Clairvoyance: Look-Ahead Compile-Time Scheduling

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

*Uppsala University, Sweden
firstname.lastname@it.uu.se

†Norwegian University of Science and Technology, Norway
firstname.lastname@idi.ntnu.no

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 geomean 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.

1. Introduction

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

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 [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. 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

978-1-5090-4931-8/17 © 2017 IEEE

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.

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, 10], program slicing [11], and decoupled access-execute [12–14] 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 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 ← Unroll(L, count_unroll)
3      L_access ← Copy(L_unrolled)
4      hoist_list ← FindLoads(L_access)
5      to_keep ← ∅
6      for load in hoist_list do
7          requirements ← FindRequirements(load)
8          to_keep ← Union(to_keep, requirements)
9      end
10     L_access ← RemoveUnlisted(L_access, to_keep)
11     L_execute ← ReplaceListed(L_access, L_unrolled)
12     L_Clairvoyance ← 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.

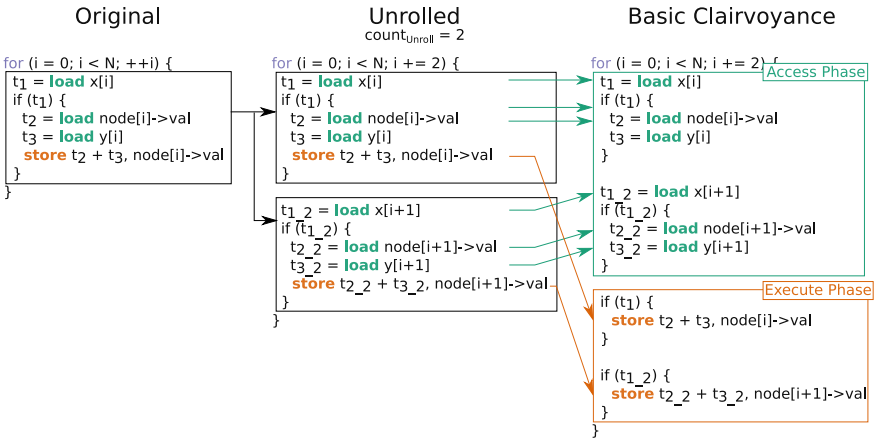

Figure 1. The basic Clairvoyance transformation. The original loop is first unrolled by $\mathrm{count}_{\mathrm{unroll}}$ which increases the number of instructions per loop iteration. Then, for each iteration, Clairvoyance hoists all (critical) loads and sinks their uses to create a memory-bound Access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

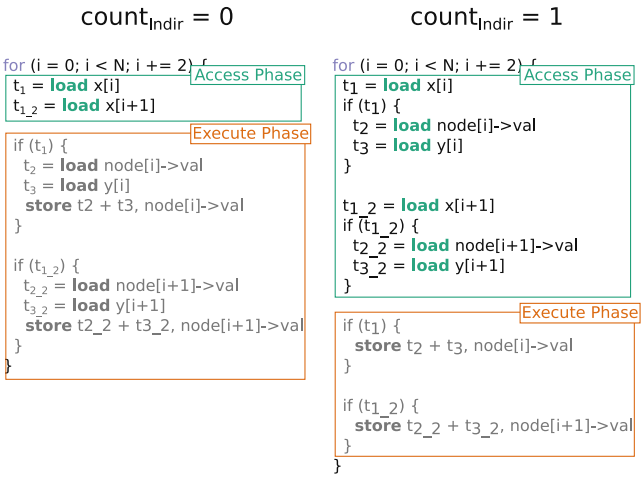

Figure 2. Selection of loads based on an indirection count $\mathrm{count}_{\mathrm{indir}}$. The Clairvoyance code for $\mathrm{count}_{\mathrm{indir}} = 0$ (left) and $\mathrm{count}_{\mathrm{indir}} = 1$ (right).

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.

2.1 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) [14] 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 [15] or VMAD [16, 17].

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 specu-*

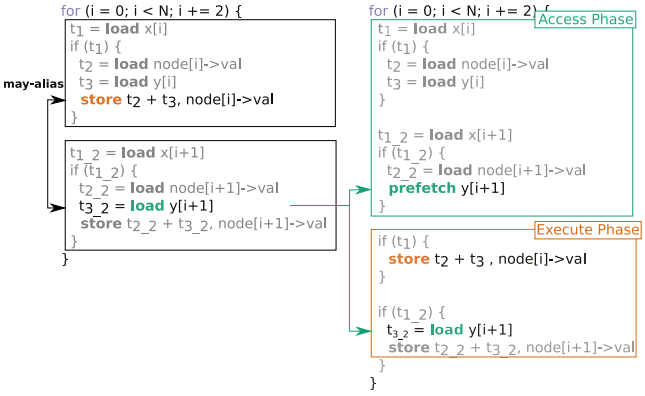

Figure 3. Handling of may-aliasing loads. Loads that may alias with any preceding store operation are not safe to hoist. Instead, we prefetch the unsafe load.

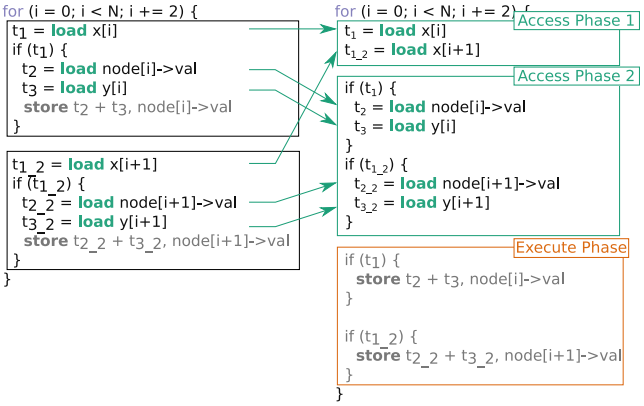

Figure 4. Splitting up dependent load chains. Clairvoyance creates one Access phase for each set of independent loads (and their requirements), which increases the distance between loads and their uses.

lation, 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 [18]. 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 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 3.

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 $\text{count}_{\text{unroll}} = 2$, 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`.

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

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.

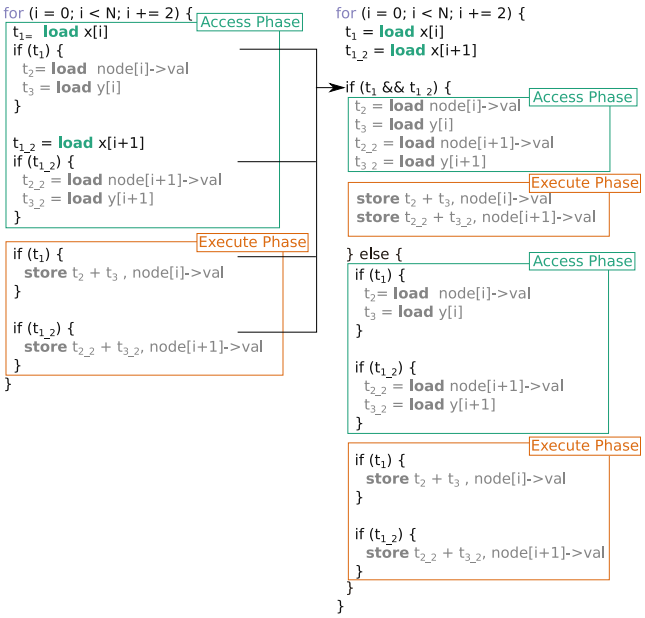

Figure 5. Early evaluation of branches enables the elimination of duplicated branches in Clairvoyance mode. Relevant branches are evaluated at the beginning of the loop. If the evaluated branches are taken, the optimized Clairvoyance code with merged basic blocks is executed; otherwise, the decoupled unrolled code (with branch duplication) is executed.

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 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 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. 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.

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.

Processor	APM X-Gene - AArch64 Octa-A57
Core Count	8
ROB size	128 micro-ops [19]
Issue Width	8 [19]
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 1. Architectural specifications of the APM X-Gene.

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.

2.7 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.

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.

3. Experimental Setup

Our transformation is implemented as a separate compilation pass in LLVM 3.8 [20]. We evaluate a range of C/C++ benchmarks from the SPEC CPU2006 [21] and NAS benchmark [22–24] suites on an APM X-Gene processor [25], 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 [26], while for NAS we identified the target functions using Valgrind [27].

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 only if the probability is above 90%.

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 2. Modified functions.

3.1 Evaluating LLVM, DAE and Clairvoyance

We compare our techniques to Software Decoupled Access-Execute (DAE) [13, 14] 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 [28]. 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.

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

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 3. Clairvoyance evaluated 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 [30].

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 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 [29] 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 example 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 speculative 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.

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.

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 speculation 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*.

4.2 Understanding Clairvoyance Best Versions

We categorize the benchmarks into memory-bound applications (`mcf`, `milc`, `soplex`, `libquantum`, `lbm`, `omnetpp`,

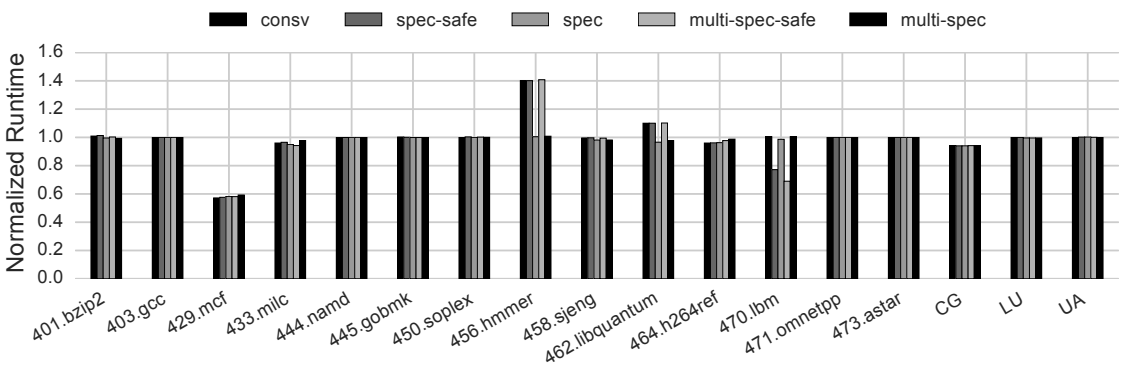

Figure 6. Normalized total runtime w.r.t original execution (-O3), for all Clairvoyance versions.

`astar`, `CG`) and compute-bound applications (`bzip2`, `gcc`, `namd`, `gobmk`, `hmmer`, `sjeng`, `h264ref`, `LU`, `UA`) [30]. 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 which can be hoisted to the *Access* phase, we are able to improve MLP while hiding the increased instruction count overhead. Clairvoyancewas 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.

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 [20, 28] are either not available for our target machine, or cannot transform our target loops. 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 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 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 `astar`.

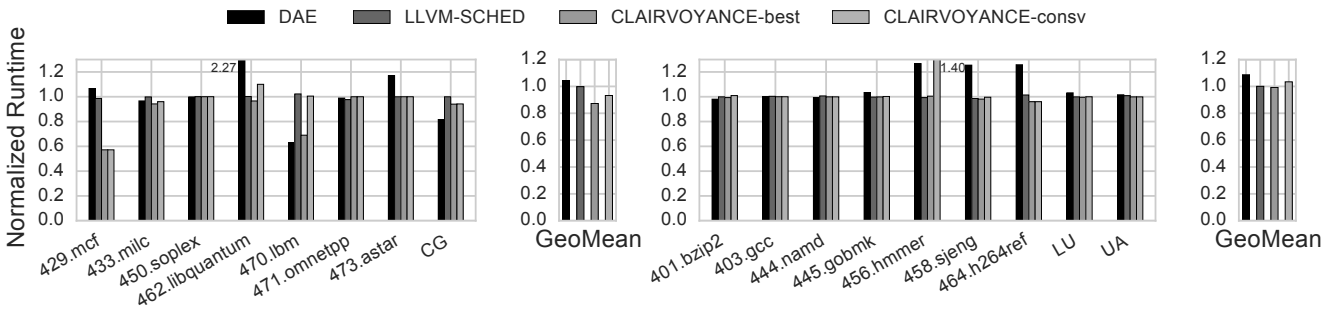

Figure 7. Normalized total runtime w.r.t original execution (-O3) for the best version of DAE, LLVM schedulers, and the conservative and the best version of Clairvoyance, categorized into memory-bound (left) and compute-bound (right) benchmarks.

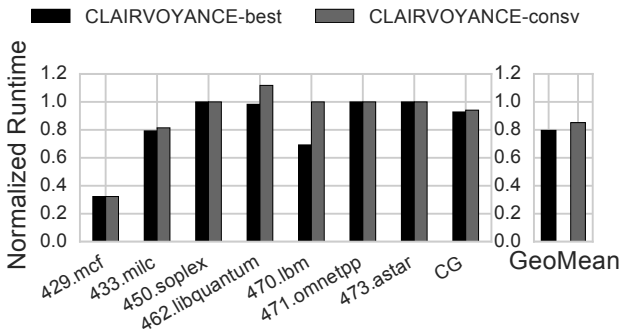

Figure 8. Normalized runtime per target loop w.r.t original loop execution (-O3) for the conservative and the best version of Clairvoyance, for memory-bound benchmarks.

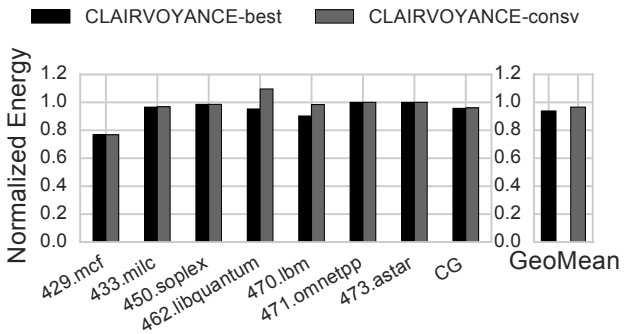

Figure 9. Normalized energy across all memory-bound benchmarks for the conservative and the best version of Clairvoyance.

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. [31]. 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.

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 [32–36] 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 [8, 9], 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 [37–39]. Previous work attempts to

provide register pressure sensitive instruction scheduling, in order to balance instruction level parallelism, latency, and spilling. Chen et al. [40] 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* [41, 42], 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 [43] 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) [13, 14] 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 [44–46] 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) [47] 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 [48]). 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 [13, 14], 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.

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.

Acknowledgments

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).

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] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

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

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

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

[28] 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.

[29] S. Yulei, D. Peng, and X. Jingling, "Sparse flow-sensitive pointer analysis for multithreaded programs," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 160–170, 2016.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

A. Artifact Description

A.1 Abstract

This artifact contains the source code to LLVM 3.8, the Clairvoyance passes, and the scripts to compile and run all benchmarks presented in our paper. To get started, we provide a Vagrantfile that will set up and install the required software. We conducted our experiments on an APM X-Gene Octa-A57.

A.2 Description

A.2.1 Check-List (Artifact Meta Information)

- **Program: C/C++ code**
- **Compilation: make**
- **Transformations: access-execute phase generation, lowering uses, branch clustering, multiple access phases**
- **Binary: O3, LLVM-SCHED, Clairvoyance**
- **Data set: NAS: C data set, SPEC CPU 2006: ref**
- **Run-time environment: Vagrant, Virtual Box**
- **Hardware: APM X-Gene AArch64 Octa-A57**
- **Execution: Experiments use perf**
- **Output: perf output**
- **Experiment workflow: build project, build binaries, run binaries, compare runtime**
- **Publicly available?: Yes**

A.2.2 How Delivered

Clone: `https://github.com/ktran/clairvoyance`. The repository structure is shown in Table 5 and contains:

- a vagrant configuration file (`Vagrantfile`)
- a setup script (`setup.sh`) that will be used by vagrant
- the compiler (`clairvoyance/compiler`)
- the experiments directory (`clairvoyance/experiments`) for building and running binaries from our paper

A.2.3 Hardware Dependencies

We evaluated on an APM X-Gene Octa-A57. Clairvoyance is designed to improve performance of limited out-of-order cores, therefore we recommend similar platforms for the evaluation. For building LLVM, provide a minimum 4GB of RAM to the virtual machine (default set up).

A.2.4 Software Dependencies

Vagrant (tested: 1.8.5) and Virtual Box (tested: 5.1.8).

A.2.5 Datasets

NAS benchmarks: ”C” data set size, SPEC CPU 2006: ref.

A.3 Installation

Currently, vagrant is set up such that it will use use 4 GB of RAM and 2 CPUs. If you have more resources you can change the Vagrantfile by modifying these lines:

Directory Name	Path	Description
clairvoyance	clairvoyance	root directory
experiments	clairvoyance/ experiments	files to build and run benchmarks
sources	clairvoyance/ experiments/ swoop/sources	benchmarks sources and scripts to build
runs	clairvoyance/ experiments/ swoop/runs	scripts to run and parse outputs

Table 5. The artifact folder structure.

Command	Description
vagrant up	starting up the virtual machine
vagrant halt	shutting down the virtual machine
vagrant ssh	ssh to your virtual machine
vagrant destroy	destroys the virtual machine

Table 6. Vagrant commands.

```
vb.memory = 4096
vb.cpus = 2
```

Table 6 contains the most important vagrant commands. In this step we will use vagrant in order to install the operating system and the required packages for the project. In the following you might need sudo rights whenever using vagrant:

```
$ cd clairvoyance
$ (sudo) vagrant up
```

Warning! If `vagrant up` does not work, see Section A.7. Never run `vagrant init` as this will overwrite the Vagrantfile. Next, ssh into the machine and install the project. Running make will compile the project (LLVM, Clairvoyance):

```
$ (sudo) vagrant ssh
$ cd /var/www/clairvoyance/compiler
$ make -j 2
```

A.4 Experiment Workflow

For benchmark configuration changes see Section A.6. This section only explains how to build and run the pre-configured benchmarks.

A.4.1 Building the Binaries

Build the pre-configured binaries by running `make`:

```
$ cd /var/www/clairvoyance/experiments/\
    swoop/sources
$ make -j 2
```

A.4.2 Running the Binaries

The `run_experiments` script in `experiments/swoop/runs` can be used to run the binaries of the provided bench-

Option	Description
-i [ref, train, test]	Select the input (all same for NAS)
-r $r	Number of times the benchmark should be run (default 1)
-n	Dry-run

Table 7. Options for the `run_experiments` script

marks. Table 7 shows the options and their semantics. Here, we run each configured binary once with the reference input:

```
$ cd /var/www/clairvoyance/experiments/\
    swoop/runs
$ ./run_experiments.py -i ref -r 1
```

This script will start the experiments, collect statistics, take the average (if repeat > 1) and write everything to the `test` directory. Each binary has a dedicated directory. The statistics are in `stderr.txt`, the output is written to `stdout.txt`, and if the output differs from the reference output, it will also contain an error file. We have configured the scripts such that they run the original (O3) and up to two Clairvoyance versions. `CG` takes approximately 15 minutes, `LU` 50 and `UA` 25 minutes per version (on our machine). This script may thus run for 270 minutes.

A.5 Evaluation and Expected Results

For `CG, UA, LU`, alias analysis can disambiguate memory accesses in the three NAS benchmarks, therefore no speculation is required (versions are similar or equivalent).

A.5.1 Parsing the Output

Parse the output that was gathered in the `test` directory:

```
$ cd /var/www/clairvoyance/\
    experiments/swoop/runs
$ ./parse_experiments.py -d test
```

This will read the `stderr.txt` files and generate a csv file `test/results.csv` which summarizes gathered statistics.

A.5.2 Plotting the Results

Run the `plot_experiments.py` script (in the `runs` directory). Pass the `test/results.csv` file and choose one out of three options: (`--print-best`, print best speculative versions, `--plot-version-comparison` plot best speculative versions, and `--plot-best`, compares LLVM schedulers, DAE (not described here) and Clairvoyance.

```
$ ./plot_experiments.py \
  -i test/results.csv --plot-best \
  --inputset ref
```

All numbers are normalized to O3 (plots require O3 runs).

A.5.3 Expected Results

We expect that Clairvoyance will reduce the total execution time of CG (memory-bound), but that the runtimes of UA and LU (compute-bound ormedium memory-bound) are less affected. The memory- vs compute-bound characteristics are target dependent, the results may vary on different hardware.

A.6 Experiment Customization

A.6.1 Compiling Existing Benchmarks

In order to compile another version of `CG,UA,LU`, follow step 3 in Section A.6.2. For changing the running scripts, look at the files `jobs.py` (which benchmarks to run) and `benchmarks.py` (which Clairvoyance parameters to use) in `experiments/swoop/runs/`.

A.6.2 Compiling Different Benchmarks

1. Add the Source Files. Copy the `sources/CG` directory and rename it to your benchmark. Then, (i) replace the source files in the new directory with your own source files, (ii) adapt the Makefile in that directory: change the flags, source files and set the benchmark name (should be different from any source file name).

2. Mark the Loops. Mark the loops that should be transformed using Clairvoyance with a pragma:

```
#pragma clang loop vectorize_width(1337)
for (int i = 0; i < N; ++i) {
  // some code
}
```

Clairvoyance only transforms inner most loops. It is best to select loops that have hard to predict data accesses.

3. Determine the Versions to Build. The `Makefile.target` file in `experiments/swoop/source/common/SWOOP/` determines which versions are build. In order to change the *unroll count* and *indirection count*, modify the `UNROLL_COUNT` and `INDIR_COUNT` variables.

4. Add Benchmark to Makefile. Add your benchmark name to the `BENCHMARKS` variable in `sources/Makefile`.

A.6.3 Running and Evaluating New Benchmarks

All scripts are already set up for SPEC CPU 2006. For new benchmarks it might be easier to run perf directly:

```
$ perf stat -r repeat -e \
    cycles,instructions benchmark
```

A.7 Notes

Box Could Not Be Found On OSX, you might need to run:

```
$ sudo rm /opt/vagrant/embedded/bin/curl
```

SWOOP Clairvoyance generates binaries that are called *SWOOP* (SW-OoO execution).

DAE If you wish to evaluate DAE, please go to `https://github.com/etascale/daedal`.

Artifact Evaluation For submission and reviewing guidelines, see `http://ctuning.org/ae/artifacts.html`.