This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2769641, IEEE Transactions on Computers

Static Instruction Scheduling for High Performance on Limited Hardware

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

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 geomean execution time improvement of 14% for memory-bound applications, on top of standard O3 optimizations, while maintaining compute-bound applications' high-performance.

Index Terms—Compilers, code generation, memory management, optimization

1 Introduction

Computer architects of the past have steadily improved performance at the cost of radically increased design complexity and wasteful energy consumption [1], [2], [3]. 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 [4].

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 [5] and the AMD A1100 Series processors [6]. 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 [7]. 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 [8], [9]) 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**

- *K. Tran, V. Spiliopoulos, S. Kaxiras and A. Jimborean are with Uppsala University*
  E-mail: first.lastname@it.uu.se
- *T. E. Carlson is with the National University of Singapore*
  E-mail: tcarlson@comp.nus.edu.sg
- *K. Koukos is with the KTH Royal Institute of Technology*
  E-mail: koukos@kth.se
- *M. Själander is with the Norwegian University of Science and Technology and Uppsala University*
  E-mail: magnus.sjalander@ntnu.no

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2769641, IEEE Transactions on Computers

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 [10] 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 performance gap between the conservative version and Clairvoyance's best speculative versions (Section 2.6, Section 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 2.2.1, Section 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 2.5.1).
4. **Comparing against State-of-the-Art Prefetching:** We extend the evaluation to include the comparison to state-of-the-art prefetching techniques (Section 4).
5. **Understanding the Performance:** We provide new static and dynamic statistics that provide new insights into the performance gains achieved by Clairvoyance (Section 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.

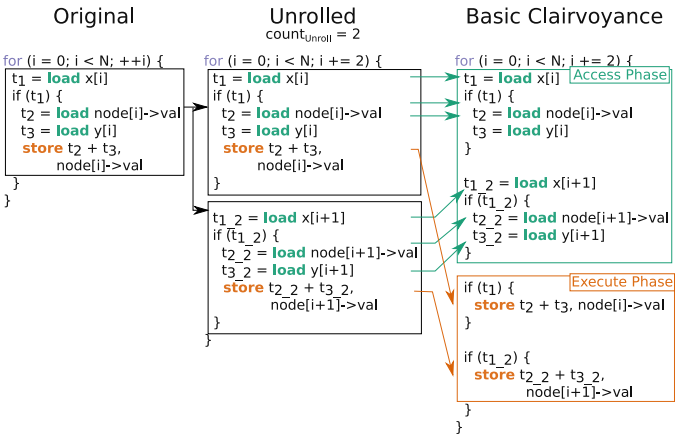


Fig. 1: The basic Clairvoyance transformation. The original loop is first unrolled by $\text{count}_{\text{unroll}}$ which increases the number of instructions per loop iteration. Then, for each iteration, Clairvoyance hoists all (critical) loads and sinks their uses to create a memory-bound Access phase and a compute-bound Execute phase.

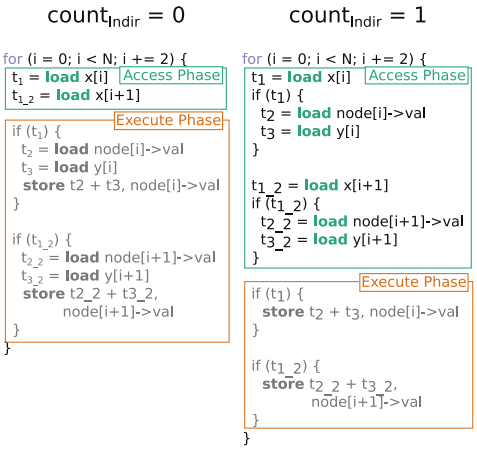


Fig. 2: Selection of loads based on an indirection count $\text{count}_{\text{indir}}$. The Clairvoyance code for $\text{count}_{\text{indir}} = 0$ (left) and $\text{count}_{\text{indir}} = 1$ (right).

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 [9], [11], program slicing [12], and decoupled access-execute [13], [14], [15] 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:

1. For simplicity we use examples with for-loop structures, but Clairvoyance is readily available for `while`, `do-while` and `goto` loops.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2769641, IEEE Transactions on Computers

- **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[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 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).

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

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.

2. We do not maintain precise exception semantics, as we reorder memory instructions that may throw an exception.

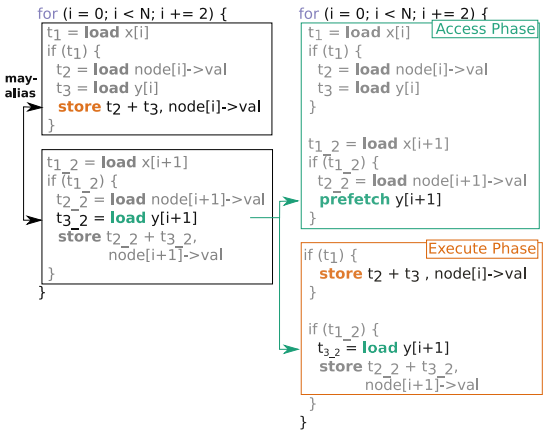


Fig. 3: Handling of may-aliasing loads. Loads that may alias with any preceding store operation are not safe to hoist. Instead, we prefetch the unsafe load.

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) [15] 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 [16] or VMAD [17], [18].

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2769641, IEEE Transactions on Computers

Name	Description
Consv	Conservative, only hoists safe loads
Spec-safe	Speculative (but safe), hoists may-aliasing load chains, but safely reloads them in *Execute*
Spec	Speculative (unsafe), hoists may-aliasing load chains and reuses all data in *Execute*
Multi-spec-safe	Multi-access version of spec-safe
Multi-spec	Multi-access version of spec

TABLE 1: Clairvoyance evaluated versions.

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 [19]. 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 3.

2.2.1 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*.

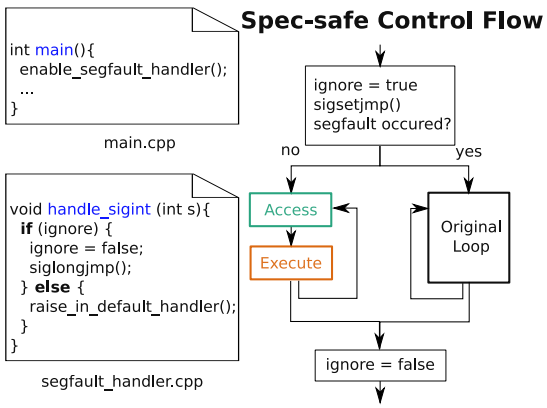


Fig. 4: Custom segmentation fault handler to safely execute spec-safe. The handler is deployed at program start (left, main.cpp). On a segmentation fault, the handler will either (1) raise the fault in the default handler, or (2) re-execute the original loop (left, segfault_handler.cpp, jump to sigsetjmp). The decision depends on a flag (*ignore*).

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 [20] 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 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2769641, IEEE Transactions on Computers

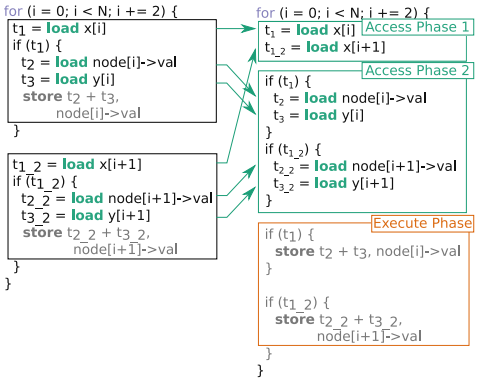


Fig. 5: Splitting up dependent load chains. Clairvoyance creates one Access phase for each set of independent loads (and their requirements), which increases the distance between loads and their uses.

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 segmentation fault occurs, the custom handler will ignore the fault, set the flag to *false*, and jump to the loop preheader (using *siglongjmp*). The loop preheader will then, on evaluation of *sigsetjmp*, restore the saved environment. Since the flag now evaluates to *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 *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 4.6.

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 *Access*. If it did, the processor might stall, because the second load represents a *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.
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 $\text{count}_{\text{unroll}} = 2$, hence there are only two independent loads to collect into the first *Access* phase and four into the second *Access* phase.

The 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`.

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

```
   Input: Set of loads
   Output: List of sets phase_loads
 1 begin
 2     remaining_loads ← loads
 3     phase_loads ← []
 4     while remaining_loads ≠ ∅ do
 5         phase ← ∅
 6         for ld in remaining_loads do
 7             reqs ← ∅
 8             FindCFGRequirements (ld, reqs)
 9             FindDataRequirements (ld, reqs)
10             is_independent ← Intersection(reqs,
               remaining_loads) == ∅
11             if is_independent then
12                 phase ← phase + ld
13             end
14         end
15         remaining_loads ← remaining_loads \ phase
16         phase_loads ← phase_loads + phase
17     end
18     return phase_loads
19 end
```

Algorithm 2: Separating loads for multiple Access phases.

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

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 *sigsetjmp / siglongjmp* calls are thread-safe [21]. Note though that the implementation of these calls may vary for each operating system.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2769641, IEEE Transactions on Computers

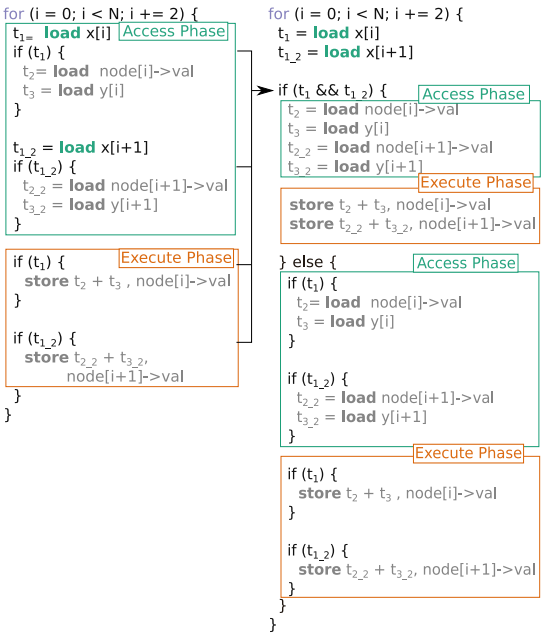


Fig. 6: Early evaluation of branches enables the elimination of duplicated branches. Relevant branches are evaluated at the beginning of the loop. If the evaluated branches are taken, the optimized Clairvoyance code with merged basic blocks is executed; otherwise, the decoupled unrolled code (with branch duplication) is executed.

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 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 4 for more details.

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.

2.5.1 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 2. Within a set, all loads are independent. A load in a set, however, depends on one or more loads of the previous set. Algorithm 3 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 reordering) (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).

2.6 Integrating State-of-the-Art Alias Analysis

We integrate the static value-flow analysis SVF [20] 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2769641, IEEE Transactions on Computers

Input: List of sets $load\_sets$, Maximum number of registers max_regs
Output: Set of $\boldsymbol{to\_reuse}$ and $\boldsymbol{to\_prefetch}$

```
1  begin
2      to_reuse ← ∅
3      to_prefetch ← ∅
4      iter ← GetIterator(load_sets)
5      while HasNext(iter) and size_of(to_reuse) <
       max_regs do
6          set ← Next(iter)
7          for ld in set do
8              if size_of(to_reuse) < max_regs then
9                  to_reuse ← to_reuse + ld
10             else
11                 to_prefetch ← to_prefetch + ld
12             end
13         end
14     end
15     if HasNext(iter) then
16         set ← Next(iter)
17         for ld in set do
18             if GetRequiredLoads(ld) ⊆ to_reuse then
19                 to_prefetch ← to_prefetch + ld
20             end
21         end
22     end
23 end
```

Algorithm 3: Heuristic to decide whether to reorder or prefetch loads.

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: $\frac{loads}{branches} < 0.7$, and disable Clairvoyance transformations if the condition is met.

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.

2.9 Limitations

2.9.1 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.

Processor	APM X-Gene - AArch64 Octa-A57
Core Count	8
ROB size	128 micro-ops [23]
Issue Width	8 [23]
L1 D-Cache	32 KB / 5-6 cycles depending on access complexity
L2 Cache	256 KB / 13 cycles Latency
L3 Cache	8 MB / 90 cycles Latency
RAM	32 GB / 89 cycles + 83 ns (for random RAM page)

TABLE 2: Architectural specifications of the APM X-Gene.

2.9.2 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.

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 [22]. 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.

3 Experimental Setup

Our transformation is implemented as a separate compilation pass in LLVM 4.0 [24]. We evaluate a range of C/C++ benchmarks from the SPEC CPU2006 [25] and NAS benchmark [26], [27], [28] suites on an APM X-Gene processor [29], 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. For SPEC, the selection

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2769641, IEEE Transactions on Computers

Benchmark	Function
401.bzip2	BZ2_compressBlock
403.gcc	reg_is_remote_constant_p
429.mcf	primal_bea_mpp
433.milc	mult_su3_na
444.namd	calc_pair_energy_fullelect calc_pair_energy calc_pair_energy_merge_fullelect calc_pair_fullelect
445.gobmk	dfa_matchpat_loop incremental_order_moves
450.soplex	entered
456.hmmer	P7Viterbi
458.sjeng	std_eval
462.libquantum	quantum_toffoli quantum_sigma_x quantum_cnot
464.h264ref	SetupFastFullPelSearch
470.lbm	LBM_performStreamCollide
471.omnetpp	shiftup
473.astar	makebound2
482.sphinx3	mgau_eval
CG	conj_grad
LU	buts
UA	diffusion

TABLE 3: Modified functions.

was made based on previous studies [30], while for NAS we identified the target functions using Valgrind [31].

In Section 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%.

3.1 Evaluating LLVM, DAE, SW-Prefetching and Clairvoyance

We compare our techniques to Software Decoupled Access-Execute (DAE) [14], [15], Software Prefetching for Indirect Memory Accesses [32] (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 [33]. 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.

CLAIRVOYANCE-DAE A hybrid of Clairvoyance and DAE: transformations are performed as for regular Clairvoyance, but always prefetch the last indirection.

CLAIRVOYANCE Best performing Clairvoyance version.

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.

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 `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*). 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, `Hmmer` 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*. 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*.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2769641, IEEE Transactions on Computers

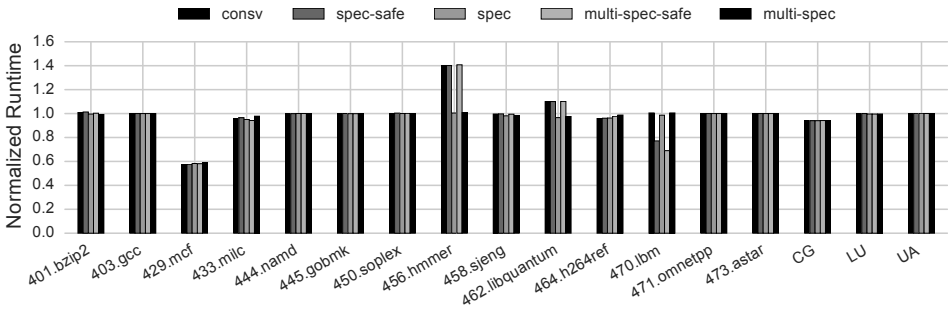


Fig. 7: Normalized total runtime w.r.t original execution (-O3), for all Clairvoyance versions.

Benchmark	Version	Unroll	Indir
429.mcf	consv	8	0
433.milc	multi-spec-safe	2	0
450.soplex	spec	2	0
462.libquantum	spec	4	1
470.lbm	multi-spec-safe	16	1
471.omnetpp	Disabled		
473.astar	Disabled		
CG	spec	4	1

TABLE 4: Best performing versions for memory-bound benchmarks [34].

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`) [34]. 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.

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 designed 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 geomean 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 `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 branches, and therefore avoids degrading performance. For the compute-bound applications, both Clairvoyance-consv and -best preserve the O3 performance, on-par with the standard LLVM instruction schedulers, except for `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 `h264ref`.

Clairvoyance-DAE shows that `lbm` and `CG` benefit from the prefetching-only scheme, which can unroll more iterations since no registers are blocked due to reordering. In contrast, `mcf` and `libquantum` profit from Clairvoyance optimizations. While DAE introduces significant overhead for `libquantum` due to the duplicated instructions,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2769641, IEEE Transactions on Computers

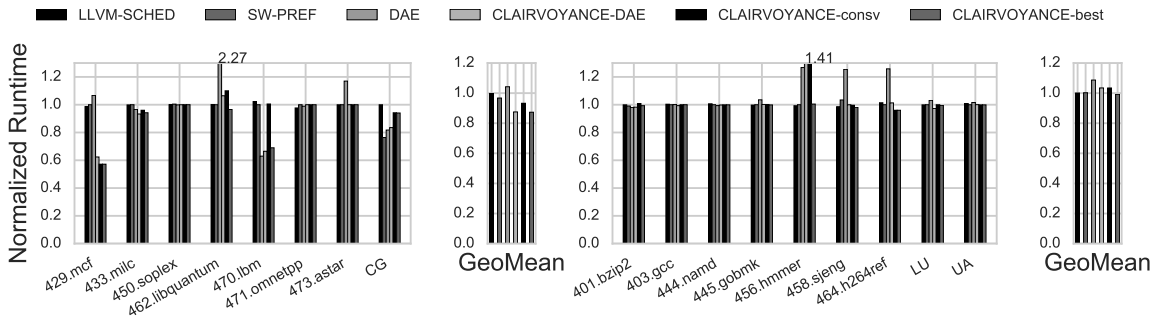


Fig. 8: Normalized total runtime w.r.t original execution (-O3) for the best version of LLVM schedulers, SW-PREF, DAE, Clairvoyance-DAE, and the conservative and the best version of Clairvoyance, categorized into memory-bound (left) and compute-bound (right) benchmarks.

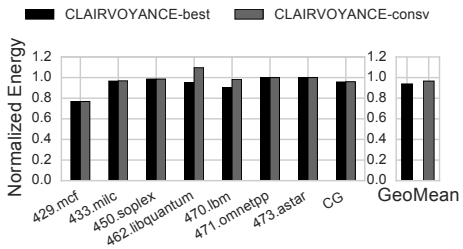


Fig. 9: Normalized energy across all memory-bound benchmarks for Clairvoyance-consv and Clairvoyance-best.

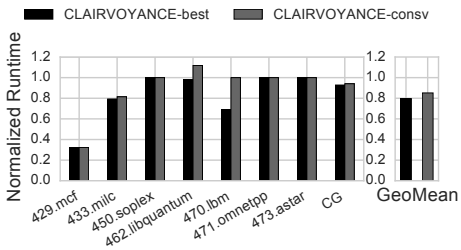


Fig. 10: Normalized runtime per target loop w.r.t original loop execution (-O3) across all memory-bound benchmarks for the Clairvoyance-consv and Clairvoyance-best.

Clairvoyance-DAE profits from branch clustering which reduces the instruction count overhead significantly. For `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 benchmarks, combining them may enable even higher gains.

SW-PREF (with a look-ahead distance of 64) inserted prefetches for `CG`, `bzip2`, `sjeng`, `soplex`, and `namd`. For most of the transformed benchmarks SW-PREF does not benefit, neither harm performance, except 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 (*mcf*), others were not inserted because there were no indirect loads to target (*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. [35]. 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 geomean improvement of 5%. By overlapping outstanding loads we increase MLP, which in turn results in shorter runtimes and thus lower total energy consumption.

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 2.6) and (*ii*) a new heuristic (Section 2.5.1) 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 conservative 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-consv versions and the number of loads and prefetches in *Access* (prefetching happens as a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2769641, IEEE Transactions on Computers

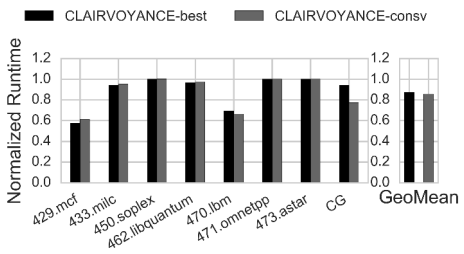


Fig. 11: Normalized runtime w.r.t. original execution (-O3) for Clairvoyance-consv (with better alias analysis and heuristic to choose between hoisting and prefetching) and the previous Clairvoyance-best.

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

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.

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-consv* 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 loads. Nevertheless, the updated Clairvoyance-consv version now reaches a geomean improvement of 14% compared to the previous 13% including speculation.

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).

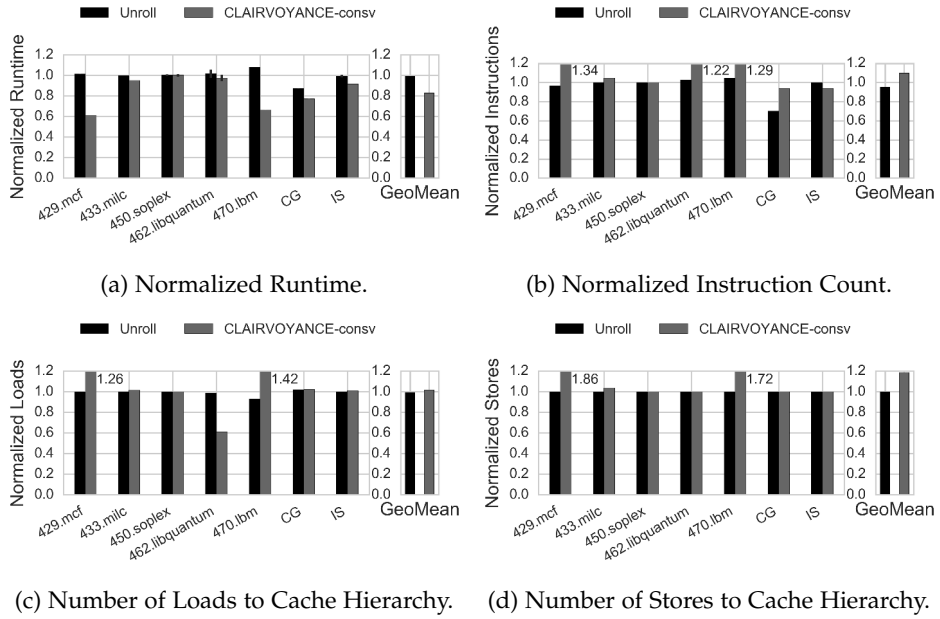
For the analysis we gather runtime, the number of dynamic instruction, and load and store operations to caches using hardware performance counters (perf) and are shown in Figure 12. The instruction count gives an insight into the *instruction count overhead* that Clairvoyance introduces, partly due to 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 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 $\rightarrow$ 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[4]. 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 $\rightarrow$ 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 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.

4. Note that in other benchmarks these branches can be successfully removed by loop unrolling – but not in all cases

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2769641, IEEE Transactions on Computers



(a) Normalized Runtime.

(b) Normalized Instruction Count.

(c) Number of Loads to Cache Hierarchy.

(d) Number of Stores to Cache Hierarchy.

Fig. 12: Normalized Dynamic Runtime, Instruction Count, Load and Store Count to the unmodified O3-version for the best Clairvoyance version and the Unrolled version with the same unroll count as Clairvoyance.

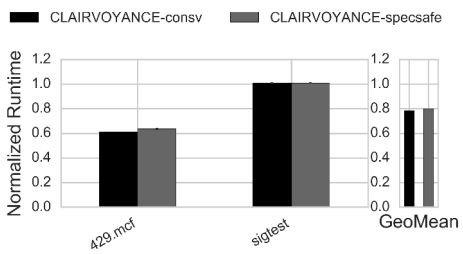


Fig. 13: Normalized runtime w.r.t. original (-O3) for Clairvoyance-consv and Clairvoyance-specsafe (with segmentation fault handling).

4.6 Safe Speculation: The Overhead of A Segmentation Fault Handler

```
1  // Fault caused if a == b (using spec-safe)
2  void seg(node_t *a, node_t *b, int n) {
3    for (int i = 0; i < n - 1; i+=2) {
4      b[i].next = &(b[i + 1]);
5      b[i].next->next = &(b[i]);
6
7      // Fault caused on accessing
8      // a[i].next->next in order to
9      // prefetch a[i].next->next->x
10     a[i].x = a[i].next->next->x;
11   }
12 }
```

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 2.2.1 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2769641, IEEE Transactions on Computers

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.

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 [36], [37], [38], [39], [40] 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 instruction schedulers is modulo scheduling [8], [9], 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 [41], [42], [43]. Previous work attempts to provide register pressure sensitive instruction scheduling, to balance ILP, latency, and spilling. Chen et al. [44] 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* [45], [46], 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 [47] 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) [14], [15] 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 [32] 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 [48], [49], [50] 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) [51] 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 [52]). 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.

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 operations across loop iterations, and controls register pressure. Using these techniques, we achieve performance improvements of up to 43% (14% geomean improvement for memory-bound benchmarks) on real hardware. Clairvoyance enables optimizations that move beyond standard instruction reordering to achieve energy efficiency and overall higher performance in the presence of long-latency loads.

Acknowledgments

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

References

[1] J.-L. Baer and T.-F. Chen, "Effective hardware-based data prefetching for high-performance processors," *Transactions on Computers*, vol. 44, no. 5, pp. 609–623, 1995.

[2] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-block ahead branch predictors," *Operating Systems Review*, vol. 30, no. 5, pp. 116–127, 1996.

[3] N. Prémillieu and A. Seznec, "Efficient out-of-order execution of guarded ISAs," *Transactions on Architecture and Code Optimization*, vol. 11, no. 4, pp. 41:1–41:21, 2014.

[4] M. Själander, M. Martonosi, and S. Kaxiras, *Power-Efficient Computer Architectures: Recent Advances*. Synthesis Lectures on Computer Architecture, 2014.

[5] H. P. Enterprise, "HPE ProLiant m400 server cartridge." Online http://www8.hp.com/us/en/products/proliant-servers/product-detail.html?oid=7398907; accessed, 2016.

[6] AMD, "AMD Opteron A-series processors." Online http://www.amd.com/en-us/products/server/opteron-a-series; accessed, 2016.

[7] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proceedings of the Annual International Symposium Computer Architecture*, pp. 76–87, 2004.

[8] A. Aiken, A. Nicolau, and S. Novack, "Resource-constrained software pipelining," in *Transactions on Parallel and Distributed Systems*, pp. 274–290, 1995.

[9] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 318–328, 1988.

[10] K. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Clairvoyance: look-ahead compile-time scheduling," in *Proceedings of the International Symposium on Code Generation and Optimization* (V. J. Reddi, A. Smith, and L. Tang, eds.), pp. 171–184, ACM, 2017.

[11] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T. han Hung, and D. I. August, "Decoupled software pipelining creates parallelization opportunities," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 121–130, 2010.

[12] M. Weiser, "Program slicing," in *Proceedings of the International Conference on Software Engineering*, pp. 439–449, 1981.

[13] K. Koukos, D. Black-Schaffer, V. Spiliopoulos, and S. Kaxiras, "Towards more efficient execution: A decoupled access-execute approach," in *Proceedings of the International Conference on Supercomputing*, pp. 253–262, 2013.

[14] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras, "Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 262–272, 2014.

[15] K. Koukos, P. Ekemark, G. Zacharopoulos, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Multiversioned decoupled access-execute: The key to energy-efficient compilation of general-purpose programs," in *Proceedings of the International Conference on Compiler Construction*, pp. 121–131, 2016.

[16] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars, "Protean code: Achieving near-free online code transformations for warehouse scale computers," in *Proceedings of the Annual International Symposium on Microarchitecture*, pp. 558–570, 2014.

[17] A. Jimborean, M. Herrmann, V. Loechner, and P. Clauss, "VMAD: A virtual machine for advanced dynamic analysis of programs," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pp. 125–126, 2011.

[18] A. Jimborean, L. Mastrangelo, V. Loechner, and P. Clauss, "VMAD: an advanced dynamic program analysis and instrumentation framework," in *Proceedings of the International Conference on Compiler Construction*, pp. 220–239, 2012.

[19] B. Hackett and A. Aiken, "How is aliasing used in systems software?," in *Proceedings of the symposium on the Foundations of Software Engineering*, pp. 69–80, 2006.

[20] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the International Conference on Compiler Construction* (A. Zaks and M. V. Hermenegildo, eds.), pp. 265–266, ACM, 2016.

[21] *Zesty sigsetjmp(3) User's Manual*. http://manpages.ubuntu.com/manpages/zesty/en/man3/sigsetjmp.3.html, accessed 29-September-2017.

[22] A. Jimborean, J. Waern, P. Ekemark, S. Kaxiras, and A. Ros, "Automatic detection of extended data-race-free regions," in *Proceedings of the International Symposium on Code Generation and Optimization* (V. J. Reddi, A. Smith, and L. Tang, eds.), pp. 14–26, ACM, 2017.

[23] Anandtech, "ARM A53/A57/T760 investigated - Samsung Galaxy Note 4 exynos review." Online http://www.anandtech.com/show/8718/the-samsung-galaxy-note-4-exynos-review/5; accessed, 2016.

[24] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 75–88, 2004.

[25] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[26] NASA, "NAS parallel benchmarks." Online https://www.nas.nasa.gov/assets/pdf/techreports/1999/nas-99-011.pdf; accessed 07-September-2016, 1999.

[27] S. Seo, G. Jo, and J. Lee, "Performance characterization of the NAS parallel benchmarks in opencl," in *Proceedings of the International Symposium on Workload Characterization*, pp. 137–148, 2011.

[28] S. Seo, J. Kim, G. Jo, J. Lee, J. Nah, and J. Lee, "SNU NPB suite." Online http://aces.snu.ac.kr/software/snu-npb/; accessed 07-September-2016, 2013.

[29] "APM X-Gene1 specification." Online http://www.7-cpu.com/cpu/X-Gene.html; accessed 07-September-2016., 2016.

[30] "SPEC CPU2006 function profile." Online http://hpc.cs.tsinghua.edu.cn/research/cluster/SPEC2006Characterization/fprof.html; accessed, 2016.

[31] N. Nethercote and J. Seward, "Valgrind: A framework for heavy-weight dynamic binary instrumentation," in *Proceedings of the Conference on Programming Language Design and Implementation*, (New York, NY, USA), pp. 89–100, ACM, 2007.

[32] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *Proceedings of the International Symposium on Code Generation and Optimization* (V. J. Reddi, A. Smith, and L. Tang, eds.), pp. 305–317, ACM, 2017.

[33] R. Jordans and H. Corporaal, "High-level software-pipelining in LLVM," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, pp. 97–100, 2015.

[34] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation." Online; accessed 07-September-2016. Web Copy: http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf, 2010.

[35] V. Spiliopoulos, A. Sembrant, and S. 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 Telecommunication Systems*, pp. 241–250, 2012.

[36] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'donnell, and J. C. Ruttenberg, "The Multiflow

trace scheduling compiler," *The Journal of Supercomputing*, vol. 7, pp. 51–142, 1993.

[37] W. Havanki, S. Banerjia, and T. Conte, "Treegion scheduling for wide issue processors," in *Proceedings of the International Symposium High-Performance Computer Architecture*, pp. 266–276, 1998.

[38] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the Annual International Symposium on Microarchitecture*, pp. 45–54, 1992.

[39] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, 1993.

[40] C. Young and M. D. Smith, "Better global scheduling using path profiles," in *Proceedings of the Annual International Symposium on Microarchitecture*, pp. 115–123, 1998.

[41] J. R. Goodman and W.-C. Hsu, "Code scheduling and register allocation in large basic blocks," in *Proceedings of the International Conference on Supercomputing*, pp. 442–452, 1988.

[42] D. G. Bradlee, S. J. Eggers, and R. R. Henry, "Integrating register allocation and instruction scheduling for RISCs," in *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, pp. 122–131, 1991.

[43] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," *Transactions on Computers*, vol. 44, pp. 353–370, 1995.

[44] G. Chen, *Effective Instruction Scheduling with Limited Registers*. PhD thesis, Harvard University Cambridge, Massachusetts, Cambridge, MA, USA, 2001.

[45] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach, Appendix H: Hardware and Software for VLIW and EPIC*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2011.

[46] J. Kim, R. M. Rabbah, K. V. Palem, and W. fai Wong, "Adaptive compiler directed prefetching for EPIC processors," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 495–501, 2004.

[47] M. Khan, M. A. Laurenzano, J. Mars, E. Hagersten, and D. Black-Schaffer, "AREP: adaptive resource efficient prefetching for maximizing multicore performance," in *Proceedings of the International Conference on Parallel Architectural and Compilation Techniques*, pp. 367–378, 2015.

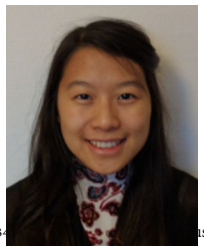
[48] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," *Computer Architecture News*, pp. 393–404, 2011.

[49] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen, "Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices," in *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 269–279, 2005.

[50] W. Zhang, D. M. Tullsen, and B. Calder, "Accelerating and adapting precomputation threads for effcient prefetching," in *Proceedings of the International Symposium High-Performance Computer Architecture*, pp. 85–95, 2007.

[51] R. Sheikh, J. Tuck, and E. Rotenberg, "Control-flow decoupling: An approach for timely, non-speculative branching," *Transactions on Computers*, vol. 64, no. 8, pp. 2182–2203, 2015.

[52] R. Sheikh, *Control-flow decoupling: An approach for timely, non-speculative branching*. PhD thesis, North Carolina State University, Department of Electrical and Computer Engineering, North Carolina, USA, 2013.



Kim-Anh Tran is a PhD student at Uppsala University since September 2014. She received her Master Degree in Computer Science at Uppsala University, Sweden, in 2013. After a year in industry, she started her PhD studies at Uppsala University with the focus on energy-efficient software-hardware co-designs.

Trevor E. Carlson is an Assistant Professor at the National University of Singapore. He received his B.S. and M.S. degrees from Carnegie Mellon University in 2002 and 2003, his Ph.D. from Ghent University in 2014, and has worked as a postdoctoral researcher at Uppsala University until 2017. His research interests include highly-efficient microarchitectures, hardware/software co-design, performance modeling and fast and scalable simulation methodologies.

Konstantinos Koukos is a PostDoc at KTH, working on the Model-based Computing Systems group. He has a PhD from Uppsala University on "efficient execution paradigms for heterogeneous architectures", and a master specialization on parallel programming and code optimization for heterogeneous systems. His research interests focuses on code optimizations for energy efficiency, and optimizations to better exploit the memory hierarchy of heterogeneous systems.

Magnus Själander is an Associate Professor at Norwegian University of Science and Technology (NTNU) and a Visiting Senior Lecturer at Uppsala University. He obtained his Ph.D. from Chalmers University of Technology in 2008. Before joining NTNU in 2016 he has been a researcher at Chalmers, Florida State University, and Uppsala University. Själander's research interests include hardware/software co-design (compiler, architecture, and hardware implementation) for high-efficiency computing.

Vasileios Spiliopoulos holds a PhD from Uppsala University. In his research, Vasileios developed analytical and statistical models aiming to improve energy efficiency of computer systems. After defending his PhD, Vasileios joined Zero-Point Technologies to work as a System Architect and Software Engineer on novel memory compression techniques.

Stefanos Kaxiras is a full professor at Uppsala University, Sweden. His research interests are in the areas of memory systems, and multiprocessor/multicore systems, with a focus on power efficiency. He is a Distinguished ACM Scientist and IEEE member.

Alexandra Jimborean is Assistant Professor at Uppsala University since 2015. She obtained her PhD from the University of Strasbourg, France in 2012, was awarded the Anita Borg Memorial Scholarship offered by Google in recognition of excellent research, along with other 25 distinctions, awards and grants. Her research focuses on compile-time and run-time code analysis and optimization for performance and energy efficiency and on software-hardware co-designs.