SWOOP: Software-Hardware Co-design for Non-speculative, Execute-Ahead, In-Order Cores

Kim-Anh Tran
Uppsala University, Sweden
kim-anh.tran@it.uu.se

Alexandra Jimborean
Uppsala University, Sweden
alexandra.jimborean@it.uu.se

Trevor E. Carlson
National University of Singapore, Singapore
tcarlson@comp.nus.edu.sg

Konstantinos Koukos
Uppsala University, Sweden
konstantinos.koukos@it.uu.se

Magnus Själander
Norwegian University of Science and Technology (NTNU), Norway
Uppsala University, Sweden
magnus.sjalander@ntnu.no

Stefanos Kaxiras
Uppsala University, Sweden
stefanos.kaxiras@it.uu.se

Abstract

Increasing demands for energy efficiency constrain emerging hardware. These new hardware trends challenge the established assumptions in code generation and force us to rethink existing software optimization techniques. We propose a cross-layer redesign of the way compilers and the underlying microarchitecture are built and interact, to achieve both performance and high energy efficiency.

In this paper, we address one of the main performance bottlenecks—last-level cache misses—through a software-hardware co-design. Our approach is able to hide memory latency and attain increased memory and instruction level parallelism by orchestrating *a non-speculative, execute-ahead paradigm in software* (SWOOP). While out-of-order (OoO) architectures attempt to hide memory latency by dynamically reordering instructions, they do so through expensive, power-hungry, speculative mechanisms. We aim to shift this complexity into software, and we build upon compilation techniques inherited from VLIW, software pipelining, modulo scheduling, decoupled access-execution, and software prefetching. In contrast to previous approaches we do not rely on either software or hardware speculation that can be detrimental to efficiency. Our SWOOP compiler is enhanced with lightweight architectural support, thus being able to transform applications that include highly complex control-flow and indirect memory accesses.

The effectiveness of our software-hardware co-design is proven on the most limited but energy-efficient microarchitectures, non-speculative, in-order execution (InO) cores, which rely entirely on compile-time instruction scheduling.

We show that (1) our approach achieves its goal in hiding the latency of the last-level cache misses and improves performance by 34% and energy efficiency by 23% over the baseline InO core, competitive with an oracle InO core with a perfect last-level cache; (2) can even exceed the performance of the oracle core, by exposing a higher degree of memory and instruction level parallelism. Moreover, we compare to a modest speculative OoO core, which hides not only the latency of last-level cache misses, but most instruction latency, and conclude that while the OoO core is still 39% faster than SWOOP, it pays a steep price for this advantage by doubling the energy consumption.

CCS Concepts • **Software and its engineering** → **Source code generation**; **Software performance**; • **Hardware** → **Power and energy**;

Keywords hardware-software co-design, compilers, memory level parallelism

1 Introduction

Conventional, high-performance demands have steered hardware design towards complex and power-hungry solutions. Yet, as energy becomes the main constraint and limiting factor for performance, energy-efficient solutions are no longer just an option, but a necessity, as shown by emerging heterogeneous platforms featuring simpler and more energy-efficient (little) cores, rack scale computing centers, etc. These platforms rely on targeted hardware and on compilers to deliver customized software and to meet performance expectations.

One of the notorious performance bottlenecks is the performance gap between memory and processor, i.e., the memory wall [78], and it has been addressed by many previous proposals [15, 19, 49, 52, 57, 67, 68, 70]. As this gap becomes wider, the time spent waiting for memory has started to dominate the execution time for many relevant applications. One way to tackle this problem is to overlap multiple memory accesses (memory level parallelism) and to hide their latency with useful computation by reordering instructions (instruction level parallelism). In-order (InO) cores rely on static instruction schedulers [2, 43, 48] to hide long latencies by interleaving independent instructions between a load and its use. Nevertheless, such techniques cannot adapt to dynamic factors, and therefore lack flexibility and in practice are very limited. Out-of-order (OoO) cores reorder instructions dynamically with hardware support, thus, being able to hide longer latencies, confined only by the limits of the hardware structures (e.g., the reorder buffer), but are significantly more power hungry.

Support for speculation contributes greatly to the energy-inefficiency of OoO execution [23, 51]. In a seminal paper Zuyban and Kogge showed that energy consumption is exponential to the speculation depth and therefore energy is exponential to performance as we achieve it today [85]. The overall effect has come to be known as Pollack's rule that states that "processor performance increase due to microarchitecture advances is roughly proportional to the square root of the increase in complexity" [9]. Since the demise of Dennard scaling (the end of energy-efficiency due to the shrinking of process technology) [8], Pollack's rule is the critical determinant of processor efficiency.

We propose SWOOP (non-speculative Software Out-of-Order Processing), a novel software/hardware co-design that exposes last-level cache misses to software to perform better scheduling decisions. SWOOP's strategy for latency tolerance is to increase memory level parallelism, driven by actual miss events, but without any speculation. Software provides flexibility and low-overhead: the application is compiled such that it can adapt dynamically to hide memory latency and to achieve improved memory and instruction level parallelism. Hardware provides the runtime information to achieve adaptability and lightweight support to overcome static limitations, without resorting to speculation. While regular, predictable programs can be offloaded to accelerators and custom functional units [7], SWOOP attacks the difficult problem of speeding up a single thread with entangled memory and control dependencies, which are not amenable to fine-grain parallelization or prefetching.

SWOOP relies on a compiler that builds upon classical optimization techniques for limited hardware: global scheduling, modulo scheduling [2, 48], software pipelining [43, 58], decoupled access-execute [33, 42, 69], etc., but uses clever techniques to extend the applicability of these proposals to general-purpose applications featuring complex control
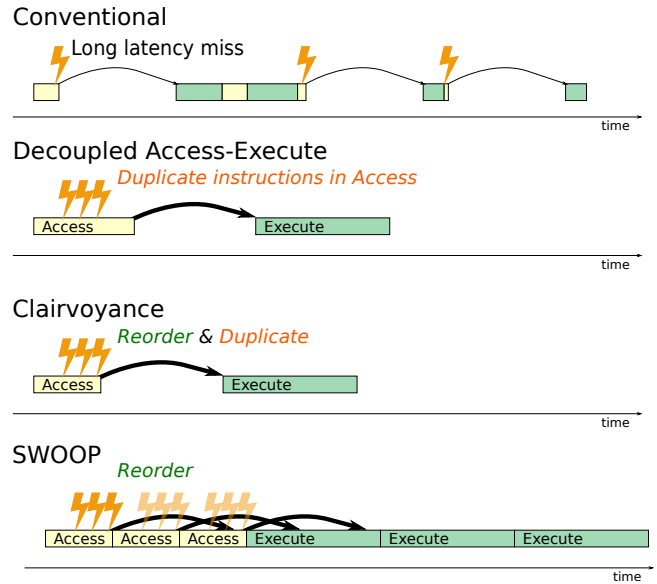

Figure 1. Conventional compilation stalls an in-order processor when using the results of a long latency memory access. DAE *prefetches* data to the cache to reduce core stall time but the overhead introduced by instruction duplication cannot be hidden by simple InO cores. Clairvoyance *combines reordering* of memory accesses from nearby iterations with data *prefetching*, but is limited by register pressure. SWOOP *reorders* and *clusters* memory accesses across iterations using frugal hardware support to avoid register pressure. Thus, SWOOP is able to access distant, critical instructions, hiding stall latencies with useful work.

flow and chained indirect memory accesses. To this end, SWOOP adds frugal hardware support to ensure runtime adaptability without speculation. The SWOOP core is an in-order based hardware-software co-designed processor that benefits from software knowledge to enable the hardware to execute *non-speculatively* past the conventional dynamic instruction stream.

The SWOOP compiler generates *Access-Execute* phases, akin to look-ahead compile-time instruction scheduling (Clairvoyance) [69] and the software decoupled access-execute (DAE) [33] model in which the target code region is transformed into *i)* a heavily memory-bound phase (i.e., the *Access*-phase for data prefetch), followed by *ii)* a heavily compute-bound phase (the *Execute*-phase that performs the actual computation). Unlike DAE, SWOOP operates on a much finer granularity and *reorders* instructions to *load* data in *Access*, instead of *duplicating* instructions for *prefetching*. Clairvoyance reduces overhead by reordering instructions as long as there are available registers, and then prefetches data to avoid register pressure. Unlike Clairvoyance, SWOOP relies on lightweight hardware mechanisms (1) to *reorder instructions without spilling registers* and (2) to select the
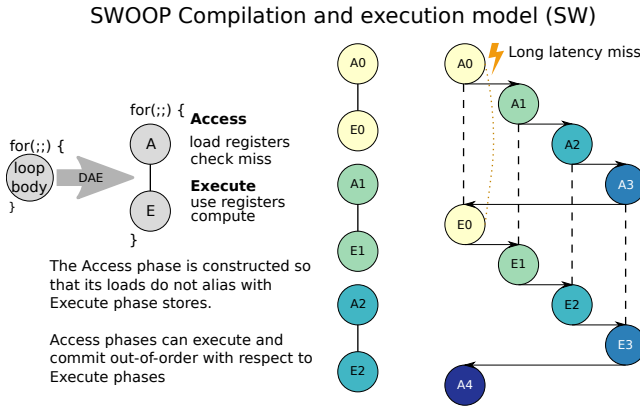
SWOOP Compilation and execution model (SW)

Figure 2. The SWOOP software model.

code to execute next (*Access* or *Execute*) depending on the observed cache misses (see Figure 1). *Access-Execute* phases are orchestrated in software guided by runtime information on cache access behavior and executed in a single superscalar pipeline, within a single thread of control. This is in stark contrast to previous *hardware* decoupled access-execute approaches [25, 65] that have separate pipelines and separate threads of control for *Access* and *Execute* code, respectively.

The SWOOP compiler and code transformations are described in Section 2, and the architectural enhancements in Section 3. Section 4 evaluates SWOOP and shows that it successfully hides memory latency and is *on-par* with an oracle that assumes perfect L3-cache (i.e., all accesses hit in the L3 cache, no memory accesses are performed). Section 5 places the work in context with respect to state-of-the-art techniques. Section 6 emphasizes the key differences between previous work and marks SWOOP's contributions. Section 7 offers conclusions.

2 SWOOP Software Components

A key component in SWOOP is the compiler that generates flexible code, prepared to dynamically adapt to the cache access behavior, assisted by targeted enhancements to the microarchitecture. SWOOP:

- Generates the *Access* and *Execute* code, decoupling the loop body as shown to the left in Figure 2.

- Enables execution to dynamically adapt by jumping to *Access* phases of future iterations, bypassing a stalling *Execute* phase. The right side of Figure 2 shows an example where three additional *Access* phases are executed (A_1, A_2, and A_3) before returning to run the *Execute* phase E_0 (that corresponds to the first *Access* phase A_0).

- Guarantees that *Access* phases can run *out-of-order* with respect to *Execute* phases, safely and efficiently, *without speculation, checkpoints, or rollbacks*.

2.1 Motivation and Background

State-of-the-art compilers[33, 69] have shown that it is possible to identify critical loads using static heuristics, to find sufficient independent instructions to hide memory latency, to break chains of dependent long-latency instructions that may stall the processor, to reuse already computed values, and to load data earlier in order to avoid branch re-execution and recomputations (e.g., in complex versions of software pipelining). While these techniques build on standard compiler methods like VLIW [22], PlayDoh, and Cydrom [59], they provide support for applications previously not amenable to global instruction scheduling and pipelining due to the complex control-flow and entangled dependencies. Yet, these techniques show their limits on very restricted hardware, such as in-order cores, where no mistakes in scheduling are forgiven, as the processor stalls immediately, and where instruction duplication comes expensively.

Our experiments show that the benefits of a state-of-the-art compiler for hiding memory latency, Clairvoyance [69], are canceled by its overheads when executing on an in-order core. More details in Section 4.

Our work leverages the compiler techniques from Clairvoyance to generate software for restricted hardware, but transcends prior work limitations when it comes to InO cores by advocating a software-hardware co-design. Responsibility on orchestrating execution is shared between software and hardware: the compiler prepares the code and offers non-speculatively optimized versions adapted for the potential execution contexts, while hardware provides runtime information on cache misses to decide upon the right version to execute. Overhead is alleviated through simple hardware extensions yielding an effective approach to exposing memory-level and instruction-level parallelism to hide memory latency.

2.2 SWOOP Compiler

The SWOOP compiler decouples critical loops in *Access* and *Execute* phases, following a technique that melds program slicing [75], software decoupled access-execute [33] and look-ahead compile-time scheduling [69]. *Access* phases are allowed to execute out-of-program-order (eagerly, before their designated time in the original program order), while *Execute* phases are kept in program order, as illustrated in Figure 2.

Access is built by hoisting loads, address computation, and the required control flow, following the use-def chain of instructions [33, 75]. The delinquent loads and their requirements (i.e., instructions required to compute the addresses and to maintain the control flow) are hoisted to an **Access phase**. We would typically choose to hoist *only* delinquent loads in an *Access* phase if such information is available, e.g., via profiling, compile-time analysis [55], or heuristics [69].

To this end, the compiler (1) creates a clone of the target loop body, (2) selects delinquent loads and the instructions

```
C/C++ code
for (i=0; i<max; i++){
    b[i] = a[i] + x->y[i];
}

Swoop IR code
i=0;
loop:
  r0 = load &x->y;
  prefetch r0+i;
  r2 = load a[i];
  if chkmiss goto AltPath;
  r1 = load r0+i;
  store r1 + r2, &b[i];
  i++;
  if (i<max) goto loop;
  goto CONT;
AltPath:
  tmp=i;
  M=N+i;
  if (i+1>=max) goto loop_E;
  i++;
loop_A:
  /*repeat Access Phase N-1 times*/
  CTX++;
  r0 = load &x->y;
  prefetch r0+i;
  r2 = load a[i];
  i++;
  if (i<M && i<max) goto loop_A;
  i=tmp;
  CTX=0;
loop_E:
  /*run Execute Phase N times*/
  r1 = load r0+i;
  store r1 + r2, &b[i];
  i++;
  CTX++;
  if (i<M && i<max) goto loop_E;
  CTX=0;
  if (i<max) goto loop;
CONT:
```

```
Legend
Access phase
Execute phase
Chkmiss
Register map
```

Figure 3. Illustration of compile-time code transformation showing an *Access* phase with may-alias memory accesses. All may-aliasing loads are transformed into safe prefetches.

required to compute their addresses (as long as they do not modify variables that escape the scope of *Access*), (3) keeps only the list of selected instructions and the control flow instructions, removes all other computations and memory accesses, and (4) simplifies the control flow to eliminate dead code (e.g., empty basic blocks and unnecessary branches). In the example in Figure 3, all loads are targeted for hoisting: the load of $a[i]$ and the pointer indirection ($x \rightarrow y[i]$). The value of $x \rightarrow y[i]$ can be loaded or prefetched by first loading the base address $r0 = \&x \rightarrow y$, and then computing the actual value's address using the offset i, i.e. $r0 + i$.

For the **Execute phase**, the compiler (5) takes the original target loop body, (6) removes instructions involved in computations of memory addresses, load operations, and computations of conditions that are part of the *Access* version, (7) replaces the use of all instructions that have counterparts in the *Access* version to avoid the overhead of duplicating computations, and (8) updates the phi nodes and the control flow to enable the execution of the subsequent *Access* phase of the next iteration. *Access* binds values to registers and *Execute* consumes these data and performs all other computations (see Figure 2 and Figure 3), without duplicating unnecessary instructions (e.g., already computed values, target addresses, and branch conditions).

The border between *Access* and *Execute* is marked by a *chkmiss* instruction, which indicates whether any of the loads in *Access* incurred a miss in the last-level cache, potentially yielding a stall in *Execute*. Chkmiss is the SWOOP hardware support to guide the execution flow through a simple and efficient mechanism, which signals an early warning regarding upcoming stalls in *Execute*. If any of the loads in *Access* are not present in the cache hierarchy, the chkmiss instruction triggers a transfer of control to prevent the corresponding *Execute* phase from stalling on the use of a long latency load (Section 3.2). Decoupling the loop into *Access* and *Execute* phases provides a clear and natural point in the code to introduce the checkmiss instruction.

Upon a miss, execution surrenders control flow to the **alternative execution path** (AltPath in Figure 3).

The *Access* phase within the alternative path is run $N - 1$ times, as long as we do not exceed the total iteration count ($i < max$). Note that one iteration of the *Access* phase has already run before entering the alternative part. By incrementing the loop iterator prior to entering *loop_A*, we skip the already executed iteration and proceed to the $N - 1$ iterations that are left to process. N is an architecture-dependent parameter whose value is determined with respect to the number of physical registers provided by the architecture and the number of registers required by an *Access* phase. At the beginning of each *Access* phase, the current active context is incremented ($CTX++$). The first context ($CTX = 0$) refers to the values loaded by the *Access* phase before entering the alternative path. After finishing in total N *Access* iterations ($i >= M$) we continue to execute the corresponding N *Execute* phases. In order to reflect the correct context and loop iteration, we reset both i and CTX, i.e. $i = tmp$ and $CTX = 0$. Each *Execute* phase therefore uses the registers set by the corresponding *Access* phase. In the end (when $i >= M$), only the context is reset ($CTX = 0$) and we conclude the alternative path by either jumping back to the regular loop execution or exit the loop.

Figure 3 sketches an implementation where the execution of *Access* and *Execute* is controlled by a loop. Alternatively, the compiler may choose to perform unrolling, when the

compiler deems it beneficial (e.g., static loop bounds, low register pressure, etc).

Memory Dependency Handling A key contribution of SWOOP is the handling of dependencies, statically, in a way that results in *non-speculative* out-of-program-order execution. The compiler ensures that instructions hoisted to *Access* do not violate memory and control dependencies. This is achieved by:

Effective handling of known dependencies. Known read-after-write (RAW) dependencies are limited to *Execute* phases, thereby freeing *Access* phases to execute out-of-program-order with respect to other *Access* or *Execute* phases. In short, dependent load-store pairs are kept in their original position in *Execute*.

Effective handling of unknown dependencies. Potential unknown dependencies between *Execute* and *Access* phases are transformed to safe prefetches in *Access*, without a register binding, which takes place only in the in-order *Execute* phase. Thus, loads that may alias stores in *Execute* (illustrated with the pair (load:$x \rightarrow y[i]$,store:&b[i]) in Figure 3) are prefetched in *Access* and safely loaded in *Execute*. Prefetching transforms long latencies into short latencies: if the load-store pair did not alias, the correct data was prefetched in L1; in the case of aliasing, the prefetch instruction fetched stall data, but there is a high probability that the required data still resides in L1 as it has been recently accessed by the store.

Our SWOOP compilation strategy is the following: *Access-Execute* communication takes place via registers loaded in *Access* and used in *Execute*, *if the absence of memory-dependency violations can be guaranteed. Execute* uses registers without the need to compute addresses, and stores registers directly to the memory address space. For the loads that *may* have dependencies we follow the approach of communicating via the cache. *Access* simply prefetches into the L1 and *Execute* loads the register from the L1.

Why Access - Execute *phases?* The SWOOP methodology centers around the idea of a change of the program execution depending on the state of the cache hierarchy. An idealized, *unstructured* approach would start with highly-optimized code (e.g., −O3 compiled) that aims to minimize execution time by reducing instruction count, register spilling and other inefficiencies. We can then use the chkmiss instruction to temporarily break out of a loop iteration *at any point* where a long-latency miss would block the rest of the instructions in the iteration, and proceed with some other profitable work, e.g., the next iteration. Unfortunately, there are several issues with this approach: (1) Memory-level-parallelism (MLP) is reduced if more misses in the same iteration could follow the break-out point. (2) Each possible exit point would have to be managed by the compiler, potentially causing significant control-flow management complexity. (3) The next iteration needs to block, possibly before any useful work

could be performed, for instance if a *known* loop carried dependency exists between the unfinished part of the first iteration and the next. (4) Worse, *unknown* memory dependencies between iterations would cause severe problems unless dynamic hardware support (dependency tracking, speculation, checkpointing, and restart) is provided. This, of course, would incur considerable costs.

Clearly, an unstructured approach is burdened with considerable complexity and could require extensive hardware support to achieve modest performance gains. In contrast, decoupling the loop iterations into *Access* and *Execute* eliminates these problems and collects the benefits of memory level parallelism.

Leveraging the decoupled execution model, SWOOP does not check for further misses once the *Access* phases start their execution out-of-order (Alternative path). This is because any miss in an out-of-order *Access* phase is sufficiently separated from its corresponding *Execute* phase so that it requires no special attention. In fact, this is how SWOOP achieves MLP. While an OoO core is constrained to the MLP of the loop iterations that fit in its reorder buffer, SWOOP executes multiple *Access* phases regardless of how far away they are spaced in the dynamic instruction stream.

Transformation Scope and Limitations SWOOP primarily targets inner loops, but can also handle long critical code paths with pointer-chasing and with subroutines that can be inlined or marked as pure (side-effect-free). SWOOP can effectively target pointer-intensive applications by combining prefetching with state-of-the-art pointer analysis.

Multi-threaded applications can be transformed within synchronization boundaries. The compiler can freely reorder instructions—with respect to other threads—both in data-race-free (DRF) [1] applications and within synchronization-free regions of code (e.g., data parallel OpenMP loops, the code within a task, section, etc). We aim to expand the compiler analysis to model aliasing across non-synchronization-free code regions in future work.

Limits are RAW loop-carried dependencies from *Execute* to *Access*. Such loads cannot be hoisted to Access, but they are not likely to incur a miss either, since the value has just been written in Execute.

3 SWOOP Architectural Support

While SWOOP compilation orchestrates non-speculative out-of-order execution of *Access* and *Execute* code, targeted enhancements of the microarchitecture are essential for efficiency.

The fundamental roadblock for a concept such as SWOOP is a lack of registers. Given enough registers one can compile for an unconstrained separation between an *Access* phase and its corresponding *Execute*, knowing that their communication will be captured in the register file (at least for the

accesses that are free of memory dependencies). Architectural registers are a scarce resource and instead, we opt for transparent register remapping in hardware: providing a larger set of physical registers and dynamically mapping the architectural registers on them. The limitation now becomes one of area and energy consumption: large physical register files are one of the most expensive structures (in both area and energy) in a microarchitecture.

Thus, the challenge of SWOOP is to provide enough separation between *Access* and *Execute* so that the technique is useful in hiding large memory latencies, but do so with the least number of registers. We choose to address this challenge by *dynamically* reacting to miss events and only then providing the necessary registers to enable a separation of *Access* and *Execute*. Periods of execution that are free of miss events on the baseline InO core cannot make use of a large physical register file, which becomes an energy burden and reduces overall efficiency.

Towards this end, we add two hardware features to the conventional stall-on-use InO processor:

- **Context Remapping:** A novel lightweight form of register renaming that: *i)* enables dynamic separation of an *Access* and its corresponding *Execute*, with a number of intervening *Access* phases from future iterations; *ii)* ensures that only the registers that are written in each *Access* phase will be remapped and encapsulated in a unit denoted as a *Context*; and *iii)* manages dependencies between *Contexts* (Section 3.1). We provide a small number of additional physical registers to dynamically populate register contexts. Our strategy reduces the architectural state compared to processors that expose register windows [63] or register rotation to the ISA [14, 37], require additional scratch register files and register file checkpointing [28], or need to store data in slice buffers, creating a secondary temporary register file [29].

- **Chkmiss instruction:** A simple and efficient mechanism to react to upcoming stalls, similar to informing memory operations [30]. A chkmiss (check miss) instruction determines the status of a set of previously executed load instructions. If any of the loads in *Access* are not present in the cache hierarchy, the chkmiss instruction triggers a transfer of control to prevent the corresponding *Execute* phase from stalling on the use of a long latency load (Section 3.2). It is at this point when Context Remapping is activated to enable the separation of *Access* and *Execute*.

The two architectural enhancements in SWOOP constitute an example of providing as few resources as possible to enable SWOOP compilation to deliver on the desired performance and energy goals. Section 4 demonstrates how well we succeed in this. However, we do not preclude other
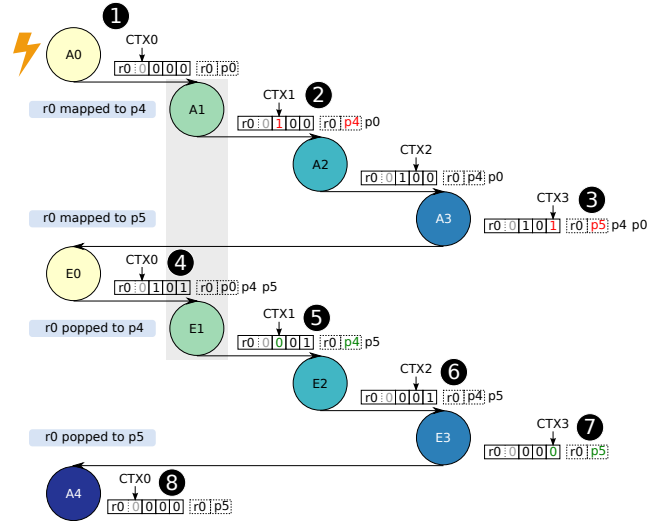

Figure 4. SWOOP Context register remapping.

possible implementations that can support the compilation framework in working equally well.

3.1 Context Register Remapping

SWOOP only relies on software prefetching when the compiler is unable to guarantee that no dependencies exist, in which case *Access* and *Execute* communicate via the L1 cache. For the remaining accesses, register-file *Access-Execute* communication is necessary to maintain much of the performance of the original optimized code. The main problem that we need to solve when using register-file communication is that it creates more live registers. When executing in program order the allocated registers in the *Access* phase are immediately consumed by the *Execute* phase. However, when executing out of program order, by injecting $Access_i$ phases, the allocated registers of each *Access* phase now have to be maintained until they get consumed by their corresponding $Execute_i$ phase.

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

SWOOP remaps only in the alternative execution path (not on normal execution). Each architectural register is remapped *only once* when it is first written in an *Access* phase, while no additional remapping is done in *Execute*. The new name is used throughout the remaining *Access* and all of its corresponding *Execute* phase, regardless of further

assignments to the architectural register. No additional register remapping is needed within an *Access* or an *Execute* or between an *Access* and *Execute* belonging to the same SWOOP context. This results in a significantly lower rate of remapping than what is found in OoO cores.

We give full flexibility to the software via two (new) instructions that directly set the current active context (CTX): CTX=0 and CTX++. The program-order (non-SWOOP execution) context is CTX 0. No remapping takes place during non-SWOOP-related execution. During SWOOP execution architectural registers are managed in a FIFO-like fashion. Each *Access* phase in the *alternative execution path* increments the context by CTX++, see Figure 3. *Returning* to a previous context, i.e., after the last *Access* or *Execute* phase, is done by a CTX=0. After exiting the alternative execution path from the last *Execute* phase, the current active context is set to 0.

Figure 4 shows an example of context register renaming. In step ① we are in normal execution (CTX$_0$), access phase A0 is running, and we have not entered SWOOP execution yet. Assume that in the normal-execution context the architectural register r0 happens to be mapped onto the physical register p0. A0 suffers a miss and because E0 will stall, we enter SWOOP execution by executing a CTX++ and proceeding to A1. We are now in CTX$_1$. If the same architectural register is written in this context (i.e., in A1) we rename it to a new physical register, e.g., p4 in step ②. We only rename registers that are written in a context and we only do it once per context.

Helper structures such as the bitmap vector per register shown in Figure 4 help us remember in which context was an architectural register renamed. In addition, the mappings of each architectural register (r0 in our example) are kept in a FIFO list.

Register r0 is not written in A2 but is written again in A3 (step ③) and gets a new mapping to p5. Once we finish the execution of access phases, because we have exhausted either the physical registers or the contexts we must return to the execution of E0. For this, in step ④, we change the context to CTX$_0$ (by a CTX=0) and start using the *first mapping in the register's FIFO list*; that would be (the original) p0 for r0. Proceeding to the next *Execute* phase EX1 in step ⑤, we check if we had renamed r0 in this context (CTX$_1$) and, if so, we pop its FIFO list to get to the next mapping (p4), and so forth. In CTX$_2$ we have done no renaming for r0, hence there is no reason to pop its FIFO, but in CTX$_3$ we have, so we pop once more to get p5 (step ⑦). Finally, when we return to normal execution (step ⑧) we keep the mappings of the last context as the mappings of the normal-execution context (i.e., p5 for r0).

Each SWOOP core implementation provides two software readable registers that specify the available number of physical registers and contexts. Before entering a SWOOP targeted loop these registers are read and the software adapts the execution to the available resources. The software is responsible for not allocating more physical registers (registers written per *Access* times *N*) or using more contexts than available.

If the software misbehaves, deadlock is prevented by monitoring requests that use too many physical registers or contexts and then raising an exception. Exceptions cause the context state of the user thread to return to CTX=0.

The SWOOP hardware facilities were developed for user-space code execution. On an operating system context switch, the hardware physical register context mapping (CTX) will remain the same. On return to the same user-space program, it is the operating system's responsibility to restore the architectural state of the user-space thread. Support for context switching to new user-space programs can either be implemented by having the hardware wait for the SWOOP execution to end, or to expose access to the context mapping state to the operating system.

3.2 Chkmiss

Chkmiss is an *informing memory operation* [30] which provides early warning on upcoming stalling code, essential for a timely control flow change to the alternative execution path. Lightweight techniques to predict misses in the cache hierarchy have been proposed [77, 80] and refined to detect a last-level cache (LLC) miss in one cycle [62]. We encode the presence of an LLC cache line in the TLB entries, using a simple bitmap (e.g., 64 bits for 64-byte cache lines in a 4kB page). This bit map is updated (out of the critical path) upon eviction from the LLC, rendering the TLB an accurate predictor for LLC misses.

Chkmiss monitors the set of loads hoisted to *Access* and reports if *any* of the loads missed. Thus, the chkmiss becomes a branch (predicated on the occurrence of misses) and takes the form: *chkmiss TargetPC* where *TargetPC* points to the alternative execution path, see AltPath in Figure 3.

We make use of breakpoints to offer a lightweight implementation of chkmiss. A prologue preceding a SWOOP loop sets chkmiss breakpoints at desired locations (e.g., at the end of *Access* phases) and specifies their *TargetPC*.

Since only a few chkmiss breakpoints are supported (e.g., four), they are encoded in a small table attached to the *Fetch* pipeline stage. The monitoring cost is, thus, kept low. Although in practice four breakpoints suffice to capture most interesting cases, if more chkmiss are required, a chkmiss instruction can be used (at the cost of an instruction issue slot).

Debugging based on dynamic events should not be an issue if a feature like Intel's Last Branch Record Facility [32] is used to record branch history information.

3.3 SWOOP Memory Model

SWOOP safely reorders instructions—with respect to other threads—within *data-race-free* (DRF) [1] regions of code and

Table 1. Simulated Microarchitecture Details. (*) The OoO is 3-way superscalar at 2.66 GHz, with: (**) 3 int/br., 1 mul, 2 fp, and 2 ld/st units.

	In-Order		**OoO**
Core	InO	SWOOP	
u-Arch	2.66 GHz, 2-way superscalar		*
ROB	-	-	32
RS	-	-	[16/32]
Phys. Reg	32x32	96x64	64/64
Br. Pred.	Pentium M (Dothan) [72]		
Br. Penalty	7	8	15
Exec. Units	2 int/br., 1 mul, 1 fp, 1 ld/st		**
L1-I	32 KB, 8-way LRU		
L1-D	32 KB, 8-way LRU, 4 cycle, 8 MSHRs		
L2 cache	256 KB, 8-way LRU, 8 cycle		
L3 cache	4 MB, 16-way LRU, 30 cycle		
DRAM	7.6 GB/s, 45 ns access latency		
Prefetcher	stride-based, L2, 16 streams		
Technology	28 nm		

as long as the region of interest does not cross synchronization points. Thus, SWOOP can handle, e.g., data parallel OpenMP applications and programs adhering to SC-for-DRF semantics as defined in contemporary language standards (C++11, C++17, Java, Scala, etc.), irrespective of the underlying consistency model provided by hardware.

Looking forward and beyond DRF restrictions, a recent breakthrough enables *non-speculative load-load reordering* in TSO (Total Store Order) [60]. This means that SWOOP can support TSO, insofar as compiler-reorder loads are concerned.

4 Evaluation

Simulation Environment. We use the Sniper Multi-Core Simulator [11] to evaluate this work. We modify the cycle-level core model [12] to support the SWOOP processor. Power and energy estimates are calculated with McPAT [45] version 1.3 in 28 nm. For our baselines, we chose two efficient processor implementations. On the low-energy side we model the ARM Cortex-A7 [6], which is an extremely energy-efficient design with support for dual-issue superscalar in-order processing, while the high-performing core is modeled with characteristics similar to a generic ARM Cortex-A15 [5]; that is, a 3-wide out-of-order core. All simulations are performed with a hardware stride-based prefetcher capable of handling 16 independent streams, leaving only hard to prefetch DRAM accesses. The processor parameters are shown in Table 1.

Benchmarks. SWOOP is targeted towards *demanding* workloads with frequent misses, i.e., *high MPKI* (misses-per-kilo-instructions) workloads even when employing hardware

prefetcher, that can be difficult even for OoO cores, yielding low performance. We select a number of benchmarks, from SPEC2006CPU [27], CIGAR [46] and NAS [50] benchmark suites, shown in Table 2. Although our primary selection criterion is MPKI, we make an effort to diversify the benchmarks to: *i*) evaluate both short (e.g., cg, libquantum) and large (e.g., lbm) loops, *ii*) give the compiler a choice of hoisting several delinquent loads in *Access* (e.g., cg, sphinx3), and *iii*) unknown dependencies that are handled with prefetching (e.g., soplex). For each benchmark, Table 2 provides: its MPKI, the *additional dynamic instruction* overhead due to decoupling, the number of loads hoisted in *Access* (for the evaluation runs), the corresponding general-purpose and SSE registers written in *Access* that determine the remapping requirements per context, the number of contexts used in the evaluation runs (N-best) and finally, the chkmiss *firing* rate, i.e., the rate at which chkmiss initiated SWOOP execution.

SWOOP Evaluation against Different Software and Hardware Solutions. To evaluate SWOOP we compare its performance and energy efficiency against different software approaches and hardware architectures. A comparison against software pipelining was excluded, as off-the-shelf software pipelining tools are not ready to cope with complex control flow [34], or are only available for other targets (LLVM's software pipeliner). We evaluate:

InO is similar to a Cortex-A7 in-order, stall-on-use core.

Clairvoyance is a state-of-the-art software-only solution to hide memory latency, by combining instruction reordering with software prefetching.

InO-Perfect-L3 is a fictitious in-order, stall-on-use core with a perfect L3 cache, i.e., L3 accesses always hit and prefetchers are disabled. This acts as an oracle for SWOOP.

OoO-r32 is similar to a Cortex-A15 out-of-order core, which hides DRAM accesses by dynamic out-of-order execution of instructions.

SWOOP with hardware support as described in Section 3. SWOOP has one additional pipeline stage over the in-order core to support context register remapping during SWOOP execution (see Section 3.1). This is reflected by the one cycle longer branch penalty shown in Table 1.

4.1 Performance

Figure 5 shows speedups normalized to the in-order, stall-on-use core (InO). SWOOP achieves significant performance improvements (34% on average) compared to InO and outpaces even the OoO when running cigar (12% faster). SWOOP targets long latency loads, i.e., main memory accesses, and avoids stalls by increasing the memory-level parallelism and executing useful instructions to hide the latency. Ideally, SWOOP could eliminate all latency caused by main memory accesses. SWOOP is competitive with the oracle InO core

Table 2. Benchmark characteristics.

Benchmark	MPKI	SWOOP Instruction Overhead	Hoisted Loads	Access GPR	Access SSE	N-Best	Chkmiss Firing	Unrolled MPKI	Unrolled Instruction Overhead	N-Best
cg	30	26%	3	5	2	4	65%	37	-0.3%	2
cigar	120	-13%	1	5	1	8	62%	123	-10.0%	4
lbm	41	1%	5	5	5	4	100%	39	3.5%	8
libquantum	32	5%	1	4	0	4	41%	37	-9.5%	16
mcf	68	3%	3	7	0	8	79%	70	0.8%	4
soplex	60	1%	3	7	7	4	18%	62	-1.6%	2
sphinx3	17	-8%	2	4	2	8	43%	18	-8.4%	4
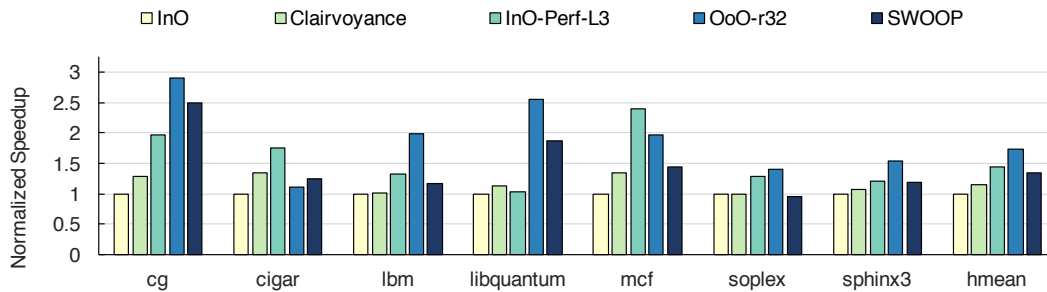
Figure 5. Speedup comparison across software and microarchitecture configurations.
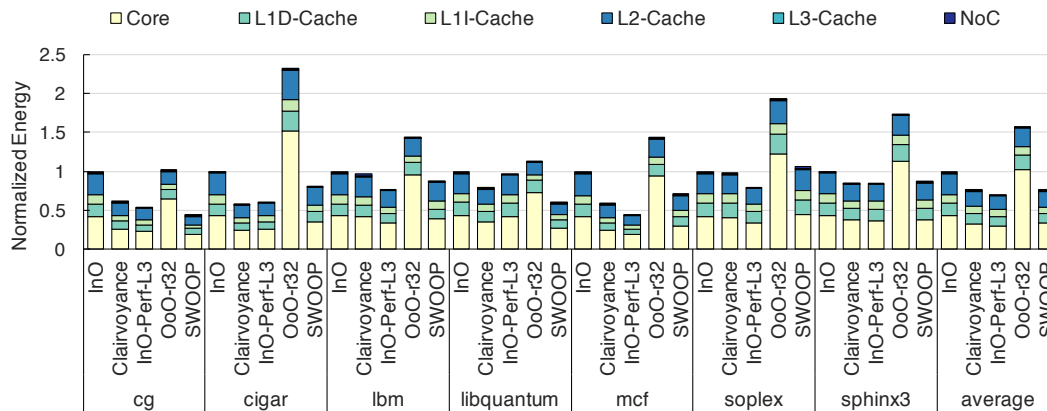
Figure 6. Energy efficiency comparison across software and microarchitecture configurations.

with a perfect L3 (InO-Perf-L3), which has no main memory latency, being on average only 7% slower. In some situations, SWOOP even outperforms the InO-Perf-L3 for cg (by 26%) and libquantum (82%). When SWOOP executes along the alternative path (Figure 3), data are not only fetched from main memory but also from the L1 and L2 caches and stored in registers for quick access in the *Execute* phases. While Clairvoyance brings modest improvements on average (15%), SWOOP outperforms Clairvoyance by 19%, resulting in more than doubling the performance improvement over the InO core (average improvement of 34%). SWOOP boosts performance significantly for two of the benchmarks (cg—120%

and libquantum—74%) over the improvement seen by Clairvoyance.

For two of the benchmarks, lbm and sphinx3, SWOOP achieves speedup over the InO, but not nearly the speedup of the OoO. The lbm benchmark contains many entangled loads and stores, making it difficult to effectively split and group loads due to may-alias dependencies. For sphinx3, the issue is that while there is a large number of delinquent loads in the application, each load contributes to a very small fraction of the total application delay, limiting total improvement. Hoisting all loads with small contributions may be expensive, as additonal registers and instructions (e.g., for control-flow)

are required, which may cancel the benefits of hoisting such loads.

For libquantum SWOOP achieves significantly better performance than InO and reaches half the speedup of the OoO core. Libquantum contains a very tight loop, hence loads and their uses are very close to each other, and therefore prolong the duration of stalls. With SWOOP, we manage to successfully separate loads from their uses, and to overlap outstanding latencies, which is reflected in the performance improvement. The OoO core on the other hand hides not only long latencies of loads, but also latencies of other not-ready-to execute instructions, thus hiding more stalls.

Finally, for soplex, SWOOP fails to achieve any speedup over the InO but does not significantly hurt performance (0.6% slower). Soplex suffers from the same problem as sphinx3: a single delinquent load in the *Access* phase (responsible for only 7% of the memory slack), exacerbated by a low chkmiss firing rate of 18% Table 2.

4.2   Energy
Figure 6 shows the energy usage normalized to the InO core. SWOOP reduces the energy usage by 23% on average and shows significant improvements compared to a low-power InO core, approaching the InO core with a perfect L3, which can only be simulated—not implementable in practice. Clairvoyance and SWOOP are mostly on-par. The energy savings stemming from better performance in SWOOP (cg, lbm, libquantum) are balanced by the energy expenditure due to additional hardware (mcf, soplex). The OoO core, which has the best average speedup, increases the energy usage by 57%. The results clearly show the exorbitant cost for the dynamic out-of-order execution that causes an OoO core to have about 3-5x higher power consumption than an in-order core [31, 79]. For the presented results McPAT estimated on average that the power consumption was 3x higher for the OoO compared to the InO.

4.3   SWOOP, Unrolling, and SW Prefetching
We compare SWOOP to two software-only techniques, forced unrolling and software prefetching.

InO-Unroll is a software-only solution based on standard compile-time transformations in which the original loop-bodies identified by SWOOP have instead been (forcedly) unrolled and executed on the InO, without any SWOOP code generation or hardware support. This version shows the limits/potential of the compiler to make use of the available architectural registers.

InO-SW-Prefetch is a software prefetch solution taken from the work of Khan et al. [38, 40][1]. InO-SW-prefetch uses profiling to identify delinquent loads to prefetch

and manually inserts prefetch instructions at the appropriate distance. We port the applications to our infrastructure and use prefetch to the L3 instructions. We use InO-SW-prefetch to substitute optimal compiler prefetching.

Figure 7 displays the execution on an InO with the hardware prefetching enabled (left) and disabled (right) for each application, all normalized to the original execution *with* hardware prefetching.

The results clearly show that forcing the loops to be unrolled is not beneficial and instead hurts performance (3% slower on average) due to register spilling, motivating the necessity for SWOOP code versions.

When running libquantum, SWOOP outperforms software prefetching on both architectures, having the advantage of acting only upon a miss, i.e., 41% firing rate of chkmiss, whereas software prefetching is unnecessarily always active.

SWOOP is on par with the software techniques when running soplex. As mentioned earlier, soplex has only one delinquent load with a low impact on the accumulated memory latency; therefore, both software prefetching and SWOOP are unable to improve its performance.

For mcf SWOOP is the clear winner compared to both software techniques, and different hardware configurations. This is because, mcf is amenable to software techniques for hiding memory latency thanks to its irregular behavior, and is hardly predictable by hardware prefetchers. Moreover, mcf contains few loads responsible for most of the L3 misses, a good target for prefetching. Yet, although software prefetching shows good results, SWOOP achieves significantly better performance, thanks to reducing the number of redundant instructions and having precision (chkmiss firing rate of 79%). These techniques enable SWOOP to fully and efficiently utilize the existing hardware resources.

Being a compute-bound application, lbm exhibits a slight improvement, both with software prefetching and with SWOOP. However, SWOOP, being concise, is competitive and reuses many of the instructions required for computing the addresses for early loading and prefetching, while software prefetching duplicates such computations in different iterations.

5   Related Work
Numerous proposals attempted hiding memory latency either in software, by employing purely hardware techniques, or a combination of both.

Software only techniques vary from inspector-executor and prefetching to global scheduling and software pipelining.

Inspector-executor methods derived from automatic parallelization [4, 83] include a compiler that generates precomputational slices grouping memory accesses [33, 42, 56, 82] and initiates runahead execution, in some cases using a prefetching helper thread [16, 35, 47, 82, 84]. Inspectors

---

[1]We contacted the authors who provided the updated applications used in their work. We evaluate the benchmarks common in their work and ours.
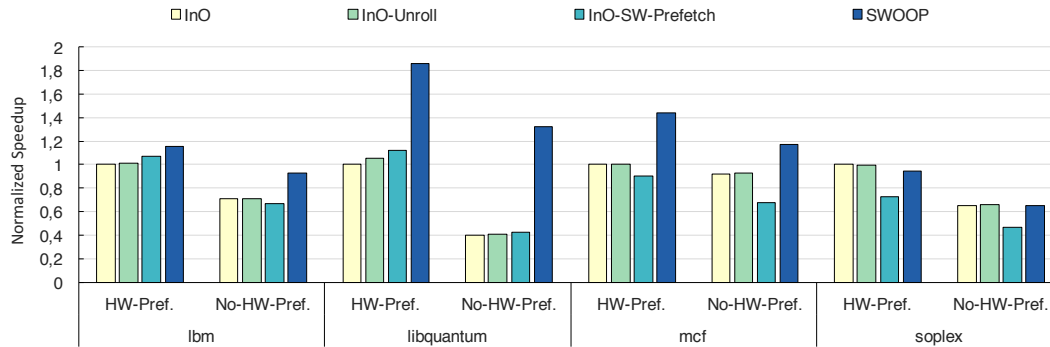
Figure 7. Comparison of unrolling and software prefetching with SWOOP, with and without hardware prefetching enabled. We evaluate the benchmarks common in the work proposing a modern software prefetcher [38, 40] and SWOOP.

replicate address computation occupying core resources during critical inner loops. Address computation can be prohibitively costly for complex, nested data structures with a high number of indirections, which are the prime target for software prefetching and can be found in abundance in sequential general-purpose applications (e.g., SPEC CPU 2006). Hence, inspector-executor methods always consume extra instruction bandwidth, regardless of the benefit. In contrast, SWOOP reuses previously computed values and loaded data to reduce overhead and its execution is triggered and hidden by hardware stalls that would be pure performance loss otherwise.

Software prefetching [3, 21, 39, 73] lacks precision. Code containing complex control flow cannot rely on software-prefetch instructions inserted a few iterations ahead. The current condition guarding the delinquent load might not correspond to the condition several iterations ahead. Hence, the prefetch may not execute or always execute (if put outside the conditional) incurring increased instruction overhead and pollution of the cache. SWOOP *Access*-phases execute precisely those instructions that are required to reach the delinquent loads in each iteration.

Static instruction schedulers [2, 43, 48] attempt to hide memory latency, however scheduling is confined by register pressure and basic block boundaries. While predicated execution provides support for global scheduling (across branches), this technique introduces hardware complexity and is sensitive to the compiler's ability to statically predict the hot path. (Software-hardware techniques are covered in the following subsection.) Software pipelining, regarded as an advanced form of instruction scheduling, is restricted by dependencies between instructions and register pressure. Solutions based on software-pipelining provide techniques to handle loop-carried dependencies customized for their goal, e.g. thread-level parallelism DSWP+ [53, 58]. DSWP decouples single-threaded code into a critical path slice and an execution slice, which run separately on distinct threads.

While DSWP overlaps stalls with computation on a separate thread, SWOOP dynamically reorders execution and aims at entirely avoiding stalls, thus saving energy by reducing the processor's idle time. In SWOOP dependencies are handled non-speculatively, combining prefetch instructions with reuses of already computed values in Execute. This combination, in addition, reduces register pressure.

Software pipelining was used to address scheduling for long-latency loads [76], but the paper does not clarify whether critical loads nested in complex control flow can be targeted and is further constrained by the limited number of registers. Clairvoyance [69] includes compiler support to cluster deeply nested critical loads and combines loads and prefetch instructions to avoid register pressure. While techniques to handle unknown dependencies, branch merging, etc., build on Clairvoyance, SWOOP addresses limits that make previous transformations non-applicable on energy-efficient platforms. The software-hardware co-design handles register pressure and reduces instruction count overhead (through context remapping) and ensures dynamic adaptation (checkmiss). SWOOP thus achieves performance improvements that Clairvoyance is not capable of on the targeted platforms.

Overall, static approaches lack flexibility and adaptability to dynamic events and are severely restricted by register pressure. SWOOP tackles this problem by retrieving dynamic information and avoiding register spilling with frugal hardware support.

Hardware techniques aim either to avoid latency by fetching data early or to hide latency by identifying and performing other useful operations.

Several proposals aimed at avoiding latency use dynamically generated threads/slices that execute ahead in an attempt to fetch necessary data [10, 15, 68]. Others create a checkpoint on a miss and continue executing in an attempt to prefetch future data [19, 20, 70]. SWOOP takes a different approach and *executes ahead*, skipping code that is likely to stall for MLP-dense code. In this respect, SWOOP shares the

execution model of the UltraSparc Rock, an *execute-ahead, checkpoint-based* processor [14]. However, while Rock relies heavily on speculation-and-rollback for correctness and may re-execute if necessary, SWOOP eliminates both costly instruction re-execution *and* the need to maintain speculative state, because the compiler guarantees the correctness of its non-speculative execution.

The indirect memory predictor [81] is an inherently speculative prefetching technique, potentially sacrificing efficiency. In addition, it requires fixed offsets to be described ahead of time in hardware, reducing its flexibility and control that is available in SWOOP.

Hardware-only proposals are general and adaptive, but incur large energy costs for rediscovering information that is statically available. This insight was the driving force for abundant research work combining compiler-hardware techniques.

Software-hardware co-designs aim to reduce hardware complexity by using software to achieve high performance with lower overhead, e.g., *very long instruction word* (VLIW) architectures [22]. Unlike compiler techniques for VLIW, HPL PlayDoh and EPIC [26, 36, 37, 41], SWOOP compiler does not require hardware support for predicated execution, speculative loads, verification of speculation, delayed exception handling, memory disambiguation, no separate thread [54] and is also not restricted by instruction bundles. Compared to VLIW and its successors, it requires minimal modifications to the target ISA, uses contemporary IO pipelines and front-end and minimizes speculation to enable higher energy efficiency.

Generally, compiler assisted techniques with hardware support rely on statically generated entities that execute efficiently on customized architectures: Braid [71] runs dataflow subgraphs on lower complexity architectures to save energy, while Outrider [17] supports highly efficient simultaneous multithreading. Speculative multithreading executes pre-computation slices [56] with architectural support to validate speculations, relies on ultra-light-weight threads to perform prefetching [13, 18, 61] or requires hardware communication channels between the prefetching and the main thread [49, 53, 58]. CFD [64] requires an architectural queue to efficiently communicate branch predicates that are loaded early in advance. Other proposals, most notably *Multiscalar* [24, 66, 74], combine software and hardware to enable instruction level parallelism using compiler-generated code structures, i.e., tasks, which can be executed simultaneously on multiple processing units.

SWOOP, in contrast, does not require additional threads and is not speculative. Since SWOOP targets all delinquent loads, it is also readily applicable for hoisting branch predicates.

6 Summary of Contributions

Given the extraordinary research efforts over the past several decades to hide memory latency and the plethora of proposals, we review SWOOP's main advancements over the state-of-the-art and how it differs from previous research.

I). Unlike hardware decoupled access-execute techniques [25, 65], **SWOOP interleaves *Access* and *Execute* code within a single thread, changing this interleaving dynamically**. Thus, SWOOP abstracts the execution order from the underlying architecture and can run *Access* and *Execute* code either in-program-order or out-of-program-order.

II). SWOOP has access to a larger portion of the dynamic instruction stream by (1) *jumping* over *Execute* code that would stall; (2) continuing with independent future *Access* code, thus exposing more memory-level parallelism (MLP); and (3) eventually resuming the *Execute* code that was skipped and is now ready for execution, increasing instruction level parallelism (ILP). Thus, SWOOP does not run-ahead: It jumps ahead compared to the conventional dynamic instruction stream, skipping over code that is likely to stall and jumping to code that is likely to produce MLP.

III). SWOOP goes beyond software and hardware techniques designed to hide memory latency [4, 15, 16, 21, 33, 35, 39, 47, 56, 68, 73, 82–84] in the following ways:

SWOOP is non-speculative: SWOOP employs advanced compilation techniques to shift hardware complexity in software without the need of speculation, checkpoints and rollbacks. Instruction reordering is performed statically to account for known and unknown dependencies and control-flow is partly duplicated in *Access* to avoid the need for predication or expensive hardware mechanisms, which limited previous proposals. Thus, SWOOP bypasses major challenges that yielded previous work impractical, such as control flow processing (predication) and handling of speculative loads.

SWOOP is frugal: SWOOP hides its execution behind hardware stalls. All *additional* instructions that are executed exploit hardware stalls that would otherwise be pure performance loss. SWOOP exploits these *occurring* stall periods to jump ahead, instead of fetching data unconditionally in an attempt to prevent future stalls. SW-prefetching, in contrast, consumes extra instruction bandwidth, which is critical for efficient in-order processors.

SWOOP is precise: *Access* phases contain the necessary code (however complex, including control flow) executed ahead of time to reach delinquent loads and hide their latency. *Access* includes precisely those instructions that are required for each iteration to reach the target loads.

SWOOP is concise (non-superfluous): SWOOP reuses conditionals, addresses and loads, and anything that has already been computed in the *Access* phase, and thus avoids replicating instructions. Hence, SWOOP aims to minimize instruction re-execution by consuming in *Execute* data produced in *Access*.

SWOOP is general: SWOOP handles complex loops which cannot be targeted by standard techniques for instruction reordering (e.g. software pipelining).SWOOP includes support for non-linear indirect memory accesses, control- and memory dependencies, `while`, `do-while`, `go-to` loops, complex control-flow etc, and uses effective compiler techniques to achieve this without speculation.

SWOOP is adaptable: SWOOP adds the *chkmiss* instruction, which triggers the execution of *Access* phases, out-of-program-order, only when a long-latency miss occurs.

SWOOP provides benefits above HW-prefetching: SWOOP shows considerable benefits on top of HW-prefetching, while SW-prefetching is largely subsumed by HW-prefetches; SW-prefetch combined with HW-prefetch can be detrimental [44].

7   Conclusions

We propose SWOOP, a new, non-speculative, software-hardware co-design to achieve out-of-program-order execution and to reach high degrees of memory and instruction level parallelism. SWOOP relies on compiler support for offering independent instructions to be executed ahead-of-time, on-demand, driven by cache miss events, and with low-overhead hardware extensions to the typical in-order architecture consisting of: miss events communicated through a control-flow instruction and a novel, *context* register remapping technique to ease register pressure. As a compilation strategy, we introduce a software decoupled access-execute model that creates *Access* phases that can execute out-of-program-order with respect to *Execute* phases, without any need for speculation. A SWOOP core jumps ahead to independent regions of code to hide hardware stalls and resumes execution of bypassed instructions once they are ready. Through a combination of software knowledge through recompilation together with efficient hardware additions, the SWOOP core shows an average performance improvement of 34% while reducing energy consumption by 23% with respect to the baseline in-order core, competitive with an oracle in-order core with perfect last level cache.

8   Acknowledgements

References

[1] Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture. Seattle, WA, June 1990*, Jean-Loup Baer, Larry Snyder, and James R. Goodman (Eds.). ACM, 2–14. https://doi.org/10.1145/325164.325100

[2] Alexander Aiken, Alexandru Nicolau, and Steven Novack. 1995. Resource-Constrained Software Pipelining. *IEEE Trans. Parallel Distrib. Syst.* 6, 12 (1995), 1248–1270. https://doi.org/10.1109/71.476167

[3] Sam Ainsworth and Timothy M. Jones. 2017. Software prefetching for indirect memory accesses. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, 305–317. http://dl.acm.org/citation.cfm?id=3049865

[4] Manuel Arenaz, Juan Touriño, and Ramon Doallo. 2004. An Inspector-Executor Algorithm for Irregular Assignment Parallelization. In *Parallel and Distributed Processing and Applications, Second InternationalSymposium, ISPA 2004, Hong Kong, China, December 13-15, 2004, Proceedings*. 4–15. https://doi.org/10.1007/978-3-540-30566-8_4

[5] ARM. [n. d.]. ARM Cortex-A15 Processor. http://www.arm.com/products/processors/cortex-a/cortex-a15.php.

[6] ARM. [n. d.]. ARM Cortex-A7 Processor. http://www.arm.com/products/processors/cortex-a/cortex-a7.php.

[7] Manish Arora, Siddhartha Nath, Subhra Mazumdar, Scott B. Baden, and Dean M. Tullsen. 2012. Redefining the Role of the CPU in the Era of CPU-GPU Integration. *IEEE Micro* 32, 6 (2012), 4–16. https://doi.org/10.1109/MM.2012.57

[8] Mark Bohr. 2007. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *IEEE Solid-State Circuits Society Newsletter* 12, 1 (2007), 11–13. https://doi.org/10.1109/N-SSC.2007.4785534

[9] Shekhar Borkar and Andrew A. Chien. 2011. The future of microprocessors. *Commun. ACM* 54, 5 (2011), 67–77. https://doi.org/10.1145/1941487.1941507

[10] Trevor E. Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. 2015. The load slice core microarchitecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, Deborah T. Marr and David H. Albonesi (Eds.). ACM, 272–284. https://doi.org/10.1145/2749469.2750407

[11] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*. 52:1–52:12. https://doi.org/10.1145/2063384.2063454

[12] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *TACO* 11, 3 (2014), 28:1–28:25. https://doi.org/10.1145/2629677

[13] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. 1999. Simultaneous Subordinate Microthreading (SSMT). In *Proceedings of the 26th Annual International Symposium on Computer Architecture, ISCA 1999, Atlanta, Georgia, USA, May 2-4, 1999*. 186–195. https://doi.org/10.1109/ISCA.1999.765950

[14] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. 2009. Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's Rock Processor. In *Proceedings of the Annual International Symposium on Computer Architecture*. ACM, New York, NY, USA, 484–495. https://doi.org/10.1145/1555754.1555814

[15] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John Paul Shen. 2001. Dynamic speculative precomputation. In *Proceedings of the 34th Annual International Symposium on Microarchitecture, Austin, Texas, USA, December 1-5, 2001*. 306–317. https://doi.org/10.1109/MICRO.2001.991128

[16] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher J. Hughes, Yong-Fong Lee, Daniel M. Lavery, and John Paul Shen. 2001. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA 2001, Göteborg, Sweden, June 30-July 4, 2001*. 14–25. https://doi.org/10.1145/379240.379248

[17] Neal Clayton Crago and Sanjay J. Patel. 2011. OUTRIDER: efficient memory latency tolerance with decoupled strands. In *38th International Symposium on Computer Architecture (ISCA 2011), June 4-8, 2011, San Jose, CA, USA*. 117–128. https://doi.org/10.1145/2000064.2000079

[18] Michel Dubois and Yong Ho Song. 1998. *Assisted Execution*. Technical Report CENG 98-25. Department of EE-Systems, University of Southern California.

[19] James Dundas and Trevor N. Mudge. 1997. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *Proceedings of the 11th international conference on Supercomputing, ICS 1997, Vienna, Austria, July 7-11, 1997.* 68–75. https://doi.org/10.1145/263580.263597

[20] Richard James Eickemeyer, Hung Qui Le, Dung Quoc Nguyen, Benjamin Walter Stolt, and Brian William Thompto. 2009. Load lookahead prefetch for microprocessors. US Patent 7,594,096.

[21] Philip G. Emma, Allan Hartstein, Thomas R. Puzak, and Viji Srinivasan. 2005. Exploring the limits of prefetching. *IBM Journal of Research and Development* 49, 1 (2005), 127–144. http://www.research.ibm.com/journal/rd/491/emma.pdf

[22] Joseph A. Fisher. 1998. Very Long Instruction Word Architectures and the ELI-512. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*. 263–273. https://doi.org/10.1145/285930.285985

[23] Daniele Folegnani and Antonio González. 2001. Energy-effective issue logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA 2001, Göteborg, Sweden, June 30-July 4, 2001.* 230–239. https://doi.org/10.1145/379240.379266

[24] Manoj Franklin. 1993. *The multiscalar architecture*. Ph.D. Dissertation. University of Wisconsin Madison.

[25] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: decoupled supply-compute communication management for heterogeneous architectures. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, Milos Prvulovic (Ed.). ACM, 191–203. https://doi.org/10.1145/2830772.2830800

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

[27] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News* 34, 4 (2006), 1–17. https://doi.org/10.1145/1186736.1186737

[28] Andrew D. Hilton, Santosh Nagarakatte, and Amir Roth. 2009. iCFP: Tolerating all-level cache misses in in-order processors. In *15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14-18 February 2009, Raleigh, North Carolina, USA.* IEEE Computer Society, 431–442. https://doi.org/10.1109/HPCA.2009.4798281

[29] Andrew D. Hilton and Amir Roth. 2010. BOLT: Energy-efficient Out-of-Order Latency-Tolerant execution. In *16th International Conference on High-Performance Computer Architecture (HPCA-16 2010), 9-14 January 2010, Bangalore, India*, Matthew T. Jacob, Chita R. Das, and Pradip Bose (Eds.). IEEE Computer Society, 1–12. https://doi.org/10.1109/HPCA.2010.5416634

[30] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. 1996. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, USA, May 22-24, 1996*, Jean-Loup Baer (Ed.). ACM, 260–270. https://doi.org/10.1145/232973.233000

[31] Mitsuhiko Igarashi, Toshifumi Uemura, Ryo Mori, Hiroshi Kishibe, Midori Nagayama, Masaaki Taniguchi, Kohei Wakahara, Toshiharu Saito, Masaki Fujigaya, Kazuki Fukuoka, Koji Nii, Takeshi Kataoka, and Toshihiro Hattori. 2015. A 28 nm High-k/MG Heterogeneous Multi-Core Mobile Application Processor With 2 GHz Cores and Low-Power 1 GHz Cores. *J. Solid-State Circuits* 50, 1 (2015), 92–101. https://doi.org/10.1109/JSSC.2014.2347353

[32] Intel. 2010. Intel™ Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide. Nehalem. https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf.

[33] Alexandra Jimborean, Konstantinos Koukos, Vasileios Spiliopoulos, David Black-Schaffer, and Stefanos Kaxiras. 2014. Fix the code. Don't tweak the hardware: A new compiler approach to Voltage-Frequency scaling. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, David R. Kaeli and Tipp Moseley (Eds.). ACM, 262. https://doi.org/10.1145/2544137.2544161

[34] Roel Jordans and Henk Corporaal. 2015. High-level software-pipelining in LLVM. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2015, Sankt Goar, Germany, June 1-3, 2015*, Henk Corporaal and Sander Stuijk (Eds.). ACM, 97–100. https://doi.org/10.1145/2764967.2771935

[35] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. 2011. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, Rajiv Gupta and Todd C. Mowry (Eds.). ACM, 393–404. https://doi.org/10.1145/1950365.1950411

[36] Vinod Kathail, Michael Schlansker, and B Ramakrishna Rau. 1994. *HPL PlayDoh architecture specification: Version 1.0*. Hewlett Packard Laboratories Palo Alto, California.

[37] Vinod Kathail, Michael S Schlansker, and B Ramakrishna Rau. 2000. *HPL-PD architecture specification: Version 1.1*. Hewlett-Packard Laboratories.

[38] Muneeb Khan and Erik Hagersten. 2014. Resource conscious prefetching for irregular applications in multicores. In *XIVth International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2014, Agios Konstantinos, Samos, Greece, July 14-17, 2014.* IEEE, 34–43. https://doi.org/10.1109/SAMOS.2014.6893192

[39] Muneeb Khan, Michael A. Laurenzano, Jason Mars, Erik Hagersten, and David Black-Schaffer. 2015. AREP: Adaptive Resource Efficient Prefetching for Maximizing Multicore Performance. In *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015.* IEEE Computer Society, 367–378. https://doi.org/10.1109/PACT.2015.35

[40] Muneeb Khan, Andreas Sandberg, and Erik Hagersten. 2014. A Case for Resource Efficient Prefetching in Multicores. In *43rd International Conference on Parallel Processing, ICPP 2014, Minneapolis, MN, USA, September 9-12, 2014.* IEEE Computer Society, 101–110. https://doi.org/10.1109/ICPP.2014.19

[41] Jinwoo Kim, Rodric M. Rabbah, Krishna V. Palem, and Weng-Fai Wong. 2004. Adaptive Compiler Directed Prefetching for EPIC Processors. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '04, June 21-24, 2004, Las Vegas, Nevada, USA, Volume 1*, Hamid R. Arabnia (Ed.). CSREA Press, 495–501.

[42] Konstantinos Koukos, Per Ekemark, Georgios Zacharopoulos, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. 2016. Multi-versioned decoupled access-execute: the key to energy-efficient compilation of general-purpose programs. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 121–131. https://doi.org/10.1145/2892208.2892209

[43] Monica S. Lam. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, Richard L. Wexelblat (Ed.). ACM, 318–328. https://doi.org/10.1145/53990.54022

[44] Jaekyu Lee, Hyesoon Kim, and Richard W. Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. *TACO* 9, 1 (2012), 2:1–2:29. https://doi.org/10.1145/2133382.2133384

[45] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2013. The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing. *TACO* 10, 1 (2013), 5:1–5:29. https://doi.org/10.1145/2445572.2445577

[46] Sushil J. Louis. [n. d.]. CIGAR - Case Injected Genetic Algortihm. http://www.cse.unr.edu/~sushil/class/gas/code/cigar/ http://ecsl.cse.unr.edu/~sushil/class/gas/code/cigar/.

[47] Chi-Keung Luk. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA 2001, Göteborg, Sweden, June 30-July 4, 2001*, Per Stenström (Ed.). ACM, 40–51. https://doi.org/10.1145/379240.379250

[48] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. 1992. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, Oregon, November 1992*, Wen-mei W. Hwu (Ed.). ACM/IEEE, 45–54. https://doi.org/10.1109/MICRO.1992.696999

[49] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. 2003. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA'03), Anaheim, California, USA, February 8-12, 2003*. IEEE Computer Society, 129–140. https://doi.org/10.1109/HPCA.2003.1183532

[50] NASA. 1999. NAS Parallel Benchmarks. http://www.nas.nasa.gov/assets/pdf/techreports/1999/nas-99-011.pdf http://www.nas.nasa.gov/assets/pdf/techreports/1999/nas-99-011.pdf.

[51] Karthik Natarajan, Heather Hanson, Stephen W. Keckler, Charles R. Moore, and Doug Burger. 2003. Microprocessor pipeline energy analysis. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design, 2003, Seoul, Korea, August 25-27, 2003*, Ingrid Verbauwhede and Hyung Roh (Eds.). ACM, 282–287. https://doi.org/10.1145/871506.871577

[52] Satyanarayana Nekkalapu, Haitham Akkary, Komal Jothi, Renjith Retnamma, and Xiaoyu Song. 2008. A simple latency tolerant processor. In *26th International Conference on Computer Design, ICCD 2008, 12-15 October 2008, Lake Tahoe, CA, USA, Proceedings*. IEEE Computer Society, 384–389. https://doi.org/10.1109/ICCD.2008.4751889

[53] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic Thread Extraction with Decoupled Software Pipelining. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-38 2005), 12-16 November 2005, Barcelona, Spain*. IEEE Computer Society, 105–118. https://doi.org/10.1109/MICRO.2005.13

[54] Emre Özer and Thomas M. Conte. 2005. High-Performance and Low-Cost Dual-Thread VLIW Processor Using Weld Architecture Paradigm. *IEEE Trans. Parallel Distrib. Syst.* 16, 12 (2005), 1132–1142. https://doi.org/10.1109/TPDS.2005.150

[55] Vlad-Mihai Panait, Amit Sasturkar, and Weng-Fai Wong. 2004. Static Identification of Delinquent Loads. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 303–314. https://doi.org/10.1109/CGO.2004.1281683

[56] Carlos García Quiñones, Carlos Madriles, F. Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. 2005. Mitosis compiler: an infrastructure for speculative threading based on precomputation slices. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 269–279. https://doi.org/10.1145/1065010.1065043

[57] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. 2006. A Case for MLP-Aware Cache Replacement. In *33rd International Symposium on Computer Architecture (ISCA 2006), June 17-21, 2006, Boston, MA, USA*. IEEE Computer Society, 167–178. https://doi.org/10.1109/ISCA.2006.5

[58] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. 2004. Decoupled Software Pipelining with the Synchronization Array. In *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France*. IEEE Computer Society, 177–188. https://doi.org/10.1109/PACT.2004.10003

[59] B. Ramakrishna Rau. 1991. Data Flow and Dependence Analysis for Instruction Level Parallelism. In *Languages and Compilers for Parallel Computing, Fourth International Workshop, Santa Clara, California, USA, August 7-9, 1991, Proceedings (Lecture Notes in Computer Science)*, Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua (Eds.), Vol. 589. Springer, 236–250. https://doi.org/10.1007/BFb0038668

[60] Alberto Ros, Trevor E. Carlson, Mehdi Alipour, and Stefanos Kaxiras. 2017. Non-Speculative Load-Load Reordering in TSO. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 187–200. https://doi.org/10.1145/3079856.3080220

[61] Amir Roth and Gurindar S. Sohi. 2001. Speculative Data-Driven Multithreading. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01), Nuevo Leone, Mexico, January 20-24, 2001*. IEEE Computer Society, 37–48. https://doi.org/10.1109/HPCA.2001.903250

[62] Andreas Sembrant, Erik Hagersten, and David Black-Schaffer. 2014. Navigating the cache hierarchy with a single lookup. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 133–144. https://doi.org/10.1109/ISCA.2014.6853203

[63] Carlo H. Séquin and David A. Patterson. 1982. *Design and Implementation of RISC I*. Technical Report UCB/CSD-82-106. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/1982/5449.html

[64] Rami Sheikh, James Tuck, and Eric Rotenberg. 2015. Control-Flow Decoupling: An Approach for Timely, Non-Speculative Branching. *IEEE Trans. Computers* 64, 8 (2015), 2182–2203. https://doi.org/10.1109/TC.2014.2361526

[65] James E. Smith. 1984. Decoupled Access/Execute Computer Architectures. *ACM Trans. Comput. Syst.* 2, 4 (1984), 289–308. https://doi.org/10.1145/357401.357403

[66] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. 1995. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 22-24, 1995*, David A. Patterson (Ed.). ACM, 414–425. https://doi.org/10.1145/223982.224451

[67] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Michael Upton. 2004. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, Shubu Mukherjee and Kathryn S. McKinley (Eds.). ACM, 107–119. https://doi.org/10.1145/1024393.1024407

[68] Karthik Sundaramoorthy, Zachary Purser, and Eric Rotenberg. 2000. Slipstream Processors: Improving both Performance and Fault Tolerance. In *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000.*, Larry Rudolph and Anoop Gupta (Eds.). ACM Press, 257–268. https://doi.org/10.1145/356989.357013

[69] Kim-Anh Tran, Trevor E. Carlson, Konstantinos Koukos, Magnus Själander, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. 2017. Clairvoyance: look-ahead compile-time scheduling. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, 171–184.

http://dl.acm.org/citation.cfm?id=3049852

[70] Marc Tremblay and Shailender Chaudhry. 2008. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC® Processor. In *2008 IEEE International Solid-State Circuits Conference, ISSCC 2008, Digest of Technical Papers, San Francisco, CA, USA, February 3-7, 2008*. IEEE, 82–83. https://doi.org/10.1109/ISSCC.2008.4523067

[71] Francis Tseng and Yale N. Patt. 2008. Achieving Out-of-Order Performance with Almost In-Order Complexity. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China*. IEEE Computer Society, 3–12. https://doi.org/10.1109/ISCA.2008.23

[72] Vladimir Uzelac and Aleksandar Milenkovic. 2009. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings*. IEEE Computer Society, 207–217. https://doi.org/10.1109/ISPASS.2009.4919652

[73] Steven P. Vanderwiel and David J. Lilja. 2000. Data prefetch mechanisms. *ACM Comput. Surv.* 32, 2 (2000), 174–199. https://doi.org/10.1145/358923.358939

[74] T. N. Vijaykumar and Gurindar S. Sohi. 1998. Task Selection for a Multiscalar Processor. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 31, Dallas, Texas, USA, November 30 - December 2, 1998*, James O. Bondi and Jim Smith (Eds.). ACM/IEEE Computer Society, 81–92. https://doi.org/10.1109/MICRO.1998.742771

[75] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981.*, Seymour Jeffrey and Leon G. Stucki (Eds.). IEEE Computer Society, 439–449. http://dl.acm.org/citation.cfm?id=802557

[76] Sebastian Winkel, Rakesh Krishnaiyer, and Robyn Sampson. 2008. Latency-tolerant software pipelining in a production compiler. In *Sixth International Symposium on Code Generation and Optimization (CGO 2008), April 5-9, 2008, Boston, MA, USA*, Mary Lou Soffa and Evelyn Duesterwald (Eds.). ACM, 104–113. https://doi.org/10.1145/1356058.1356073

[77] Carole-Jean Wu, Aamer Jaleel, William Hasenplaugh, Margaret Martonosi, Simon C. Steely Jr., and Joel S. Emer. 2011. SHiP: signature-based hit predictor for high performance caching. In *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, Carlo Galuzzi, Luigi Carro, Andreas Moshovos, and Milos Prvulovic (Eds.). ACM, 430–441. https://doi.org/10.1145/2155620.2155671

[78] William A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News* 23, 1 (1995), 20–24. https://doi.org/10.1145/216585.216588

[79] Xin-Xin Yang. 2014. An Introduction to the QorIQ LS1 Family. Presentation slides. https://cache.freescale.com/files/training/doc/dwf/DWF14_APF_NET_T0162.pdf.

[80] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stéphan Jourdan. 1999. Speculation Techniques for Improving Load Related Instruction Scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture, ISCA 1999, Atlanta, Georgia, USA, May 2-4, 1999*, Allan Gottlieb and William J. Dally (Eds.). IEEE Computer Society, 42–53. https://doi.org/10.1109/ISCA.1999.765938

[81] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, Milos Prvulovic (Ed.). ACM, 178–190. https://doi.org/10.1145/2830772.2830807

[82] Weifeng Zhang, Dean M. Tullsen, and Brad Calder. 2007. Accelerating and Adapting Precomputation Threads for Efficient Prefetching. In *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*. IEEE Computer Society, 85–95. https://doi.org/10.1109/HPCA.2007.346187

[83] Chuan-Qi Zhu and Pen-Chung Yew. 1987. A Scheme to Enforce Data Dependence on Large Multiprocessor Systems. *IEEE Trans. Software Eng.* 13, 6 (1987), 726–739. https://doi.org/10.1109/TSE.1987.233477

[84] Craig B. Zilles and Gurindar S. Sohi. 2001. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA 2001, Göteborg, Sweden, June 30-July 4, 2001*, Per Stenström (Ed.). ACM, 2–13. https://doi.org/10.1145/379240.379246

[85] Victor V. Zyuban and Peter M. Kogge. 2001. Inherently Lower-Power High-Performance Superscalar Architectures. *IEEE Trans. Computers* 50, 3 (2001), 268–285. https://doi.org/10.1109/12.910816