Transcending hardware limits with software out-of-order execution

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

Department of Information Technology, Uppsala University, Sweden

ABSTRACT

Reducing the widening gap between processor and memory speed has been steering processors’ design over the last decade, as memory accesses became the main performance bottleneck. Out-of-order architectures attempt to hide memory latency by dynamically reordering instructions, while in-order architectures are restricted to static instruction schedules. We propose a software-hardware co-design to break out of the hardware limits of existing architectures and attain increased memory and instruction level parallelism by orchestrating coarse-grain out-of-program-order execution in software (SWOOP). On in-order architectures, SWOOP acts as a virtual reorder buffer (ROB) while out-of-order architectures are endowed with the ability to jump ahead to independent code, far beyond the reach of the ROB. We build upon the decoupled access-execute model, however, executed in a single superscalar pipeline and within a single thread of control. The compiler generates the Access and Execute code slices and orchestrates their execution out-of-order, with the support of frugal microarchitectural enhancements to maximize efficiency. SWOOP significantly improves the performance of memory-bound applications by 42% on in-order cores, and by 43% on out-of-order architectures. Furthermore, not only is SWOOP competitive with out-of-order cores which benefit from double-sized reorder buffers, but it is also considerably more energy efficient.

1. INTRODUCTION

Closing the performance gap between memory and processor, i.e., the memory wall [1], has been a long pursued goal and vivid research topic [2–15]. As this gap has been becoming 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 (IO) cores rely on static instruction schedulers [16–18] to hide long latencies by interleaving independent instructions between a load and its use. Nevertheless, such techniques 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 ROB). Yet, contemporary processors’ support for out-of-order execution cannot cope with misses in the entire cache hierarchy.

Figure 1: SWOOP: Interleaved Decoupled Access-Execute on a Single Core (In-Order or Out-Of-Order).

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

SWOOP is a decoupled access-execute approach, built upon coarse-grain Access and Execute phases generated and orchestrated in software, and executed (either in-order or out-of-order) in a single superscalar pipeline, within a single thread of control. This is in stark contrast to previous decoupled access-execute (DAE) approaches [19] that have separate pipelines and separate threads of control for Access code and Execute code, respectively.

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, both in order and on out-of-order cores.
In an in-order core, SWOOP emulates a virtual reorder buffer (ROB) by (1) jumping over Execute code that would stall; (2) continuing with independent future Access code, thus exposing more MLP; and (3) eventually resuming the Execute code that was skipped and is now ready for execution, increasing ILP (Figure 1).

In an out-of-order core, SWOOP cooperates with existing hardware structures and expands the reach of the ROB by jumping over stalling Execute code to future Access code. Thereby, SWOOP exposes farther independent instructions, otherwise not available for the out-of-order core. Compared to the conventional dynamic instruction stream, SWOOP does not run-ahead, but jumps ahead, bypassing stalling instructions and exposing independent code that is likely to produce MLP (Figure 1).

A key component in SWOOP is the compiler that orchestrates the execution assisted by targeted enhancements to the microarchitecture. The SWOOP compiler:

- Generates the Access and Execute code, decoupling the loop body as shown in Figure 2;
- Enables execution to adapt by jumping to Access phases of future iterations, bypassing a stalling Execute phase;
- Guarantees that Access phases can run and commit out-of-order with respect to Execute phases, safely and efficiently, without speculation, checkpoints, or rollbacks.

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

- **Context Renaming:** A novel register renaming approach that: (i) Enables unrestricted dynamic separation of an Access and its corresponding Execute, due to the interleaving of other Access phases corresponding to future iterations. **Context Renaming** ensures that registers written in each Access phase will be renamed and encapsulated in a unit denoted as Context. (ii) Manages dependencies between Contexts. (Section 3.1)
- **Chkmiss:** A simple and efficient mechanism to signal an early warning regarding upcoming stalls in Execute, with a new take on an old idea: informing memory operations [20]. A chkmiss instruction determines the status of any of the 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. The right side of Figure 2 shows an example where three additional Access phases are executed (A1, A2, and A3) before returning to run the Execute phase E0 (that corresponds to the first Access phase A0). (Section 3.2)
- **Early Commit of Loads (ECL):** Access phase loads are allowed to commit before their data is returned form the memory system, as soon as it is determined that they can cause no exception. This allows whole Access phases to commit without blocking in both the in-order cores and OoO cores. (Section 3.3)

SWOOP goes beyond software and hardware techniques designed to hide memory latency [3,7,21,32] in the following ways:

- **SWOOP is frugal:** It exploits occurring stall periods to execute ahead, instead of fetching data unconditionally in an attempt to prevent future stalls.
- **SWOOP is precise:** Access contains the necessary code (however complex, including control flow) executed ahead of time to reach delinquent loads and hide their latency.
- **SWOOP is concise (non-superfluous):** Aims to minimize instruction re-execution by consuming in Execute data produced in Access (computations and fetched data).

The SWOOP compiler and code transformations are outlined in Section 2 and the architectural enhancements (Section 3) as follows: Section 3.1 details the mechanism for renaming registers and managing contexts, Section 3.2 presents the chkmiss instruction, and Section 3.3 the out-of-order commit of loads. The proposal is thoroughly evaluated in Section 4 placed in context and compared with state of the art techniques. Section 5 Finally we conclude in Section 6.

2. **SWOOP COMPILER**

The SWOOP compiler decouples critical loops in Access and Execute phases, following a technique that melds program slicing [33] and decoupled access-execute [25]. Access phases are allowed to execute out-of-program-order (earlier, before their designated time in the original program order), while Execute phases are kept in program order, as illustrated in Figure 2. One Access - Execute pair is generated per loop body. Access is built by hoisting loads [2] address computation and the required control flow, following the use-def chain of instructions [25,33]. Access binds values to registers and Execute consumes these data and performs all other computations (see Figure 2 and left side of Figure 3), without duplicating unnecessary instructions. Decoupling provides a clear and natural point in the code to introduce the informing memory operations. The border between Access and Execute is marked by a chkmiss instruction, which guides the execution flow. Chkmiss indicates whether any of the loads in Access incurred a miss in the last level cache, potentially yielding a stall in Execute. Upon a miss, execution surrenders control flow to the alternative execution path.

Typically, however, we would choose to hoist only delinquent loads in an Access phase if such information is available, e.g., via profiling or static compile time analysis [34].
C/C++ code

for (i=0; i<max; i++)
  b[i] = x[i] * n->y[i];

Swoop IR code with single Access (A)

{0;
  loop:
    L1 = load y[i];
    L2 = load x[L1];
    L3 = load a[i];
    if chkmiss(L1,L3) goto AltPath_A;
    prefect x[L1];
    if chkmiss(prefetch) goto AltPath_B;
    store L2 + L3, &b[i];
    i++;
    if (i<max) goto loop;
  AltPath: /
  if (i+1>=max) goto loop_E;
  tmp=i;

Swoop IR code with multi-Access (A, B)

and may-alias (L2:x->y[i],store:&b[i])

{0;
  loop:
    L1 = load y[i];
    L2 = load x[L1];
    L3 = load a[i];
    if chkmiss(L1,L3) goto AltPath_A;
    prefect x[L1];
    if chkmiss(prefetch) goto AltPath_B;
    store L2 + L3, &b[i];
    i++;
    if (i+1>=max) goto loop_E;
    tmp=i;
    AltPath_A:
    tmp;  
    if (i<M && i<max) goto loop_B;
    AltPath_B:
    tmp;  
    if (i<M && i<max) goto loop_A;
    i++;  
    loop_A:
    store L2 + L3, &b[i];
    i++;
    if (i<M && i<max) goto loop_B;
    if (i+1>=max) goto loop_E;
    tmp=i;
    AltPath_A:
    tmp;  
    if (i<M && i<max) goto loop_B;
    AltPath_B:
    tmp;  
    if (i<M && i<max) goto loop_A;
    i++;  
    loop_B:
    L1 = load y[i];
    L2 = load x[L1];
    L3 = load a[i];
    if (i+1>=max) goto loop_E;
    tmp=i;
    AltPath_A:
    tmp;  
    if (i<M && i<max) goto loop_B;
    AltPath_B:
    tmp;  
    if (i<M && i<max) goto loop_A;
    i++;  
    loop_A:
    prefetch x[L1];
    i++;
    if (i<M && i<max) goto loop_B;
    i++;  
    loop_B:
    store L2 + L3, &b[i];
    i++;
    if (i<M && i<max) goto loop_E;
    tmp=i;
    AltPath_A:
    tmp;  
    if (i<M && i<max) goto loop_B;
    AltPath_B:
    tmp;  
    if (i<M && i<max) goto loop_A;
    i++;  
    loop_B:

Figure 3: Swoop: Compile-Time Code Transformations: Single Access (left), Multiple Access Phases With May-Aliasing Memory Accesses (right).

which consists of the Access phases of the following N iterations and the corresponding N Execute phases as shown on the left-side of Figure 3.

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 dependences. This is achieved by:

**Effective handling of known dependencies:** Known read-after-write (RAW) dependencies are constrained only in 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 are prefetched in Access and safely loaded in Execute, illustrated with the pair (L2:x->y[i],store:&b[l]) on the right of Figure 3. Prefetching transforms long latencies into short latencies: if the load-store pair did not alias, the correct data was prefetched in L1; in 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.

One would assume that the hardware disambiguation and memory dependence prediction [35] available on out-of-order architectures would be sufficient to handle the may-aliasing pointers without special treatment from the compiler. This is not the case, since Swoop enables out-of-program-order execution by software, yet, from the hardware perspective, this is a correct execution of the program. Thus, the architecture is unaware that loads in Access phases are executed out-of-program-order with respect to stores in other Execute phases. We note here that the prefetch can be substituted by advanced mechanisms such as speculative memory bypassing (SMB) [36], NoSQ [37], and renaming based techniques [35,38,41] but this is out of the scope of this paper and is left for future work.

**Multiple access phases.**

Normally, the Swoop compiler hoists loads in Access and sinks their use in Execute, ensuring separation for performing other useful work in the interim. However, when delinquent loads depend on other delinquent loads (Figure 3) Swoop requires L1, either via a control or a register dependency, the miss penalty is paid in the Access phase. Under these circumstances, the compiler splits the old Access phase into multiple Access phases (A and B in Figure 3 and Figure 4) and allocates different dependents to the different Access phases, separating the dependent loads. Hence, if one of the leading loads misses, the whole Access phase does not stall. A chkmiss is inserted after each Access phase, so that execution can proceed along any alternative execution paths depending on which Access phase incurred the miss.

**Multi-threaded code.**

The compile-time code decoupling as described above is readily applicable on multi-threaded applications within synchronization boundaries, as Swoop transforms the code of a single thread. The compiler can freely reorder instructions with respect to other threads within data-race-free (DRF) [42] regions of code (e.g., data parallel OpenMP loops, the code within a task, section, etc.). We aim to ex-
3. SWOOP ARCHITECTURAL SUPPORT

3.1 Context Register Renaming

Register-file Access-Execute communication is necessary to maintain much of the performance of the original optimized code. SWOOP, therefore, only relies on software prefetching when the compiler is unable to guarantee that no dependencies exist.

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 phases, the allocated registers of each Access phase now have to be maintained until they get consumed by their corresponding Execute phase. An alternative would be to have the compiler solve this issue through advanced register allocation, with the result of increased register pressure and likely subpar performance. Instead, we propose a novel register renaming technique, based on execution contexts, that alleviates the burden for additional register allocation and exposes more registers to the software.

The key observation for an efficient renaming 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.

One can draw an analogy between SWOOP contexts and register windows such as those in the EPIC architecture [43]. SWOOP contexts are “virtual” windows provided solely via register renaming. Similarly to register windows, SWOOP contexts are driven by software. But, whereas register windows are automatically managed as a stack via call and return instructions, SWOOP contexts are managed by setting the active context with two functions, CTX++ and CTX=number (typically number=0), invoked in the alternative execution path. We implement these functions with explicit instructions.

The program-order (non-SWOOP execution) context is CTX 0. No SWOOP-related renaming activity takes place during non-SWOOP execution. During SWOOP execution contexts are managed in a FIFO-like fashion. Each Access phase in the alternative execution path increments CTX. Returning to a previous context (either for another Access phase, or to the in-order execution of the Execute phases) 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. CTX management is shown on the left side of the mappings of the previous Map with the next. In a merge, lower-context renamings cannot replace higher-context renamings (which represent future mappings that occurred out of order). CTX=0 simply sets the active map to 0. When this is done by the last Execute phase that returns to non-SWOOP execution, the current Map replaces Map 0.

In more detail, merge works as follows. During the execution of a phase in context CTX, a number of new, native (to the context) renamings are established. These are protected from being overwritten by renamings coming from lower contexts. A number of other renamings were also transferred to CTX, from the previous context CTX−1 in the last merge. These are also new (for the execution of the current phase) but they are not native to CTX, hence not protected in this context. All the new renamings (both the transferred from CTX−1 and the new, native to CTX) are “pushed” to the next context CTX+1, when execution moves to the higher context (e.g., with a CTX++). They cannot replace renamings that were already there and are protected (i.e., native to the higher context).

CTX management takes place in the front-end of the pipeline just after the decode stage(s), in the renaming stage(s). For the architectures we examine in our evaluation (Section 4), we estimate that two to four additional rename stages are needed for cloning and merging of context maps (depending on copy and merge bandwidth). These rename stages add to the branch misprediction penalty but for the regions of codes we are targeting, branch misprediction is not an issue. Further, as shown in a slew of industry patents [44,50] the extra rename stages can be disabled and skipped in non-SWOOP execution imposing no performance  

Figure 5: SWOOP Context Register Renaming. Example is for In-Order Cores.
penalty on non-SWOOP code.

This new renaming scheme is compatible both with in-order cores, which normally do not perform any renaming, and out-of-order cores, which require register renaming to avoid false dependencies.

**In-order cores:** SWOOP renames only in the alternative execution path (not on normal execution). Each architectural register is renamed *only once* when it is first written in an *Access* phase. *Execute* phases do not rename. The key observation is that we only need to rename registers when we intermix *Access* and *Execute* phases belonging to different SWOOP contexts. In an in-order core, no register renaming 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 renaming than that which is found in OoO cores. Figure 5 shows this case.

Since *Execute* phases do not rename, the only case where the full merge mechanism (as described above) is needed is with multi-*Access* B phases during a second sweep of the contexts (from CTX0 to CTX5). To eliminate this complexity, and simplify merging of Maps in the in-order cores, the compiler enforces writes to different sets of registers between A and B phases. This way only a single *Access* phase version can cause a register to be renamed avoiding conflict with “future” mappings of the same register. In a merge of Map, with Map+1 the latest round of renamings in Map, are simply copied in bulk to Map+1 without further checking or bookkeeping for protection of native renamings. The restriction on compilation limits the reuse of registers between *Access* versions but has limited impact in practice as the *Access* phases are small and use a limited number of registers, see Table 2.

**Out-of-order cores:** In contrast to in-order cores, OoO cores rename continuously, in both *Access* and *Execute* phases. This means that CTX++ causes a Map merge even in *Execute* phases. In the OoO case the compiler optimization for the merge does not apply, since the *Execute* phases require the full merge mechanism.

### 3.2 Chkmiss

Chkmiss 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 [51, 52] and refined to detect a last level cache (LLC) miss in one cycle [53]. We encode the presence of an LLC cache line in the TLB as a 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 N, TargetPC where TargetPC points to the alternative execution path.

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

**Debugging.** SWOOP execution based on chkmiss instructions depends on dynamic events. As this may complicate debugging, a “TakenNot-Taken” vector can be maintained for the last few chkmiss instructions and made available for inspection in a debugging mode.

### 3.3 Early Commit of Loads

SWOOP out-of-program-order execution of *Access* phases can cause a vast number of instructions to be executed between the time of a delinquent load and its use. In a conventional in-order, stall-on-use core a delinquent load would reside at the head of a commit buffer until the load has been serviced by the memory hierarchy. This would require either a large commit buffer or cause a SWOOP core to stall once the buffer becomes full. Similarly, in an OoO core, a delinquent load at the head of the reorder buffer (ROB) blocks commit, which then causes the ROB to fill up and stall execution.

An alternative solution that we employ in SWOOP is to do *early commits of loads* (ECLs). We commit a delinquent load as soon as we are sure that the load cannot cause an exception, i.e., once the TLB access has been made and the loaded address has been verified. This enables the expedient commit of *Access* phases even when they cause a large number of misses. An ECL resides in the MSHR, which causes the correct register to be updated once the data is returned by the memory hierarchy.

Early commits of loads can cause a slight increase in the response time of exceptions and interrupts as the data requested by the ECL have to be received and written to the register before switching context.

**Memory Model.**

As discussed previously, SWOOP safely reorder instructions —with respect to other threads— within data-race-free (DRF) [42] regions of code and as long as the region of interest does not cross synchronization points. In this context, early commit of loads (for race-free data) cannot affect the memory consistency model as the reordering cannot be observed [54]. Thus, SWOOP can handle e.g., data parallel OpenMP codes and programs adhering to SC-for-DRF semantics as defined in contemporary language standards (C++11, Java, and the latest C standard), irrespective of the underlying consistency model.

However, in the presence of *races* (i.e., without explicit DRF guarantees) early commit of loads can violate memory ordering constraints as for example load→load required by TSO [55] and SC [56]. In this case, SWOOP can only be compatible with consistency models that fully relax ordering (weak ordering, release consistency [57], ARM, SPARC relaxed memory order—RMO—Intel IA 64, HP Alpha, and IBM PowerPC) where memory fences must be used to enforce ordering. In a SWOOP core, fences order ECL loads by waiting for their completion (register write) and, as
stated, the SWOOP compiler does not reorder across fences.

4. EVALUATION

Simulation Environment. We use the Sniper Multi-Core Simulator \[58\] to evaluate this work. We modify the cycle-level core model \[59\] to support the SWOOP processor. Power and energy estimates are calculated with McPAT \[60\] version 1.3 in 28 nm. For our baselines, we chose two efficient processor implementations. The ARM Cortex-A7 \[61\] is an extremely energy-efficient design with support for dual-issue superscalar in-order processing. On the high-end, we compare to the ARM Cortex-A15 \[62\], 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) even when employing hardware prefetcher, that can be difficult even for OoO cores, yielding low performance. We select a number of benchmarks, from SPEC2006CPU \[63\], CIGAR \[64\] and NAS benchmark suite \[65\], shown in Table 2. The primary criterion for our selection is MPKI; however, we made an effort to diversify benchmarks to: \(i\) stress our methodology with 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), \(iii\) have dependent delinquent loads for multiple Access phases (e.g., lbm, mcf, soplex), and \(iv\) unknown dependencies that are handled with prefetching (e.g., soplex). For each benchmark, Table 2 also 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 renaming 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.

Evaluated SWOOP, Software, and Hardware Solutions. To evaluate the success of SWOOP we compare a number of potential solutions:

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

**IO-perfectL3** a fictitious in-order, stall-on-use core with a perfect L3 cache, i.e., L3 accesses always hit.

**IO-UNROLLED** is a software-only solution based on standard compile-time transformations in which the original loop-bodies identified by SWOOP have instead been (forcefully) unrolled and executed on the IO, without any SWOOP code generation or hardware support.

**SW-prefetch** SW-prefetch is a software prefetch solution taken from the work of Khan et al. \[66\],\[67\]. SW-prefetch, uses profiling to identify delinquent loads to prefetch, and manually inserts prefetch instructions at the appropriate distance. We port the codes to our infrastructure and use prefetch to the L3 instructions. We use SW-prefetch to substitute optimal compiler prefetching.

Runahead Runahead is an in-order implementation of \[10\] in our simulation infrastructure. We compare against Runahead to show that i) re-execution is costly in both performance and energy and ii) for the benchmarks we use it is important to execute further down the dynamic stream to find MLP.

OoO 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 comprises the support for multiple access phase (in the presence of dependent delinquent loads) and for early load commit. SWOOP in-order has an additional pipeline stage to support context register renaming, while SWOOP out-of-order employs a variable length pipeline where three additional pipeline stages are used for register renaming during SWOOP execution (see Section 3.1). This is reflected by the variable branch penalty shown in Table 1.

4.1 In-Order Performance

Figure 6 shows speedups normalized to the in-order, stall-on-use core (IO). The in-order SWOOP achieves significant performance improvements (42% on average) compared to IO and outpaces even the OoO in some cases. In particular, SWOOP outperforms the OoO when running cigar (5% faster), mcf (18% faster), and on average when no hardware prefetching is used (1.5% faster). 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. Runahead is only showing a speedup over IO for mcf and hurts performance in all other cases (5% slower on average).

For two of the benchmarks lbm and sphinx3, SWOOP achieves speedup over the IO, but not nearly the speedup of the OoO. Lbm is primarily compute-bound, limiting the overall benefit of this technique to the delay caused by the cache hierarchy. For sphinx3, the issue is that while there is a large number of delinquent loads in the application, each contributes a very small portion to the total application delay, limiting total improvement.

For libquantum SWOOP achieves significantly better performance than IO and reaches half the speedup of the OoO core. Libquantum is a very tight loop, hence any instruction overhead has a non-negligible impact. Part of the optimization opportunities are hidden by the OoO core, or, OoO can reach sufficiently far to cover the existing latency.

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

Out-of-Order

Core | IO | Runahead | SWOOP | OoO | SWOOP
--- | --- | --- | --- | --- | ---

| u-Architecture | 1.5 GHz, 2-way superscalar | 1.5 GHz, 3-way superscalar |
| ROB | - | - | - | 32 | 64 |
| Phys. Reg | 32x32 32x32 | 96x64 64/64 | 96/96 160/128 |
| Branch Penalty | 7 7 8 | 15 15 15 | 15/18 15/18 |
| Execution units | 2 int/br., 1 mul, 1 fp, 1 ld/st | 3 int/br., 1 mul, 2 fp, 2 ld/st |
| MSHRs | 8 |
| L1-D | 32 KB, 8-way LRU |
| L2 cache | 256 KB, 8-way LRU, 8 cycle |
| L3 cache | 4 MB, 16-way LRU, 30 cycle |
| Main memory | 7.6 GB/s, 45 ns access latency |
| Technology node | 28 nm |

Table 1: Simulated Microarchitecture Details.

In terms of energy consumption we note that a 32-ROB SWOOP core is approximately equivalent to a 64-ROB OoO core, with a slight penalty on SWOOP because of its larger register file, and similarly a 64-ROB SWOOP core equivalent to an 128-ROB OoO core. For this reason we concentrate on their relative performance.

4.3 Out-of-Order Performance

Figure 8 shows speedups compared to out-of-order cores with 16 (left) and 32 (right) reservation stations and reorder buffer (ROB) sizes of 32 to 128 entries. The SWOOP core has an ROB size of 32. SWOOP outperforms even the four times larger OoO core with a 128 ROB (3% faster on average) when the reservation stations are limited to 16. SWOOP effectively uses its larger number of registers (without scaling any other part of the architecture) to effectively expand its ROB to quadruple size. With a more aggressively sized reservation station the SWOOP core performs on par with the twice as large OoO core (only 1% slower on average).

In general, the out-of-order SWOOP shows better performance due to significant re-execution of instructions.

4.2 Energy

Figure 7 shows the energy usage normalized to the IO core. SWOOP reduces the energy usage by 23% on average and is the only technique that shows any significant improvements compared to the IO core. 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.

Runahead has a 5% increase in energy expenditure compared to IO. This can be attributed to its limited performance improvements and wasted energy due to significant re-execution of instructions.

In terms of energy consumption we note that a 32-ROB SWOOP core is approximately equivalent to a 64-ROB OoO core, with a slight penalty on SWOOP because of its larger register file, and similarly a 64-ROB SWOOP core equivalent to an 128-ROB OoO core. For this reason we concentrate on their relative performance.

4.3 Out-of-Order Performance

Figure 8 shows speedups compared to out-of-order cores with 16 (left) and 32 (right) reservation stations and reorder buffer (ROB) sizes of 32 to 128 entries. The SWOOP core has an ROB size of 32. SWOOP outperforms even the four times larger OoO core with a 128 ROB (3% faster on average) when the reservation stations are limited to 16. SWOOP effectively uses its larger number of registers (without scaling any other part of the architecture) to effectively expand its ROB to quadruple size. With a more aggressively sized reservation station the SWOOP core performs on par with the twice as large OoO core (only 1% slower on average). Similarly, Figure 9 shows the speedup of a SWOOP core with an ROB size of 64 over OoO cores. The trends are similar, only SWOOP performs better, this time matching the performance of the OoO core with a 128-entry ROB even in the case of 32 reservation stations. With fewer reservation stations, SWOOP outperforms the 128-entry ROB OoO by a clear margin.

In general, the out-of-order SWOOP shows better perfor-
formance for more benchmarks than the in-order SWOOP. The out-of-order SWOOP shows similarly good results for cg, libquantum and mcf, but relatively better results for cigar and sphinx. Lbm and soplex is not improved by either the in-order or out-of-order version of SWOOP.

4.4 SWOOP vs. Unrolling vs. SW Prefetching

We compare SWOOP to two software-only techniques, forced unrolling and software prefetching. Figure 10 and Figure 11 display speedup normalized to the original execution on an in-order and on an out-of-order architecture, respectively, with hardware prefetcher enabled. The left side of the figures displays the runs performed with the hardware prefetching enabled and the right side with the hardware prefetching disabled (all normalized to the original execution with hardware prefetching).

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. Partly, the memory latency is hidden by the out-of-order architecture, reducing the optimization opportunities for SWOOP, however, our proposal is clearly outperforming the other software-only techniques. SWOOP is on par with the software techniques when running soplex, both on an in-order and on an out-of-order architecture. As explained before, 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.

Mcf is a clear winner on both architectures, with respect to both software techniques and to the original runs. Mcf is amenable to software techniques for hiding memory latency thanks to its irregular behavior, 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 using multiple access phases, 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, lqm exhibits merely a slight improvement on the in-order core, both with software prefetching and with SWOOP, while the out-of-order core is able to hide its memory latency without any SWOOP support. However, SWOOP, being concise, is competitive and does not harm when run on the out-of-order architecture, whereas software prefetching significantly degrades performance. SWOOP reuses many of the instructions required for computing the addresses for early loading and prefetching, while software prefetching duplicates such computations in different iterations.
4.5 Effectiveness of Early Commit of Loads

Figure 12 shows the speedup achieved when supporting early commit of loads (ECL). The results are normalized against the respective baseline (in-order and out-of-order SWOOP) without ECL support. ECL improves the performance with 4% and 8% on average for the in-order and out-of-order SWOOP, respectively, and as much as 24% for cigar on the out-of-order SWOOP core. Mcf shows an increased importance of supporting ECL on out-of-order architectures when hoisting more loads into the Access phase, three loads (mcf.3) compared to a single load (mcf.1), and with more complex Access phases (mcf.multiaccess). ECL enables resources to be released by continuing committing instructions, succeeding in fetching more instructions and exposing more ILP and MLP.

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 aims to hide latency by timely prefetching data deriving inspector-executor methods borrowed from automatic parallelization [21][22]. The compiler generates pre-computational slices grouping memory accesses [23][25] and initiates runahead execution with prefetching helper threads [24][26][29]. Hence, software prefetching techniques always consume extra instruction bandwidth, regardless of the benefit. In contrast, SWOOP execution is hidden inside hardware stalls that would be pure performance loss otherwise. While static approaches lack flexibility and adaptability to dynamic events, SWOOP tackles this problem by retrieving dynamic information with minimal hardware support, using the chkmiss instruction.

Software prefetching [30][32] 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 evaluation of the condition several iterations ahead. Hence, the prefetch may not execute or always execute (if put outside the conditional) incurring increased instruction overhead and polluting the cache. SWOOP Access-phases contain the required control flow and, there-

Figure 9: Out-of-Order Speedup (ROB=64, RS entries=32) versus a 32-entry ROB baseline.

Figure 10: In-Order Prefetch

Figure 11: Out-of-Order Prefetch
Therefore, execute precisely those instructions that are required to reach the delinquent loads in each iteration. Moreover, timely prefetching requires replicating address computation few iterations ahead, 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). SWOOP strives to eliminate redundant instructions and reuses as many of the computed values as possible (conditionals, addresses, and loads binding to registers—resorting to non-binding prefetching only if the target address may alias). In practice, accurate software prefetch is largely subsumed by hardware prefetchers and it has been shown that software prefetch on top of hardware prefetch can be detrimental [70]. SWOOP shows considerable benefits on top of (16-stream-strided) hardware prefetch.

**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 generate threads/slices that execute ahead in an attempt to fetch necessary data [3, 7, 71]. Other, create a checkpoint on a miss and continue executing in an attempt to prefetch future data [8–10, 72]. The former require additional hardware in the form of dynamic identification of threads and multi-thread execution support and the latter in the form of checkpoint and restore. These techniques also cause instructions to be executed and discarded that give rise to increased energy. SWOOP takes a completely different approach on miss events. It jumps instead of attempting to compute along the conventional dynamic instruction stream, skipping over code that is likely to stall (and only take up hardware resource), going for the code that is likely to produce MLP. Furthermore, SWOOP does not require large hardware structures and does not cause costly re-execution of instructions.

Latency can be hidden by increasing the instruction window to expose more independent instructions. However, increasing the instruction window is normally costly as many of the necessary hardware structures scale unfavorably. Several proposals attempt to limit the hardware cost by creating a second storage where dependent instructions of long latency loads are stored until they are ready to be executed [11–14] or by passing them on to a second pipeline [73–74]. Others, use check points and recovery [75, 76, 79] to early recycle hardware resources and, thus, reduce the total hardware requirements to support a large number of instructions in flight. Another approach to limit the hardware cost of large structures is through dynamic resizing. For compute intensive code with few long latency instructions many hardware resources can be scaled down to save energy while memory bound computations necessitates larger structures to expose MLP and independent instructions to execute [80–83]. SWOOP is an orthogonal approach that could be used in conjunction with such techniques that increase the window size to improve ILP for compute bound applications. However, SWOOP reduces the need for very large window sizes. Independent code can simply be reached by jumping forward in the program order.

Other hardware proposals describe cache access optimizations based on informing memory operations [20], hit-miss predictions [51, 52], providing direct access to data residing in different cache levels [53], or MLP aware replacement policies [15].

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 software to achieve high performance with lower overhead, e.g., *very long instruction word* (VLIW) architectures [84]. While VLIW compilers schedule instructions without regard to actual miss events, EPIC architectures and compilation methods employ predicated prefetching to specifically target delinquent loads [43, 85]. Generally, compiler assisted techniques with hardware support rely on statically generated entities that execute efficiently on customized architectures: Braid [86] runs dataflow subgraphs on lower complexity architectures to save energy, while Outrider [87] supports highly efficient simultaneous multithreading. Speculative multithreading executes pre-computation slices [23] with architectural support to validate speculations, relies on ultra-light-weight threads to perform prefetching [88, 90] or requires hardware communication channels between the prefetching and the main thread [2, 91, 92]. Other proposals, most notably *Multiscalar* [23, 95], 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. Speculative precomputation is shown to outperform OoO engines for memory bound codes, while OoO is suitable for compute intensive applications, on an Itanium processor [96].

**Out-of-Order Commit** for early commit of loads has been proposed by creating a checkpoint and then predicting the value to be returned from the load [97]. On a miss predicted value the checkpoint is restored and execution restarted. A more generic approach specifies six rules that need to be fulfilled before an instruction can be committed out-of-order [98]. SWOOP makes an effective use of
ECL with compiler support and with an efficient implementation. ECL is performed when we simply determine that the load cannot cause an exception (i.e., when next in line to be committed and once the TLB access has been made and the loaded address has been verified). Checkpointing and rollback is eliminated as the compiler guarantees correctness.

6. CONCLUSIONS

We propose SWOOP, a new technique to achieve out-of-program-order execution and to reach high degrees of memory and instruction level parallelism. SWOOP is a hardware-software approach, with modest changes to the typical (in-order or OoO core) architecture: communicating miss events through a control-flow instruction and a novel, context register renaming technique to ease register pressure, and early commit of loads to allow expedited commit of Access phases. As a compilation strategy, we introduce a software decoupled access-execute model that creates Access phases that can run “out-of-program-order” with respect to Execute phases, without any need for speculation. On in-order cores, SWOOP acts as a “virtual” reorder buffer that enables out-of-order execution in software, while on an out-of-order architecture SWOOP extends the reach of the physical reorder buffer by jumping ahead to independent regions of code.

7. REFERENCES

[29] C.-K. Luk, “Tolerating memory latency through software-controlled


