Transcending Hardware Limits with Software Out-of-order Processing

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

Abstract—Building high-performance, next-generation processors require novel techniques to enable improved performance given today's power- and energy-efficiency requirements. Additionally, a widening gap between processor and memory performance makes it even more difficult to improve efficiency with conventional techniques. While out-of-order architectures attempt to hide this memory latency with dynamically reordered instructions, they lack the energy efficiency seen in in-order processors. Thus, our goal is to reorder the instruction stream to avoid stalls and improve utilization for energy efficiency and performance.
To accomplish this goal, we propose an enhanced stall-on-use in-order core that improves energy efficiency (and therefore performance in these power-limited designes) through out-of-program-order execution. During long latency loads, the SWOOP (Software Out-of-Order Processing) core exposes additional memory- and instruction-level parallelism to perform useful, non-speculative work. The resulting instruction lookahead of the SWOOP core reaches beyond the conventional fixed-sized processor structures with the help of transparent hardware register contexts. Our results show that SWOOP demonstrates a 34% performance improvement on average compared with an in-order, stall-on-use core, with an energy reduction of 23%.

Index Terms—Compilation, decoupled access-execute, energy, memory level parallelism

1 Introduction

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

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

SWOOP is a hardware/software *decoupled access-execute* approach, built upon *Access* phases (containing loads and their requirements) and *Execute* phases (consuming data from *Access* and containing all computation). SWOOP relies on a compiler to generate *Access-Execute* phases, akin to the decoupled access-execute (DAE) model [1], [2] in which the target code region is transformed into (i) a heavily memory-bound phase (i.e., the access-

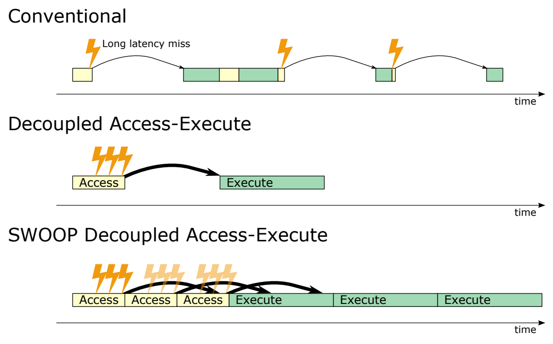

Fig. 1. Conventional execution stalls an InO processor when using the results of a long latency memory access. DAE clusters cache accesses and prefetches them ahead of time to reduce core stall time. SWOOP, on the other hand, reorders software across iterations to access distant, critical instructions, hiding load latencies with useful work.

phase for data prefetch), followed by (ii) a heavily compute-bound phase (i.e., the execute-phase that performs the actual computation). SWOOP operates on a much finer granularity and prefers loads over prefetches. *Access* - *Execute* phases are orchestrated in software guided by runtime information and executed in a single superscalar pipeline, within a single thread of control (Fig. 1).

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

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

- A novel register remapping approach (a lightweight

• *Trevor E. Carlson, Kim-Anh Tran, Alexandra Jimborean, Konstantinos Koukos, Magnus Själander, and Stefanos Kaxiras are with Uppsala University. Magnus Själander is also with the Norwegian University of Science and Technology (NTNU).*

Manuscript received September 13, 2016; revised February 17, 2017.

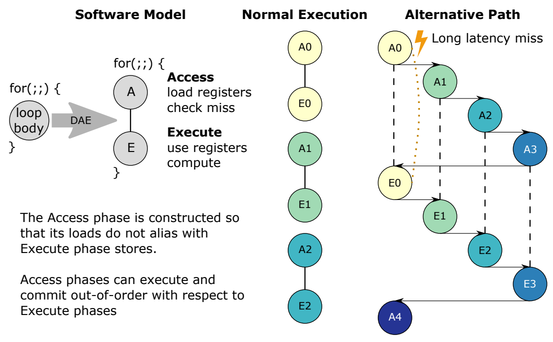

Fig. 2. The SWOOP software and execution model.

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

- A simple yet efficient check-miss mechanism (chkmiss) to quickly inform the core about long-latency loads in *Access*, akin to informing memory operations [3], to adapt and prevent memory stalls (Sec. 3.2).

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

2 SWOOP Software Components

The SWOOP core is a hardware-software co-designed processor that benefits from software knowledge to enable the hardware to execute past the conventional dynamic instruction stream. The compiler transforms loops to safely and efficiently cluster memory accesses into *Access* regions, similar to SW-DAE [1], [2], which has been successful in decoupling a large number of complex, general-purpose applications. Unlike SW-DAE, loads without read-after-write dependencies are hoisted to *Access*, and only memory dependent loads are replaced by safe-prefetches. Together with hardware enhancements, SWOOP executes useful, non-speculative instructions to improve energy efficiency and performance of memory-bound applications.

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

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

3 SWOOP Architectural Support

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

3.1 Context Register Remapping

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

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

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

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

3.1.1 *Implementing SWOOP Contexts*

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

Non-SWOOP execution does not require any remapping since the code is executed in its original program order.

SWOOP execution is governed by three simple rules: (1) For each new *Access* phase the context is incremented (`CTX++`) and the corresponding bit of the context remapping vector is set to zero. Upon the first write to an architectural register, the corresponding bit gets set and a new physical register name is pushed to the head of the FIFO. Future writes to the same architected register in this phase do not generate new mappings. The new physical register (i.e., the head) becomes the new effective architectural to physical map and is used until it gets remapped in a future *Access* or *Execute* phase. (2) When returning to the first *Execute* phase (E_0) the original mapping of A_0 is to be used. This is equivalent with the oldest register in the remapping FIFO

TABLE 1
Microarchitecture. OoO has (*) 2 int, 1 int/br., 1 mul, 2 fp, and 2 ld/st.

	In-Order			OoO
Core	InO	Runahead	SWOOP	
u-Arch	1.5 GHz, 2-way superscalar			
ROB	-	-	-	32
RS	-	-	-	[16/32]
Phys. Reg	32/32	32/32	96/64	64/64
Br. Penalty	7	7	8	15
Exec. Units	1 int, 1 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			

(the element that would get popped). Upon returning to CTX=0, the effective map is changed from the youngest (i.e., the head) to the oldest (i.e., the tail) physical register in the FIFO. (3) For each new *Execute* phase where the corresponding bit in the remapping vector is set a physical register is popped from the FIFO and the new register at the tail is the new effective map.

3.2 Chkmiss

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

3.3 Early Commit of Loads

SWOOP out-of-program-order execution of *Access* phases can cause a vast number of instructions to be executed. A delinquent load would reside at the head of a commit buffer causing a SWOOP core to stall once the buffer becomes full. To avoid stalling, we employ early commits of loads (ECLs) [4] to commit loads that cannot cause an exception.

4 Evaluation

We use the Sniper Multi-Core Simulator to evaluate this work. We modify the cycle-level core model to support the SWOOP processor. Power and energy estimates are calculated with McPAT version 1.3 in 28 nm. The processor parameters are shown in Table 1. SWOOP is evaluated on *demanding* workloads with frequent misses even when employing hardware prefetcher, from the SPEC2006CPU, CIGAR, and NAS benchmark suites.

We compare a number of potential solutions in performance and energy efficiency. InO is a two-wide in-order, stall-on-use core. InO-Unroll and InO-Perf-L3 are extensions to the in-order core with software unrolling and a perfect L3 cache. SW-prefetching [5][1] and in-order Runahead [6] were also evaluated. Finally a 2-wide moderately-aggressive out-of-order core (OoO) with and without software unrolling is compared to our (SWOOP) implementation.

1. Received benchmarks from authors, those in common evaluated.

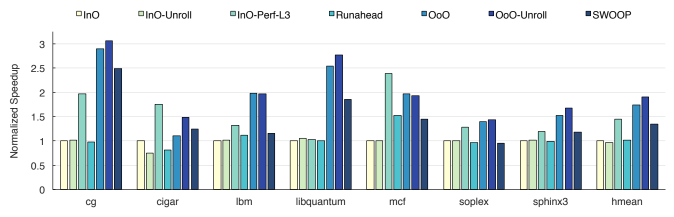

Fig. 3. Speedup comparison.

4.1 Performance

Fig. 3 shows speedups normalized to the in-order, stall-on-use core (InO). SWOOP achieves significant performance improvements (34% on average) compared to InO and outperforms the base OoO when running cigar (12% faster). The results clearly show that forcing the loops to be unrolled on an InO is not beneficial and instead hurts performance (3% slower on average) due to register spilling, motivating the necessity of the SWOOP approach. Runahead is only showing a speedup over InO for lbm and mcf. One of the main reasons for this is the limited reach of runahead execution compared to SWOOP. On average, InO runahead is able to execute an additional 21 instructions per runahead phase (with six containing load operations). As misses can be sparse, maximizing the instruction distance travelled is necessary to expose additional long latency loads. SWOOP brings *Access* phases together, reducing the amount of work needed to discover long latency loads, improving performance for more applications than is possible with runahead. For sphinx3, SWOOP achieves speedup over the InO, but not nearly the speedup of the OoO. While there is a large number of delinquent loads in the application, each contributes a very small portion to the total application delay, limiting total improvement. For libquantum, SWOOP achieves significantly better performance than InO and reaches half the speedup of OoO. Libquantum contains a very tight loop, hence any instruction overhead has a non-negligible impact. Some of the optimization opportunities are hidden by the OoO core, or, OoO can reach sufficiently far to cover the existing latency. Finally, for soplex, SWOOP fails to achieve any speedup over the InO but does not significantly hurt performance (0.6% slower). Soplex suffers from the same problem as sphinx3: a single delinquent load in the *Access* phase (responsible for only 7% of the memory slack), exacerbated by a lower chkmiss firing rate of 18%.

4.2 Energy

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

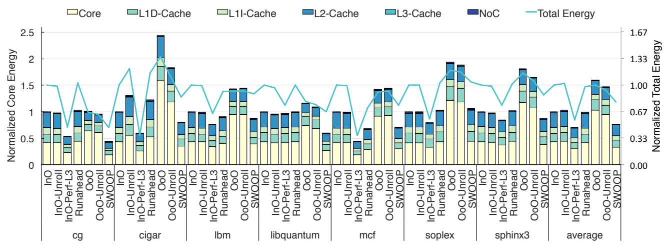

Fig. 4. Energy efficiency comparison. DRAM energy is included in the total.

3x higher for OoO compared to InO. Runahead increases energy expenditure by 5% compared to InO due to significant re-execution of instructions, which limits performance improvements and wastes energy.

4.3 SWOOP, Unrolling, and SW Prefetching

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

5 Related Work

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

Basic Block Execution (BBE) hardware [7] identifies blocks of instructions to execute in parallel and builds dependence graphs in hardware for data sharing. Speculation can be used to enable more parallelism, reducing efficiency. In contrast, SWOOP proposes a disciplined execution with software-controlled block splitting to allow for multiple *Access* phases to execute out-of-order with low hardware overhead and without speculation. Since BBE allows for a more general splitting of a program into basic blocks, the hardware is also much more complex to ensure correctness (handling dependences between blocks, speculation, etc). Furthermore, SWOOP does not pre-define the scheduling of *Access* and *Execute* phases, but uses dynamic information to orchestrate the execution. SW-DAE [1], [2] generates *Access* and *Execute* phases, but, unlike SWOOP, *Access* prefetches data and, therefore, introduces significant instruction count overhead. SW-DAE is suitable for OoO engines, but code rematerialization becomes critical on InO machines. SWOOP on the other-hand prioritizes instruction reordering rather than rematerialization and triggers the OoO execution only upon a miss. Control-Flow Decoupling (CFD) [8] clusters long latency loads used in branches, similar to *Access*, and communicates their values through architectural queues to the consumer phase, i.e., *Execute*. The number of target loads is considerably less for CFD compared to SWOOP. Dundas et al. [6] employ runahead execution to prefetch data, but requires instruction re-execution and Ozer et al. [9] uses a speculative thread, requiring recovery. Finally, Multiscalar processors [7] speculate aggressively on data dependencies and replicate execution units. SWOOP uses compile-time analysis to avoid both additional speculation and instruction re-execution to reduce additional overheads.

6 Conclusion

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

References

[1] A. Jimborean et al., “Fix the code. Don’t tweak the hardware: A new compiler approach to voltage-frequency scaling,” in *CGO’14*.
[2] K. Konstantinos et al., “Multiversioned decoupled access-execute: the key to energy-efficient compilation of general-purpose programs,” in *CC’16*.
[3] M. Martonosi et al., “Informing memory operations: Providing memory performance feedback in modern processors,” in *ISCA’96*.
[4] T. J. Ham et al., “DeSC: Decoupled supply-compute communication management for heterogeneous architectures,” in *MICRO’15*.
[5] M. Khan et al., “A case for resource efficient prefetching in multicores,” in *ICPP’14*.
[6] J. Dundas et al., “Improving data cache performance by pre-executing instructions under a cache miss,” in *ICS’97*.
[7] G. S. Sohi et al., “Multiscalar processors,” in *ISCA’95*, 1995.
[8] R. Sheikh et al., “Control-flow decoupling: An approach for timely, non-speculative branching,” *IEEE Trans. Computers*, 2015.
[9] E. Ozer et al., “High-performance and low-cost dual-thread VLIW processor using weld architecture paradigm,” *IEEE TPDS*, 2005.