Static Instruction Scheduling for High Performance on Energy-Efficient Processors

Kim-Anh Tran
Static Instruction Scheduling for High Performance on Energy-Efficient Processors

Kim-Anh Tran
kim-anh.tran@it.uu.se

January 2018

Division of Computer Systems
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

Dissertation for the degree of Licentiate of Philosophy in Computer Science

© Kim-Anh Tran 2017
ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden
Abstract

New trends such as the internet-of-things and smart homes push the demands for energy-efficiency. Choosing energy-efficient hardware, however, often comes as a trade-off to high-performance. In order to strike a good balance between the two, we propose software solutions to tackle the performance bottlenecks of small and energy-efficient processors.

One of the main performance bottlenecks of processors 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, if data has not been cached in advance. Load instructions that require an access to memory are thereby referred to as long-latency or delinquent loads. Long latencies caused by delinquent loads are putting a strain on small processors, which have few or no resources to effectively hide the latencies. As a result, the processor may stall.

In this thesis we propose compile-time transformation techniques to mitigate the penalties of delinquent loads on small out-of-order processors, with the ultimate goal to avoid processor stalls as much as possible. Our code transformation is applicable for general-purpose code, including unknown memory dependencies, complex control flow and pointers. We further propose a software-hardware co-design that combines the code transformation technique with lightweight hardware support to hide latencies on a stall-on-use in-order processor.
Acknowledgments

I would like to thank my supervisors Alexandra Jimborean and Stefanos Kaxiras for their guidance, their support and their encouragement. Thank you, Alexandra, for always taking your time to meet me and to discuss, and for being such an inspiring person! Thank you, Stefanos, for showing me exciting new ideas and directions for the future.

In addition I would like to thank my co-authors and friends Magnus Själander, Trevor E. Carlson, and Konstantinos Koukos for the great time and the great work. I learned a lot from all of you!

Furthermore I would like to thank my office mates, Mehdi Alipour and Johan Janzén, for exciting discussions, fun internal office jokes and unforgettable trips to Italy. Without you, the office would be too sad to be around. I would also like to thank the whole UART group and especially Grégory Vaumourin, Andreas Sembrant, Germán Ceballos, Christos Sakalis, Gustaf Borgström and Ricardo Alves for being great team mates.

Finally, I would like to thank my family: Stephan Brandauer for always supporting and believing in me, and my parents and my sister for their endless love.
List of papers

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.
Contents

Part I: Introduction ............................................................................................................. 11
  1 Introduction .................................................................................................................. 11
  2 Overcoming Memory Latency ......................................................................................... 15
  3 Latency-aware Instruction Scheduling for General Purpose Code ......................... 18
     3.1 Contribution 1: Targeting Limited Out-of-Order (OoO) Processors ....................... 18
     3.2 Contribution 2: Stressing the Limitations of Conservative Scheduling ................. 21
     3.3 Contribution 3: Targeting In-Order (InO) Processors ........................................... 22
  4 Conclusion and Future Work ......................................................................................... 24

References .......................................................................................................................... 25

Part II: Papers ..................................................................................................................... 29

Paper I: ............................................................................................................................... 31
  I.1 Introduction .................................................................................................................. 34
  I.2 The Clairvoyance Compiler ......................................................................................... 36
     I.2.1 Identifying Critical Loads ...................................................................................... 38
     I.2.2 Handling Unknown Dependencies ......................................................................... 39
     I.2.3 Handling Chains of Dependent Loads .................................................................... 40
     I.2.4 Overcoming Instruction Count Overhead .............................................................. 41
     I.2.5 Overcoming Register Pressure .............................................................................. 44
     I.2.6 Heuristic to Disable Clairvoyance Transformations ............................................ 44
     I.2.7 Parameter Selection: Unroll Count and Indirection ............................................ 44
     I.2.8 Limitations ............................................................................................................ 45
  I.3 Experimental Setup ..................................................................................................... 45
     I.3.1 Evaluating LLVM, DAE and Clairvoyance .......................................................... 47
     I.3.2 A Study on Speculation Levels ............................................................................ 47
  I.4 Evaluation .................................................................................................................... 48
     I.4.1 Comparing Clairvoyance’s Speculation Levels .................................................... 48
     I.4.2 Understanding Clairvoyance Best Versions ......................................................... 49
     I.4.3 Runtime and Energy ............................................................................................ 50
  I.5 Related Work ............................................................................................................... 52
Acronyms

DAE  Decoupled Access-Execute
ILP  Instruction-Level Parallelism
InO  In-Order
MLP  Memory-Level Parallelism
OoO  Out-of-Order
RAW  Read-after-Write
SWOOP  Software Out-of-Order Processing
Part I:
Introduction
1. Introduction

Energy-efficiency has not only become a limiting factor for performance, but is also a necessity in the development of mobile devices, in the domain of IoT, and for the cooling of large scale data centers. Existing energy-efficient processors such as InO and small OoO processors, however, are significantly slower than aggressive OoO processors, and often struggle to meet performance demands.

In comparison to their aggressive and fast pendant, small OoO cores quickly suffer from performance penalties due to loads that miss in the last-level cache (the delinquent loads). These last-level cache misses cause a long latency, as they require a fetch of data from main memory. In the meantime, the processor is required to find enough independent instructions in the instruction stream that are ready to execute, such that the latency can be hidden. In memory-bound applications, i.e. applications that exhibit many memory accesses, the processor may struggle to completely hide the latency, and in small cores, these delinquent loads may cause the processor to stall. In this stage, the processor remains idle (while on high frequency), waiting for data to arrive to resume execution, and thus wasting both energy and valuable cycles for execution. As the gap between processor and memory speed continues to widen (known as the memory wall [36]), avoiding last level cache misses becomes a critical factor in achieving high performance.

In order to overcome the penalty of delinquent loads, previous work have, among others, proposed solutions involving software prefetching [6, 20, 112, 72, 95, 98, 120], software pipelining [30, 4, 35, 86, 31, 93], and software-hardware co-designs [17, 18, 32, 12, 21, 112]. 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 avoid last-level cache misses. Clustering accesses to memory can overlap outstanding latencies and achieve high levels of Memory-Level Parallelism (MLP), 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. 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, and reordering across basic blocks may become difficult in the presence of complex control-flow.
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-aliases 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-aliases rarely turn out to be must-aliases at runtime [82], 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 this 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 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 improves performance, and how much overhead it introduces.

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 and IV). Unlike OoO processors, stall-on-use InO processors stall directly 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.

The rest of this thesis will be divided into two parts. The first part consists of a related work section (Section 2), an introduction into the three contributions of the thesis (Section 3), and a conclusive remark (Section 4). The related work will highlight previous work on hiding memory-latency, showing both software-only and software-hardware co-designs. The contributions will outline Clairvoyance and SWOOP, and describe their motivation and their scope. In the end, the conclusion summarizes the thesis and gives an outlook on future work. The second part of the thesis (Part II) includes all four papers that are referred to in the contributions.
2. Overcoming Memory Latency

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 [6, 20, 95, 112, 72] 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 [24] – 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 [24]) 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 [29], cache modeling [20], simulation [3] or multi-versioning [98] 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 [72].

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) [97, 120, 98] 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 \cite{8, 26, 9, 94} 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 \cite{81, 76, 77, 25, 2}, 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 \cite{2}.

Software Pipelining \cite{30, 4, 35, 86, 31, 93} re-structures the loop body with the aim to gain Instruction-Level Parallelism (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 restructure 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 [17, 18, 32, 12, 21] 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 [28, 112, 27] decouple the computation of the branch predicate from the actual branch and other instructions or predict the control-flow for highly predicable 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.
3. 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 processors (small OoO and InO processors), with the goal to achieve MLP and ultimately better performance. We first introduce the concept of latency-aware instruction scheduling on small OoO processors in Section 3.1. We explain the transformation, the advantages and the drawbacks of our technique. In Section 3.2 we dissect the technique even further, and analyze where these drawbacks come from, and how to overcome them. Finally, we apply our latency-aware instruction scheduler on code running on a stall-on-use InO core, and pair the so far software-only work with lightweight hardware support.

3.1 Contribution 1: Targeting Limited 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

![Clairvoyance Transformation](image)

*Figure 3.1. Creating Access and Execute Phases for hot loops by reordering long latency loads.*
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 3.1 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 the remaining instructions, mainly computation and uses of loads. Figure 3.2 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.
Prefetch data if loads may-alias with preceding store, otherwise hoist.

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 3.3. 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.
3.2 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 [115] 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 3.3). 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. The heuristic takes the available registers into consideration, and switches to prefetch mode as soon as no more registers are available. When integrating the state-of-the-art alias analysis, the most conservative version meets the performance gains of the previous speculative one (oracle).

Paper I also evaluated a speculative, but safe version (spec-safe) that would allow speculative prefetching of may-aliasing dependent load chains. For an access $A[B[i]]$ where both and $B[i]$ and $A[B[i]]$ are may-aliasing loads, spec-safe would load $B[i]$ in order to be able to prefetch $A[B[i]]$ in the access phase, but safely re-load both in the execute phase to guarantee correctness. Nevertheless, an access to $B[i]$ may already throw a segmentation fault, if the memory address is invalid (which never happened in practice for the studied applications), even if this value is never used for the final computation. In cases 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 [72].

3.3 Contribution 3: Targeting 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, in contrast to OoO processors, stall directly 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 hard-
ware checks at the end of an Access phase whether any last-level cache misses occurred, and if so, more Access phases are executed.

Figure 3.4 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 and IV 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 simulations show an average performance improvement of 34% and energy savings of 23% over an InO processor.
4. Conclusion and Future Work

While InO and small OoO processors are energy-efficient, they lack the performance benefit that aggressive OoO processors can provide. In order to make small processors more competitive, we address the issue of long-latency load execution, which may stall small processors if not enough independent instructions can be found to hide the outstanding latency.

We present 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. By overlapping their latencies we achieve MLP and ultimately performance benefits. We apply our ideas on InO and small OoO cores, and achieve a performance improvement of 34% for the former, and of 14% for the latter, on average.

So far our techniques have only been evaluated on single-threaded applications. The extension towards multi-threaded applications is work in progress. While Clairvoyance can already be applied – as is – on loops without any synchronization for data-race free applications, work remains to expand Clairvoyance towards any multi-threaded application. Currently, Clairvoyance is oblivious of synchronization, and therefore, would reorder load instructions across synchronization points where actually not admitted. Clairvoyance needs to be extended to be aware of (i) synchronization points, and to be able to (ii) apply the transformation on parts of the loop, or even on non-loop code.
References


Part II:
Papers
Paper I
Paper I
Clairvoyance: Look-ahead Compile-Time Scheduling

Kim-Anh Tran, Trevor E. Carlson, Konstantinos Koukos, Magnus Själander, Vasileios Spiliopoulos, Stefanos Kaxiras and Alexandra Jimborean

In Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO) Austin, Texas, USA, February 2017
Abstract
To enhance the performance of memory-bound applications, hardware designs have been developed to hide memory latency, such as the out-of-order (OoO) execution engine, at the price of increased energy consumption. Contemporary processor cores span a wide range of performance and energy efficiency options: from fast and power-hungry OoO processors to efficient, but slower in-order processors. The more memory-bound an application is, the more aggressive the OoO execution engine has to be to hide memory latency.

This proposal targets the middle ground, as seen in a simple OoO core, which strikes a good balance between performance and energy efficiency and currently dominates the market for mobile, hand-held devices and high-end embedded systems. We show that these simple, more energy-efficient OoO cores, equipped with the appropriate compile-time support, considerably boost the performance of single-threaded execution and reach new levels of performance for memory-bound applications.

Clairvoyance generates code that is able to hide memory latency and better utilize the OoO engine, thus delivering higher performance at lower energy. To this end, Clairvoyance overcomes restrictions which yielded conventional compile-time techniques impractical: (i) statically unknown dependencies, (ii) insufficient independent instructions, and (iii) register pressure. Thus, Clairvoyance achieves a geometric mean execution time improvement of 7% for memory-bound applications with a conservative approach and 13% with a speculative but safe approach, on top of standard O3 optimizations, while maintaining compute-bound applications’ high-performance.

I.1 Introduction
Computer architects of the past have steadily improved performance at the cost of radically increased design complexity and wasteful energy consumption [75, 110, 106]. Today, power is not only a limiting factor for performance; given the prevalence of mobile devices, embedded systems, and the Internet of Things, energy efficiency becomes increasingly important for battery lifetime [113].

Highly efficient designs are needed to provide a good balance between performance and power utilization and the answer lies in simple, limited out-of-order (OoO) execution cores like those found in the HPE Moonshot m400 [80] and the AMD A1100 Series processors [73]. Yet, the effectiveness of moderately-aggressive OoO processors is limited when executing memory-bound applications, as they are unable to match the performance of the high-end devices, which use additional hardware to hide memory latency.

This work aims to improve the performance of highly energy-efficient, limited OoO processors, with the help of advanced compilation techniques. The static code transformations are specially designed to hide the penalty of last-level cache misses and to better utilize the hardware resources.

One primary cause for slowdown is last-level cache (LLC) misses, which, with conventional compilation techniques, result in a sub-optimal utilization of the limited OoO engine that may stall the core for an extended period of time. Our method identifies potentially critical memory instructions through
advanced static analysis and hoists them earlier in the program’s execution, even across loop iteration boundaries, to increase memory-level parallelism (MLP). We overlap the outstanding misses with useful computation to hide their latency and thus increase instruction-level parallelism (ILP).

There are a number of challenges that need to be met to accomplish this goal.

1. **Finding enough independent instructions:** A last level cache miss can cost hundreds of cycles [79]. Conventional instruction schedulers operate on the basic-block level, limiting their reach, and, therefore, the number of independent instructions that can be scheduled in order to hide long latencies. More sophisticated techniques (such as software pipelining [71, 122]) schedule across basic-block boundaries, but instruction reordering is severely restricted in general-purpose applications when pointer aliasing and loop-carried dependencies cannot be resolved at compile-time. Solutions are needed that can cope with statically unknown dependencies in order to effectively increase the reach of the compiler while ensuring correctness.

2. **Chains of dependent long latency instructions are serialized:** Dependence chains of long latency instructions would normally serialize, as the evaluation of one long latency instruction is required to execute another (dependent) long latency instruction. This prevents parallel accesses to memory and may stall a limited OoO core. Novel methods are required to increase memory level parallelism and to hide latency, which is particularly challenging in tight loops and codes with numerous (known and unknown) dependencies.

3. **Increased register pressure:** Separating loads and their uses in order to overlap outstanding loads with useful computation increases register pressure. This causes additional register spilling and increases the dynamic instruction count. Controlling register pressure, especially in tight loops, is crucial.

**Contributions:**

Clairvoyance looks ahead, reschedules long latency loads, and thus improves MLP and ILP. It goes beyond static instruction scheduling and software pipelining techniques, and optimizes general-purpose applications, which contain large numbers of indirect memory accesses, pointers, and complex control-flow. While previous compile-time techniques are inefficient or simply inapplicable to such applications, we provide solutions to well-known problems, such as:

1. Identifying potentially delinquent loads at compile-time;
2. Overcoming scheduling limitations of statically unknown memory dependencies;
3. Reordering chains of dependent memory operations;
4. Reordering across multiple branches and loop iterations, without speculation or hardware support;
5. Controlling register pressure.

Clairvoyance code runs on real hardware prevalent in mobile, hand-held devices and in high-end embedded systems and delivers high-performance, thus alleviating the need for power-hungry hardware complexity. In short, Clairvoyance increases the performance of single-threaded execution by up to 43% for memory bound applications (13% geomean improvement) on top of standard O3 optimizations, on hardware platforms which yield a good balance between performance and energy efficiency.

I.2 The Clairvoyance Compiler

This section outlines the general code transformation performed by Clairvoyance while each subsection describes the additional optimizations, which make Clairvoyance feasible in practice. Clairvoyance builds upon techniques such as software pipelining [122, 86], program slicing [117], and decoupled access-execute [97, 120, 98] and generates code that exhibits improved memory-level parallelism (MLP) and instruction-level parallelism (ILP). For this, Clairvoyance prioritizes the execution of critical instructions, namely loads, and identifies independent instructions that can be interleaved between loads and their uses.

Figure 1 shows the basic Clairvoyance transformation, which is used as a running example throughout the paper. The transformation is divided into two steps:

Loop Unrolling
To expose more instructions for reordering, we unroll the loop by a loop unroll factor \( \text{count}_{\text{unroll}} = 2^n \) with \( n = \{0, 1, 2, 3, 4\} \). Higher unroll counts significantly increase code size and register pressure. In our examples, we set \( n = 1 \) for the sake of simplicity.

Access-Execute Phase Creation
Clairvoyance hoists all load instructions along with their requirements (control flow and address computation instructions) to the beginning of the loop. The group of hoisted instructions is referred to as the Access phase. The respective uses of the hoisted loads and the remaining instructions are sunk in a so-called Execute phase.

Access phases represent the program slice of the critical loads, whereas Execute phases contain the remaining instructions (and guarding conditionals). When we unroll the loop, we keep non-statically analyzable exit blocks. All exit blocks (including goto blocks) in Access are redirected to Execute, from
Figure 1. The basic Clairvoyance transformation. The original loop is first unrolled by \( \text{count}_\text{unroll} \) which increases the number of instructions per loop iteration. Then, for each iteration, Clairvoyance hoists all (critical) loads and sinks their uses to create a memory-bound Access phase and a compute-bound Execute phase.

where they will exit the loop after completing all computation. The algorithm is listed in Algorithm 1 and proceeds by unrolling the original loop and creating a copy of that loop (the Access phase, Line 3). Critical loads are identified (FindLoads, Line 4) together with their program slices (instructions required to compute the target address of the load and control instructions required to reach the load, Lines 5 - 9). Instructions which do not belong to the program slice of the critical loads are filtered out of Access (Line 10), and instructions hoisted to Access are removed from Execute (Line 11). The uses of the removed instructions are replaced with their corresponding clone from Access. Finally, Access and Execute are combined into one loop (Line 12).

This code transformation faces the same challenges as typical software pipelining or global instruction scheduling: (i) selecting the loads of interest statically; (ii) disambiguating pointers to reason about \textit{reordering memory instructions}; (iii) finding sufficient independent instructions in applications with \textit{entangled dependencies}; (iv) reducing the instruction count overhead (e.g., stemming from partly duplicating control-flow instructions); and (v) overcoming register pressure caused by unrolling and separating loads from their uses. Each of these challenges and our solutions are detailed in the following subsections.
**Input:** Loop $L$, Unroll Count $\text{count}_{unroll}$

**Output:** Clairvoyance Loop $L_{\text{Clairvoyance}}$

1. **begin**
2. \[ L_{\text{unrolled}} \leftarrow \text{Unroll}(L, \text{count}_{unroll}) \]
3. \[ L_{\text{access}} \leftarrow \text{Copy}(L_{\text{unrolled}}) \]
4. \[ \text{hoist\_list} \leftarrow \text{FindLoads}(L_{\text{access}}) \]
5. \[ \text{to\_keep} \leftarrow \emptyset \]
6. **for** load \textit{in} hoist\_list \textbf{do}
7. \[ \text{requirements} \leftarrow \text{FindRequirements}(\text{load}) \]
8. \[ \text{to\_keep} \leftarrow \text{Union}(\text{to\_keep}, \text{requirements}) \]
9. **end**
10. \[ L_{\text{access}} \leftarrow \text{RemoveUnlisted}(L_{\text{access}}, \text{to\_keep}) \]
11. \[ L_{\text{execute}} \leftarrow \text{ReplaceListed}(L_{\text{access}}, L_{\text{unrolled}}) \]
12. \[ L_{\text{Clairvoyance}} \leftarrow \text{Combine}(L_{\text{access}}, L_{\text{unrolled}}) \]
13. **return** $L_{\text{Clairvoyance}}$
14. **end**

**Algorithm 1:** Basic Clairvoyance algorithm. The Access phase is built from a copy of the unrolled loop. The Execute phase is the unrolled loop itself, while all already computed values in Access are reused in Execute.

### I.2.1 Identifying Critical Loads

**Problem:** Selecting the right loads to be hoisted is essential in order to avoid code bloat and register pressure and to ensure that long latency memory operations overlap with independent instructions.

**Solution:** We develop a metric, called indirection count, based on the number of memory accesses required to compute the memory address (indirections) [98] and the number of memory accesses required to reach the load. For example, $x[y[z[i]]]$ has an indirection count of two, as it requires two loads to compute the address. The latter interpretation of indirection count is dependent on the control flow graph (CFG). If a load is guarded by two if-conditions that in turn require one load each, then the indirection count for the CFG dependencies is also two. Figure 2 shows an example of load selection with indirection counts. A high value of indirection indicates the difficulty of predicting and prefetching the load in hardware, signaling an increased likelihood that the load will incur a cache miss. For each value of this metric, a different code version is generated (i.e., hoisting all loads that have an indirection count less than or equal to the certain threshold). We restrict the total number of generated versions to a fixed value to control code size increase. Runtime version selection (orthogonal to this proposal) can be achieved with dedicated tools such as Protean code [101] or VMAD [89, 91].
I.2.2 Handling Unknown Dependencies

Problem: Hoisting load operations above preceding stores is correct if and only if all read-after-write (RAW) dependencies are respected. When aliasing information is not known at compile-time, detecting dependencies (or guaranteeing the lack of dependencies) is impossible, which either prevents reordering or requires speculation and/or hardware support. However, speculation typically introduces considerable overhead by squashing already executed instructions and requiring expensive recovery mechanisms.

Solution: We propose a lightweight solution for handling statically known and unknown dependencies, which ensures correctness and efficiency. Clairvoyance embraces safe speculation, which brings the benefits of going beyond conservative compilation, without sacrificing simplicity and lightness.

We propose a hybrid model to hide the latency of delinquent loads even when dependencies with preceding stores are unknown (i.e., may-alias). Thus, loads free of dependencies are hoisted to Access and the value is used in Execute, while loads that may alias with stores are prefetched in Access and safely loaded and used in their original position in Execute. May-aliases, however, are an opportunity, since in practice may-aliases rarely materialize into real aliasing at runtime [82]. Prefetching in the case of doubt is powerful: (1) if the prefetch does not alias with later stores, data will have been correctly prefetched; (2) if aliasing does occur, the prefetched data becomes overwritten and correctness is ensured by loading the data in the original program order. Figure 3 shows an example in which an unsafe load is turned into a prefetch-load pair.
I.2.3 Handling Chains of Dependent Loads

**Problem:** When a long latency load depends on another long latency memory operation, Clairvoyance cannot simply hoist both load operations into Access. If it did, the processor might stall, simply because the second critical load represents a use of the first long latency load. As an example, in Figure 4 we need to load the branch predicate $t_1$ before we can load $t_2$ (control dependency). If $t_1$ is not cached, an access to $t_1$ will stall the processor if the out-of-order engine cannot reach ahead far enough to find independent instructions and hide the load’s latency.

**Solution:** We propose to build multiple Access phases, by splitting dependent load chains into chains of dependent Access phases. As a consequence, loads and their uses within access phase are separated as much as possible, enabling more instructions to be scheduled in between. By the time the dependent load is executed, the data of the previous load may already be available for use.

Each phase contains only independent loads, thus increasing the separation between loads and their uses. In Figure 4 we separate the loads into two Access phases. For the sake of simplicity, this example uses $count_{\text{unroll}} = 2$, hence there are only two independent loads to collect into the first Access phase and four into the second Access phase.

The proposed solution is safe. In addition to this solution, we will analyze variations of this solution that showcase the potential of Clairvoyance when assuming a stronger alias analysis. These more speculative variations are allowed to hoist whole chains of may-aliasing loads and will be introduced during the experimental setup in Section I.3.
The algorithm to decide how to distribute the loads into multiple Access phases is shown in Algorithm 2. The compiler first collects all target loads in remaining_loads, while the distribution of loads per phase phase_loads is initialized to empty-set. As long as the loads have not yet been distributed (Line 4), a new phase is created (Line 5) and populated with loads whose control-requirements (Line 8) and data-requirements (Line 9) do not match any of the loads that have not yet been distributed in a preceding Access phase (Line 10 and 11-14). Loads distributed in the current phase are removed from the remaining_loads only at the end (Line 15), ensuring that no dependent loads are distributed to the same Access phase. The newly created set of loads phase is added to the list of phases (Line 16) and the algorithm continues until all critical loads have been distributed. Next, we generate each Access phase by following Algorithm 1 corresponding to a set of loads from the list phase_loads.

I.2.4 Overcoming Instruction Count Overhead

Problem: The control-flow-graph is partially duplicated in Access and Execute phases, which, on one hand, enables instruction reordering beyond basic block boundaries, but, on the other hand, introduces overhead. As an example, the branch using predicate $t_1$ (left of Figure 5) is duplicated in each Access phase, significantly increasing the overhead in the case of multi-Access phases. Branch duplication not only complicates branch prediction but also increases instruction overhead, thus hurting performance.

Solution: To overcome this limitation, Clairvoyance generates an optimized version where selected branches are clustered at the beginning of a loop. If the
**Input**: Set of loads  
**Output**: List of sets `phase_loads`

```
begin
    remaining_loads ← loads
    phase_loads ← []
    while remaining_loads ≠ ∅ do
        phase ← ∅
        for ld in remaining_loads do
            reqs ← ∅
            FindCFGRequirements(ld, reqs)
            FindDataRequirements(ld, reqs)
            is_independent ← Intersection(reqs, remaining_loads) == ∅
            if is_independent then
                phase ← phase + ld
            end
        end
        remaining_loads ← remaining_loads \ phase
        phase_loads ← phase_loads + phase
    end
    return phase_loads
end
```

**Algorithm 2**: Separating loads for multiple Access phases.
Figure 5. Early evaluation of branches enables the elimination of duplicated branches in Clairvoyance mode. Relevant branches are evaluated at the beginning of the loop. If the evaluated branches are taken, the optimized Clairvoyance code with merged basic blocks is executed; otherwise, the decoupled unrolled code (with branch duplication) is executed.

respective branch predicates evaluate to true, Clairvoyance can then execute a version in which their respective basic blocks are merged. The right of Figure 5 shows the transformed loop, which checks $t_1$ and $t_2$ and if both predicates are true (i.e., both branches are taken), execution continues with the optimized version, in which the duplicated branch is eliminated. If $t_1$ or $t_2$ are false, then a decoupled unrolled version is executed.

The branches selected for clustering affect how often the optimized version will be executed. If we select all branches, the probability of all of them evaluating to true shrinks. Deciding the optimal combination of branches is a trade-off between branch duplication and the ratio of executing the optimized vs. the unoptimized version. As a heuristic, we only cluster branches if they statically have a probability above a given threshold. See Section I.4 for more details.
I.2.5 Overcoming Register Pressure

**Problem:** Early execution of loads stretches registers’ live ranges, which increases register pressure. Register pressure is problematic for two reasons: first, spilling a value represents an immediate *use* of the long latency load, which may stall the processor (assuming that Clairvoyance targets critical loads, whose latency cannot be easily hidden by a limited OoO engine); second, spill code increases the number of instructions and stack accesses, which hurts performance.

**Solution:** The Clairvoyance approach for selecting the loads to be hoisted to *Access* and for transforming the code *naturally* reduces register pressure. First, the compiler identifies potentially critical loads, which significantly reduces the number of instructions hoisted to *Access* phases. Second, critical loads that entail memory dependencies are prefetched instead of being hoisted, which further reduces the number of registers allocated in the *Access* phase. Third, multi-*Access* phases represent *consumers* of prior *Access* phases, releasing register pressure. Fourth, merging branches and *consuming* the branch predicate early releases the allocated registers. Furthermore, should register pressure become a bottleneck, one can decide to combine prefetching and instruction reordering (i.e., prefetch rather than hoist some of the critical loads), thus turning long latencies into short latencies, which can be hidden easily without increasing register pressure.

In a nutshell, each of the optimizations mentioned above, designed to overcome the typical problems of global instruction scheduling and software pipelining, contribute to reduced register pressure.

I.2.6 Heuristic to Disable Clairvoyance Transformations

Clairvoyance may cause performance degradation despite the efforts to reduce the overhead. This is the case for loops with long latency loads guarded by many nested if-else branches. We define a simple heuristic to decide when the overhead of branches may outweigh the benefits, namely, if the number of targeted loads is low in comparison to the number of branches. To this end, we use a metric which accounts for the number of loads to be hoisted and the number of branches required to reach the loads: \( \frac{\text{loads}}{\text{branches}} < 0.7 \), and disable Clairvoyance transformations if the condition is met.

I.2.7 Parameter Selection: Unroll Count and Indirection

We rely on state-of-the-art runtime version selectors to select the best performing version. In addition, simple static heuristics are used to simplify the configuration selection: small loops with few loads profit from a high unroll count to increase MLP; loops containing a high number of nested branches should have a low unroll and indirection count to reduce instruction count.
overhead; loops with large basic blocks containing both loads and computation may profit from a hybrid model using loads and prefetches to balance register pressure and instruction count overhead.

### I.2.8 Limitations

Currently, Clairvoyance relies on the LLVM loop unrolling, which is limited to inner-most loops. To tackle outer-loops, standard techniques such as unroll and jam are required. Unroll and jam refers to partially unrolling one or more loops higher in the nest than the innermost loop, and then fusing ("jamming") the resulting loops back together.

### I.3 Experimental Setup

Our transformation is implemented as a separate compilation pass in LLVM 3.8 [100]. We evaluate a range of C/C++ benchmarks from the SPEC CPU2006 [85] and NAS benchmark [104, 108, 109] suites on an APM X-Gene processor [69], see Table 1 for the architectural specifications. The remaining benchmarks were not included due to the difficulties in compilation with LLVM or simply because they were entirely compute-bound.

Clairvoyance targets loops in the most time-intensive functions (listed in Table 2), such that the benefits are reflected in the application’s total execution time. For SPEC, the selection was made based on previous studies [70], while for NAS we identified the target functions using Valgrind [105].

In Section I.2.4 we introduced an optimization to merge basic blocks if the static branch prediction indicates a probability above a certain threshold. For the following evaluation, we cluster branches only if the probability is above 90%.
Table 2. Modified functions.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Function</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>401.bzip2</td>
<td>BZ2_compressBlock</td>
<td></td>
</tr>
<tr>
<td>403.gcc</td>
<td>reg_is_remote_constant_p</td>
<td></td>
</tr>
<tr>
<td>429.mcf</td>
<td>primal_bea_mpp</td>
<td></td>
</tr>
<tr>
<td>433.milc</td>
<td>mult_su3_na</td>
<td></td>
</tr>
<tr>
<td>444.namd</td>
<td>calc_pair_energy_fullelect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>calc_pair_energy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>calc_pair_energy_merge_fullelect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>calc_pair_fullelect</td>
<td></td>
</tr>
<tr>
<td>445.gobmk</td>
<td>dfa_matchpat_loop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>incremental_order_moves</td>
<td></td>
</tr>
<tr>
<td>450.soplex</td>
<td>entered</td>
<td></td>
</tr>
<tr>
<td>456.hmmer</td>
<td>P7Viterbi</td>
<td></td>
</tr>
<tr>
<td>458.sjeng</td>
<td>std_eval</td>
<td></td>
</tr>
<tr>
<td>462.libquantum</td>
<td>quantum_toffoli</td>
<td></td>
</tr>
<tr>
<td></td>
<td>quantum_sigma_x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>quantum_cnot</td>
<td></td>
</tr>
<tr>
<td>464.h264ref</td>
<td>SetupFastFullPelSearch</td>
<td></td>
</tr>
<tr>
<td>470.lbm</td>
<td>LBM_performStreamCollide</td>
<td></td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>shiftup</td>
<td></td>
</tr>
<tr>
<td>473.astar</td>
<td>makebound2</td>
<td></td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>mgau_eval</td>
<td></td>
</tr>
<tr>
<td>CG</td>
<td>conj_grad</td>
<td></td>
</tr>
<tr>
<td>LU</td>
<td>buts</td>
<td></td>
</tr>
<tr>
<td>UA</td>
<td>diffusion</td>
<td></td>
</tr>
</tbody>
</table>
### I.3.1 Evaluating LLVM, DAE and Clairvoyance

We compare our techniques to Software Decoupled Access-Execute (DAE) [120, 98] and the LLVM standard instruction schedulers list-ilp (prioritizes ILP), list-burr (prioritizes register pressure) and list-hybrid (balances ILP and register pressure). DAE reduces the energy consumption by creating a duplicated loop that prefetches data ahead of time, while running at low frequency and maintaining its original performance. We further attempt to compare Clairvoyance against software pipelining and evaluate a target-independent, readily available software pipelining pass [93]. The pass fails to pipeline the targeted loops (all except of one fail) due to the high complexity (control-flow and memory dependencies). LLVM’s software pipeliner is not readily applicable for the target architecture, and could thus not be evaluated in this work. In the following, we will evaluate three techniques:

- **LLVM-SCHED** LLVM’s best-performing scheduling technique (one of list-ilp, list-burr, and list-hybrid).
- **DAE** Best performing DAE version.
- **CLAIRVOYANCE** Best performing Clairvoyance version.

### I.3.2 A Study on Speculation Levels

For Clairvoyance we evaluate a number of versions that vary in their speculative nature. **Consv** is a conservative version which only hoists safe loads. In case of a chain of dependent loads, it turns the first unsafe load into a prefetch and does not target the remaining loads. **Spec-safe** is a speculative but safe version. It hoists safe loads, but unlike the **consv** version, in case of a chain...
of dependent loads, *spec-safe* duplicates unsafe loads in *Access* such that it is
able to reach the entire chain of dependent loads. Then it turns the last unsafe
load of each chain into a prefetch, and reloads the unsafe loads in *Execute*. *Spec*
is a speculative but unsafe version which hoists all safe and unsafe loads
and reuses them in *Execute*. Finally, *multi-spec-safe* and *multi-spec* represent
the multiple-access versions of the previous two.

The exploration of different speculation levels is a study to give an overview
on Clairvoyance’s performance assuming increasingly accurate pointer anal-
ysis. The conservative *consv* version shows what we can safely transform at
the moment, while *spec* indicates a *perfect alias analyzer*. We expect that
state-of-the-art pointer analyses [67] approach the accuracy of *spec*. *Spec-
safe* demonstrates the effect of combining both prefetches and loads. A better
pointer analysis would enable Clairvoyance to safely load more values, and
consequently we would have to cope with increased register pressure. To this
end, *spec-safe* is a version that balances between loads and prefetches, and
thus between register spills and increased instruction count overhead.

The speculative but safe versions (*spec-safe, multi-spec-safe*) may cause a
segmentation fault in *Access* when speculatively accessing memory locations
to compute the target address of the prefetch. Since our transformation ensures
that only safely loaded values are reused in *Execute*, segmentation faults that
are triggered during an *Access* can be safely caught and ignored, for exam-
ple by overwriting the segmentation fault handler. During code generation we
can differentiate between speculative loads (loads hoisted above may-aliasing
stores) and non-speculative loads (no-aliasing loads). If the address of a spec-
ulative load matches the address of the segmentation fault, the fault handler
ignores it; otherwise, the fault is exposed to the user. In practice, however,
none of the analyzed benchmarks caused such a fault.

*Spec* and *multi-spec* are intended as an oracle with perfect alias-analysis.
We do not implement a correction mechanism, as it would require alternative
execution paths and is beyond the goal of this proposal. The results for *spec*,
despite the speculative nature, are verified at runtime as being correct.

I.4 Evaluation

In this section, we first compare different versions of Clairvoyance, starting
with the conservative approach and gradually increasing the speculation level.
Next we discuss the performance and energy improvements of the applications
compiled with Clairvoyance.

I.4.1 Comparing Clairvoyance’s Speculation Levels

Figure 6 compares the normalized runtimes of all Clairvoyance versions across
all benchmarks. For the majority of workloads, the different degrees of spec-
Figure 6. Normalized total runtime w.r.t original execution (-O3), for all Clairvoyance versions.

ulation do not play a major role for the final performance. For hmmer and libquantum we observe a significant difference between the more conservative versions (consv, spec-safe, multi-spec-safe) and the speculative ones (spec, multi-spec). Hmmer is a compute bound benchmark whose workload fits in the cache; therefore, there is little expected improvement. Furthermore, the target loop consists of one main basic block that contains a large number of loads interleaved with store instructions. The prefetches added by the conservative versions are not separated enough from their actual uses and thus translate to pure instruction count overhead, especially for a compute-bound application. Since the speculative versions only reorder the instructions, there is no additional overhead. This also applies to libquantum: libquantum consists of very small and tight loops, such that any added instruction count overhead quickly outweighs the benefits of Clairvoyance.

On the other hand, there are workloads that benefit from a less aggressive hoisting of loads, such as lbm—which shows best results with spec-safe and multi-spec-safe. Separating a high number of potentially delinquent loads from their uses can increase register pressure significantly. Since spec-safe and its multiple access version multi-spec-safe use a combination of reordering loads and prefetches, these versions introduce less register pressure compared to spec.

I.4.2 Understanding Clairvoyance Best Versions

We categorize the benchmarks into memory-bound applications (mcf, milc, soplex, libquantum, lbm, omnetpp, astar, CG) and compute-bound applications (bzip2, gcc, namd, gobmk, hmmer, sjeng, h264ref, LU, UA) [88]. Table 4 lists the best performing Clairvoyance version for each memory-bound benchmark. Typically, the best performing versions rely on a high unroll count and a low indirection count. The branch-merging optimization that allows for a higher unroll count is particularly successful for mcf, as the branch operations connecting the unrolled iterations are merged, showing low overhead across
Table 4. Best performing versions for memory-bound benchmarks [88].

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Version</th>
<th>Unroll</th>
<th>Indir</th>
</tr>
</thead>
<tbody>
<tr>
<td>429.mcf</td>
<td>conv</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>433.milc</td>
<td>multi-spec-safe</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>450.soplex</td>
<td>spec</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>spec</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>470.libquantum</td>
<td>multi-spec-safe</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>Disabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>473.astar</td>
<td>Disabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CG</td>
<td>spec</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 7. Normalized total runtime w.r.t original execution (-O3) for the best version of DAE, LLVM schedulers, and the conservative and the best version of Clairvoyance, categorized into memory-bound (left) and compute-bound (right) benchmarks.

loop iterations. As the memory-bound applications contain a high number of long latency loads which can be hoisted to the Access phase, we are able to improve MLP while hiding the increased instruction count overhead. Clairvoyance was disabled for omnetpp and astar by the heuristic that prevents generating heavy-weight Access phases that may hurt performance.

For compute-bound benchmarks the best performing versions have a low unroll count and a low indirection count, yielding versions that are very similar to the original. This is expected as Clairvoyance cannot help if the entire workload fits in the cache. However, if applied on compute-bound benchmarks, Clairvoyance will reorder instructions hiding even L1 cache latency.

I.4.3 Runtime and Energy

Figure 7 compares the normalized runtimes when applying Clairvoyance and state-of-the-art techniques designed to hide memory latency: DAE and the optimal LLVM instruction scheduler selected for each particular benchmark. Clairvoyance-consv shows the performance achieved with the most conservative version, while Clairvoyance-best shows the performance achieved by
the best Clairvoyance version (which may be $consv$ or any of the speculative versions $spec-safe$, $multi-spec-safe$, $spec$, $multi-spec$). The baseline represents the original code compiled with -O3 using the default LLVM instruction scheduler. Measurements were performed by executing the benchmarks until completion. We attempted a comparison with available software pipeliners [93, 100] are either not available for our target machine, or cannot transform our target loops. For memory-bound applications we observe a geometric mean improvement of 7% with Clairvoyance-$consv$ and 13% with Clairvoyance-best, outperforming both DAE and the LLVM instruction schedulers. The best performing applications are $mcf$ (both Clairvoyance versions) and $lbm$ (with Clairvoyance-best), which show considerable improvements in the total benchmark runtime (43% and 31% respectively). These are workloads with few branches and very “condensed” long latency loads (few loads responsible for most of the LLC misses).

DAE is competitive to Clairvoyance, but fails to leverage the same performance for $mcf$. An analysis of the generated code suggests that DAE fails to identify the correct set of delinquent loads. Benchmarks with small and tight loops such as $libquantum$ suffer from the additional instruction count overhead introduced by DAE, which duplicates target loops in order to prefetch data in advance. A slight overhead is observed with Clairvoyance-$consv$ for tight loops, due to partial instruction duplication, but this limitation would be alleviated by a more precise pointer analysis, as indicated by Clairvoyance-best.

We further observe that $astar$ suffers from performance losses when applying DAE. $astar$ has multiple nested if-then-else branches, which are duplicated in $Access$ and thus hurt performance. In contrast, our simple heuristic disables Clairvoyance optimization for loops with a high number of nested

\[ \text{Figure 8. Normalized runtime per target loop w.r.t original loop execution (-O3) for the conservative and the best version of Clairvoyance, for memory-bound benchmarks.} \]
branches, and therefore avoids degrading performance. For the compute-bound applications, both the conservative and best versions of Clairvoyance preserve the O3 performance, on-par with the standard LLVM instruction schedulers, except for hmmer, where Clairvoyance-consv introduces an overhead. Again, a precise pointer analysis could alleviate this overhead and allow Clairvoyance to hide L1 latency, as in the case of h264ref.

Figure 8 shows per loop runtimes, normalized to the original loop execution. Highly memory-bound benchmarks show significant speed-ups, mcf-68%, milc-20% and lbm-31%. Clairvoyance-consv introduces a small overhead for libquantum, which is corrected by Clairvoyance-best (assuming a more precise pointer analysis). As mentioned previously, Clairvoyance was disabled for omnetpp and astart. Overall, Clairvoyance-consv improves per loop runtime by 15%, approaching the performance of Clairvoyance-best (20%).

We collect power numbers using measurement techniques similar to Spiliopoulos et al. [114]. Figure 9 shows the normalized energy consumption for all memory-bound benchmarks. The results align with the corresponding runtime trends: benchmarks as mcf and lbm profit the most with an energy reduction of up to 25%. For memory-bound benchmarks, we achieve a geometric improvement of 5%. By overlapping outstanding loads we increase MLP, which in turn results in shorter runtimes and thus lower total energy consumption.

I.5 Related Work

Hiding long latencies of memory accesses to deliver high-performance has been a monumental task for compilers. Early approaches relied on compile-time instruction schedulers [102, 83, 103, 87, 118] to increase instruction level parallelism (ILP) and hide memory latency by performing local- or global-
scheduling. Local scheduling operates within basic block boundaries and is the most commonly adopted algorithm in mainstream compilers. Global scheduling moves instructions across basic blocks and can operate on cyclic or acyclic control-flow-graph. One of the most advanced forms of static instruction schedulers is modulo scheduling [71, 122], also known as software pipelining, which interleaves different iterations of a loop.

Clairvoyance overcomes challenges that led static instruction schedulers to generate suboptimal code: (1) Clairvoyance identifies potential long latency loads to compensate for the lack of dynamic information; (2) Clairvoyance combines prefetching with safe-reordering of accesses to address the problem of statically unknown memory dependencies; (3) Clairvoyance performs advanced code transformations of the control-flow graph, yielding Clairvoyance applicable on general-purpose applications, which were until now not amenable to software-pipelining. We emphasize that off-the-book-shelf software pipelining is tailored for independent loop iterations and is readily applicable on statically analyzable code, but it cannot handle complex control-flow, statically unknown dependencies, etc. Furthermore, among the main limitations of software pipelining are the prologues and epilogues, and high register pressure, typically addressed with hardware support.

Clairvoyance advances the state-of-the-art by demonstrating the efficiency of these code transformations on codes that abound in indirect memory accesses, pointers, entangled dependencies, and complex, data-dependent control-flow.

Typically, instruction scheduling and register allocation are two opposing forces [81, 76, 77]. Previous work attempts to provide register pressure sensitive instruction scheduling, in order to balance instruction level parallelism, latency, and spilling. Chen et al. [78] propose code reorganization to maximize ILP with a limited number of registers, by first applying a greedy superblock scheduler and then pushing over-hoisted instructions back. Yet, such instruction schedulers consider simple code transformations and compromise on other optimizations for reducing register pressure. Clairvoyance naturally releases register pressure by precisely increasing the live-span of certain loads only, by combining instruction reordering with prefetching and by merging branches.

Hardware architectures such as Very Long Instruction Word (VLIW) and EPIC [121, 96], identify independent instructions suitable for reordering, but require significant hardware support such as predicated execution, speculative loads, verification of speculation, delayed exception handling, memory disambiguation, etc. In contrast, Clairvoyance is readily applicable on contemporary, commodity hardware. Clairvoyance decouples the loop, rather than simply reordering instructions; it generates optimized code that can reach delinquent loads, without speculation or hardware support for predicated execution and handles memory and control dependencies purely in software. Clairvoy-
ance provides solutions that can re-enable decades of research on compiler techniques for VLIW-like and EPIC-like architectures.

Software prefetching [95] instructions, when executed timely, may transform long latencies into short latencies. Clairvoyance attempts to fully hide memory latency with independent instructions (ILP) and to cluster memory operations together and increase MLP by decoupling the loop. Software Decoupled Access-Execute (DAE) [120, 98] targets reducing energy expenditure using DVFS, while maintaining performance, whereas Clairvoyance focuses on increasing performance. DAE generates Access-Execute phases that merely prefetch data and duplicate a significant part of the original loop (control instructions and address computation). Clairvoyance’s contribution consists in finding the right balance between code rematerialization and instruction reordering, to achieve high degrees of ILP and MLP, without the added register pressure. DAE uses heuristics to identify the loads to be prefetched which take into consideration memory-dependencies. In addition, Clairvoyance combines information about memory- and control- dependencies, which increases the accuracy and effectiveness of the long latency loads identification.

Helper threads [94, 107, 119] attempt to hide memory latency by warming up the cache using a prefetching thread. Clairvoyance uses a single thread of execution, reuses values already loaded in registers (between Access and Execute phases) and resorts to prefetching only as a mechanism to safely handle unknown loop carried dependencies.

Software-hardware co-designs such as control-flow decoupling (CFD) [112] prioritize the evaluation of data-dependent branch conditions, and support a similar decoupling strategy for splitting load-use chains as our multi-access phases (however, their multi-level decoupling is done manually [111]). Contrary to Clairvoyance, CFD requires hardware support to ensure low-overhead communication between the decoupled phases. A software only version, Data-flow Decoupling (DFD), relies on prefetch instructions and ensures communication between phases by means of caches, akin to DAE [120, 98], using code duplication. As the CFD solution is not entirely automatic and requires manual intervention, Clairvoyance provides the missing compiler support and is readily applicable to decouple the CFG and hoist branch-evaluation, in lieu of long latency loads. Moreover, Clairvoyance provides software solutions to replace the hardware support for efficient communication between the decoupled phases. CFD makes use of decoupled producer phases for branches, similar to Clairvoyance’s multi-access phases, but low-overhead communication is achieved with hardware support.

I.6 Conclusion

Improving the performance, and therefore the energy-efficiency, of today’s power-limited, modern processors is extremely important given the end of
Dennard scaling. While aggressive out-of-order designs achieve high performance, they do so at a high cost.

Instead of improving performance with aggressive out-of-order processors, limited, efficient out-of-order processors can be used. Unfortunately, the reach of these efficient processors – as measured by the number of dynamic instructions that they can track before becoming stalled – tends to be much less than in aggressive cores. This limits the performance of the more efficient out-of-order processors for memory-intensive applications with high latency, distant independent instructions.

In this work, we propose a new technique to improve a processor’s performance by increasing both memory and instruction-level-parallelism and therefore the amount of useful work that is done by the core. The Clairvoyance compiler techniques introduced by this work overcome limitations imposed by may-alias loads, reordering dependent memory operations across loop iterations, and controlling register pressure. Using these techniques, we achieve performance improvements of up to 43% (7% geomean improvement for memory-bound applications with a conservative approach and 13% with a speculative but safe approach) on real hardware. The use of Clairvoyance enables optimizations that move beyond standard instruction reordering to achieve energy efficiency and overall higher performance in the presence of long latency loads.

**Acknowledgements**

We would like to thank Andreas Scherman for his contribution to the branch clustering strategy. This work is supported, in part, by the Swedish Research Council UPMARC Linnaeus Centre and by the Swedish VR (grant no. 2010-4741).
1. References


[62] Sangmin Seo, Jungwon Kim, Gangwon Jo, Jun Lee, Jeongho Nah, and Jaejin...
Lee. SNU NPB suite. Online

Multiple-block ahead branch predictors. *Operating Systems Review*,

Architectures: Recent Advances*. Synthesis Lectures on Computer Architecture,
2014.

[65] Vasileios Spiliopoulos, Andreas Sembrant, and Stefanos Kaxiras. Power-sleuth:
A tool for investigating your program’s power behavior. In *Proceedings of the
International Symposium on Modeling, Analysis Simulation of Computer and


[67] Sui Yulei, Di Peng, and Xue Jingling. Sparse flow-sensitive pointer analysis for
multithreaded programs. In *Proceedings of the International Symposium on
Paper II

Static Instruction Scheduling for High Performance on Limited Hardware

Kim-Anh Tran, Trevor E. Carlson, Konstantinos Koukos, Magnus Själander, Vasileios Spiliopoulos, Stefanos Kaxiras and Alexandra Jimborean

In IEEE Transactions on Computers, 2018 (to appear)
Abstract

Complex out-of-order (OoO) processors have been designed to overcome the restrictions of outstanding long-latency misses at the cost of increased energy consumption. Simple, limited OoO processors are a compromise in terms of energy consumption and performance, as they have fewer hardware resources to tolerate the penalties of long-latency loads. In worst case, these loads may stall the processor entirely. We present Clairvoyance, a compiler based technique that generates code able to hide memory latency and better utilize simple OoO processors. By clustering loads found across basic block boundaries, Clairvoyance overlaps the outstanding latencies to increases memory-level parallelism. We show that these simple OoO processors, equipped with the appropriate compiler support, can effectively hide long-latency loads and achieve performance improvements for memory-bound applications. To this end, Clairvoyance tackles (i) statically unknown dependencies, (ii) insufficient independent instructions, and (iii) register pressure. Clairvoyance achieves a geometric execution time improvement of 14% for memory-bound applications, on top of standard O3 optimizations, while maintaining compute-bound applications’ high-performance.

II.1 Introduction

Computer architects of the past have steadily improved performance at the cost of radically increased design complexity and wasteful energy consumption [75, 110, 106]. Today, power is not only a limiting factor for performance; given the prevalence of mobile devices, embedded systems, and the Internet of Things, energy efficiency becomes increasingly important for battery lifetime [113].

Highly efficient designs are needed to provide a good balance between performance and power utilization and the answer lies in simple, limited out-of-order (OoO) cores like those found in the HPE Moonshot m400 [80] and the AMD A1100 Series processors [73]. Yet, the effectiveness of moderately-aggressive OoO processors is limited when executing memory-bound applications, as they are unable to match the performance of the high-end devices, which use additional hardware to hide memory latency and extend the reach of the processor.

This work aims to improve the performance of the limited, more energy-efficient OoO processors, through the help of advanced compilation techniques specifically designed to hide the penalty of last level cache misses and better utilize hardware resources.

One primary cause for slowdown is last-level cache (LLC) misses, which, with conventional compilation techniques, result in a sub-optimal utilization of the limited OoO engine that may stall the core for an extended period of time. Our method identifies potentially critical memory instructions and hoists them earlier in the program’s execution, even across loop iteration boundaries, to increase memory-level parallelism (MLP). We overlap the outstanding misses with useful computation to hide their latency and, thus, also increase instruction-level parallelism (ILP).
Modern instruction schedulers for out-of-order processors are not designed for optimizing MLP or memory overlap, and assume that each memory access is a cache hit. Because of the limits of these simple out-of-order cores, hiding LLC misses is extremely difficult, resulting in the processor stalling, unable to perform additional useful work. Our instruction scheduler targets these specific problems directly, grouping loads together to increase memory-level parallelism, in order to increase performance and reduce energy dissipation. We address three challenges that need to be met to accomplish this goal:

1. **Finding enough independent instructions:** A last level cache miss can cost hundreds of cycles [79]. Conventional instruction schedulers operate on the basic-block level, limiting their reach, and, therefore, the number of independent instructions that can be scheduled in order to hide long latencies. More sophisticated techniques (such as software pipelining [71, 122]) schedule across basic-block boundaries, but instruction reordering is severely restricted in general-purpose applications when pointer aliasing and loop-carried dependencies cannot be resolved at compile-time. Clairvoyance introduces a hybrid load reordering and prefetching model that can cope with statically unknown dependencies in order to increase the reach of the compiler while ensuring correctness.

2. **Chains of dependent long-latency instructions may stall the processor:** Dependence chains of long-latency instructions prevent parallel accesses to memory and may stall a limited OoO core, as the evaluation of one long-latency instruction is required to execute another (dependent) long-latency instruction. Clairvoyance splits up dependent load chains and schedules independent instructions in-between to enable required loads to finish before their dependent loads are issued.

3. **Increased register pressure:** Separating loads and their uses to overlap outstanding loads with useful computation increases register pressure. This causes additional register spilling and increases the dynamic instruction count. Controlling register pressure, especially in tight loops, is crucial. Clairvoyance naturally reduces register pressure by prefetching loads that are not safe to reorder. This, however, assumes that the compiler cannot statically determine whether loads are safe to reorder.

   In our previous work [116] the compiler was too conservative. To ensure correctness, Clairvoyance re-ordered only those that were statically known not to alias with any preceding store. The full potential can, however, only be unlocked by targeting a wider range of long-latency loads.

   In this work we aim to reach the performance of the most speculative version of Clairvoyance, while guaranteeing correctness. We extend our previous work through the following contributions:

   1. **Improving Alias Analysis:** We improve Clairvoyance’s conservative version by integrating a more powerful pointer analysis, which is able to disambiguate more memory operations. As a result, we can reduce the per-
formance gap between the conservative version and Clairvoyance’s best speculative versions (Section II.2.6, Section II.4.4).

2. **Making the Speculative Version Reliable:** While previously speculation acted as an oracle, we now add support for mis-speculation such that the speculative version can be safely invoked if necessary (Section II.2.2, Section II.4.6).

3. **Handling of Register Pressure:** Reordering instructions and purposely increasing register lifetime – by separating loads from their uses – increases register pressure. To mitigate register pressure and spilling, we propose heuristics that determine which loads to reorder (and keep the loaded values in registers) and which loads to prefetch without increasing register pressure (Section II.2.5).

4. **Comparing against State-of-the-Art Prefetching:** We extend the evaluation to include the comparison to state-of-the-art prefetching techniques (Section II.4).

5. **Understanding the Performance:** We provide new static and dynamic statistics that provide new insights into the performance gains achieved by Clairvoyance (Section II.4.5).

Clairvoyance generated code runs on real hardware prevalent in mobile devices and in high-end embedded systems and delivers high-performance, thus alleviating the need for power-hungry hardware complexity. In short, Clairvoyance increases the performance of single-threaded execution by 17% (geomean improvement) on top of standard O3 optimizations, on hardware platforms that yield a good balance between performance and energy efficiency.

II.2 The Clairvoyance Compiler

This section outlines the general code transformation performed by Clairvoyance while each subsection describes the additional optimizations, which make Clairvoyance feasible in practice. Clairvoyance builds upon techniques such as software pipelining [122, 86], program slicing [117], and decoupled access-execute [97, 120, 98] and generates code that exhibits improved memory-level parallelism (MLP) and instruction-level parallelism (ILP). For this, Clairvoyance prioritizes the execution of critical instructions, namely loads, and identifies independent instructions that can be interleaved between loads and their uses.

Figure 1 shows the basic Clairvoyance transformation, which is used as a running example throughout the paper\(^1\). The transformation is divided into two steps:

- **Loop Unrolling** To expose more instructions for reordering, we unroll the loop by a loop unroll factor count\(_{unroll} = 2^n\) with \(n = \{0, 1, 2, 3, 4\}\).

\(^1\)For simplicity we use examples with for-loop structures, but Clairvoyance is readily available for while, do-while and goto loops.
Figure 1. The basic Clairvoyance transformation. The original loop is first unrolled by $\text{count}_{\text{unroll}}$ which increases the number of instructions per loop iteration. Then, for each iteration, Clairvoyance hoists all (critical) loads and sinks their uses to create a memory-bound Access phase and a compute-bound Execute phase.

Figure 2. Selection of loads based on an indirection count $\text{count}_{\text{indir}}$. The Clairvoyance code for $\text{count}_{\text{indir}} = 0$ (left) and $\text{count}_{\text{indir}} = 1$ (right).
Higher unroll counts significantly increase code size and register pressure. In our examples, we set $n = 1$ for the sake of simplicity.

- **Access-Execute Phase Creation** Clairvoyance hoists all load instructions along with their requirements (control flow and address computation instructions) to the beginning of the loop\(^2\). The group of hoisted instructions is referred to as the *Access* phase. The respective uses of the hoisted loads and the remaining instructions are sunk in a so-called *Execute* phase.

*Access* phases represent the program slice of the critical loads, whereas *Execute* phases contain the remaining instructions (and guarding conditionals). When we unroll the loop, we keep non-statically analyzable exit blocks. All exit blocks (including goto blocks) in *Access* are redirected to *Execute*, from where they will exit the loop after completing all computation. The algorithm is listed in Algorithm 3 and proceeds by unrolling the original loop and creating a copy of that loop (the *Access* phase, Line 3). Critical loads are identified (*FindLoads*, Line 4) together with their program slices (instructions required to compute the target address of the load and control instructions required to reach the load, Lines 5 - 9). Instructions which do not belong to the program slice of the critical loads are filtered out of *Access* (Line 10), and instructions hoisted to *Access* are removed from *Execute* (Line 11). The uses of the removed instructions are replaced with their corresponding clone from *Access*. Finally, *Access* and *Execute* are combined into one loop (Line 12).

This code transformation faces the same challenges as typical software pipelining or global instruction scheduling: (i) selecting the loads of interest statically; (ii) disambiguating pointers to reason about *reordering memory instructions*; (iii) finding sufficient independent instructions in applications with *entangled dependencies*; (iv) reducing the instruction count overhead (e.g., stemming from partly duplicating control-flow instructions); and (v) overcoming register pressure caused by unrolling and separating loads from their uses. Each of these challenges and our solutions are detailed in the following subsections.

### II.2.1 Identifying Critical Loads

**Problem:** Selecting the right loads to be hoisted is essential in order to avoid code bloat and register pressure and to ensure that long-latency memory operations overlap with independent instructions.

**Solution:** We develop a metric, called indirection count, based on the number of memory accesses required to compute the memory address (indirections) [98] and the number of memory accesses required to reach the load. For

\(^2\)We do not maintain precise exception semantics, as we reorder memory instructions that may throw an exception.
**Input**: Loop $L$, Unroll Count $\text{count}_{\text{unroll}}$

**Output**: Clairvoyance Loop $L_{\text{Clairvoyance}}$

1. begin
2. $L_{\text{unrolled}} \leftarrow \text{Unroll}(L, \text{count}_{\text{unroll}})$
3. $L_{\text{access}} \leftarrow \text{Copy}(L_{\text{unrolled}})$
4. hoist_list $\leftarrow \text{FindLoads}(L_{\text{access}})$
5. to_keep $\leftarrow \emptyset$
6. for load in hoist_list do
7. requirements $\leftarrow \text{FindRequirements}(\text{load})$
8. to_keep $\leftarrow \text{Union}(\text{to_keep}, \text{requirements})$
9. end
10. $L_{\text{access}} \leftarrow \text{RemoveUnlisted}(L_{\text{access}}, \text{to_keep})$
11. $L_{\text{execute}} \leftarrow \text{ReplaceListed}(L_{\text{access}}, L_{\text{unrolled}})$
12. $L_{\text{Clairvoyance}} \leftarrow \text{Combine}(L_{\text{access}}, L_{\text{unrolled}})$
13. return $L_{\text{Clairvoyance}}$
14. end

**Algorithm 3**: Basic Clairvoyance algorithm. The Access phase is built from a copy of the unrolled loop. The Execute phase is the unrolled loop itself, while all already computed values in Access are reused in Execute.

example, $x[y[z[i]][$]]$ has an indirection count of two, as it requires two loads to compute the address. The latter interpretation of indirection count is dependent on the control flow graph (CFG). If a load is guarded by two if-conditions that in turn require one load each, then the indirection count for the CFG dependencies is also two. Figure 2 shows an example of load selection with indirection counts. A high value of indirection indicates the difficulty of predicting and prefetching the load in hardware, signaling an increased likelihood that the load will incur a cache miss. For each value of this metric, a different code version is generated (i.e., hoisting all loads that have an indirection count less than or equal to the certain threshold). We restrict the total number of generated versions to a fixed value to control code size increase. Runtime version selection (orthogonal to this proposal) can be achieved with dedicated tools such as Protean code [101] or VMAD [89, 91].

**II.2.2 Handling Unknown Dependencies**

**Problem**: Hoisting load operations above preceding stores is correct if and only if all read-after-write (RAW) dependencies are respected. When aliasing information is not known at compile-time, detecting dependencies (or guaranteeing the lack of dependencies) is impossible, which either prevents reordering or requires speculation and/or hardware support. However, speculation typically introduces considerable overhead by squashing already executed instructions and requiring expensive recovery mechanisms.
Solution: We propose a lightweight solution for handling statically known and unknown dependencies, which ensures correctness and efficiency. Clairvoyance embraces safe speculation, which brings the benefits of going beyond conservative compilation, without sacrificing simplicity and lightness.

We propose a hybrid model to hide the latency of delinquent loads even when dependencies with preceding stores are unknown (i.e., may-alias). Thus, loads free of dependencies are hoisted to Access and the value is used in Execute, while loads that may alias with stores are prefetched in Access and safely loaded and used in their original position in Execute. May-aliases, however, are an opportunity, since in practice may-aliases rarely materialize into real aliasing at runtime [82]. Prefetching in the case of doubt is powerful: (1) if the prefetch does not alias with later stores, data will have been correctly prefetched; (2) if aliasing does occur, the prefetched data becomes overwritten and correctness is ensured by loading the data in the original program order.

Figure 3 shows an example in which an unsafe load is turned into a prefetch-load pair.

The proposed solution is safe. In addition to this solution, we analyze variations of this solution that showcase the potential of Clairvoyance when assuming a stronger alias analysis. These more speculative variations are allowed to hoist whole chains of may-aliasing loads and will be introduced during the experimental setup in Section II.3.
Table 1. Clairvoyance evaluated versions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consv</td>
<td>Conservative, only hoists safe loads</td>
</tr>
<tr>
<td>Spec-safe</td>
<td>Speculative (but safe), hoists may-aliasing load chains, but safely reloads them in <em>Execute</em></td>
</tr>
<tr>
<td>Spec</td>
<td>Speculative (unsafe), hoists may-aliasing load chains and reuses all data in <em>Execute</em></td>
</tr>
<tr>
<td>Multi-spec-safe</td>
<td>Multi-access version of spec-safe</td>
</tr>
<tr>
<td>Multi-spec</td>
<td>Multi-access version of spec</td>
</tr>
</tbody>
</table>

A Study on Speculation Levels

*Problem:* Prefetching the first may-aliasing load in a chain of may-aliasing loads is restrictive. First, this may prevent us from reaching the loads that actually miss in the cache. Second, it may become impossible to find enough loads to overlap the outstanding latencies.

*Solution:* In order to reach the target load when its address depends on a chain of may-aliasing loads, we evaluate three versions that vary in their speculative nature: Consv, spec-safe, and spec.

*Consv* is a conservative version which only hoists safe loads. In case of a chain of dependent loads, it turns the first unsafe load into a prefetch and does not target the remaining loads. *Spec-safe* is a speculative but safe version. It hoists safe loads, but unlike the *consv* version, in case of a chain of dependent loads, *spec-safe* duplicates unsafe loads in *Access* such that it is able to reach the entire chain of dependent loads. Then it turns the last unsafe load of each chain into a prefetch, and reloads the unsafe loads in *Execute*. *Spec* is a speculative but unsafe version which hoists all safe and unsafe loads and reuses them in *Execute*.

The exploration of different speculation levels is a study to give an overview on Clairvoyance’s performance assuming increasingly accurate pointer analysis. The conservative *consv* version shows what we can safely transform at the moment, while *spec* indicates a *perfect alias analyzer*. We expect that state-of-the-art pointer analyses [115] approach the accuracy of *spec*. *Spec-safe* demonstrates the effect of combining both prefetches and loads. A better pointer analysis would enable Clairvoyance to safely load more values, and consequently we would have to cope with increased register pressure. To this end, *spec-safe* is a version that balances between loads and prefetches, and thus between register spills and increased instruction count overhead. Table 1 shows the evaluated versions (multi-access is explained in Section II.2.3).

71
The speculative but safe version (spec-safe) may cause a segmentation fault in Access when speculatively accessing memory locations to compute the target address of the prefetch. Since only safely loaded values are reused in Execute, segmentation faults that are triggered during an Access can be safely caught and ignored.

In order to avoid fine-grain differentiation between speculative loads (loads hoisted above may-aliasing stores) and non-speculative loads (no-aliasing loads), which may be expensive, we perform coarse-grain differentiation at loop level. The idea is to restore a previously saved state and execute a back up version of the original loop, whenever a segmentation fault occurred for spec-safe. During the execution of the original loop, Clairvoyance reordering will not cause any segmentation fault. If, however, the original program is faulty, the segmentation fault is triggered.

Although a differentiation on loop-iteration level (instead of loop-level) would allow for more flexibility, it would have required one call to sigsetjump and one additional branch instruction per loop iteration. This overhead would unnecessarily penalize the case where all may-aliases turn out to be no-aliases.

Figure 4 shows the set up of a segmentation fault handler that enables spec-safe to continue execution without faulting at runtime. The handler is deployed on program entry (main.cpp). The behavior of the segmentation fault handler depends on a flag (ignore). Before entering the reordered loop, the flag is set to true. It follows a call to sigsetjmp, which stores the current environment. If a
Figure 5. Splitting up dependent load chains. Clairvoyance creates one Access phase for each set of independent loads (and their requirements), which increases the distance between loads and their uses.

segmentation fault occurs, the custom handler will ignore the fault, set the flag to \textit{false}, and jump to the loop preheader (using \texttt{siglongjmp}). The loop preheader will then, on evaluation of \texttt{sigsetjmp}, restore the saved environment. Since the flag now evaluates to \textit{false}, the execution will continue with executing the original loop. If a segmentation fault was caused by Clairvoyance reordering, the loop will conclude without any error; otherwise, it will raise a segmentation fault as expected. After successful execution of the loop, the flag is set back to \textit{false} (ignore = true)\(^3\).

In practice, none of the analyzed benchmarks caused a fault and therefore none of our benchmarks makes use of this safety measure. Nevertheless, we evaluate the overhead of the segmentation fault handler in Section II.4.6.

II.2.3 Handling Chains of Dependent Loads

**Problem:** When a long-latency load depends on another long-latency load, Clairvoyance cannot simply hoist both load operations into \textit{Access}. If it did, the processor might stall, because the second load represents a \textit{use} of the first long-latency load. As an example, in Figure 5 we need to load the branch predicate \(t_1\) before we can load \(t_2\) (control dependency). If \(t_1\) is not cached, an access to \(t_1\) will stall the processor if the OoO engine cannot reach ahead far enough to find independent instructions and hide the load’s latency.

\(^3\)In a multi-threaded setting, one flag per thread is required. The segmentation fault signal is then delivered only to the offending thread. The \texttt{sigsetjmp} / \texttt{siglongjmp} calls are thread-safe [68]. Note though that the implementation of these calls may vary for each operating system.
Solution: We propose to build multiple Access phases, by splitting dependent load chains into chains of dependent Access phases. As a consequence, loads and their uses within access phase are separated as much as possible, enabling more instructions to be scheduled in between. By the time the dependent load is executed, the data of the previous load may already be available for use.

Each phase contains only independent loads, thus increasing the separation between loads and their uses. In Figure 5 we separate the loads into two Access phases. For the sake of simplicity, this example uses count<sub>unroll</sub> = 2, hence there are only two independent loads to collect into the first Access phase and four into the second Access phase.

The algorithm to decide how to distribute the loads into multiple Access phases is shown in Algorithm 4. The compiler first collects all target loads in remaining_loads, while the distribution of loads per phase phase_loads is initialized to empty-set. As long as the loads have not yet been distributed (Line 4), a new phase is created (Line 5) and populated with loads whose control-requirements (Line 8) and data-requirements (Line 9) do not match any of the loads that have not yet been distributed in a preceding Access phase (Line 10 and 11-14). Loads distributed in the current phase are removed from the remaining_loads only at the end (Line 15), ensuring that no dependent loads are distributed to the same Access phase. The newly created set of loads phase is added to the list of phases (Line 16) and the algorithm continues until all critical loads have been distributed. Next, we generate each Access phase by following Algorithm 3 corresponding to a set of loads from the list phase_loads.

In Section II.4 evaluate the multi-access phases on top of the speculative versions (thus noted as multi-spec, multi-spec-safe).

II.2.4 Overcoming Instruction Count Overhead

Problem: The control-flow-graph is partially duplicated in Access and Execute phases, which, on one hand, enables instruction reordering beyond basic block boundaries, but, on the other hand, introduces overhead. As an example, the branch using predicate \( t_1 \) (left of Figure 6) is duplicated in each Access phase, significantly increasing the overhead in the case of multi-Access phases. Branch duplication not only complicates branch prediction but also increases instruction overhead, thus hurting performance.

Solution: To overcome this limitation, Clairvoyance generates an optimized version where selected branches are clustered at the beginning of a loop. If the respective branch predicates evaluate to true, Clairvoyance can then execute a version in which their respective basic blocks are merged. The right of Figure 6 shows the transformed loop, which checks \( t_1 \) and \( t_2 \) and if both predicates are true (i.e., both branches are taken), execution continues with the optimized
**Input:** Set of loads

**Output:** List of sets \( \text{phase}\_\text{loads} \)

1 begin
2 
3 \hspace{1em} \text{remaining}\_\text{loads} \leftarrow \text{loads}
4 \hspace{1em} \text{phase}\_\text{loads} \leftarrow \[]
5 \hspace{1em} \text{while} \ \text{remaining}\_\text{loads} \neq \emptyset \text{ do}
6 \hspace{2em} \text{phase} \leftarrow \emptyset
7 \hspace{2em} \text{for} \ \text{ld} \ \text{in} \ \text{remaining}\_\text{loads} \text{ do}
8 \hspace{3em} \text{reqs} \leftarrow \emptyset
9 \hspace{3em} \text{FindCFGRequirements}(\text{ld}, \text{reqs})
10 \hspace{3em} \text{FindDataRequirements}(\text{ld}, \text{reqs})
11 \hspace{3em} \text{is}\_\text{independent} \leftarrow \text{Intersection}(\text{reqs}, \text{remaining}\_\text{loads})
12 \hspace{3em} \text{==} \emptyset \text{ then}
13 \hspace{4em} \text{phase} \leftarrow \text{phase} + \text{ld}
14 \hspace{2em} \text{end}
15 \hspace{1em} \text{remaining}\_\text{loads} \leftarrow \text{remaining}\_\text{loads} \setminus \text{phase}
16 \hspace{1em} \text{phase}\_\text{loads} \leftarrow \text{phase}\_\text{loads} + \text{phase}
17 \text{end}
18 \text{return} \text{phase}\_\text{loads}
19 end

**Algorithm 4:** Separating loads for multiple Access phases.
Figure 6. Early evaluation of branches enables the elimination of duplicated branches. Relevant branches are evaluated at the beginning of the loop. If the evaluated branches are taken, the optimized Clairvoyance code with merged basic blocks is executed; otherwise, the decoupled unrolled code (with branch duplication) is executed.

version, in which the duplicated branch is eliminated. If $t_1$ or $t_2$ are false, then a decoupled unrolled version is executed.

The branches selected for clustering affect how often the optimized version will be executed. If we select all branches, the probability of all of them evaluating to true shrinks. Deciding the optimal combination of branches is a trade-off between branch duplication and the ratio of executing the optimized vs. the unoptimized version. As a heuristic, we only cluster branches if they statically have a probability above a given threshold. See Section Section II.4 for more details.
II.2.5 Overcoming Register Pressure

Problem: Early execution of loads stretches registers’ live ranges, which increases register pressure. Register pressure is problematic for two reasons: first, spilling a value represents an immediate use of the long-latency load, which may stall the processor (assuming that Clairvoyance targets critical loads, whose latency cannot be easily hidden by a limited OoO engine); second, spill code increases the number of instructions and stack accesses, which hurts performance.

Solution: The Clairvoyance approach for selecting the loads to be hoisted to Access and for transforming the code naturally reduces register pressure. First, the compiler identifies potentially critical loads, which significantly reduces the number of instructions hoisted to Access phases. Second, critical loads that entail memory dependencies are prefetched instead of being hoisted, which further reduces the number of registers allocated in the Access phase. Third, multi-Access phases represent consumers of prior Access phases, releasing register pressure. Fourth, merging branches and consuming the branch predicate early releases the allocated registers.

If still more loads are hoisted than registers may exist, we introduce a heuristic to select which critical loads to hoist, and which ones to prefetch instead, in order to release register pressure. Prefetching is used as a mechanism to turn long latencies into short latencies, which can be easily hidden by the OoO core, without increasing register pressure.

Limiting the Number of Registers in Use

Given the number of architectural registers $R$ we limit the number of loads hoisted to the Access phase by $R$. The intuition is to keep all hoisted loads in registers. These hoisted loads may be consumed by other loads and thus, in practice, not require a register for the whole duration of an Access phase, if not reused in Execute. Note that this strategy does not guarantee that no spilling will occur. First, we do not only keep loaded values alive, but also all other computed values that can be safely reused in the Execute phase. Second, register allocation is a separate step that has not yet happened. The chosen heuristic is a means to have a handle on register pressure.

In the following, we explain the details of how to determine when to hoist a load and when to prefetch the value instead. Similar to the creation of multiple access phases, we begin by separating the loads into sets of loads, see Algorithm 4. Within a set, all loads are independent. A load in a set, however, depends on one or more loads of the previous set. Algorithm 5 shows how we select the loads to hoist or prefetch after having created the sets of loads. First we loop through the load sets one by one while we still have registers left (Line 6). For each set, we decide if the load should be hoisted or simply prefetched. If the current number of loads to hoist has not yet exceeded the maximum number of available registers (Line 8), we hoist the load (instruction reorder-
ing) (Line 9), otherwise we prefetch the value from the target address (Line 11). If, by the end of looping through the current set of loads, some loads were prefetched instead of being reordered, we stop the main loop. Since the next set may contain some loads that can be prefetched (if all of their dependencies might be already hoisted to the Access phase, i.e. contained in to_reuse, we look through the next set (Line 17), and choose to prefetch each load that has all its requirements hoisted (Line 19).

Input: List of sets load_sets, Maximum number of registers max_regs
Output: Set of to_reuse and to_prefetch

begin
  to_reuse ← ∅
  to_prefetch ← ∅
  iter ← GetIterator(load_sets)
  while HasNext(iter) and size_of(to_reuse) < max_regs do
    set ← Next(iter)
    for ld in set do
      if size_of(to_reuse) < max_regs then
        to_reuse ← to_reuse + ld
      else
        to_prefetch ← to_prefetch + ld
      end
    end
  end
  if HasNext(iter) then
    set ← Next(iter)
    for ld in set do
      if GetRequiredLoads(ld) ⊆ to_reuse then
        to_prefetch ← to_prefetch + ld
      end
    end
  end
end

Algorithm 5: Heuristic to decide whether to reorder or prefetch loads.

II.2.6 Integrating State-of-the-Art Alias Analysis

We integrate the static value-flow analysis SVF [115] for single-threaded applications to improve the precision of the alias analysis. We make the alias analysis available to LLVM as well, thus all versions, including the original ones (baseline), make use of the alias information, allowing for a more fair
comparison. Note that LLVM requires passes to explicitly preserve analysis information, such that all passes can make use of the analysis. We integrated the SVF analysis into O3, however it may be that not all passes preserve the analysis.

II.2.7 Heuristic to Disable Clairvoyance Transformations
Clairvoyance may cause performance degradation despite the efforts to reduce the overhead. This is the case for loops with long-latency loads guarded by many nested if-else branches. We define a simple heuristic to decide when the overhead of branches may outweigh the benefits, namely, if the number of targeted loads is low in comparison to the number of branches. To this end, we use a metric which accounts for the number of loads to be hoisted and the number of branches required to reach the loads: \( \frac{\text{loads}}{\text{branches}} < 0.7 \), and disable Clairvoyance transformations if the condition is met.

II.2.8 Parameter Selection: Unroll Count and Indirection
We rely on state-of-the art runtime version selectors to select the best performing version. In addition, simple static heuristics are used to simplify the configuration selection: small loops with few loads profit from a high unroll count to increase MLP; loops containing a high number of nested branches should have a low unroll and indirection count to reduce instruction count overhead; loops with large basic blocks containing both loads and computation may profit from a hybrid model using loads and prefetches to balance register pressure and instruction count overhead.

II.2.9 Limitations

Outer Loop Transformations
Currently, Clairvoyance relies on the LLVM loop unrolling, which is limited to inner-most loops. To tackle outer-loops, standard techniques such as unroll and jam are required. Unroll and jam refers to partially unrolling one or more loops higher in the nest than the innermost loop, and then fusing ("jamming") the resulting loops back together.

Support for Multi-threaded Applications
Standard compilation techniques rely on the memory model sequential consistency for data race free code (SC-for-DRF) and perform optimizations within synchronization free regions as if the code was sequential. In the same manner, Clairvoyance is readily applicable within synchronization free regions, but instructions cannot be moved (reordered) across synchronization boundaries.
<table>
<thead>
<tr>
<th>Processor</th>
<th>APM X-Gene - AArch64 Octa-A57</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Count</td>
<td>8</td>
</tr>
<tr>
<td>ROB size</td>
<td>128 micro-ops [74]</td>
</tr>
<tr>
<td>Issue Width</td>
<td>8 [74]</td>
</tr>
<tr>
<td>L1 D-Cache</td>
<td>32 KB / 5-6 cycles depending on access complexity</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256 KB / 13 cycles Latency</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>8 MB / 90 cycles Latency</td>
</tr>
<tr>
<td>RAM</td>
<td>32 GB / 89 cycles + 83 ns (for random RAM page)</td>
</tr>
</tbody>
</table>

Typically, multi-threaded applications include synchronization points within loop bodies, for example, critical sections or even the simple lock taken by a thread to check if there are any iterations left to execute. Synchronization prevents instructions to be safely hoisted across these points. One approach to apply Clairvoyance on multi-threaded programs is to generate access-execute phases for each data-race-free region. However, these regions are small and would limit the ability of Clairvoyance to reorder, and thus cluster loads, as Clairvoyance unrolls several loop iterations to gather loads from different iterations.

To enable Clairvoyance instruction reordering on large code regions (i.e. across synchronization points) requires non-trivial inter-thread and inter-procedural compile-time analysis [92]. Our expectation is that with an increasing number of threads that compete for the shared cache, fewer data can be kept in the last level cache for each thread. Therefore, as load latencies are more likely to increase, we expect that Clairvoyance will benefit even more multi-threaded applications that are not embarrassingly parallel. A thorough evaluation of Clairvoyance on multi-threaded applications is left as future work.

## II.3 Experimental Setup

Our transformation is implemented as a separate compilation pass in LLVM 4.0 [100]. We evaluate a range of C/C++ benchmarks from the SPEC CPU2006 [85] and NAS benchmark [104, 108, 109] suites on an APM X-Gene processor [69], see Table 2 for the architectural specifications. The remaining benchmarks were not included due to the difficulties in compilation with LLVM or simply because they were entirely compute-bound. Although we have not run experiments on x86-processors, we expect that for more aggressive out-of-order processors Clairvoyance will not provide benefit, but will also not harm execution.

Clairvoyance targets loops in the most time-intensive functions (see Table 3), such that the benefits are reflected in the application’s total execution time.
Table 3. **Modified functions.**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>401.bzip2</td>
<td>BZ2_compressBlock</td>
</tr>
<tr>
<td>403.gcc</td>
<td>reg_is_remote_constant_p</td>
</tr>
<tr>
<td>429.mcf</td>
<td>primal_bea_mpp</td>
</tr>
<tr>
<td>433.milc</td>
<td>mult_su3_na</td>
</tr>
<tr>
<td>444.namd</td>
<td>calc_pair_energy_fullelect</td>
</tr>
<tr>
<td></td>
<td>calc_pair_energy</td>
</tr>
<tr>
<td></td>
<td>calc_pair_energy_merge_fullelect</td>
</tr>
<tr>
<td></td>
<td>calc_pair_fullelect</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>dfa_matchpat_loop</td>
</tr>
<tr>
<td></td>
<td>incremental_order_moves</td>
</tr>
<tr>
<td>450.soplex</td>
<td>entered</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>P7Viterbi</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>std_eval</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>quantum_toffoli</td>
</tr>
<tr>
<td></td>
<td>quantum_sigma_x</td>
</tr>
<tr>
<td></td>
<td>quantum_cnot</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>SetupFastFullPelSearch</td>
</tr>
<tr>
<td>470.lbm</td>
<td>LBM_performStreamCollide</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>shiftup</td>
</tr>
<tr>
<td>473.astar</td>
<td>makebound2</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>mgau_eval</td>
</tr>
<tr>
<td>CG</td>
<td>conj_grad</td>
</tr>
<tr>
<td>LU</td>
<td>buts</td>
</tr>
<tr>
<td>UA</td>
<td>diffusion</td>
</tr>
</tbody>
</table>
time. For SPEC, the selection was made based on previous studies [70], while for NAS we identified the target functions using Valgrind [105].

In Section II.2.4 we introduced an optimization to merge basic blocks if the static branch prediction indicates a probability above a certain threshold. For the following evaluation, we cluster branches if probability is above 90%.

II.3.1 Evaluating LLVM, DAE, SW-Prefetching and Clairvoyance

We compare our techniques to Software Decoupled Access-Execute (DAE) [120, 98], Software Prefetching for Indirect Memory Accesses [72] (SW-PREF) and the LLVM standard instruction schedulers list-ilp (prioritizes ILP), list-burr (prioritizes register pressure) and list-hybrid (balances ILP and register pressure). DAE reduces the energy consumption by creating an accesses phase that prefetches data ahead of time, while running at low frequency. These access phases can span tens to hundreds of iterations. Comparing DAE and Clairvoyance will showcase the difference between prefetching vs. loading, and coarse-grain vs. fine-grain handling of loads. SW-PREF is a software prefetching technique that targets indirect memory accesses. It inserts prefetches for each indirect load whose address can be generated by adding an offset to a referenced induction variable. We attempted to compare Clairvoyance against software pipelining and evaluated a target-independent, readily available software pipelining pass [93]. The pass fails to pipeline the targeted loops (except one loop) due to the high complexity (control-flow and memory dependencies). LLVM’s software pipeliner is not readily applicable for the target architecture, and could thus not be evaluated in this work.

We also compare to a hybrid of DAE and Clairvoyance, which performs the same transformations as Clairvoyance but, borrowing from DAE, always prefetches the last indirection and does not reuse any of the computed values in Access. In other words, Clairvoyance-DAE (1) unrolls the loop, (2) uses Clairvoyance heuristics (indirection count reflects memory and control-flow indirections), and (3) applies Clairvoyance-optimizations (branch clustering), just as other Clairvoyance versions. However, instead of keeping loaded values in registers, it only prefetches them, as in DAE. This version may also, as spec-safe, throw a segmentation fault during Access, if invalid memory addresses are accessed during address computation. The prefetch-only version serves as a comparison point to our reordering scheme.

In the following, we will evaluate four techniques:

**LLVM-SCHED** LLVM’s best-performing scheduling technique (one of list-ilp, list-burr, and list-hybrid).

**DAE** Best performing DAE version.

**SW-PREF** Software prefetch for indirect memory accesses.

82
II.4 Evaluation

In this section, we first compare different versions of Clairvoyance, starting with the conservative approach and gradually increasing the speculation level. We first discuss the performance and energy consumption of Clairvoyance’s best versions (among all speculation levels). Next, we compare the optimized but conservative version of Clairvoyance (which includes a state-of-the-art alias analysis and a heuristic to mitigate register pressure) to the previously known best version. Finally, we analyze the performance penalty that comes with ensuring correctness of the spec-safe version.

II.4.1 Comparing Clairvoyance’s Speculation Levels

Figure 7 compares the normalized runtimes of all Clairvoyance versions across all benchmarks. For the majority of workloads, the different degrees of speculation do not play a major role in the final performance. For hìmmer and libquantum we observe a significant difference between the more conservative versions (consv, spec-safe, multi-spec-safe) and the speculative ones (spec, multi-spec). The benchmarks contain small and tight loops, thus any added instructions introduce overhead that quickly outweighs the benefits of Clairvoyance. Since the speculative versions only reorder instructions, the overhead is minimal. Furthermore, hìmmer is a compute bound benchmark whose workload fits in the cache; therefore, there is little expected improvement.

On the other hand, there are workloads that benefit from hoisting loads, such as lbm—which shows best results with spec-safe and multi-spec-safe.
Table 4. Best performing versions for memory-bound benchmarks [88].

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Version</th>
<th>Unroll</th>
<th>Indir</th>
</tr>
</thead>
<tbody>
<tr>
<td>429.mcf</td>
<td>conv</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>433.milc</td>
<td>multi-spec-safe</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>450.soplex</td>
<td>spec</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>spec</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>470.lbm</td>
<td>multi-spec-safe</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>Disabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>473.astar</td>
<td>Disabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CG</td>
<td>spec</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Since spec-safe and its multiple access version multi-spec-safe use a combination of reordering loads and prefetches, these versions provide a better balance between register pressure and memory-level-parallelism compared to spec.

II.4.2 Understanding Clairvoyance Best Versions

We categorize the benchmarks into memory-bound applications (mcf, milc, soplex, libquantum, lbm, omnetpp, astar, CG) and compute-bound applications (bzip2, gcc, namd, gobmk, hmmer, sjeng, h264ref, LU, UA) [88]. Table 4 lists the best performing Clairvoyance version for each memory-bound benchmark. Typically, the best performing versions rely on a high unroll count and a low indirection count. The branch-merging optimization that allows for a higher unroll count is particularly successful for mcf, as the branch operations connecting the unrolled iterations are merged, showing low overhead across loop iterations. As the memory-bound applications contain a high number of long-latency loads that can be hoisted to the Access phase, we are able to improve MLP while hiding the increased instruction count overhead. Clairvoyance was disabled for omnetpp and astar by the heuristic that prevents generating heavy-weight Access phases that may hurt performance.

For compute-bound benchmarks the best performing versions have a low unroll count and a low indirection count, yielding versions that are very similar to the original. This is expected as Clairvoyance cannot help if the entire workload fits in the cache. However, when applied on compute-bound benchmarks, Clairvoyance will reorder instructions partly hiding even L1 cache latency.

II.4.3 Runtime and Energy

Figure 8 compares the normalized runtimes when applying Clairvoyance, its prefetch-only pendant (Clairvoyance-DAE), and state-of-the-art techniques de-
Figure 8. Normalized total runtime w.r.t original execution (-O3) for the best version of LLVM schedulers, SW-PREF, DAE, Clairvoyance-DAE, and the conservative and the best version of Clairvoyance, categorized into memory-bound (left) and compute-bound (right) benchmarks.

signed to hide memory latency: DAE, SW-PREF and the optimal LLVM instruction scheduler selected for each particular benchmark. Clairvoyance-consv shows the performance achieved with the most conservative version, while Clairvoyance-best shows the performance achieved by the best Clairvoyance version (which may be consv or any of the speculative versions spec-safe, multi-spec-safe, spec, multi-spec). The baseline represents the original code compiled with -O3 using the default LLVM instruction scheduler. Measurements were performed by executing the benchmarks until completion. For memory-bound applications we observe a geometric improvement of 7% with Clairvoyance-consv and 13% with Clairvoyance-best, outperforming both DAE and the LLVM instruction schedulers. The best performing applications are mcf (both Clairvoyance versions) and lbm (with Clairvoyance-best), which show considerable improvements in the total benchmark runtime (43% and 31% respectively). These are workloads with few branches and very “condensed” long-latency loads (few static load instructions responsible for most of the last level cache misses).

DAE is competitive to Clairvoyance, but fails to leverage the same performance for mcf. An analysis of the generated code suggests that DAE fails to identify the correct set of delinquent loads. Benchmarks with small and tight loops such as libquantum suffer from the additional instruction count overhead, since DAE duplicates target loops to prefetch data in advance. A slight overhead is observed with Clairvoyance-consv for tight loops, due to partial instruction duplication, but this limitation would be alleviated by a more precise pointer analysis, as indicated by Clairvoyance-best.

We further observe that a-star suffers from performance losses when applying DAE. A-star has multiple nested if-then-else branches, which are duplicated in Access and thus hurt performance. In contrast, our simple heuristic disables Clairvoyance optimization for loops with a high number of nested branches, and therefore avoids degrading performance. For the compute-bound applications, both Clairvoyance-consv and -best preserve the O3 per-
formance, on-par with the standard LLVM instruction schedulers, except for 
\texttt{hmmer}, where Clairvoyance-consv introduces an overhead due to prefetching 
instead of reordering. A precise pointer analysis could alleviate this overhead 
and enable Clairvoyance to hide L1 latency, as in the case of \texttt{h264ref}.

Clairvoyance-DAE shows that \texttt{lbm} and \texttt{CG} benefit from the prefetching-only 
scheme, which can unroll more iterations since no registers are blocked due 
to reordering. In contrast, \texttt{mcf} and \texttt{libquantum} profit from Clairvoyance op-
timizations. While DAE introduces significant overhead for \texttt{libquantum} due 
to the duplicated instructions, Clairvoyance-DAE profits from branch clustering 
which reduces the instruction count overhead significantly. For \texttt{mcf}, 
Clairvoyance-DAE identifies the correct set of delinquent loads, just as the 
other versions of Clairvoyance, as it makes use of the Clairvoyance-indirection 
heuristic which considers both memory and control-flow indirections. For 
compute-bound benchmarks instruction overhead is critical, thus Clairvoyance-
best still performs better. Overall, Clairvoyance-DAE and -best show similar 
geomean improvements, but since they provide benefits for different bench-
marks, combining them may enable even higher gains.

SW-PREF (with a look-ahead distance of 64) inserted prefetches for \texttt{CG}, 
\texttt{bzip2}, \texttt{sjeng}, \texttt{soplex}, and \texttt{namd}. For most of the transformed benchmarks 
SW-PREF does not benefit, neither harm performance, except \texttt{CG}, where SW-
 PREF outperforms all other techniques (improvement of 24%, compared to 
Clairvoyance-best 6%). SW-PREF is designed to prefetch indirect loads that 
do not require complex control flow for their address computation, and thus 
the targeted benchmarks differ from the ones we study. For example, some 
prefetches could not be inserted as they depend on non-loop-induction phi 
nodes (\texttt{mcf}), others were not inserted because there were no indirect loads to 
target (\texttt{milc}).
Figure 10 shows per loop runtimes, normalized to original. Highly memory-bound benchmarks show significant speed-ups, mcf-68%, milc-20% and lbm-31%. Clairvoyance-consv introduces a small overhead for libquantum, which is corrected by Clairvoyance-best (assuming a more precise pointer analysis). As mentioned previously, Clairvoyance was disabled for omnetpp and astar. Overall, Clairvoyance-consv improves per loop runtime by 15%, approaching the performance of Clairvoyance-best (20%).

We collect power numbers using measurement techniques similar to Spiliopoulos et al. [114]. Figure 9 shows the normalized energy consumption for all memory-bound benchmarks. The results align with the corresponding runtime trends: benchmarks as mcf and lbm profit the most with an energy reduction of up to 25%. For memory-bound benchmarks, we achieve a geometric improvement of 5%. By overlapping outstanding loads we increase MLP, which in turn results in shorter runtimes and thus lower total energy consumption.

II.4.4 Closing the Gap between Clairvoyance-best and Clairvoyance-consv

In Figure 8 Clairvoyance-best includes speculative versions. In fact, all benchmarks profited most from speculation except for mcf. In order to close the gap between Clairvoyance-best and Clairvoyance-consv, we introduce (i) an improved alias analyzer to disambiguate memory operations (Section II.2.6) and (ii) a new heuristic (Section II.2.5) to determine whether to hoist or prefetch a disambiguated load.

Figure 11 shows the updated comparison between Clairvoyance-best (or better, Clairvoyance-previous-best) and Clairvoyance-consv. The conserva-
tive version is now competitive with Clairvoyance-best, but without the need of any speculation. In fact, all targeted load store pairs can be successfully determined to be a no-alias or a must-alias, and thus no speculation is even required. Table 5 reflects the best performing Clairvoyance-conservative versions and the number of loads and prefetches in Access (prefetching happens as a result of our register balancing heuristic, and not because of unknown memory dependencies). The numbers reflect the loads and prefetches after running Clairvoyance and O3. O3 optimizations may remove or insert new load instructions in the Access and Execute phases. So, even though the AARCH64 execution state provides in total 31 general purpose registers, the total number of loads may be less or more than 31. Note that multi-conserv is now among the best versions: since more loads can be disambiguated, more and longer dependency chains exist that can be split into multiple access phases.

For the majority of the benchmarks the previous best versions are on-par or slightly outperform the corresponding Clairvoyance-conservative (e.g., 4% for mcf). For two benchmarks, CG and lbm, Clairvoyance-conservative outperforms the previous Clairvoyance-best (17% for CG and 3% for lbm). This difference can be traced back to the efficiency of the chosen heuristic, as the heuristic may choose to prefetch other loads than the previous prefetch/load scheme. While we previously unrolled and hoisted all no-aliasing loads (and only relied on prefetches of may-aliasing loads), we now only hoist loads as long as there are registers still left to use, while the rest of the addresses are prefetched instead.

The combination of improved alias analysis and applied heuristic to consider register pressure enables Clairvoyance to explore higher unroll counts while being able to handle register pressure. The heuristic chooses to prefetch other loads than Clairvoyance-best, which would use prefetches for may-aliasing

![Normalized runtime w.r.t. original execution (-O3) for Clairvoyance-conserv (with better alias analysis and heuristic to choose between hoisting and prefetching) and the previous Clairvoyance-best.](image)

*Figure 11. Normalized runtime w.r.t. original execution (-O3) for Clairvoyance-conserv (with better alias analysis and heuristic to choose between hoisting and prefetching) and the previous Clairvoyance-best.*
II.4.5 A Close Look Into Clairvoyance’s Performance Gains for Memory-Bound Benchmarks

In order to better understand Clairvoyance’s performance gains this section focuses on the relevant memory-bound benchmarks: mcf, lbm, milc CG, soplex, and libquantum. In addition to the already presented benchmarks, we further include IS (NAS benchmark suite).

For the analysis we gather runtime, the number of dynamic instruction, and load and store operations to caches using hardware performance counters (perf) and are shown in Figure 12. The instruction count gives an insight into the instruction count overhead that Clairvoyance introduces, partly due to additional prefetch instructions and branch duplication. The load and store counters serve as an estimate of inserted spill code that results from register pressure overhead. All numbers are normalized to the original (O3) execution. Each graph shows two bars: one for the best Clairvoyance-consv version and one for the unrolled version it is based on (e.g., if the best Clairvoyance-consv
Table 5. Best performing versions for memory-bound benchmarks using the improved Clairvoyance-consv, and their number of loads hoisted or prefetches inserted for each target loop.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Version</th>
<th>Unroll</th>
<th>Indir</th>
<th>#Loads</th>
<th>#Pref</th>
</tr>
</thead>
<tbody>
<tr>
<td>429.mcf</td>
<td>multi-consv</td>
<td>8</td>
<td>3</td>
<td>32</td>
<td>17</td>
</tr>
<tr>
<td>433.milc</td>
<td>multi-consv</td>
<td>2</td>
<td>0</td>
<td>11</td>
<td>28</td>
</tr>
<tr>
<td>450.soplex</td>
<td>consv</td>
<td>1</td>
<td>16</td>
<td>8,3,2,3</td>
<td>0,0,0,0</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>consv</td>
<td>4</td>
<td>2</td>
<td>9,9,6</td>
<td>0,0,0</td>
</tr>
<tr>
<td>470.lbm</td>
<td>multi-consv</td>
<td>16</td>
<td>1</td>
<td>30</td>
<td>358</td>
</tr>
<tr>
<td>CG</td>
<td>consv</td>
<td>16</td>
<td>1</td>
<td>32</td>
<td>17</td>
</tr>
<tr>
<td>IS</td>
<td>consv</td>
<td>8</td>
<td>1</td>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

version had an unroll count of 2, the evaluated unrolled version would have the same unroll count).

Figure 12a shows the normalized total runtime. For all benchmarks we see a performance gain from applying Clairvoyance on top of unrolling. Looking at the geometric mean, unrolling improves performance by 1%, while applying Clairvoyance on top of it allows for an improvement of 17%. Clairvoyance has its biggest impact on mcf (39%), lbm (34%) and CG (23%). All three of them, despite of their runtime gains, show a significant increase in the number of dynamically executed instructions (see Figure 12b). All three insert prefetch instructions; see Table 5 for the number of loads hoisted and prefetches inserted. Most of the instruction count overhead in CG is due to the added prefetches, and only a few are related to additional spill code (small increase in number of stores and loads). For lbm and mcf, load and store counts go up, by 26% and 42% for loads, and by 86% and 72% for stores, thus indicating that registers are spilled to memory. The overhead of additional instructions can, nevertheless, be hidden by overlapping the long-latency loads. Generated versions with less or no register pressure do not achieve the same benefit as the ones shown here.

Libquantum is a case in which the number of loads drops by 39%, as a consequence of our branch clustering technique. Branch clustering enables the reuse of loaded values. As an example, the loop condition depends on a variable reg → size, which is loaded and used in each iteration. Branch clustering in libquantum targets the unrolled loop branches that determine whether the next iteration is valid to be executed. It calculates all loop iteration variable values \(i, \ldots, i + \text{count}_{\text{unroll}} - 1\) and only compares the last value \((i + \text{count}_{\text{unroll}} - 1)\) against reg → size. In total, branch clustering combined with O3 enables the removal of seven out of 12 loads for each iteration for one of the targeted loops. Even though the number of loads is reduced, Clairvoyance...

\(^{4}\text{Note that in other benchmarks these branches can be successfully removed by loop unrolling – but not in all cases}\)
Figure 13. Normalized runtime w.r.t. original (-O3) for Clairvoyance-consv and Clairvoyance-specsafe (with segmentation fault handling).

Clairvoyance still introduces more instructions than the original, see Figure 12b. The additional instructions can stem from evaluating the branches at an early stage: as we target the branches in between the unrolled iterations, we may compute the loop iteration variables \( i, \ldots, i + \text{count}_{\text{unroll}} - 1 \) unnecessarily, if we only have one iteration left to execute.

II.4.6 Safe Speculation: The Overhead of A Segmentation Fault Handler

```c
// Fault caused if a == b (using spec-safe)
void seg(node_t *a, node_t *b, int n) {
    for (int i = 0; i < n - 1; i+=2) {
        b[i].next = &(b[i + 1]);
        b[i].next->next = &(b[i]);

        // Fault caused on accessing
        // a[i].next->next in order to
        // prefetch a[i].next->next->x
        a[i].x = a[i].next->next->x;
    }
}
```

Listing 1: Microbenchmark causing a segmentation fault when applying speculation (spec-safe) and if \( a == b \).

None of our evaluated benchmarks actually requires speculation, as all targeted load store pairs can be successfully disambiguated (or are known to be must aliases). Nevertheless, we evaluate the overhead of the segmentation fault handler for spec-safe. For this purpose we created a microbenchmark that contains a must-alias for the given input, see Listing 1. None of the loads
in the given microbenchmark can be fully disambiguated by the compiler. As a result, speculation will try to prefetch the address with the highest indirection (Line 10). To compute the address of that value, two other loads need to be hoisted into the Access phase (load instructions in Line 4 and 5). As these loads alias with the stores in the loop, accessing their values will cause a segmentation fault. We implemented the segmentation fault handler described in Section II.2.2 to recover from the erroneous execution.

The benchmark is not memory-bound and is thus not an actual target of Clairvoyance. The estimated overhead is a worst-case estimate, as (i) the segmentation fault will happen once for every iteration, (ii) the loop is tight and any overhead will directly reflect in the runtime, and (iii) none of the values can be reused (all actual must-aliases at runtime), thus any reordering will lead to an unnecessary overhead.

Figure 13 (right) shows the normalized runtime of the microbenchmark for consv and spec-safe. The segmentation fault is thrown directly in the first iteration. The execution is then directed to our custom segmentation fault handler, which then resumes execution at the original, unmodified loop. Both Clairvoyance-consv and Clairvoyance-spec-safe do not differ in runtime and only introduce a negligible overhead compared to the original (1%).

We also evaluate the overhead of our segmentation fault handling procedure on mcf, our most promising benchmark. Mcf does not throw a fault at runtime, as opposed to our crafted microbenchmark. Figure 13 (left) shows the overhead that our safety measure introduces: for mcf we introduce a performance degradation of 2% over the conservative version when adding the segfault handler.

This version of the segmentation fault handler favors cases, in which the speculative but safe version would cause a segmentation fault in many iterations. If the segmentation fault would only happen seldom, a more fine grain approach may give better results.

Since none of our benchmarks, except of the manually crafted microbenchmark, actually throw a segmentation fault, we have not further investigated potential improvements of this safety feature.

II.5 Related Work

Hiding long latencies of memory accesses to deliver high-performance has been a monumental task for compilers. Early approaches relied on compile-time instruction schedulers [102, 83, 103, 87, 118] to increase instruction level parallelism (ILP) and hide memory latency by performing local- or global-scheduling. Local scheduling operates within basic block boundaries and is the most commonly adopted algorithm in mainstream compilers. Global scheduling moves instructions across basic blocks and can operate on cyclic or acyclic control-flow-graphs. One of the most advanced forms of static in-
struction schedulers is modulo scheduling [71, 122], also known as software pipelining, which interleaves different iterations of a loop.

Clairvoyance tackles challenges that led static instruction schedulers to generate suboptimal code: (1) Clairvoyance identifies potential long latency loads to compensate for the lack of dynamic information; (2) Clairvoyance combines prefetching with safe-reordering of accesses to address the problem of statically unknown memory dependencies; (3) Clairvoyance performs advanced code transformations of the control-flow graph, yielding Clairvoyance applicable on general-purpose applications, which were until now not amenable to software-pipelining. We emphasize that off-the-book-shelf software pipelining is tailored for independent loop iterations and is readily applicable on statically analyzable code, but it cannot handle complex control-flow, statically unknown dependencies, etc. Furthermore, among the main limitations of software pipelining are the prologues and epilogues, and high register pressure, typically addressed with hardware support.

Clairvoyance advances the state-of-the-art by demonstrating the efficiency of these code transformations on codes that abound in indirect memory accesses, pointers, entangled dependencies, and complex, data-dependent control-flow.

Typically, instruction scheduling and register allocation are two opposing forces [81, 76, 77]. Previous work attempts to provide register pressure sensitive instruction scheduling, to balance ILP, latency, and spilling. Chen et al. [78] propose code reorganization to maximize ILP with a limited number of registers, by first applying a greedy superblock scheduler and then pushing overhoisted instructions back. Yet, such instruction schedulers consider simple code transformations and compromise on other optimizations for reducing register pressure. Clairvoyance naturally releases register pressure by precisely increasing the live-span of certain loads only, by combining instruction reordering with prefetching and by merging branches.

Hardware architectures such as Very Long Instruction Word (VLIW) and EPIC [121, 96], identify independent instructions suitable for reordering, but require significant hardware support such as predicated execution, speculative loads, verification of speculation, delayed exception handling, memory disambiguation, etc. In contrast, Clairvoyance is readily applicable on contemporary, commodity hardware. Clairvoyance decouples the loop, rather than simply reordering instructions; it generates optimized code that can reach delinquent loads, without speculation or hardware support for predicated execution and handles memory and control dependencies purely in software. Clairvoyance provides solutions that can re-enable decades of research on compiler techniques for VLIW-like and EPIC-like architectures.

Software prefetching [95] instructions, when executed timely, may transform long latencies into short latencies. Clairvoyance attempts to fully hide memory latency with independent instructions (ILP) and to cluster memory operations together and increase MLP by decoupling the loop. Software De-
coupled Access-Execute (DAE) [120, 98] targets reducing energy expenditure using DVFS, while maintaining performance, whereas Clairvoyance focuses on increasing performance. DAE generates Access-Execute phases that merely prefetch data and duplicate a significant part of the original loop (control instructions and address computation). Clairvoyance’s contribution consists in finding the right balance between code rematerialization and instruction reordering, to achieve high degrees of ILP and MLP, without the added register pressure. DAE uses heuristics to identify the loads to be prefetched, which take into consideration memory-dependencies. In addition, Clairvoyance combines information about memory- and control- dependencies, which increases the accuracy and effectiveness of the long latency loads identification. Software prefetching for indirect memory accesses [72] prefetches indirect loads; loads that are not detected by a strided prefetcher. Similarly, Clairvoyance targets loads of all indirections, but manages also to hoist loads that require complex control flow for address generation, at the expense of instruction count overhead.

Helper threads [94, 107, 119] attempt to hide memory latency by warming up the cache using a prefetching thread. Clairvoyance uses a single thread of execution, reuses values already loaded in registers (between Access and Execute phases) and resorts to prefetching only as a mechanism to safely handle unknown loop carried dependencies.

Software-hardware co-designs such as control-flow decoupling (CFD) [112] prioritize the evaluation of data-dependent branch conditions, and support a similar decoupling strategy for splitting load-use chains as our multi-access phases (however, their multi-level decoupling is done manually [111]). Contrary to Clairvoyance, CFD requires hardware support to ensure low-overhead communication between the decoupled phases. A software only version, Data-flow Decoupling (DFD), relies on prefetch instructions and ensures communication between phases by means of caches, using code duplication. As the CFD solution is not entirely automatic, Clairvoyance provides the missing compiler support and is readily applicable to decouple the CFG and hoist branch predicates, in lieu of long latency loads. Moreover, Clairvoyance provides software solutions to replace the hardware support for efficient communication between the decoupled phases. CFD makes use of decoupled producer phases for branches, but low-overhead communication is achieved with hardware support.

II.6 Conclusion

In this work, we propose a new technique to improve a processor’s performance by increasing both memory and instruction-level-parallelism and therefore the amount of useful work that is done by the core. Clairvoyance handles limitations imposed by may-alias loads, reorders dependent memory opera-
tions across loop iterations, and controls register pressure. Using these tech-
niques, we achieve performance improvements of up to 43% (14% geomean
improvement for memory-bound benchmarks) on real hardware. Clairvoy-
ance enables optimizations that move beyond standard instruction reordering
to achieve energy efficiency and overall higher performance in the presence of
long-latency loads.

Acknowledgment
This work is supported, in part, by the Swedish Research Council UPMARC
Linnaeus Centre and by the Swedish VR (grant no. 2016-05086).
1. References


York, NY, USA, 2007. ACM.


[119] Weifeng Zhang, Dean M. Tullsen, and Brad Calder. Accelerating and adapting
Paper III

Student Research Poster: Software Out-of-Order Execution for In-Order Architectures

Kim-Anh Tran

In Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT)
Haifa, Israel, September 2016
Abstract
Processor cores are divided into two categories: fast and power-hungry out-of-order processors, and efficient, but slower in-order processors. To achieve high performance with low-energy budgets, this proposal aims to deliver out-of-order processing by software (SWOOP) on in-order architectures.

**Problem:** A primary cause for slowdown in in-order processors is last-level cache misses (caused by difficult to predict data-dependent loads), resulting in cores stalling.

**Solution:** As loads are non-blocking operations, independent instructions are scheduled to run before the loads return. We execute critical load instructions earlier in the program for a three-fold benefit: increasing memory and instruction level parallelism, and hiding memory latency.

**Related work:** Some instruction scheduling policies attempt to hide memory latency, but scheduling is confined by basic block limits and register pressure. Software pipelining [122] is restricted by dependencies between instructions and decoupled access-execute (DAE) [120] suffers from address re-computation. Unlike EPIC [121] (evolved from VLIW), SWOOP does not require hardware support for predicated execution, speculative loads and their verification, delayed exception handling, memory disambiguation etc.

III.1 The SWOOP Compiler

**Contribution:** (1) *Overcoming dependent loads.* SWOOP detects dependent load chains and splits them up into multiple access phases, enabling independent instructions to be scheduled in-between. (2) *Handling of unknown memory dependencies.* May-aliasing loads are turned into safe prefetches. (3) *Balancing of latency and register pressure.* SWOOP selectively turns safe loads to prefetches to control register pressure. We transform the code using LLVM in a way similar to DAE. Figure 1 shows the transformation and execution. SWOOP (1) hoists critical loads, (2) creates multiple access phases to maximize separation, (3) turns may-aliasing loads to prefetches to overcome unknown dependencies, and (4) inserts a *ckmiss* instruction. On a cache miss in an access phase, the *ckmiss* will trigger a branch to the next itera-
tion’s access phase. We introduce a register remapping technique to ensure that execute phases consume the registers of their respective access phases.

Figure 2. Normalized speed up.

Figure 2 shows the normalized speedup of memory-bound benchmarks (SPEC 2006CPU, CIGAR, NAS) evaluated with the Sniper Simulator. We compare an in-order core (InO~Cortex-A7), the SWOOP core (InO with register remapping) and an out-of-order core (OoO~Cortex-A15), and achieve an average improvement of 34% in runtime and a reduction by 23% in energy consumption (not shown) over the InO.

III.2 Acknowledgments

I would like to thank my advisors Alexandra Jimborean and Stefanos Kaxiras for their guidance and support.
1. References


Paper IV
Paper IV

Transcending Hardware Limits with Software Out-of-order Processing

Trevor E. Carlson, Kim-Anh Tran
Alexandra Jimborean, Konstantinos Koukos
Magnus Själander and Stefanos Kaxiras

In IEEE Computer Architecture Letters (2017)
Abstract
Building high-performance, next-generation processors require novel techniques to enable improved performance given today’s power- and energy-efficiency requirements. Additionally, a widening gap between processor and memory performance makes it even more difficult to improve efficiency with conventional techniques. While out-of-order architectures attempt to hide this memory latency with dynamically reordered instructions, they lack the energy efficiency seen in in-order processors. Thus, our goal is to reorder the instruction stream to avoid stalls and improve utilization for energy efficiency and performance.

To accomplish this goal, we propose an enhanced stall-on-use in-order core that improves energy efficiency (and therefore performance in these power-limited designs) through out-of-program-order execution. During long latency loads, the SWOOP (Software Out-of-Order Processing) core exposes additional memory- and instruction-level parallelism to perform useful, non-speculative work. The resulting instruction lookahead of the SWOOP core reaches beyond the conventional fixed-sized processor structures with the help of transparent hardware register contexts. Our results show that SWOOP demonstrates a 34% performance improvement on average compared with an in-order, stall-on-use core, with an energy reduction of 23%.

IV.1 Introduction
Increasing energy efficiency while maintaining high performance is the holy grail of software and hardware design. One way to tackle this problem is to overlap multiple memory accesses (memory level parallelism) and to hide their latency with useful computation by reordering instructions (instruction level parallelism). In-order (InO) cores rely on static instruction schedulers to hide long latencies by interleaving independent instructions between a load and its use. Nevertheless, such techniques cannot adapt to dynamic factors and, in practice, are very limited.

We propose SWOOP (Software Out-of-Order Processing), a novel software/hardware co-design that exceeds conventional hardware limits to hide memory latency and to achieve improved memory level parallelism (MLP) and instruction level parallelism (ILP). While regular, predictable programs can be offloaded to accelerators and custom functional units, SWOOP attacks the difficult problem of speeding up a single thread with entangled memory and control dependencies that are not amenable to fine-grain parallelization or prefetching.

SWOOP is a hardware/software decoupled access-execute approach, built upon Access phases (containing loads and their requirements) and Execute phases (consuming data from Access and containing all computation). SWOOP relies on a compiler to generate Access-Execute phases, akin to the decoupled access-execute (DAE) model [123, 127] in which the target code region is transformed into (i) a heavily memory-bound phase (i.e., the access-phase for data prefetch), followed by (ii) a heavily compute-bound phase (i.e., the execute-phase that performs the actual computation). SWOOP operates on a much finer granularity and prefers loads over prefetches. Access
Conventional execution stalls an InO processor when using the results of a long latency memory access. DAE clusters cache accesses and prefetches them ahead of time to reduce core stall time. SWOOP, on the other hand, reorders software across iterations to access distant, critical instructions, hiding load latencies with useful work.

- *Execute* phases are orchestrated in software guided by runtime information and executed in a single superscalar pipeline, within a single thread of control (section 1).

SWOOP interleaves *Access* and *Execute* code within a single thread, changing this interleaving dynamically. Thus, SWOOP abstracts the execution order from the underlying architecture and can run *Access* and *Execute* code either in-program-order or out-of-program-order.

While SWOOP orchestrates the out-of-order execution of *Access* and *Execute* code, the targeted enhancements of the microarchitecture are essential for efficiency. Specifically, the SWOOP architecture provides:

- A novel register remapping approach (a lightweight form of register renaming) that: i) Enables unrestricted dynamic separation of an *Access* and its corresponding *Execute*, with a number of *Access* phases from future iterations. Context Remapping ensures that registers written in each *Access* phase will be remapped and encapsulated in a unit denoted as *Context*. ii) Manages dependencies between *Contexts* (subsection IV.3.1).

- A simple yet efficient check-miss mechanism (chkmiss) to quickly inform the core about long-latency loads in *Access*, akin to informing memory operations [129], to adapt and prevent memory stalls (subsection IV.3.2).

SWOOP outperforms InO cores significantly and approaches, for memory-bound codes, the performance of OoO engines.
IV.2 SWOOP Software Components

The SWOOP core is a hardware-software co-designed processor that benefits from software knowledge to enable the hardware to execute past the conventional dynamic instruction stream. The compiler transforms loops to safely and efficiently cluster memory accesses into Access regions, similar to SW-DAE [123, 127], which has been successful in decoupling a large number of complex, general-purpose applications. Unlike SW-DAE, loads without read-after-write dependencies are hoisted to Access, and only memory dependent loads are replaced by safe-prefetches. Together with hardware enhancements, SWOOP executes useful, non-speculative instructions to improve energy efficiency and performance of memory-bound applications.

We list in what follows the main steps in transforming the code for the SWOOP hardware. First, the critical loops need to be identified as candidates for modification. This can be done by means of user inserted pragmas, running a profiling step up-front, or a runtime step for JIT-enabled environments. Next, the loops must be split into Access and Execute phases by hoisting address computation and loads into the Access phase.

The border between Access and Execute is marked by a chkmiss, which guides the execution flow. Chkmiss indicates whether any of the loads in Access incurred a miss in the last level cache, potentially yielding a stall in Execute. Upon a miss, execution surrenders control flow to the alternative execution path, which consists of the Access phases of the following \( N \) iterations and the corresponding \( N + 1 \) Execute phases (section 2). \( N \) is determined by the number of physical registers provided by the microarchitecture and the number of registers required by an Access phase.
IV.3 SWOOP Architectural Support

We add three hardware features to the conventional stall-on-use in-order processor to enable the software and hardware to work together more efficiently. Context-based register remapping allows for hardware variety (physical registers) to exploit long latency loads with a single SWOOP software version (subsection IV.3.1). In addition, the check-miss mechanism (chkmiss) is used to dynamically enable context register remapping in the presence of long latency loads (subsection IV.3.2), and the early commit of loads (subsection IV.3.3) reduces stalls by committing loads early, even before results have returned.

IV.3.1 Context Register Remapping

For the majority of cases, the aim is to use register-file *Access-Execute* communication to maintain much of the performance of the original optimized code.

We propose a novel register remapping technique, based on *execution contexts*, that alleviates the burden for additional register allocation and exposes additional physical registers to the software. The key observation for an efficient remapping scheme is that we only need to handle the case when we intermix *Access* and *Execute* phases belonging to different SWOOP contexts. A SWOOP context comprises an *Access* and its corresponding *Execute*, which share a single program-order view of the architectural register state (each pair of *Access-Execute* during normal execution in section 2).

SWOOP remaps only in the alternative execution path (out-of-program-order execution). Each architectural register is remapped only once when it is first written in an *Access* phase, while no additional remapping is done in *Execute*. No additional register remapping is needed within an *Access* or an *Execute* or between an *Access* and *Execute* belonging to the same SWOOP context resulting in a significant lower rate of remapping compared to conventional OoO cores.

We give full flexibility to the software via two new directives that set the current active context (CTX): CTX=0 and CTX++. The program-order (non-SWOOP execution) context is CTX 0. No remapping takes place during non-SWOOP-related execution.

**Implementing SWOOP Contexts**

The register remapping is implemented for each architectural register through: (1) a context remapping vector, which contains as many bits as the number of supported contexts and (2) a context remapping FIFO, which holds mappings from the architectural register to physical registers.

**Non-SWOOP execution** does not require any remapping since the code is executed in its original program order.
**SWOOP execution** is governed by three simple rules: (1) For each new *Access* phase the context is incremented (CTX++) and the corresponding bit of the context remapping vector is set to zero. Upon the first write to an architectural register, the corresponding bit gets set and a new physical register name is pushed to the head of the FIFO. Future writes to the same architected register in this phase do not generate new mappings. The new physical register (i.e., the head) becomes the new effective architectural to physical map and is used until it gets remapped in a future *Access* or *Execute* phase. (2) When returning to the first *Execute* phase (E₀) the original mapping of A₀ is to be used. This is equivalent with the oldest register in the remapping FIFO (the element that would get popped). Upon returning to CTX=0, the effective map is changed from the youngest (i.e., the head) to the oldest (i.e., the tail) physical register in the FIFO. (3) For each new *Execute* phase where the corresponding bit in the remapping vector is set a physical register is popped from the FIFO and the new register at the tail is the new effective map.

### IV.3.2 Chkmiss

Chkmiss provides for a timely control flow change to the alternative execution path for upcoming stalling code. We encode the presence of an LLC cache line in the TLB entries, using a simple bitmap (e.g., 64 bits for 64-byte cache lines in a 4kB page). Chkmiss monitors the set of *any* loads missed that were hoisted to *Access*. The border between *Access* and *Execute* can be marked, e.g., by an x86 instruction prefix (used in this implementation), a break point, or a full instruction, which guides the execution flow.

### IV.3.3 Early Commit of Loads

SWOOP out-of-program-order execution of *Access* phases can cause a vast number of instructions to be executed. A delinquent load would reside at the head of a commit buffer causing a SWOOP core to stall once the buffer becomes full. To avoid stalling, we employ early commits of loads (ECLs) [131] to commit loads that cannot cause an exception.

### IV.4 Evaluation

We use the Sniper Multi-Core Simulator to evaluate this work. We modify the cycle-level core model to support the SWOOP processor. Power and energy estimates are calculated with McPAT version 1.3 in 28 nm. The processor parameters are shown in subsection 1. SWOOP is evaluated on *demanding* workloads with frequent misses even when employing hardware prefetcher, from the SPEC2006CPU, CIGAR, and NAS benchmark suites.
### Table 1. Microarchitecture. OoO has (*) 2 int, 1 int/br., 1 mul, 2 fp, and 2 ld/st.

<table>
<thead>
<tr>
<th>Core</th>
<th>In-Order</th>
<th>Runahead</th>
<th>SWOOP</th>
<th>OoO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u-Arch</td>
<td>1.5 GHz, 2-way superscalar</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>RS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>[16/32]</td>
</tr>
<tr>
<td>Phys. Reg</td>
<td>32/32</td>
<td>32/32</td>
<td>96/64</td>
<td>64/64</td>
</tr>
<tr>
<td>Br. Penalty</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>Exec. Units</td>
<td>1 int, 1 int/br., 1 mul, 1 fp, 1 ld/st</td>
<td>*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| L1-I     | 32 KB, 8-way LRU |          |       |      |
| L1-D     | 32 KB, 8-way LRU, 4 cycle, 8 MSHRs |          |       |      |
| L2 cache | 256 KB, 8-way LRU, 8 cycle |          |       |      |
| L3 cache | 4 MB, 16-way LRU, 30 cycle |          |       |      |
| DRAM     | 7.6 GB/s, 45 ns access latency |          |       |      |
| Prefetcher | stride-based, L2, 16 streams |          |       |      |
| Technology | 28 nm |          |       |      |

We compare a number of potential solutions in performance and energy efficiency. InO is a two-wide in-order, stall-on-use core. InO-Unroll and InO-Perf-L3 are extensions to the in-order core with software unrolling and a perfect L3 cache. SW-prefetching [128] and in-order Runahead [126] were also evaluated. Finally a 2-wide moderately-aggressive out-of-order core (OoO) with and without software unrolling is compared to our (SWOOP) implementation.

### IV.4.1 Performance

Subsection 3 shows speedups normalized to the in-order, stall-on-use core (InO). SWOOP achieves significant performance improvements (34% on average) compared to InO and outperforms the base OoO when running cigar (12% faster). The results clearly show that forcing the loops to be unrolled on an InO is not beneficial and instead hurts performance (3% slower on average) due to register spilling, motivating the necessity of the SWOOP approach. Runahead is only showing a speedup over InO for lbm and mcf. One of the main reasons for this is the limited reach of runahead execution compared to SWOOP. On average, InO runahead is able to execute an additional 21 instructions per runahead phase (with six containing load operations). As misses can be sparse, maximizing the instruction distance travelled is necessary to expose additional long latency loads. SWOOP brings Access phases together, reducing the amount of work needed to discover long latency loads, improv-

---

1 Received benchmarks from authors, those in common evaluated.
ing performance for more applications than is possible with runahead. For sphinx3, SWOOP achieves speedup over the InO, but not nearly the speedup of the OoO. While there is a large number of delinquent loads in the application, each contributes a very small portion to the total application delay, limiting total improvement. For libquantum, SWOOP achieves significantly better performance than InO and reaches half the speedup of OoO. Libquantum contains a very tight loop, hence any instruction overhead has a non-negligible impact. Some of the optimization opportunities are hidden by the OoO core, or, OoO can reach sufficiently far to cover the existing latency. Finally, for soplex, SWOOP fails to achieve any speedup over the InO but does not significantly hurt performance (0.6% slower). Soplex suffers from the same problem as sphinx3: a single delinquent load in the Access phase (responsible for only 7% of the memory slack), exacerbated by a lower chkmiss firing rate of 18%.

IV.4.2 Energy

subsection 4 shows the energy usage normalized to the InO core, estimated with McPAT. SWOOP reduces the energy usage by 23% on average and is the only technique that shows any significant improvements compared to the InO core. The OoO core, which has the best average speedup, increases the energy usage by 60% (47% for software unrolling). SWOOP improves EDP by 41% over the in-order baseline on average, while the out-of-order core and the unrolled version improves EDP by 43% and 51% respectively. While energy increases by 60% on average, power consumption is 3x higher for OoO compared to InO. Runahead increases energy expenditure by 5% compared to InO due to significant re-execution of instructions, which limits performance improvements and wastes energy.

IV.4.3 SWOOP, Unrolling, and SW Prefetching

We compare SWOOP to two software-only techniques, forced unrolling and software prefetching across the four benchmarks (libquantum, soplex, mcf and
lbm) that overlap between this and software prefetching work [128]. In many cases, software prefetching and unrolling fail to show a benefit on the baseline InO core. For libquantum, software unrolling provides a 5% performance improvement and unrolling a 12% improvement. SWOOP, on the other hand, improves performance by 86%. In addition, lbm sees a 7% improvement with software prefetching, while SWOOP achieves almost a 16% improvement. SWOOP is on par with the software techniques when running soplex, which has only one delinquent load with a low impact on the accumulated memory latency. SWOOP performs better than software-only solutions as it reuses address computation instructions for early loading and prefetching of data.

IV.5 Related Work

Standard techniques to hide memory latency rely on sophisticated static instruction schedulers, which by definition are inflexible and do not adapt to runtime factors. SWOOP overcomes this by means of the chkmiss instruction. EPIC adds runtime adaptivity, but significantly increases hardware complexity and energy expenditure through expensive speculative, software-based recovery mechanisms, and instruction bundles to expose parallelism. In contrast, SWOOP merely uses a simple and more efficient context register remapping technique, which is more general than register rotation. The SWOOP compiler does not require hardware support for predicated execution, speculative loads, verification of speculation, delayed exception handling, memory disambiguation, etc. and is not restricted by instruction bundles to schedule independent code. Consequently, SWOOP moves only complexity of instruction scheduling for improved MLP (Access/Execute phases) to the compiler, freeing the CPU from unnecessary complexity. Compared to Itanium (EPIC), SWOOP requires few modifications to the target ISA, uses contemporary InO pipelines and front-end, and does not introduce additional speculation.

Basic Block Execution (BBE) hardware [125] identifies blocks of instructions to execute in parallel and builds dependence graphs in hardware for data
sharing. Speculation can be used to enable more parallelism, reducing efficiency. In contrast, SWOOP proposes a disciplined execution with software-controlled block splitting to allow for multiple Access phases to execute out-of-order with low hardware overhead and without speculation. Since BBE allows for a more general splitting of a program into basic blocks, the hardware is also much more complex to ensure correctness (handling dependences between blocks, speculation, etc). Furthermore, SWOOP does not pre-define the scheduling of Access and Execute phases, but uses dynamic information to orchestrate the execution. SW-DAE [123, 127] generates Access and Execute phases, but, unlike SWOOP, Access prefetches data and, therefore, introduces significant instruction count overhead. SW-DAE is suitable for OoO engines, but code rematerialization becomes critical on InO machines. SWOOP on the other-hand prioritizes instruction reordering rather than rematerialization and triggers the OoO execution only upon a miss. Control-Flow Decoupling (CFD) [130] clusters long latency loads used in branches, similar to Access, and communicates their values through architectural queues to the consumer phase, i.e., Execute. The number of target loads is considerably less for CFD compared to SWOOP. Dundas et al. [126] employ runahead execution to prefetch data, but requires instruction re-execution and Ozer et al. [124] uses a speculative thread, requiring recovery. Finally, Multiscalar processors [125] speculate aggressively on data dependencies and replicate execution units. SWOOP uses compile-time analysis to avoid both additional speculation and instruction re-execution to reduce additional overheads.

IV.6 Conclusion

SWOOP is a new technique to achieve out-of-program-order execution and to reach high degrees of memory and instruction level parallelism. SWOOP is a hardware-software approach, with low-overhead additions to the typical inorder architecture consisting of: miss events communicated through a control-flow instruction and novel context register remapping to ease register pressure. Following the software decoupled access-execute model, the SWOOP compiler creates Access phases that can run out-of-program-order with respect to Execute phases, without any need for speculation. A SWOOP core jumps ahead to independent regions of code to hide hardware stalls and resumes execution of bypassed instructions once they are ready. The SWOOP core shows 34% performance improvement while reducing energy usage by 23% on average with respect to an InO core.
1. References

[127] K. Konstantinos et al. Multiversioned decoupled access-execute: the key to energy-efficient compilation of general-purpose programs. In *CC’16*.
Recent licentiate theses from the Department of Information Technology


2017-002 Germán Ceballos: Modeling the Interactions Between Tasks and the Memory System

2017-001 Diana Yamalova: Hybrid Observers for Systems with Intrinsic Pulse-Modulated Feedback

2016-012 Peter Backeman: New Techniques for Handling Quantifiers in Boolean and First-Order Logic

2016-011 Andreas Svensson: Learning Probabilistic Models of Dynamical Phenomena Using Particle Filters

2016-010 Aleksandar Zeljić: Approximations and Abstractions for Reasoning about Machine Arithmetic

2016-009 Timofey Mukha: Inflow Generation for Scale-Resolving Simulations of Turbulent Boundary Layers

2016-008 Simon Sticko: Towards Higher Order Immersed Finite Elements for the Wave Equation

2016-007 Volkan Cambazoglou: Protocol, Mobility and Adversary Models for the Verification of Security

2016-006 Anton Axelsson: Context: The Abstract Term for the Concrete

2016-005 Ida Bodin: Cognitive Work Analysis in Practice: Adaptation to Project Scope and Industrial Context

2016-004 Kasun Hewage: Towards a Secure Synchronous Communication Architecture for Low-power Wireless Networks