Static Memory Dependence Analysis for Hardware Optimizations

Jean-Michel Gorius

Univ Rennes, ENS Rennes – F-35000, Rennes, France
jean-michel.gorius@ens-rennes.fr

Supervisors:

Alexandra Jimborean
Uppsala University
Uppsala – Sweden

Alberto Ros
University of Murcia
Murcia – Spain

Internship location and dates: University of Murcia, 03/02/20–26/06/20

Abstract. Modern Out-of-Order hardware can determine the memory dependencies between individual instructions and change their execution order to maximize pipeline usage, executing non-dependent instructions during long-latency memory accesses. However, this kind of computation is redundant, as the information computed by the hardware at run-time is available to the compiler at compile-time. We introduce a collaborative software-hardware approach that leverages the LLVM compiler infrastructure to perform memory disambiguation at compile-time. We show that a majority of load instructions can benefit from such a technique for a wide range of workloads.

Keywords: LLVM, static analysis, memory dependence, store queue, HW/SW co-design

1 Introduction

Modern high-performance processors are based on out-of-order (OoO) architectures. When executing on such a processor, a program's instructions can be reordered before being executed, which can lead to increased performance for a wide range of workloads. On current hardware, the gap between CPU speed and memory speed is increasing, with memory operations becoming more and more of a bottleneck for program execution [4]. Out-of-order execution tolerates the latency of multi-cycle operations by executing independent instructions concurrently. The new execution order must preserve program correctness, and thus, the processor needs to execute inter-dependent instructions in the right order. There are two different kinds of dependence between instructions: register dependencies and memory dependencies. The former are known statically, and the processor can use register renaming to resolve true and false dependencies between instructions. On the other hand, memory dependencies have to be determined dynamically by the processor during program execution. This task, which is often referred to as *memory disambiguation*, introduces the need for complex hardware components designed for low-level load-store alias analysis.

The following sections describe memory dependence handling in modern hardware and motivate the use of cooperative software-hardware approaches to reduce the hardware logic's complexity for memory disambiguation. Section 1.1 gives a broad overview of memory dependence handling in hardware. Section 1.2 focuses on store-load dependencies and hardware implementations for data forwarding between stores and loads. The memory disambiguation system is one of the least scalable parts of modern processor architectures because of its intrinsic complexity. Section 1.3 presents a few existing approaches to improve the scalability of memory disambiguation.

1.1 Memory dependence handling

Out-of-order hardware needs to obey memory dependencies between instructions while providing high-performance execution. The issue that arises when trying to handle load and store instructions is that, in many cases, the memory address is not known until the load or store executes. Therefore, determining the dependence or independence of loads and stores has to be handled after their execution, or at least after partially executing them to resolve the source or destination address. Since the CPU executes instructions out of program order, a recent load instruction can have its address resolved before an older store's address is known. If the store location happens to overlap with the load's address and width, then the load should only be executed after knowing what value would be stored to said location. There are three different ways to deal with such unknown addresses. The *conservative* approach is to stall the load until all previous stores have computed their addresses or even retired. The CDC 6600 computer [14] used such a conservative memory disambiguation scheme. On the other hand, a very *aggressive* method would be to assume that every load is independent of unknown-address stores and schedule the load as soon as it is ready to be processed. This approach comes at a high cost, since the hardware needs to revert execution in case there is an actual dependence between the load and an unknown-address store. Modern processors follow a more elaborate strategy that makes use of a specialized *predictor* to assess if a load is independent of unknown-address stores that precede it [7, 2]. A recovery mechanism is still needed in case of a misprediction.

1.2 Out-of-Order completion of memory operations

In order to enable the OoO execution of memory instructions, the OoO engine needs to detect the memory dependence of a given load instruction on a previous store instruction. Even assuming that we know all addresses of past stores when resolving the load address, we still need to check whether it is dependent on a past store and if so, we need to forward the data that is written by the store to the load instruction. In modern high-performance processors, instead of waiting until all previous stores commit to memory, the hardware keeps a list of pending stores in a *store queue* (SQ), and a list of pending loads in a *load queue* (LQ) [9]. Those two queues can be combined in a single load/store queue. The following paragraph illustrates the operation of the SQ.

When the processor encounters a store instruction in the program, it registers it in the SQ. When such an instruction finishes execution, it writes its address and data in its SQ entry. When a next load instruction computes its address, it searches the SQ with said address for a store that writes at least to a part of the loaded memory location. Missing parts of the load value are queried from the memory hierarchy. If there are one or more entries in the SQ that write to the load location, then the hardware needs to combine those values and forward them to the load.

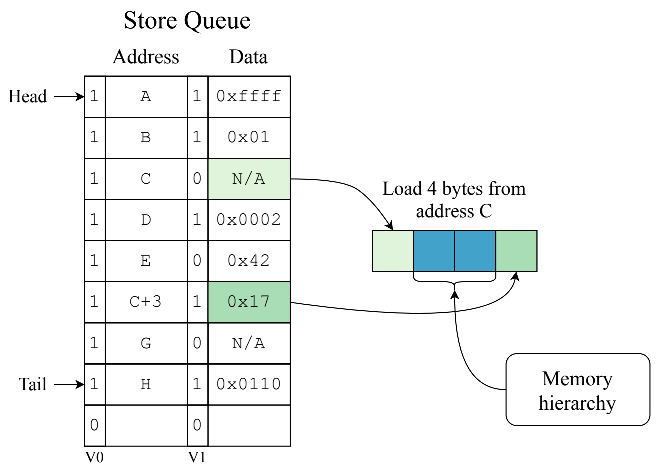

Fig. 1: Reconstructing load data from the store queue and the memory hierarchy.

Figure 1 illustrates the process of resolving memory dependencies between a load instruction and several store instructions from the SQ. The store queue contains one store in each row, with the most recent store located at the tail of the queue. Each entry contains the destination address and the written data, alongside a validity bit for each field. In this example, we assume that all store addresses have been resolved and are valid. Let us further assume that the program's next instruction is a load of four bytes from address `C`. Once the load address has been resolved, the SQ is traversed from tail to head and the range of loaded memory locations is compared against the stored addresses. The hardware finds a first match for the store at address `C+3` and a second one for a store at address `C`. Since there is no additional match in the SQ for the two remaining bytes, they need to be fetched from the memory hierarchy. Once the data for the store at address `C` is known, the value is constructed and stored in the load's destination.

1.3 Memory disambiguation scalability issues

Searching the store queue for every load can be very expensive, as one SQ traversal involves three different types of searches at the hardware level: a content addressable search to check for address matches, a range search based on the address and size of both the load and earlier store instructions, and an age-based search that needs to determine the last written values in the SQ. Moreover, as illustrated in figure 1, load data can sometimes be the combination of multiple source values. Reducing the complexity of the memory disambiguation logic would make it possible to design more resource-efficient processors and increase the scalability of the hardware [13, 5]. Several techniques have been developed towards this goal, some at the hardware level and others at the software level.

The NoSQ design [13] introduces a microarchitecture that performs store-load communication without a store and a load queue. Instead, it introduces a new hardware component designed for speculative memory bypassing [8] based on an accurate store-load communication predictor. Another way to achieve higher architectural scalability is to use cooperative software-hardware designs that lift some of the operations usually performed in hardware to the software level. Huang et al. [5] introduce such an approach to memory disambiguation. By using a binary parser to identify loads that can safely avoid the memory disambiguation checks, the authors can insert special-purpose instructions to inform the underlying hardware not to perform a store queue traversal when encountering said loads. This approach is based on the assumption that the physical ISA of the processor can be extended by decoupling it from the architected ISA.

We propose a similarly cooperative approach to memory disambiguation that leverages the existing alias analysis infrastructure of the compiler to detect store-load dependencies at compile-time.

2 Analyzing memory dependencies at compile time

We base our work on the observation that hardware replicates at run-time the memory dependence analysis performed statically by the compiler, thus paying a performance overhead and added complexity (which translates to higher energy costs). We transfer part of this complexity to the compiler and implement a compiler pass that performs static memory dependence analysis and communicates this information to the target processor. We implement our analysis pass using LLVM [6]. Our work focuses on x86 platforms, but it could be extended to other architectures by taking advantage of LLVM's target-independent code generation infrastructure.

The following sections describe how we communicate alias information to the hardware (section 2.1) and briefly introduce the LLVM alias analysis infrastructure (section 2.2). Section 2.3 presents our analysis pass in more details.

2.1 Communicating store-load dependences to the hardware

When traversing the store queue upon encountering a load, the hardware is looking for the nearest store that writes to parts of the loaded memory location.

We introduce a method that gives hints to the processor about where the first aliasing store is likely to be located in the store buffer by providing a lower-bound estimate on the number of non-aliasing stores that precede a given load instruction in the program. We forward this number to the hardware by marking load instructions using a combination of no-op instructions inserted before the load. By adding a marker detection mechanism to the memory disambiguation logic, we can instruct the processing logic to skip the given number of stores in the store buffer when looking for an alias. The following method is based on the assumption that the hardware includes such a marker detector.

We explored different marking strategies to find one that would best fit our purpose. Figure 2 illustrates the successive iterations we went through.

```
mov rdi, 10
xchg rdi, rdi
mov rax, [0x1234]
```

(a) Register value.

```
xor rdi, 10
xor rdi, 10
mov rax, [0x1234]
```

(b) Involutive operations.

```
xchg r9, r9     ; 1
xchg r10, r10   ; 2
mov rax, [0x1234]
```

(c) Base-8 encoding.

Fig. 2: Different load marking strategies for an aliasing distance of 10.

Our first approach stored the estimated number of non-aliasing stores (or *alias distance*) in a known register, `rdi` (figure 2a), before signaling the presence of a marker to the hardware by issuing an `xchg rdi, rdi` instruction. The main advantage of this approach is that the marker's value is easily retrieved by reading the current value of `rdi`. However, for this marking strategy to be effective, we need to reserve a register that will be used exclusively for markers. Reserving a register could lead to increased register pressure and possible performance degradations due to spilling. On the other hand, saving and restoring the marker register's value is not always possible since block terminating instructions such as `ret` also load from memory. If we were to insert a `push`/`pop` pair around the marker and the marked load, the `pop` following a block terminator would never be executed. The x86 instruction set provides a few involutive operations that we can use to avoid having to reserve a given register for our markers. A good candidate for such an approach is the `xor` instruction (figure 2b). However, since the `xor` operation modifies the value of the flags register, the insertion of markers that use such an instruction can potentially change the execution of loads that need to access the value of the flags. To avoid such changes, we need to use true no-op instructions. The x86 `nop` instructions are used by the compiler for alignment and padding purposes and are thus excluded. In order not to interfere with the compiler's output, we chose to encode the alias distance using `xchg` instructions. We chose to encode the computed alias distance in base 8, using registers `r8-15` as our digits. Figure 2c gives an example of how we would encode an alias distance of 10_{10}, or 12_8.

2.2 Alias analysis in LLVM

LLVM provides a generic alias analysis infrastructure that allows several analysis passes to be plugged in transparently. LLVM alias analysis passes work on LLVM IR, an SSA-based [3] representation of the input program. An alias analysis pass provides a method to query if two SSA values x and y alias, and returns one of three answers according to their aliasing state: `NoAlias` if x and y do not alias, `MustAlias` if x and y are guaranteed to never alias, and `MayAlias` if x and y might refer to the same memory location or object.

```
define i1 @load_fold(i64* %x, i64 %l) {
entry:
  %0 = load i64, i64* %x, align 8
  %and = and i64 %0, %l
  %tobool = icmp ne i64 %and, 0
  ret i1 %tobool
}
```

```
%1:gr64 = COPY $rsi
%0:gr64 = COPY $rdi
TEST64mr %0, 1, $noreg, 0, $noreg,
  %1, implicit-def $eflags
    :: (load 8 from %ir.x)
%2:gr8 = SETCCr 5, implicit $eflags
$al = COPY %2
RET 0, $al
```

Fig. 3: MIR load annotation example.

To be able to consider target-specific details, we implement our analysis pass at the LLVM Machine IR (MIR) level. MIR is LLVM's machine-specific intermediate representation, which provides analysis and transformation mechanisms similar to those available at the LLVM IR level. Alias information is forwarded to MIR from LLVM IR by using instruction annotations that build a correspondence between low-level instructions and LLVM IR operations. Figure 3 shows an example LLVM IR module next to the MIR code generated by the instruction selector. A load annotation is inserted at the end of the `TEST64mr` instruction that links the low-level memory load to the `%x` SSA value in the original module. By exploiting such annotations as well as target-specific knowledge about some instructions (*e.g.*, load and store width, implicit loads and stores), we can take full advantage of an alias analysis that operates at the LLVM IR level and use its results lower in the compilation pipeline.

2.3 Static memory dependence analysis

Our pass runs just before target assembly code generation, which allows us to make sure that we have access to the code after the compiler has applied all optimization passes, including MIR-level optimizations. We distinguish three different types of load instructions. We consider a load to be *`ret`-like* if it is an implicit load that does not alias with any store instruction in the generated assembly, under a few safe assumptions about compiler-generated code. *Stack load instructions* like `pop` and `leave` read values from the stack exclusively. *Explicit loads* encompass all the instructions that explicitly access memory through a memory operand. The remaining loads implicitly read from memory and are conservatively marked with an alias distance of zero.

We perform identical computations for explicit loads and `ret`-like loads, with the particularity that for the latter we consider that the load does not alias with any store in the same machine function. We can safely make this assumption for compiler-generated code for non-malicious programs, since overwriting the return address would cause several security issues [11]. For loads from the stack, we keep track of the depth of the stack by updating a depth value for each stack-manipulating instruction: an alias is found when the sign of the stack depth changes.

The alias distance computation function is described in algorithm 1 (appendix A). To get the alias distance for a given load L, we perform a depth-first traversal of the reversed control-flow graph (CFG), starting at L and working towards the machine function's root by visiting each instruction on the path until we find an alias. If L aliases with a store S in the same basic block, then we can directly set the alias distance D to the number of stores between L and S. If L's parent block B has two or more successors, we compute the number of non-aliasing stores on each path from L to an aliasing store and return the minimum of those values. In order not to loop indefinitely during the CFG traversal, we traverse back-edges at most once, and only if their origin is reachable from B. We find the back-edges and compute the associated reachability information using the dominator tree of the current machine function as well as a specialized LLVM structure that records low-level loop information, which is retrieved a the beginning of the analysis. To avoid expensive computations, we compute the reachability information only once for each basic block in the function. This approach is made possible by the intrinsic structure of the LLVM MIR representation, which does not allow arbitrary control flow incoming a block. Instead, each instruction in a basic block is guaranteed to dominate all the instructions contained in basic blocks dominated by its parent. The worklist used during the CFG traversal records the current instruction as well as the distance that was computed so far on the currently traversed path. The `AssignDistance` function makes sure that we do not change the value of D if $d \geqslant D$ and is used to perform additional checks when handling loops, as described in section 3.2.

The table in figure 4 gives an example of the alias distance D computed by our pass for a few load-store pairs from the given CFG. If we assume that L_0 aliases with S_{14}, then we traverse two different paths in the CFG. The first path goes through blocks $\{0, 1, 2, 4, 5\}$ and the second one through blocks $\{0, 1, 3, 4, 5\}$. In the first case, the alias distance is 9, and in the second one, it is 10. The SMDA pass marks L_0 with a distance of 9. Note that we do not traverse the back-edge from block 1 to block 4 since its origin does not dominate the parent block of L_0. A similar reasoning obtains the other distances in figure 4.

The heart of the alias analysis is performed using target-specific information alongside high-level alias information. Algorithm 2 (appendix A) gives an overview of the alias checking algorithm. First, we check whether the I and L instructions can alias by making sure that they access memory. Sometimes the

Load	Aliasing Stores	D
L_0	$\{S_{14}\}$	9
L_0	$\{S_5, S_{11}\}$	4
L_1	$\{S_3\}$	2
L_1	$\{S_{12}\}$	7
L_3	$\{S_1, S_{14}\}$	0
L_5	$\emptyset$	0

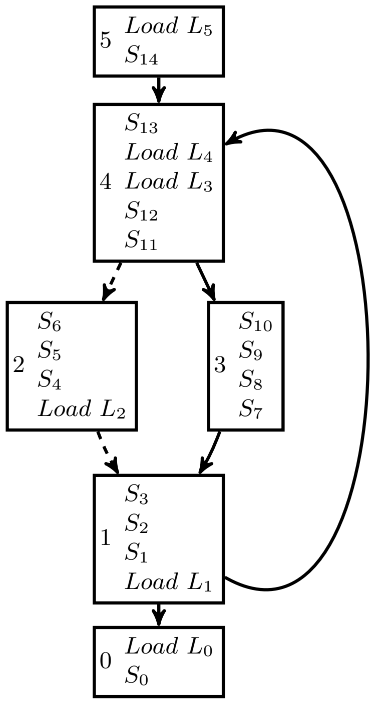
5 Load L_5 S_{14}
4 S_{13} Load L_4 Load L_3 S_{12} S_{11}
2 S_6 S_5 S_4 Load L_2
3 S_{10} S_9 S_8 S_7
1 S_3 S_2 S_1 Load L_1
0 Load L_0 S_0

Fig. 4: Example alias distance computation by the SMDA pass.

target can determine if two memory accesses are disjoint by using target-specific information about the ISA. The `MemAccessesAreDisjoint` function ensures that we do not perform unneeded computations if the target can readily determine if there can be no alias between L and I. After performing these initial checks, we loop over all combinations of memory operands from L and I to find a potential alias. Since querying the alias analysis can be expensive, we further rely on target-known information to exclude common cases.

The first common case arises when either I or L accesses a *pseudo-location.* A pseudo-location is a virtual memory location that the compiler uses to represent memory areas not known by the LLVM IR. These locations include the function's stack frame (*e.g.* an access to a spill slot), the area below the stack frame (*e.g.* argument space) and the function's constant pool. If L or I accesses such a pseudo-location, we fall back to the LLVM-provided `MayAliasPseudo` function. The latter uses the machine function's frame information to determine if the pseudo-value can alias with another value or not. If not, we proceed to the next pair of memory operands. If the base addresses of two memory operands M_L and M_I is known to be the same, or if they access the same pseudo-location, then we can quickly determine if there is an alias by checking if the two memory locations overlap. The last remaining case arises when the base addresses of the memory accesses are different. In this case, we cannot straightforwardly determine if there is an alias, so we query the alias analysis for M_I and M_L's aliasing status.

3 Discussion

Several challenges emerge when trying to analyze code using the approach exposed in section 2.3. The static memory dependence analysis pass implicitly relies on the compiler infrastructure to provide accurate and complete information regarding instructions, which may not always be available. Section 3.1 discusses some of the issues we faced when designing our analysis pass. In order to increase the accuracy of our analysis, we have to take loops into account when computing alias distances. Section 3.2 exposes some of the tradeoffs that go into implementing proper loop handling. The implementation detailed in section 2.3 focuses on the `MayAlias` metric. Section 3.3 exposes the key differences between the `MayAlias` and `MustAlias` metrics in the SMDA pass.

3.1 Dealing with incomplete load and store information

The LLVM compiler infrastructure provides a variety of different low-level transformation passes that operate on the MIR. These passes modify existing instructions, remove some of them, insert new ones and update the associated metadata accordingly. However, information can sometimes be lost during such a transformation. Our analysis pass conservatively handles instructions that are known to be loads or stores from the x86 ISA specification but are not reported as loads or stores by LLVM. At the time of this writing, information loss often occurs when the compiler backend optimizes away stack allocations that were inserted as part of a function's prologue to accommodate local parameter storage. In case a load instruction is not reported as accessing memory by LLVM, the SMDA pass marks it with an alias distance of zero. For store instructions, we currently consider them as aliasing with every load if the compiler does not report them as writing to memory. This approach results in a slight loss of precision in our markers but preserves their correctness.

3.2 Handling loops

The alias distance computation described in section 2.3 is slightly simplified. The SMDA pass integrates a mechanism to detect and mark loops, and to report a special alias distance $D^\star$ in case there is a loop on the path from an aliasing store S and the load L to mark. The hardware can use the loop entry and exit markers to record the amount of executed stores in a loop and its trip count. With this information, whenever the processor executes a load marked with $D^\star$, it can retrieve the amount of non-aliasing stores from the last executed loop. By doing so, we assume that the number of non-aliasing stores in a loop will likely be significantly larger than the number of non-aliasing stores on the path from S to L outside of the loop.

3.3 MustAlias analysis

We conduct a static dependence analysis for both `MayAlias` and `MustAlias` metrics based on compiler-provided alias information. The latter metric is more aggressive and can provide us with more accurate results in some cases. The main difference between the `MayAlias` and `MustAlias` analyses resides in the way we implement the alias query function (algorithm 2). While the procedure exposed in section 2.3 describes how we find stores that may alias with a given load, when looking for must-alias stores, we try querying the alias analysis even in the case where the base address or width of the access is unknown. In case there is no alias information available, we consider that S and L do not alias.

4 Validation

To validate our results and compare the marked alias distances with the expected alias distances at run-time, we develop a plugin for Intel PIN [10], a tool that allows us to instrument the binaries generated by our modified version of `clang` with instruction-level granularity. Section 4.1 briefly presents the operation of our plugin and section 4.2 exposes and comments on results obtained using this tool.

4.1 Dynamic binary instrumentation

Our PIN plugin analyzes the given binary before executing it, inserting predicated calls to routines that register memory accesses for each load and store instruction in the program. The binary is then executed and its execution flow is diverted each time it encounters a memory accessing instruction. In the case of a store, it records the store location into a ring buffer representing the store queue. When executing a sequence of load marking instructions, the alias distance is recorded and passed to the next predicated call that registers a load instruction. This call checks if the load address is found in the store buffer and, if so, indicates whether the marked alias distance is less than or equal to the distance to the nearest aliasing store in the store buffer. The size of the store buffer is configured for both the SMDA pass and the validation tool, and all alias distances are clamped to this value in the compiler.

The results presented in section 4.2 were obtained by running benchmarks instrumented by the tool described in the preceding paragraph. Because of such instrumentation's dynamic nature, we can report results only for instructions that are encountered during a particular execution of the benchmark. Going back to the CFG in figure 4, if we assume that, for a given program execution, the dashed edges are never executed, load L_2 will not be reported as marked by the validation tool.

4.2 Evaluation

We evaluate our static memory dependence analysis pass on two sets of benchmarks: the Splash-3 [12] and SPEC CPU 2017 [1] benchmark suites. All results were obtained using the SMDA pass based LLVM version `11.0.0git-931a68f`.

Benchmark	**Loads** (% total instr.)	**Marked 0** (% total loads)	**Marked 1–5** (% total loads)	**Marked 6–25** (% total loads)	**Marked 26–99** (% total loads)	**Marked $D^\star$** (% total loads)
`perlbench`	16.775	70.620	28.415	0.965	0	0
`gcc`	11.682	76.621	20.892	2.403	0.042	0.042
`mcf`	15.054	93.736	5.291	0.010	0	0.963
`lbm`	14.882	67.083	14.208	18.709	0	0
`omnetpp`	10.851	74.915	19.773	5.247	0.065	0
`xalancbmk`	14.821	86.150	12.944	0.865	0	0.042
`deepsjeng`	13.951	80.437	15.645	3.892	0	0.026
`imagick`	8.705	98.818	1.073	0.048	0	0.060
`leela`	14.441	81.846	15.654	2.500	0	0
`nab`	14.504	87.470	12.062	0.430	0.037	0.001
`xz`	11.668	77.355	19.999	1.727	0	0.919

Table 1: `MayAlias` results for C/C++ SPEC CPU 2017 benchmarks.

Table 1 gives an overview of the marker distribution for SPEC CPU benchmarks using the `MayAlias` metric. In most cases, we notice that load instructions represent up to 15% of the total number of executed instructions. Improving the store queue mechanism described in section 1.2 can thus have a significant impact on the overall execution of a program, which further motivates the need to develop scalable load/store handling mechanisms. Even with a conservative handling of alias information, we are still able to show that, in nearly all benchmarks, more than 12% of executed loads could bypass at least one aliasing check in the store queue. In some particularly favorable cases such as for the `lbm` executable, nearly 20% of loads can skip at least five store queue entries, which represents 10% of the size of the store queue in Skylake-based processors and more than 15% for Haswell platforms. We also note that the amount of loop markers $D^\star$ is not significant in our measurements for the `MayAlias` metric.

Benchmark	**Loads** (% total instr.)	**Marked 0** (% total loads)	**Marked 1–5** (% total loads)	**Marked 6–25** (% total loads)	**Marked 26–99** (% total loads)	**Marked $D^\star$** (% total loads)
`barnes`	15.550	22.619	73.006	4.199	0.176	0
`fmm`	6.478	41.191	48.066	10.694	0.043	0.006
`ocean` (cont.)	14.589	8.456	40.684	31.562	10.086	9.212
`ocean` (non-cont.)	15.874	36.535	29.674	4.984	6.857	21.949
`radiosity`	13.505	32.988	59.143	6.568	1.301	0
`volrend`	10.775	6.523	15.846	44.528	33.103	0
`cholesky`	11.429	14.573	28.700	52.116	4.610	0
`fft`	5.273	2.232	74.714	18.873	4.181	0
`lu` (cont.)	11.183	4.052	8.440	24.384	63.093	0.031
`lu` (non-cont.)	11.845	1.671	8.824	14.801	74.605	0.099
`radix`	6.422	43.696	0	18.719	37.585	0

Table 2: `MustAlias` results for Splash-3 benchmarks (error rate $< 2\%$).

The conservative nature of the `MayAlias` analysis limits the amount of information we can gain from it when targeting larger marker values. Table 2 illustrates how the `MustAlias` metric can help us achieve higher marking distances with the help of the compiler. Since this analysis handles aliases more aggressively, it can sometimes introduce marking errors in the binary. In our experiments, we record an erroneous marking rate of less than 2% overall, and less than 0.5% on average. Our results show that in nearly all cases, more than 60% of loads can skip at least one entry in the store queue, and more than 10% of them can skip the first five entries. We notice that in quite a few benchmarks, nearly half of a typical store queue with 56 entries can be skipped by more than 30% of the program's loads. Even though the `MustAlias` marking strategy is more aggressive, we observe that the number of $D^\star$ distances stays near zero in most benchmarks. This suggests that the loop handling described in section 3.2 could likely be skipped to further reduce area cost and design complexity.

5 Conclusion and future work

The memory disambiguation logic is one of the least scalable parts in modern processor architectures. Several techniques have been developed to try to reduce the complexity of the store queue and improve the scalability of the processor. In this report, we presented a collaborative approach that allows us to leverage the alias analysis performed by the compiler to lift memory disambiguation from hardware to software. We showed that a significant portion of the program's loads could benefit from bypassing aliasing checks in the store queue.

Future work will focus on both the software and hardware aspects of this project. On the compiler side, extending the existing analysis pass to support inter-procedural alias queries would allow us to produce more accurate markers by modeling the actual flow of instructions more precisely. Work has started on implementing the logic to support our load markers in the SNIPER hardware simulator.

Acknowledgements

First and foremost, I would like to thank my supervisors, Alexandra Jimborean (Uppsala University, Sweden) and Alberto Ros (University of Murcia, Spain), for their insights and support during the development of this project. I feel lucky to have been able to continue to work remotely during this internship despite recent events. This project will serve as a starting point for Sawan Singh (University of Murcia, Spain) in his work on compiler-assisted hardware optimizations. I would like to thank him for his patience during the SMDA debugging sessions and wish him success during his PhD. at the University of Murcia.

References

1. Bucek, J., Lange, K.D., v. Kistowski, J.: Spec cpu2017: Next-generation compute benchmark. In: Companion of the 2018 ACM/SPEC International Conference on

Performance Engineering. p. 41–42. ICPE '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3185768.3185771

2. Chrysos, G.Z., Emer, J.S.: Memory dependence prediction using store sets. In: Proceedings of the 25th Annual International Symposium on Computer Architecture. p. 142–153. ISCA '98, IEEE Computer Society, USA (1998). https://doi.org/10.1145/279358.279378
3. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '89, ACM Press, New York, NY, USA (1989)
4. Hennessy, J.L., Patterson, D.A.: Computer Architecture, Sixth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edn. (2017)
5. Huang, R., Garg, A., Huang, M.: Software-hardware cooperative memory disambiguation. In: The 12th International Symposium on High-Performance Computer Architecture. pp. 244–253 (2006)
6. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO '04, IEEE, Washington, DC, USA (2004)
7. Moshovos, A., Breach, S.E., Vijaykumar, T.N., Sohi, G.S.: Dynamic speculation and synchronization of data dependences. In: Proceedings of the 24th Annual International Symposium on Computer Architecture. p. 181–193. ISCA '97, Association for Computing Machinery, New York, NY, USA (1997). https://doi.org/10.1145/264107.264189
8. Moshovos, A., Sohi, G.S.: Speculative memory cloaking and bypassing. Int. J. Parallel Program. **27**(6), 427–456 (Dec 1999). https://doi.org/10.1023/A:1018776132598
9. Park, I., Chong Liang Ooi, Vijaykumar, T.N.: Reducing design complexity of the load/store queue. In: Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36. pp. 411–422 (2003)
10. Patil, H., Cohn, R., Charney, M., Kapoor, R., Sun, A., Karunanidhi, A.: Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In: 37th International Symposium on Microarchitecture (MICRO-37'04). pp. 81–92 (2004)
11. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur. **15**(1) (Mar 2012). https://doi.org/10.1145/2133375.2133377
12. Sakalis, C., Leonardsson, C., Kaxiras, S., Ros, A.: Splash-3: A properly synchronized benchmark suite for contemporary research. In: 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 101–111 (2016)
13. Sha, T., K. Martin, M.M., Roth, A.: NoSQ: Store-load communication without a store queue. In: 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06). pp. 285–296 (2006)
14. Thornton, J.E.: The CDC 6600 project. IEEE Ann. Hist. Comput. **2**(4), 338–348 (Oct 1980). https://doi.org/10.1109/MAHC.1980.10044

A Algorithms

Algorithm 1: ComputeAliasDistance(L, TII, MFI, AAR)

```
/* L: Load instruction                                          */
/* TII: Target instruction information                          */
/* MFI: Machine function frame information                      */
/* AAR: Alias analysis results                                  */
$D \leftarrow 0$; Skip ←false; WL ← $\{L, 0\}$;
while WL is not empty do
    $E \leftarrow$ PopBack(WL);
    $(I, d) \leftarrow$ ($E$.CurrInstr, $E$.CurrDist);
    $B \leftarrow$ GetParentBB($I$);
    while $I$ does not refer to the start of $B$ do
        if MayAlias($I$, $L$, TII, MFI, AAR) $\vee$ IsCall($I$) then
            AssignDistance($D$, $d$); Skip ← true;
        else
            foreach memory operand $m$ in $I$'s operands do
                if $m$ is a store then $d \leftarrow d + 1$;
            end
            if $d \geqslant D$ then Skip ← true, break;
            $I \leftarrow$ instruction preceding $I$ in $B$;
        end
    end
    if Skip then Skip ← false;
    else
        if $B$ has no predecessors in $F$ then AssignDistance($D$, $d$) ;
        else
            foreach predecessor $P$ of $B$ in $F$ do
                if (($P$, $B$) is not a back-edge) $\vee$
                    (($P$ is reachable from $B$) $\wedge$ (($P$, $B$) was never taken)) then
                    WL ← Append(WL, $P.Last$, $d$)
                end
            end
        end
    end
end
return $D$
```

Algorithm 2: MayAlias(S, L, TII, MFI, AAR)

```
/* $S$: Potential store instruction                          */
/* $L$: Load instruction                                     */
/* TII: Target instruction information                       */
/* MFI: Machine function frame information                   */
/* AAR: Alias analysis results                               */
if $S$ nor $L$ loads from or stores to memory then
    return false
else if both $S$ and $L$ don't store to memory then
    return false
else if MemAccessesAreDisjoint(TII, $S$, $L$) then
    return false
end
Alias ← false;
foreach memory operand $M_L$ in $L$'s operands do
    foreach memory operand $M_S$ in $S$'s operands do
        $B_{S,L} \leftarrow$ base address of $M_{S,L}$;
        if $B_S$ or $B_S$ is unknown then return true;
        $O_{S,L} \leftarrow$ offset from the base address for $M_{S,L}$;
        $W_{S,L} \leftarrow$ width of the access for $M_{S,L}$;
        $m_{\text{offset}} \leftarrow \min(O_S, O_L)$;
        if $B_S \neq B_L$ then
            if ($M_S$ accesses a pseudo-location $P_S$) $\wedge \neg$ MayAliasPseudo($P_S$,
              MFI) then
                continue
            end
            if ($M_L$ accesses a pseudo-location $P_L$) $\wedge \neg$ MayAliasPseudo($P_L$,
              MFI) then
                continue
            end
        end
        if $B_S = B_L \vee P_S = P_L$ then
            if $W_S$ or $W_L$ is unknown then return true;
            $M_{\text{offset}} \leftarrow \max(O_S, O_L)$;
            $w \leftarrow$ if $M_{offset} = O_S$ then $W_S$ else $W_L$;
            if $m_{offset} + w > M_{offset}$ then return true else continue;
        end
        if there are no alias analysis results for $M_L$ and $M_S$ in AAR then
            return true
        end
        if AAR does not contain NoAlias for the ($M_S$, $M_L$) pair then
            return true
        end
    end
end
return false
```