Finding and Exploiting Memory-Level-Parallelism in Constrained Speculative Architectures

KIM-ANH TRAN
Abstract

One of the main performance bottlenecks of processors today is the discrepancy between processor and memory speed, known as the memory wall. While the processor executes instructions at a high pace, the memory is too slow to provide data in a timely manner. Load instructions that require an access to memory are referred to as long-latency or delinquent loads. To prevent the processor from stalling, independent instruction past the load may execute, including independent loads. Overlapping load operations and thus their latency is referred to as memory-level parallelism. Memory-level parallelism (MLP) can significantly improve performance. Today's out-of-order processors are therefore equipped with complex hardware that allows them to look into the future and to select independent loads that can be overlapped. However, the ability to choose future instructions and speculatively execute them in advance introduces complexity, increased power consumption and potential security risks. In this thesis we look at constrained speculative architectures that struggle to hide memory latencies as they are constrained by design, by their resources, or by security. We investigate ways for the compiler to help them in finding MLP, with the ultimate goal to avoid processor stalls as much as possible. This includes small energy-efficient processors that lack the ability to look-ahead far enough to find independent loads, but also large processors that are disallowed to speculatively execute independent loads due to enforced security measures to circumvent side-channel attacks. We identify the reason for their limitation and propose software transformations and hardware extensions to overcome their restrictions.

Keywords: Memory-level-parallelism, Energy-efficiency, Performance, Compiler, Instruction Scheduling, SW/HW Co-Design

Kim-Anh Tran, Department of Information Technology, Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Kim-Anh Tran 2020

ISSN 1651-6214
urn:nbn:se:uu:diva-402642 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-402642)
To Stephan and Felix.
This thesis is based on the following papers, which are referred to in the text by their Roman numerals.


Reprints were made with permission from the publishers.
The Author’s Contribution

I Main author. Implementation and experiments by main author, manuscript with co-authors.

II Main author. Implementation and experiments by main author, manuscript with co-authors.

III Main author. Implementation on compiler side by main author, experiments and manuscript with co-authors.

IV Main author. Implementation on compiler side and experiments by main author, manuscript with co-authors.

Related Publications


Contents

1 Introduction ................................................................................................ 11
  1.1 Finding MLP on Small Processors ............................................... 12
  1.2 Finding MLP on Secure Processors .............................................. 15
  1.3 Thesis Structure .............................................................................. 17

2 Overcoming Memory Latency on Small Processors ........................................ 20
  2.1 Latency-aware Instruction Scheduling for General Purpose Code ......................................................... 22
    2.1.1 Contribution 1: Targeting Resource-Constrained Out-of-Order (OoO) Processors .................................... 23
    2.1.2 Contribution 2: Stressing the Limitations of Conservative Scheduling ................................................. 26
    2.1.3 Contribution 3: Targeting Design-Constrained In-Order (InO) Processors ............................................... 27

3 Overcoming Memory Latency on Secure Speculative Processors .................. 30
  3.1 Efficient Invisible Speculation through Delay-on-Miss ......... 31
  3.2 Shadow-aware Instruction Scheduling for Delay-on-Miss Hardware .............................................................................. 33
    3.2.1 Contribution 4: Targeting Security-Constrained Processors ............................................................. 33

4 Summary and Discussion .......................................................................... 38
  4.1 Discussion: Clairvoyance under the Light of Security ................ 39

5 Svensk Sammanfattning ............................................................................ 42

6 Acknowledgements ................................................................................... 44

References ........................................................................................................ 46
Abstract

One of the main performance bottlenecks of processors today is the discrepancy between processor and memory speed, known as the memory wall. While the processor executes instructions at a high pace, the memory is too slow to provide data in a timely manner. Load instructions that require an access to memory are referred to as long-latency or delinquent loads. To prevent the processor from stalling, independent instruction past the load may execute, including independent loads. Overlapping load operations and thus their latency is referred to as memory-level parallelism.

Memory-level parallelism (MLP) can significantly improve performance. Today’s out-of-order processors are therefore equipped with complex hardware that allows them to look into the future and to select independent loads that can be overlapped. However, the ability to choose future instructions and speculatively execute them in advance introduces complexity, increased power consumption and potential security risks.

In this thesis we look at constrained speculative architectures that struggle to hide memory latencies as they are constrained by design, by their resources, or by security. We investigate ways for the compiler to help them in finding MLP, with the ultimate goal to avoid processor stalls as much as possible. This includes small energy-efficient processors that lack the ability to look-ahead far enough to find independent loads, but also large processors that are disallowed to speculatively execute independent loads due to enforced security measures to circumvent side-channel attacks. We identify the reason for their limitation and propose software transformations and hardware extensions to overcome their restrictions.
1. Introduction

While processor speed is continuously increasing, memory speed keeps lagging behind. This phenomenon, named the *memory wall* by Wulf et al. [51], still continues to limit the performance on systems even two decades later. As an effort to mitigate the long delays caused by memory accesses, processors continue execution past unresolved loads to find independent instructions that can be scheduled while waiting for the data to arrive. If other independent loads can be found, scheduling them close to each other even allows for parallel accesses to memory and exploits Memory-Level Parallelism (MLP). The processor’s ability to look-ahead is determined by the *instruction window size*, which defines the number of instructions that it sees. The bigger the window size, the more flexibility the processor has to execute other instructions while waiting for memory accesses to complete.

There are however reasons to limit the size of the instruction window, and even the ability to execute instructions speculatively. One of the reasons is energy-efficiency. Energy-efficiency has not only become a limiting factor for performance, but is also a necessity in the development of mobile devices or IoT. A big instruction window requires hardware resources that exceed the energy budget. Different demands therefore prefer smaller cores and trade performance for efficiency. Another reason to limit the speculativeness of processors is security. *Speculative execution*, i.e. the execution of instructions ahead of time that may turn out not to be needed, has recently been discovered to be vulnerable for speculative side-channel attacks [27, 32]. Although speculative execution does not change the architectural state, it does leave traces in the micro-architectural state, which can be exploited. As a response to the newly discovered security vulnerabilities, new hardware solutions now delay speculative execution to prevent sensitive data being leaked through side-channels [52, 45, 49].

In this thesis we look into the potential of the compiler to find MLP if the hardware cannot. The thesis is divided into two parts. The first part analyzes resource- and design-constrained speculative architectures that lack the required hardware to look ahead far enough to find loads to achieve MLP (Section 1.1). The second part analyzes security-constrained hardware designs that mitigate speculative side-channel attacks by disallowing speculative MLP (Section 1.2).
1.1 Finding MLP on Small Processors

In the first part of our thesis (contributions I-III) we focus on techniques to efficiently hide memory latencies on small in-order and out-of-order processors by clustering loads to increase MLP and overlapping their latencies with independent loads, and thus increase Instruction-Level Parallelism (ILP). These small processors especially suffer from delinquent loads as they, in contrast to their aggressive out-of-order processor counter-part, lack the required hardware to look ahead and find independent instructions to compensate for the slow memory; i.e. they are more prone to stall if an application has many memory accesses. Their selling point, however, is the energy efficiency, which is fundamental in the development of hand-held devices.

In order to overcome the penalty of long latency loads, also known as delinquent loads, previous work has proposed solutions involving software prefetching [7, 25, 47, 1, 24, 29, 19], software pipelining [40, 5, 50, 17, 41, 20], and software-hardware co-designs [22, 23, 46, 16, 26, 47]. The common idea is to fetch data early in advance (whether into the cache, registers, or custom architectural structures), and to provide data to the consuming instructions in a timely manner to avoid latencies that cannot be hidden. Fetching data early in advance can help to avoid last-level cache misses. If data fetches are further more clustered the outstanding latencies can be overlapped and high levels of MLP can be achieved, which allows for better hardware resource usage. Clustering and executing prefetches or loads early in advance is not easy to get right. Software prefetching inserts additional instructions, and only pays off if they are accurate and timely, otherwise unnecessary data is fetched that thrashes the cache and decreases performance. Reordering loads, on the other hand, does not introduce additional instructions, but requires care: instructions may not be arbitrarily reordered, due to Read-after-Write (RAW) dependencies. In addition reordering across basic blocks may become difficult in the presence of complex control-flow.

We focus on load reordering as a technique to hide latencies on these processors, since instruction count overhead can only be tolerated to a limited extent. Load reordering introduces a number of challenges that need to be addressed, to be able to efficiently cluster them to achieve MLP and ultimately to hide long latencies. In the following, we list the challenges that come with load reordering.

**Unknown Memory Dependencies.** Memory instructions cannot be arbitrarily reordered in code. Loads and stores may refer to the same memory location (they may be *aliases*).

Figure 1.1a shows a load and store instruction that may be aliasing according to static alias analysis. If the *may-aliasing* instructions turn out to be *must-alias*es at runtime, the load instruction is not allowed to be executed before the store instruction, as we would otherwise offend RAW dependencies. However, given that may-aliasing rarely turn out to be must-aliasing at runtime [15],
Figure 1.1. Challenges to consider when reordering load instructions to achieve MLP and hide memory latency. The example code is focused on loop bodies, since our work targets hot loops.
not reordering may-aliasing loads would severely limit the potential for hiding outstanding latencies. As the compiler has to be conservative, many load-store pairs are marked as *may-aliasing*, and as such, unknown memory dependencies hinder early execution of long latency loads.

**Chains of Dependent Loads.** Chains of dependent loads cannot access memory in parallel, as the loads depend on the result of the previous load, consider Figure 1.1b. The dependent loads are an immediate use of the previous long-latency load, and if executed close in time, they are serialized, and occupy hardware resources even though they cannot be executed in parallel. On a stall-on-use InO processor, the second load would directly stall the processor. In order to prevent these loads to unnecessarily occupy hardware resources, reordering techniques need to find ways to schedule them as far as possible apart from each other.

**Complex Control-Flow.** Loads may be part of (nested) conditional blocks, see Figure 1.1c. Traditional software pipelining techniques rely on if-conversion in order to transform control dependencies into data dependencies. For more complex code (involving pointers and indirections), nested or big basic blocks, if-conversion is not applicable or not advantageous. Since general purpose programs contain complex control-flow, we also need to find answers to dealing with control-flow when reordering load instructions.

**Register Pressure.** Splitting loads from their uses naturally increases register lifetime. The more loads that are clustered, the higher the register pressure. Consider Figure 1.1d: registers are not only allocated for the loads to be clustered, but also used for address computation and others (such as, branch conditions). As a result, register pressure is a limiting factor for how many loads can be clustered, and therefore a deciding factor for performance. If too many registers are occupied unnecessary spill code will be introduced, which increases the instruction count. For an InO core, the spilled code, which represents a new use of the load, would introduce stalls.

In the first part of the thesis we introduce instruction scheduling techniques that aim at overcoming the penalty of long latency loads on small processors (InO and limited OoO processors), while addressing the issues of unknown memory-dependencies, chains of dependent loads, complex-control flow and register pressure. We make the following contributions:

1. We present Clairvoyance, a code transformation technique that allows small resource-constrained OoO processors to hide long latency penalties (Paper I). Clairvoyance creates Access and Execute phases using loop unrolling and instruction reordering, and overlaps outstanding loads to increase MLP. For its most conservative version it achieves a performance improvement of 7% compared to the code optimized with standard O3 on memory-bound benchmarks.
2. We extend Clairvoyance to address the limitations of conservative scheduling (Paper II). Paper I shows that a speculative version (i.e.
an oracle) that assumes perfect alias analysis can bring significant performance improvements. In order to accelerate the conservative version, Paper II integrates better alias analysis and introduces new heuristics to insert prefetches instead of loads in cases of register pressure. We further extend its evaluation to include hardware counters (the number of cycles, instructions, loads, and stores) to gain more insight into the results, such as how Clairvoyance transformations improve performance, and how much overhead it introduces. With its extension Clairvoyance achieves a performance improvement of 14% on top of O3 standard optimizations for memory-bound applications.

3. We present Software Out-of-Order Processing (SWOOP), a software-hardware co-design technique that achieves Clairvoyance-like code transformations, but customized for InO processors (Paper III). Stall-on-use InO processors are constrained by design, as they lack the ability to look ahead-of-time and find independent instructions to execute. Unlike OoO processors, they directly stall on the first use of the load. This requires extra care, as every use will directly cause a processor stall, if data has not been fetched in a timely manner. While Clairvoyance works for limited OoO processors, it introduces too much overhead on the InO core. Therefore, we introduce lightweight hardware support to allow Clairvoyance-transformations without register spilling, while giving the hardware the control to execute more Access phases in case last-level cache misses have been detected. Overall, SWOOP achieves to improve performance by 34% and energy-efficiency by 23% compared to the baseline InO core.

1.2 Finding MLP on Secure Processors

Speculative execution refers to the execution of speculative instructions ahead of time, i.e. instructions that are speculated to be needed in the future. One example is the execution of instructions past unresolved branches. If the branch target is not yet resolved, the branch predictor predicts the future path based on its previous history, such that instructions from that path can be executed ahead-of-time. Speculation can significantly increase performance as it makes better use of existing resources. In the rare case of a misprediction, these speculatively execute instructions that have been fetched and executed prematurely are removed (i.e., squashed). Afterwards the processor resumes execution at the point where the misprediction started. Speculatively executed instructions do not alter the architectural state, but they leave traces to the micro-architectural state, including the memory system. Speculative loads fetch data from memory into cache that remain in cache even if the loads happen to be squashed.
Meltdown and Spectre [32, 27] are probably the best known hardware vulnerabilities that make use of speculative execution. Meltdown leaks kernel memory to an unauthorized user process by exploiting that speculative execution will continue even if the access raises an exception. Spectre leaks data that only the victim process may access, by exploiting speculative execution past branches. Figure 1.2 (victim code) and Figure 1.3 (attacker code) show a simplified example for Spectre variant 1, which allows attackers to access the victim code’s secret that is protected through a branch check. The victim code includes a function which accesses its array if the passed index is in bounds (victim, line 5). The attacker code starts it attack by training the branch predictor to record that the index of the branch check is historically valid, by calling the \texttt{victim(index)} function multiple times for a valid index (attacker, line 7). The attacker then flushes the probe array from the cache, which will be used to identify the secret value (attacker, line 12). At this point all accesses to the probe array would result in a cache miss. Assuming that the secret value lies past the victim owned array (victim, Line 2), the attacker now wants to access the secret value, which in this example is at \texttt{lstinlinearray[10000]}, and note that the index is past the actual array length. The attacker now uses the wrong index to access the secret value (attacker, Line 15). Although the index should not pass the branch check, it actually does, since the branch predictor was trained to predict that the index is valid. The load is performed and returned to the attacker (attacker, Line 15). In order to make the secret visible, the attacker now uses the secret value as an index to access its probe array (attacker, Line 18). With this access the value \texttt{probe[secret * 64]} will be fetched into cache. As soon as the processor detects that the predicted path was the wrong one, it squashes the wrongly executed instructions. Since the side-effects of the cache access to the attacker’s probe array are still visible, the attacker can find out the secret value by checking which index (the secret value) to the probe array will have the shortest loading time, since that probe array value has been speculatively loaded during the wrong execution (attacker, Line 24).

Speculative execution is a core technique for processors to achieve high performance. Completely disallowing it for the sake of security is therefore not an option. Solutions to these vulnerabilities thus need to find a way to close off the vulnerability while minimizing the performance degradation introduced by it. Hardware solutions to prevent speculative side-channel attacks on caches rely on hiding speculative updates until they become resolved [52, 44], undoing the speculative side-effects [43], or delaying speculative execution [53, 31, 45, 11]. As expected, all techniques, incur a slowdown compared to the unsecure OoO. Hiding speculative updates comes at the cost of keeping a buffer around in which the updates can be kept until they are known to be correct. When undoing the effects of speculation, overhead is added when a misspeculation is detected. Finally, delaying loads restricts the ability of the processor to execute loads and overlap latencies, which hurts performance. Existing software
solutions [13, 36, 14] only target specific variants of the vulnerabilities or rely on the user to make use of barriers and conditional move/select instructions.

In part two of the thesis (contribution IV) we focus on improving performance of hardware defenses that rely on delaying speculative loads. By delaying loads the security measures restrict the ability of the OoO to do what it is best at: to find independent instructions, including loads, and to execute them in an out-of-order fashion. It thus significantly affects its flexibility, and similarly to part one of the thesis, the potential for MLP. As a proof-of-concept, we analyze the delay-on-miss work on efficient invisible speculative execution by Sakalis et al. [45], which delays speculative loads to prevent speculative side channel attacks on the cache. Nevertheless our analysis applies to any of the delay-based solutions available in literature.

Our contribution is a software-hardware co-design that aims at finding MLP in the presence of security countermeasures that delay speculative loads. We find ways to shorten the period of time in which loads are considered as speculative, such that more loads become safe to execute at an early time, which unlocks the potential for MLP. To this end, paper IV proposes a delay-aware instruction scheduling technique and combines it with a coherence protocol that allows for non-speculative load reordering. With our changes in place we achieve, on average, a 9% improvement on the delay-on-miss work proposed by Sakalis et al., and thus reduce the gap between the insecure OoO and the secure delay-on-miss OoO by 48% (on average).

1.3 Thesis Structure

The rest of this thesis will be divided into three parts. The first part (Section 2.1) focuses on contributions I-III that apply techniques to compensate for the missing hardware to look ahead-of-time and find and overlap loads on InO and OoO processors. The second part (Section 3) covers the contribution IV, which evaluates techniques to find MLP in the presence of delayed loads.
/* Attacker code */
uint8_t probe[256 * 64];

void attacker() {
    /* Multiple calls to victim(index) with
     * valid index, thus training the branch
     * predictor */
    train();

    /* Flush probe array from cache, such that
     * all
     * accesses to probe will require a
     * long latency */
    flush(probe);

    /* Access secret value. */
    uint8_t secret = victim(10000);

    /* Use secret to fetch probe data. */
    uint8_t dummy = probe[secret * 64];

    /* At this point we try to find out the
     * secret
     * data. The secret data is the index
     * for which loading probe[index] * 64 is
     * fast,
     * since that data was speculative fetched
     * . */
    uint8_t secret = find_secret();
}

Figure 1.3. An example for Spectre variant 1, attacker code.
without compromising the security guarantees. Both parts include an overview on the related work and on the contributions of this thesis. Finally, we discuss and conclude the thesis (Section 4).
2. Overcoming Memory Latency on Small Processors

Previous work has introduced software-only and software-hardware techniques to mitigate the discrepancy between processor and memory speed. We categorize them into (i) software prefetching-based and (ii) reordering-based techniques.

**Software Prefetching.** Software prefetching \cite{7, 25, 24, 47, 1} aims at fetching data well ahead of time. In order to achieve a performance benefit, these prefetches need to be *timely* (i.e., data fetched before it is required) and *accurate* (i.e., eliminate misses). Software prefetches are inserted at a certain distance, and in batches of consecutive memory locations, the degree.

Software prefetching is especially beneficial if it targets irregular memory addresses, short streams or a large number of streams \cite{30} – basically when targeting data that the hardware prefetcher struggles to prefetch. However, software prefetching instructions need to be inserted with care, as they introduce additional instructions, may affect the performance of the hardware prefetcher (disturb the observed pattern, and consume memory bandwidth \cite{30}) and they may replace useful cache lines if applied wrongly.

In order to decide which data to prefetch, previous work has made use of profiling \cite{39}, cache modeling \cite{25}, simulation \cite{4} or multi-versioning \cite{29} to find the loads that are most likely to miss in the cache. There are, however, also a number of software prefetchers that statically decide what loads to target \cite{1}.

While software prefetching can help to eliminate long latencies, it is very difficult to achieve a high coverage of delinquent loads in general purpose code. To overcome complex control-flow, software prefetches may be executed speculatively, or require code duplication which both increase the instruction count overhead. Since this work targets energy-efficient processors, an increased number of instruction count may directly reflect in decreased performance.

Decoupled Access-Execute (DAE) \cite{28, 19, 29} transforms hot loops to contain a memory-bound *access* and compute-bound *execute* phase. The access phase is created by duplicating the loop body, and discarding all instructions that are unrelated to address computation and prefetch generation. The access phase is a prefetching slice, which fetches data into cache for the execute phase to consume. DAE targets energy efficiency, and to this end it runs the access phase on low frequency (processor is mainly accessing memory in parallel), and the execute phase on high frequency (processor can directly consume data).
In order to get a good balance between performance and energy, DAE relies on aggressive OoO cores to deal with the additional instruction count overhead that stems from the code duplication in the access phase. For small processors, the instruction count overhead may overweight the benefit of prefetching.

Software pre-execution [9, 34, 10, 21] makes use of multi-threading to boost the performance of a single main thread. The helper threads that are spawned by the main thread or by the helper threads themselves execute a prefetching slice that computes and fetches future addresses and values from memory into cache. These prefetching slices hide the latency of delinquent loads and improve branch prediction accuracy and the cache hit rate.

Software pre-execution is only applicable in scenarios where idle threads are available. The benefit of the pre-execution slices further need to outweigh the overhead of spawning new threads.

**Instruction Reordering.** There are a number of compiler optimizations that aim at reordering instructions in loop bodies, to separate instructions and their dependencies as much as possible.

Code scheduling [12, 6, 8, 33, 3], is traditionally solved separately from register allocation (whether pre-, post-scheduling or a mixture of both). While code scheduling focuses on hiding latencies and optimizing the schedule for the pipeline, register allocation aims at reducing register lifetimes, and avoiding register spills. These two compiler stages have opposing optimization goals. Code may trade increased register spilling for higher ILP scheduling, and register allocation introduces anti- (write after read dependency) and output-dependencies (write after write dependency) by assigning registers, thus limiting code motion (due to the introduced dependencies between registers). Code scheduling work has evaluated to combine the two into an integrated approach, or to perform several scheduling stages, in order to find a balance between latency hiding, optimal pipeline usage, and register spills. However, local instruction scheduling techniques can only overlap latencies of loads within one basic blocks, and global code motion considering register spills may create schedules that decrease performance [3].

Software Pipelining [40, 5, 50, 17, 41, 20] re-structures the loop body with the aim to gain ILP. The new loop body then consists of instructions of different iterations of the former loop, which can be executed in parallel (given enough resources). The basic idea is to overlap different iterations, such that one can start before the previous one is finished executing. The new loop, the kernel loop, is proceeded by a prologue, that feeds the pipeline, and an epilogue, that drains it. The dependent instructions are separated as much as required (or possible) across loop iterations, to avoid serialization of their execution through data dependencies. The advantage of software pipelining is the minimal code size increase compared to unroll-based reordering schemes that require the duplication of the loop body. However, software pipelining struggles with code with complex control-flow dependencies, and it cannot restruc-
ture the loop body if the compiler cannot successfully disambiguate memory dependencies between instructions.

**Software-Hardware Co-Design.** Software-Hardware Co-Designs rely on the software to make use of its static knowledge, while providing hardware support to exploit the gained insight.

VLIW/EPIC architectures, such as the Itanium and HPL PlayDoh [22, 23, 46, 16, 26] rely on the compiler to optimize the code schedule to achieve high ILP. The compiler thereby optimizes code on a large scale, exceeding a processor’s regular dynamic instruction window. As the processor is designed to be efficient, hardware elements such as the reorder buffer are eliminated, and instead the processor is equipped with a large number of execution resources to support the EPIC style, including but not limited to predication, rotating registers, and speculation. EPIC builds on the philosophy of close communication between the software and the hardware.

While providing plenty of resources to overcome the penalties of long-latency loads, VLIW/EPIC designs rely on speculation, thus requiring support for verification, delayed exception handling, memory disambiguation and predication.

Software-hardware co-designs that tackle branch mispredictions [38, 47, 35] decouple the computation of the branch predicate from the actual branch and other instructions or predict the control-flow for highly predictable branches at an early stage. Branch mispredictions can be very costly, as all instructions executed past a mispredicted branch have to be discarded. Without prediction, however, there are less opportunities to exploit ILP, and as a result, performance suffers. By finding a general solution to overcome any delinquent loads, both branch mispredictions and processor stalls can be avoided.

Previous work has extensively looked into instruction scheduling for hiding load latencies. However, presented solutions require additional threads or speculation, struggle with general-purpose code, or add too much instruction count overhead which is harmful for performance when it comes to small processors. In the next section we will introduce our contributions on overcoming long-latency loads through instruction scheduling for general purpose code on small processors.

### 2.1 Latency-aware Instruction Scheduling for General Purpose Code

A processor’s tolerance for delinquent loads is dependent on the available resources the processor has (reorder buffer, registers, MSHR size, ..) to keep track of outstanding loads and to be able to execute enough independent instructions to hide the latencies. Processors with fewer resources are naturally at risk of stalling faster, but have the benefit of being more energy-efficient. In this thesis we present instruction reordering techniques tailored for small pro-
2.1.1 Contribution 1: Targeting Resource-Constrained OoO Processors

OoO processors are designed to be able to cope with long latency loads. However, limited OoO processors have fewer resources limiting their ability to find and execute independent instructions. As a result, the processor may stall if outstanding loads cannot be resolved in time, and no other instructions can be executed to hide the latency.

We present Clairvoyance, a compiler reordering technique, with the aim to (i) cluster loads to increase MLP, and (ii) to overlap their latencies with independent instructions to increase ILP. Clairvoyance is applicable on general purpose code, and handles unknown memory dependencies, pointers, and control-flow. Similar to DAE, Clairvoyance transformations generate a memory-bound Access phase containing the clustered loads and their requirements, and a compute-bound Execute phase containing the uses of the loads.

Figure 2.1 shows the basic Clairvoyance idea. We target hot loops and create (by reordering instructions) two phases, the Access phase that contains the long latency loads and their address computation, and an execute phase with...
Figure 2.2. Basic Clairvoyance transformation. Hoisting and clustering load instructions, and lowering all remaining ones; creating a memory-bound Access phase and a compute-bound Execute phase.
Figure 2.3. Prefetch data if loads may-alias with preceding store, otherwise hoist.

the remaining instructions, mainly computation and uses of loads. Figure 2.2 shows a step-by-step transformation for an example loop (pseudo code), which consists of an unrolling step (to find more instructions to reorder) and a reordering step (to create the phases). The green arrows indicate the positioning of the load instructions before and after reordering, the orange ones respectively for the store instructions. For simplicity, this example only shows the case with an unroll count of 2, but normally we experiment with unroll counts of up to 16. In order to cope with control-flow, Clairvoyance duplicates the branches within the Access phase, similar to DAE.

Reordering can be difficult to accomplish given unknown memory dependencies (loads cannot be hoisted before stores if a RAW dependency exists). Often, the compiler reports that a load store pair may alias, i.e. a RAW dependency may exist but cannot be statically proven, even though they turn out to be not aliasing at runtime. Only reordering loads that are known not to alias with any stores is therefore very restrictive, and would limit the amount of MLP we could gain given an imperfect alias analyzer.

Paper I addresses these issues, and introduces (i) load-prefetch hybrids to overcome these restrictions. No-aliasing loads are reordered, while may-aliasing loads are prefetched in Access, and safely reloaded in Execute, see Figure 2.3. The may-aliasing load in the second iteration may be aliasing with the preceding store of the first iteration – therefore we only prefetch its value in the Access phase, and keep the load in the correct order in the Execute phase.

Clairvoyance further overcomes complex control-flow (by duplicating branches in Access), reduces register pressure (by applying its load-prefetch
hybrid) and instruction count overhead (by performing early branch clustering to mitigate control-flow duplication) and presents a solution to split up dependent load chains (by creating multiple access phases).

We evaluate Clairvoyance on a range of memory- and compute-bound benchmarks, and show that Clairvoyance achieves 7% geomean performance improvement in average for memory-bound benchmarks for its most conservative reordering, and a respective 12% for its most speculative (oracle) one.

2.1.2 Contribution 2: Stressing the Limitations of Conservative Scheduling

While Clairvoyance’s conservative version is safe, it is still too limited in what loads it can target compared to its most speculative version (the oracle). Ideally we would like to achieve the performance seen by the oracle, while still guaranteeing correctness. In order to achieve the same performance improvement as the best speculative version, while remaining conservative, we incorporate state-of-the-art alias analysis [48] to successfully disambiguate more load store pairs. With more loads known to be no-aliases, more registers will be allocated to hold the values, and therefore, register pressure will become a problem also for the most conservative version.

In Paper I we relied on a hybrid version that would prefetch data, instead of loading them into registers, if a load store pair cannot be successfully disambiguated (Figure 2.3). The hybrid model was not only a handle to bridge unknown memory dependencies, but also one to naturally reduce register pressure. With an improved alias analyzer, even the conservative version would hoist more loads than general purpose registers exist. Register pressure will cause register spills, and therefore increased instruction count overhead and early uses of outstanding loads.

To tackle the register pressure, Paper II introduces a new heuristic to handle register pressure given many reordable loads, choosing to prefetch selected loads, even if they can be completely disambiguated (Figure 2.3). The hybrid model was not only a handle to bridge unknown memory dependencies, but also one to naturally reduce register pressure. With an improved alias analyzer, even the conservative version would hoist more loads than general purpose registers exist. Register pressure will cause register spills, and therefore increased instruction count overhead and early uses of outstanding loads.

To tackle the register pressure, Paper II introduces a new heuristic to handle register pressure given many reordable loads, choosing to prefetch selected loads, even if they can be completely disambiguated (Figure 2.3). The hybrid model was not only a handle to bridge unknown memory dependencies, but also one to naturally reduce register pressure. With an improved alias analyzer, even the conservative version would hoist more loads than general purpose registers exist. Register pressure will cause register spills, and therefore increased instruction count overhead and early uses of outstanding loads.

In Paper I we relied on a hybrid version that would prefetch data, instead of loading them into registers, if a load store pair cannot be successfully disambiguated (Figure 2.3). The hybrid model was not only a handle to bridge unknown memory dependencies, but also one to naturally reduce register pressure. With an improved alias analyzer, even the conservative version would hoist more loads than general purpose registers exist. Register pressure will cause register spills, and therefore increased instruction count overhead and early uses of outstanding loads.

To tackle the register pressure, Paper II introduces a new heuristic to handle register pressure given many reordable loads, choosing to prefetch selected loads, even if they can be completely disambiguated (Figure 2.3). The hybrid model was not only a handle to bridge unknown memory dependencies, but also one to naturally reduce register pressure. With an improved alias analyzer, even the conservative version would hoist more loads than general purpose registers exist. Register pressure will cause register spills, and therefore increased instruction count overhead and early uses of outstanding loads.

To tackle the register pressure, Paper II introduces a new heuristic to handle register pressure given many reordable loads, choosing to prefetch selected loads, even if they can be completely disambiguated (Figure 2.3). The hybrid model was not only a handle to bridge unknown memory dependencies, but also one to naturally reduce register pressure. With an improved alias analyzer, even the conservative version would hoist more loads than general purpose registers exist. Register pressure will cause register spills, and therefore increased instruction count overhead and early uses of outstanding loads.

In Paper I we relied on a hybrid version that would prefetch data, instead of loading them into registers, if a load store pair cannot be successfully disambiguated (Figure 2.3). The hybrid model was not only a handle to bridge unknown memory dependencies, but also one to naturally reduce register pressure. With an improved alias analyzer, even the conservative version would hoist more loads than general purpose registers exist. Register pressure will cause register spills, and therefore increased instruction count overhead and early uses of outstanding loads.

In Paper I we relied on a hybrid version that would prefetch data, instead of loading them into registers, if a load store pair cannot be successfully disambiguated (Figure 2.3). The hybrid model was not only a handle to bridge unknown memory dependencies, but also one to naturally reduce register pressure. With an improved alias analyzer, even the conservative version would hoist more loads than general purpose registers exist. Register pressure will cause register spills, and therefore increased instruction count overhead and early uses of outstanding loads.

To tackle the register pressure, Paper II introduces a new heuristic to handle register pressure given many reordable loads, choosing to prefetch selected loads, even if they can be completely disambiguated (Figure 2.3). The hybrid model was not only a handle to bridge unknown memory dependencies, but also one to naturally reduce register pressure. With an improved alias analyzer, even the conservative version would hoist more loads than general purpose registers exist. Register pressure will cause register spills, and therefore increased instruction count overhead and early uses of outstanding loads.

To tackle the register pressure, Paper II introduces a new heuristic to handle register pressure given many reordable loads, choosing to prefetch selected loads, even if they can be completely disambiguated (Figure 2.3). The hybrid model was not only a handle to bridge unknown memory dependencies, but also one to naturally reduce register pressure. With an improved alias analyzer, even the conservative version would hoist more loads than general purpose registers exist. Register pressure will cause register spills, and therefore increased instruction count overhead and early uses of outstanding loads.

To tackle the register pressure, Paper II introduces a new heuristic to handle register pressure given many reordable loads, choosing to prefetch selected loads, even if they can be completely disambiguated (Figure 2.3). The hybrid model was not only a handle to bridge unknown memory dependencies, but also one to naturally reduce register pressure. With an improved alias analyzer, even the conservative version would hoist more loads than general purpose registers exist. Register pressure will cause register spills, and therefore increased instruction count overhead and early uses of outstanding loads.

To tackle the register pressure, Paper II introduces a new heuristic to handle register pressure given many reordable loads, choosing to prefetch selected loads, even if they can be completely disambiguated (Figure 2.3). The hybrid model was not only a handle to bridge unknown memory dependencies, but also one to naturally reduce register pressure. With an improved alias analyzer, even the conservative version would hoist more loads than general purpose registers exist. Register pressure will cause register spills, and therefore increased instruction count overhead and early uses of outstanding loads.

To tackle the register pressure, Paper II introduces a new heuristic to handle register pressure given many reordable loads, choosing to prefetch selected loads, even if they can be completely disambiguated (Figure 2.3). The hybrid model was not only a handle to bridge unknown memory dependencies, but also one to naturally reduce register pressure. With an improved alias analyzer, even the conservative version would hoist more loads than general purpose registers exist. Register pressure will cause register spills, and therefore increased instruction count overhead and early uses of outstanding loads.
where the alias analysis cannot disambiguate load store pairs, spec-safe is the version that may give better performance, as it allows Clairvoyance to target more loads for improved MLP.

Paper II introduces a safety net for safe speculation, which would ignore segmentation faults if caused by Clairvoyance. Even if the most conservative version is preferred in case load-store pairs can be successfully disambiguated, the speculative but safe version may allow for further exploration for those benchmarks, where alias analysis is still imperfect.

In addition, Paper II provides an extended evaluation, including an in-depth analysis of the hardware counters (cycles, instruction count, load/store count), a comparison to a prefetch-only Clairvoyance version, and to state-of-the-art prefetching [1].

2.1.3 Contribution 3: Targeting Design-Constrained InO Processors

Clairvoyance performs reordering, but to ensure correctness, it is required to duplicate parts of the code (duplicated branch instructions in the Access phases, prefetch instructions for may-aliasing loads). On OoO processors, code duplication, i.e. instruction count overhead, can be tolerated, but InO processors suffer significantly from additional instructions. In its current state, Clairvoyance transformations are not a good fit for InO cores.

Stall-on-use InO processors are designed to be energy-efficient and lack the flexibility of the OoO to hide outstanding latencies. In contrast to OoO...
processors, they directly stall on the first use of data that is not yet available. Artifacts such as spilling or dependent long-latency load chains, which would access data that has not yet been fetched from memory, are directly impacting the performance of InO processors.

In order to find a solution for latency-hiding on InO processors, we must therefore take into account that (i) any accesses of outstanding loads or any early uses of these loads, and (ii) any additional overhead of instructions should be avoided, if possible. In Paper III we present SWOOP, a software-hardware co-design technique to accelerate InO processors, while adding lightweight hardware support to tackle the challenges that arise on InO processors, but still executing more energy-efficient than OoO processors.

SWOOP performs Clairvoyance-like code transformations, but restricts itself statically to one iteration. The software communicates the access and execute phase boundaries to the underlying hardware. At runtime, the hardware checks at the end of an Access phase whether any last-level cache misses occurred, and if so, more Access phases are executed.

Figure 2.4 shows the SWOOP software model, and the execution compared to Clairvoyance. After applying Clairvoyance on one iteration (i.e. hoisting loads, lowering uses), we run on the SWOOP processor, and if a long-latency load is detected within the current Access phase (here $A_0$), SWOOP executes several more access phases to overlap the outstanding latencies before executing their corresponding execute phases. How many Access phases are run depends on the number of available registers. SWOOP guarantees that no spills are required, by only executing as many Access phases as registers are available.

The motivation to transfer this control to hardware (instead of unrolling and reordering), is that the compiler can only see the architectural registers, but not the physical registers that may be available for use. The architectural registers are strictly fewer than the physical registers, and thus, the compiler would statically introduce spill code even if more registers are available. By reordering, and communicating the access and execute boundaries to the SWOOP core, we create a lean access phase without additional spill code, while allowing more loads from future iterations to be executed ahead of time, using hardware. This addresses the problem of stalling on spills.

Next, we replace the static load selection heuristic in Clairvoyance by an offline profiling step, in order to identify the delinquent loads. This allows us to create smaller access phases that focus on the loads that are responsible for most of the penalty, and at the same time, we reduce the number of instructions that are additionally executed (reduces the required control flow to hoist loads).

Paper III focus on the software transformation and the hardware changes required to accomplish software out-of-order execution on InO processors. We evaluate our solution on a range of memory-bound applications, and the simu-
lations show an average performance improvement of 34% and energy savings of 23% over an InO processor.
3. Overcoming Memory Latency on Secure Speculative Processors

Side-channel attacks allow attackers to leak sensitive data by observing the system. Lately, a number of side-channel attacks have been discovered that exploit speculative execution, two of the best known are Meltdown and Spectre [27, 32]. The idea is the following: in order to prevent the processor from stalling, the out-of-order processor fetches instructions ahead-of-time and executes them speculatively. In the rare case of a mispeculation, the speculatively executed instructions are undone, and the processor can resume execution from that point where the mis-speculation began. Speculative side-channel attacks exploit the fact that speculative execution leaves traces in the microarchitectural state, and this includes updates to the memory system.

We can identify three groups of hardware defenses that have been proposed: defenses based on (i) hiding the side-effects, (ii) undoing the side-effects, and (ii) delaying speculative execution.

**Hiding the side-effects** allows speculation to continue, but requires a separate buffer that acts as a speculative cache that keeps the speculative values around [52, 45]. These values are not propagated to memory until the load is regarded as safe to execute (i.e., as soon as it becomes unsquashable). Since the memory hierarchy is not modified until a load is safe, no secrets can be leaked. However, hiding the execution incurs a cost that is relative to how long it takes until they can be made public. Furthermore, an update to the memory hierarchy may require an expensive validation step, as the load may have violated the memory consistency model.

**Undoing the side-effects** of speculative execution [43] focuses on an undo approach. They allow speculative execution to modify microarchitectural state, but undo them if a misprediction is discovered. Undoing wrong updates introduces costs on a misprediction, but not if the processor speculated correctly, which happens to be the common case. While significantly improving the performance for hardware defenses, the approach is complex: it requires cache line tracking for updates from transient loads, cache randomization for L2/LLC, relies on random cache replacement for L1, and delay of loads affected by coherency.

**Delaying speculative execution** [52, 45] entirely delay selective instructions until they are unsquashable. The advantage of delaying speculative execution is its effectiveness and simplicity. On the other hand delaying loads restricts the OoO core in what it can do best: to find and execute instructions out-of-order, and thus to effectively hide latencies and avoid stalls.
**Software mitigations** were introduced to mitigate specific variants of speculative side-channel attacks. Retpote line (return trampoline) [13] replaces indirect branches with a call-return procedure and tricks speculative execution into an infinite loop that is only exited as soon as the branch target is known. Indirect branches can otherwise be misused to steer speculative execution into jumping to a different location. Condition masking [36] poisons load addresses on mispredicted paths, by keeping a poison register and updating it based on the outcome of the condition. KAISER [14] enforces strict user and kernel space isolation and protects against Meltdown, but not against Spectre. Other mitigations include barriers and conditional select/move instructions [2, 18], but are relying on the programmer to decide when they are to be used, since automated detection on the compiler side for vulnerable code is not complete [37].

3.1 Efficient Invisible Speculation through Delay-on-Miss

Sakalis et al. [45] propose a hardware defense scheme that is called **delay-on-miss**, in which speculative loads that miss in cache have to be delayed. On the contrary, loads that hit in the cache do not fetch new data and are thus allowed to be executed speculatively. Loads are classified as speculative, if they are under the speculative shadow of an older instruction. An instruction casts a speculative shadow if its execution may affect following instructions. A shadow is cast on all following load instructions, which are not allowed to execute until all shadows that are covering it are lifted, i.e., when the reason for speculation is resolved. The authors identify four major causes for speculation and name the shadows accordingly: Exception (E)-, Control (C)-, Data (D)-, and Memory (M)-shadows. Table 3.1 shows an example for each shadow type. One instruction may cast several shadow types. For example, loads cast E-shadows (may throw an exception) and M-shadows (subsequent loads are required to wait until a load has finished execution under Total Store Order (TSO)). The M-shadow enforces a strict order on all loads, as the observable memory order has to be preserved.

Shadows that are cast are lifted as soon as the reason for their speculation is resolved: E-shadows are lifted as soon as the target address of memory operations is known\(^1\), C-shadows as soon as the target branch is known, D-shadows when the data dependency is resolved, and M-shadows when the corresponding load has finished execution.

\(^1\)arithmetic instructions may also cast E-shadows, but since they only make up for a fraction of all shadows [45] we do not focus on removing them in this work.
<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-shadow (Exception)</td>
<td><code>int x = a[invalid] /* throws */ int y = a[i] /* E-shadows are cast by any instruction that may throw an exception*/</code></td>
</tr>
<tr>
<td>C-shadow (Control)</td>
<td><code>if (test(i)) { /* unknown path */ int y = a[i] } /* C-shadows are cast by unresolved branches.*/</code></td>
</tr>
<tr>
<td>D-shadow (Data)</td>
<td><code>a[i] = compute() int y = b[i] /* a == b? */ /* D-shadows are cast by potential data dependencies.*/</code></td>
</tr>
<tr>
<td>M-shadow (Memory)</td>
<td><code>int x = a[i] /* ld order in TSO */ int y = a[i+1] /* M-shadows conserve the observable load ordering under TSO.*/</code></td>
</tr>
</tbody>
</table>

Table 3.1. Examples for shadow types identified by Sakalis et al.
for (int i = 0; i < 1000; ++i) {
    int addr1 = ...;
    int l1 = p1->a[addr1];
    params.fullf[0] = l1;
    int addr2 = ...;
    l2 = p2->a[addr2];
    params.fullf[1] = l2;
    bool cond = l1 < l2;
    if (cond) {
        ...
    }
}

Figure 3.1. Example code showing (i) the type of shadows cast by a line of code (E,C,M,D) and (ii) their overlap. Instructions towards the end of the code excerpt are blocked by several overlapping shadows.

3.2 Shadow-aware Instruction Scheduling for Delay-on-Miss Hardware

As shadows are cast on all following load instructions, one single shadow is enough to block all following loads from executing. Given that the majority of shadow-casting instructions is loads, stores and branches [45], these shadows are occurring often and may require a long time to be lifted. Even if one single shadow is lifted, following loads may not be able to execute, as they are shadowed by overlapping shadows, as demonstrated in Figure 3.1. The shadows are visualized through the gray overlapping boxes. On the right we annotate which types of shadows are cast by that specific line. In order to successfully remove shadows to the extent that loads can be safely executed, one needs to remove a chain of shadows leading up to that load.

3.2.1 Contribution 4: Targeting Security-Constrained Processors

In part two of this thesis we focus on improving performance on hardware defenses that rely on delaying speculative execution. For analysis, implementation and experimentation we focus on a delay-based hardware solution proposed by Sakalis et al. [45], but our work is a general analysis on the effects of ordering and delaying loads for security, and what the compiler can do to improve performance.

In paper IV we look into the possibility to use a combination of static transformation and hardware support to completely remove or shorten shadow durations, with the ultimate goal to unshadow loads and execute them out-of-order,
Table 3.2. Overview on shadow-casting instructions, the shadows they cast (x), and the solutions analyzed in this work. We exclude shadow-casting instructions that are not memory or control instructions, as their total share is negligible. By excluding them, D- and E-shadows can be combined into one category.

<table>
<thead>
<tr>
<th>Inst</th>
<th>LD</th>
<th>ST</th>
<th>BR</th>
<th>Lifted when.</th>
<th>Unshadowing Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-shadow</td>
<td>x</td>
<td>x</td>
<td></td>
<td>Target address known</td>
<td>Early Target Address Computation</td>
</tr>
<tr>
<td>C-shadow</td>
<td></td>
<td></td>
<td>x</td>
<td>Branch target address known</td>
<td>Early Condition Evaluation</td>
</tr>
<tr>
<td>M-shadow</td>
<td>x</td>
<td></td>
<td></td>
<td>Load has executed</td>
<td>Non-speculative Load-Reordering</td>
</tr>
</tbody>
</table>

Table 3.2 shows an overview of the techniques we evaluate to tackle each shadow type. In the following we shortly present our work on unshadowing loads for the different shadow types.

Mitigating E- and C-shadows through Early Evaluation of Conditions And Addresses

C-shadows are caused by unresolved branches. Any load that comes after an unresolved branch is considered as speculative, as the future program path is yet unknown. E-shadows are caused by load and store operations whose target address is not yet resolved. As long as the target is not known the memory operations may throw exceptions which would, if instructions were executed in-order, not allow any operations to be executed past that point.

In order to mitigate E- and C-shadows we propose a solution that hoists branch condition computation (tackling C-shadows) as well as load and store target address computation (tackling E-shadows) to shorten the cast shadow duration. The general idea is the following: we identify the instructions we want to hoist, collect their requirements and dependencies, and schedule them as early as possible, but earliest at the beginning of a basic block.
Our scheduling techniques do not cross boundaries, contrary to the scheduling techniques presented in Section 2.1, as both techniques target different causes to why the processor cannot exhibit enough MLP. Scheduling for small processors needs to look across basic block boundaries in order to be able to find independent loads to cluster and other instructions to schedule simultaneously, as the hardware does not provide a large enough look ahead (the instruction window size). Scheduling for the secure OoO is different: the hardware for looking ahead exists, and the branch predictor still does a good job at predicting the future paths, however, the loads that are found are not allowed to execute, as they are speculative. The scheduling goal for delay-based hardware defenses is to cluster computation in advance to avoid unresolved, speculative instructions to block all others. While scheduling within a basic block is unnecessary on unsecure OoO, it becomes relevant again in the secure OoO. In a way, the shadowing techniques can be seen as restricting the OoO in its out-of-orderness. Small changes in the schedule (within a basic block) can therefore already make a difference. On top of this, we want to strike a balance between hoisting instructions just enough to allow several otherwise speculative instructions to get resolved early, and not hoisting too far to burden the OoO as address computation instructions are partly visible and executable within the large instruction window.

Figure 3.2 shows a small example of our reordering. Figure 3.2(a) contains the original code; an example basic block. First, we identify which instructions we want to hoist (marked with $\times$) to the left of the code. Afterwards, we start creating buckets from the marked instructions. A bucket $b_{i+1}$ contains only instructions that are independent of each other, that have, however, a dependency on at least one instruction of the previous bucket $b_i$. Therefore all instructions within one bucket may be scheduled together without violating any dependencies. Figure 3.2(b) shows the three buckets $b_0, b_1, b_3$ that are created for the marked instructions of the original code. Afterwards, we hoist the instructions and order them according to the buckets. Figure 3.2(c) shows the resulting reordered code, where the marked instructions are now hoisted to start at the beginning of the basic block.

More details on the reordering and its algorithms can be found in Paper IV. Our evaluation in Paper IV will also show that selecting all instructions (instead of only the ones computing branch conditions and load and store target addresses) can deliver better performance. Though the selective hoisting is a more intuitive solution, reordering everything makes better use of the hardware resources. We can see two reasons for why this may be the case. First, by overlapping the branch condition and address computation with independent instructions, enough time might pass such that the condition and the addresses are ready as soon as their consuming instructions start executing. Second, by grouping independent instructions and scheduling them together, we also group loads that may further exploit MLP, similar to what we achieved in our previous work in Paper I-III.
Figure 3.2. The original code and the selected instructions to hoist (physical address computation for memory operations and branch condition evaluation, marked with $\times$) are shown in Figure (a). The selected instructions and their dependencies are ordered into buckets as in Figure (b). Instructions within a bucket are independent of each other. An instruction of a bucket has at least one dependency in its preceding bucket. The buckets determine the order in which they will be hoisted to the beginning of the basic block. The remaining instructions are kept in their original order. Figure (c) shows the resulting reordering.
Mitigating M-shadows through Enabling Non-Speculative Reordering of Loads

M-shadows only exist under consistency models that enforce load ordering, such as TSO. If two processors work on shared data, and have conflicting accesses, i.e. one processor writes while the other reads, it needs to be ensured that the memory reordering that may happen on one of the cores is not visible to the other. The M-shadows disallow any memory reordering that may turn out to be invalid under TSO by entirely sequentializing loads, i.e. every load casts an M-shadow that is only lifted as soon as the load has finished execution. This also means that loads that miss in cache can never be overlapped. This has a significant impact on performance as it diminishes the potential of MLP.

To overcome M-shadows we make use of a coherence protocol that allows memory reordering with the guarantee that the reordering will be valid under TSO and invisible to any other core that shares that data. This coherence protocol was developed by Ros et al. and applied on the use case of out-of-order commit [42], but can be applied in our case to allow the complete removal of M-shadows. Usually a conflicting update from another processor requires the squashing of the reordered memory operations. However, the coherence protocol is designed such that it delays acknowledging the update (a so called invalidation) from the other core, until the affected memory operations have finished execution, such that the reordering can no longer be observed. The coherence protocol is described in detail in Ros et al. [42]. For us, this means that M-shadows are no longer required, as memory reordering is guaranteed not be invalid under TSO. By lifting M-shadows we open the door to MLP on TSO.

We evaluate our ideas on a range of benchmarks of the SPEC2006 benchmark suite. The implementation on the software side is based on LLVM 8, and the simulation is run in GEM5. We evaluate our software-hardware extension on top of delay-on-miss [45] and improve their delay-on-miss version by 9% (on average), thus reducing the gap between delay-on-miss and the unsafe baseline by 48% (on average).
4. Summary and Discussion

With the widening gap between memory and processor speed, overcoming memory latency is a cornerstone for higher performance. In this thesis we look at constrained speculative architectures that struggle to hide memory latencies and propose compiler techniques and hardware extensions to overcome their limitations.

Paper I: Clairvoyance

| Problem | Small resource-constrained OoO processors lack the ability to look far enough ahead to find independent loads that can be scheduled together to achieve high MLP and ILP. |
| Solution | A latency-aware instruction reordering technique, which prioritizes and clusters loads, to fetch data early into registers, and lowers their uses to split the load-use chain as much as possible. |
| Results | A performance improvement of 7% compared to the code optimized with standard O3 on memory-bound benchmarks. |

Paper II: Clairvoyance Extension

| Problem | Restrictions on Clairvoyance’s performance benefit imposed by introduced register pressure and weak alias analysis. |
| Solution | Inclusion of better alias analysis, usage of prefetches instead of loads in cases of register pressure and a cost model to decide when and how to combine the two, and finally a safety net that makes it possible to be more speculative. |
| Results | 14% geomean improvement for memory-bound benchmarks on top of O3 optimizations. |
Paper III: SWOOP

Problem
Stall-on-use InO processors are constrained by design and stall as soon as the data of an unresolved load is accessed, leading to performance degradation on memory-bound benchmarks. Clairvoyance as-is on InO introduces too much overhead, as many additional instructions are added which cancel out potential performance improvements.

Solution
Clairvoyance-like transformation on a single loop iteration combined with lightweight hardware support. Communication of Access and Execute phases allows hardware to take care of the unrolling. Hardware determines the number of Access phases to run, depending on the number of available registers.

Results
A performance improvement of 34% and energy savings of 23% over an InO processor.

Paper IV: Software-Hardware to Improve Delay-on-Miss

Problem
Secure processor designs to mitigate speculative side-channel attacks such as Spectre may rely on delaying loads to prevent unauthorized loads from fetching sensitive data. Delaying speculative loads limits the potential for MLP and thus introduces a performance degradation.

Solution
Hoisting branch condition and target address computation, and overlapping them with independent instructions. Employing a coherence protocol to safely reorder loads. In combination our techniques achieve to increase the number of loads that can be safely executed and ultimately exploits MLP.

Results
A performance improvement of 9% on top of state-of-the-art work on delaying loads for security.

4.1 Discussion: Clairvoyance under the Light of Security

Contributions I-III were developed prior to the discovery of speculative side-channel attacks. Especially since our work on contribution IV deals with security issues we need to reflect on whether or not our Clairvoyance-based scheduling techniques are safe.
Clairvoyance’s most conservative version Clairvoyance-consv is, as the name suggests, conservative. After applying hardware solutions to only allow safe speculation, it is similarly safe as other code. The most speculative one, Clairvoyance-spec, is speculative, and reorders memory operations in a way such that loads may not be safe to execute. Since this is software speculation, this introduces a vulnerability that is not taken care of by proposed hardware solutions. However, in our paper we already mentioned that the speculative version can be seen as an oracle given perfect alias analysis that is not considered to be safe.

In the following we shed light on whether or not Clairvoyance’s safe-spec version, the version that combines some speculation while being safe, would be actually safe to apply under the light of side-channel attacks. For each shadow, we check if any our transformation would violate the guarantees of that shadow. Clairvoyance’s safe-spec version hoists may-aliasing loads chains in its Access phase, but reloads them during its Execute phase.

**M-shadows**
Clairvoyance as such is only applicable for single-threaded applications (or for data-race free regions) and its reordering is therefore not visible to other cores.

**D-shadows**
Clairvoyance may violate read-after-write dependencies if the compiler statically could not tell whether or not they alias. The safe-spec version hoists loads above may-aliasing stores with the hope to access the correct data and to fetch it into cache early in advance. The loaded value is not used in the final computation (in the Execute phase) but is reloaded; nevertheless the wrong value is cached during the Access phase. Given that this speculation happens in software, proposed hardware solutions to mitigate the speculative vulnerability do not protect against malicious attacks. Clairvoyance’s safe-spec version can thus be misused to leak sensitive data. One way to make safe-spec resilient against attacks would be to annotate that this access has to be handled as any other speculative access, e.g. by having a buffer that keeps the speculative data. A load then remains speculative until its previous execute phases have been executed.

**C-shadows**
Clairvoyance hoists loads across basic blocks, and clusters them to overlap their delays. However, to be on the safe side, we designed Clairvoyance to not hoist loads outside of their guarding blocks, i.e. all loads that are hoisted are hoisted together with their guarding if-cases. Our transformation is therefore not violating the guarantees of the C-shadow.
E-shadows

Clairvoyance does not alter or violate the E-shadow guarantees.


I del två kollar vi på processorer som har förväntat att hitta många oberoende instruktioner men som inte får exekvera dem av säkerhetsskäl. Ett antal sårbarheter har upptäcks nyligen som kan utnyttja spekulativ exekvering på processorer. Spekulativ exekvering möjliggor att processorn spekulativt exekverar instruktioner även om de kanske inte behövs. Ett exempel är instruktioner som följer efter en branch-instruktion; dess mål är inte känt ännu. Problemet är att ändringar på mikroarkitekturen kan användas för att extrahera information genom cachessidorkanaler. För att åtgärda de mest bekanta, Spectre och Meltdown, har forskare presenterat lösningar som fördröjer spekulativ exekvering och därmed spekulativa load-instruktioner. I och med att
6. Acknowledgements

First and foremost I would like to thank my supervisor, Alexandra Jimborean. Alexandra, you have been a great advisor. I truly believe that I have been lucky with my advisors in the past, and this continued with you becoming my PhD advisor. You were always supportive, had advice when I got stuck and took your time to meet and discuss ideas. There were many things that I would have never imagined to be able to do, but you always believed in me. Thank you for all these years, Alexandra! I learned a lot from you and had a lot of fun working with you!

Next, I would like to thank my second supervisor, Stefanos Kaxiras, for all the big ideas, the guiding, the discussions, and for your kindness; you always made me feel like part of the family. It was inspiring to be working with you!

Thank you to my co-authors, and especially Trevor Carlson, Magnus Själander, and Christos Sakalis. Trevor and Magnus, it was invaluable to me that you accompanied my first two years. You two were great colleagues and even greater friends. Thank you Chris for always being cheerful, helpful, and fun to work with. It has been a pleasure for me to work with you!

Thank you to the entire UART group and to my peers Mehdi, Johan, Marina, Chris, Gustaf, Per, Ricardo, Hassan, Chang-Hyun, Yuan, and former UART members Andreas, Gérman, Greg and Mihail, for all the fun and the long fikas. I met many friends at UART and I am happy that I got the chance to meet all of you!

A special thanks to Johan and Mehdi for our common PhD journey. I consider myself lucky to have gathered so many nice memories with you, from our our trips to Italy, our search for wine while going through shady alleys, to our nights out. I will always remember them when thinking back to Uppsala. Mehdi, thank you for being my best buddy, for sharing the ups and downs, for all the chats, laughter and tears, and just for being you.

Thanks to my peers and friends in the department, especially Sonja and Kiko. Kiko, thanks for the (exhausting), but fun ACM-W time. I am still very proud that we managed to organize a conference together! Sonja, thank you for the fikas, the time we spent on parental leave together, the swimming lessons that we took, and for all the chats we had. Felix and I miss Kilian and you!

Furthermore I would like to thank my parents and my sister for always having by back. Thank you dad, for sharing your love of technology and maths with me. Thank you mom, for your care and for always preparing tons of food to take home for me. Thank you Chichen, for being a loving and amazing sister. I love you three!
Thank you Stephan, for being a wonderful husband and father and for always supporting and believing in me. Thank you, for all the comfort and hugs that you give me when things get rough. Without you I would have never learned to believe in myself and neither to enjoy life as I do. A lot happened during these 5 years: our common PhD studies, our marriage, our first child, and our move to Germany. I cannot wait to see what adventures our future will bring, but I am happy that I scored lucky and will be able to share it with you! Finally, thank you Felix; for your hugs, your laughter, your singing and dancing, your calls for mummy, and for all the joy you bring. I love you two!
References


2018.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1897

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title “Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology”.)