Simulating Energy-Efficient Hardware
The Software Out-of-order Processor

Simon Lövgren
Abstract

Simulating Energy-Efficient Hardware: The Software Out-of-order Processor

Simon Lövgren

The modern trends for technology scaling are not extremely bright. The cost of transistors have leveled off recently, effectively halting the ability to put additional transistors on a chip for the same price. In addition, Dennard Scaling, what has allowed for switching additional transistors whilst scaling to smaller nodes is slowing significantly. This thesis, with focus on the hardware, proposes an enhanced stall-on-use in-order core hardware/software co-design which improves performance and energy efficiency by allowing out-of-program-order execution through allowing the hardware and software to communicate with one another -- allowing the hardware to make dynamic decisions on how to direct execution flow to expose additional memory- and instruction level parallelism.

The results are very promising where we see an increase in both performance (up to 3.7x speedup) and energy efficiency (up to 59% increase). While additional work is needed to evaluate the extent of the benefits across a wide range of applications, SWOOP looks to be a good option for energy efficiency without compromising performance for memory-bound applications with MLP.
Acknowledgements

Thanks to Alexandra Jimborean and Trevor E. Carlson for being helpful supervisors.
Thanks to Trevor E. Carlson for much needed insight into hardware design and general help with the simulation platform.
Thanks to Kim-Anh Tran for helping solve software-side bugs and providing much needed insight into LLVM IR and assembly.
Contents

1 Introduction ........................................ 7
  1.1 Goals ............................................. 8

2 Background ....................................... 8
  2.1 Decoupled access-execute (DAE) .................. 9
  2.2 Out-of-Order (OoO) ............................... 9
  2.3 Hardware loops ................................... 9
  2.4 Loop unrolling ................................... 9
  2.5 Prior work ....................................... 11
  2.6 Tools- and frameworks ......................... 12
    2.6.1 Sniper Multi-Core Simulator .................. 12
    2.6.2 Intel Pin ..................................... 12
    2.6.3 McPAT ...................................... 12
    2.6.4 LLVM/Clang ................................. 12

3 Theory .......................................... 12
  3.1 SWOOP software model ........................... 12
    3.1.1 SWOOP-sessions ............................. 13
  3.2 SWOOP execution model ........................... 15
    3.2.1 SWOOP Contexts .............................. 16
    3.2.2 Difference from prior work .................. 16
  3.3 Architectural Support ............................ 16
    3.3.1 Context register remapping .................. 16
    3.3.2 Hardware-unrolling of access and execute phases .. 17
    3.3.3 Software/Hardware communication .............. 17
    3.3.4 Stack interface ............................. 18
    3.3.5 Early commit of loads (ECL) ................. 19

4 Implementation .................................. 19
  4.1 Implementing SWOOP Contexts ................... 19
  4.2 Software/Hardware communication ............... 21
  4.3 The SWOOP alternate execution path in hardware .. 22
    4.3.1 Determining unroll factor .................... 25
    4.3.2 Hardware unrolling .......................... 25
    4.3.3 SWOOP physical registers .................... 26
    4.3.4 False contexts .............................. 26
    4.3.5 Handling of writes in access ................ 27
  4.4 Stack usage analysis ............................ 28
    4.4.1 Static analysis in Pin ....................... 28

5 Evaluation ..................................... 29
  5.1 Benchmarks ...................................... 30
  5.2 Results ......................................... 30
    5.2.1 Performance .................................. 31
    5.2.2 Energy ....................................... 32
    5.2.3 Sweeping hardware and software parameters .. 33
    5.2.4 SWOOP-registers versus performance- and energy 35
  5.3 Discussion ...................................... 36
1 Introduction

The modern trends for technology scaling are not extremely bright. The cost of transistors have leveled off recently, effectively halting the ability to put additional transistors on a chip for the same price. In addition, Dennard Scaling, what has allowed for switching additional transistors whilst scaling to smaller nodes is slowing significantly. It is becoming very difficult to increase the number of transistors on a chip because of physical and practical reasons, one being the thermal energy produced [1–3].

Because of this it is only possible to switch on a certain number transistors at a time, and it becomes more important to switch these in newer and smarter ways in order to preserve energy efficiency while increasing performance [3]. One way to increase performance is to improve its efficiency; doing the same amount of work with fewer transistors. One attempt at solving this problem is a new, novel software/hardware co-design called SWOOP.

SWOOP (Software Out-of-Order Processor) [4] is a novel software/hardware co-design that is able to hide memory latency and achieve increased MLP (memory level parallelism) which results in an overall increase in ILP (instruction level parallelism). It does so by successfully targeting single-threaded, entangled memory and computation dependencies which traditionally are difficult to prefetch or parallelize. The hardware/software co-design is built upon access- and execute phases, where the access phase contains loads and their requirements and the execute phase consumes data from access with the aim to contain the majority of the computation. This software-approach is called decoupled access-execute (DAE). In SWOOP, an access- and execute phase pair is referred to as a SWOOP-session. SWOOP then adds additional architectural support, enabling an alternate execution path that is able to exploit this increased MLP to not only increase performance, but also increase energy efficiency. This alternate execution path is simply the path taken during execution on a SWOOP-core in comparison to the original in-program-order execution path, which executes instructions in the order they occur. This is further described in Sections 2 and 3.2.

This thesis proposes a version of the SWOOP software/hardware co-design, with focus on the hardware, wherein the software is able to communicate with the hardware as to allow the hardware to make dynamic decisions on how to direct execution flow. To enable this behavior an additional setup phase has been added to the SWOOP-session, which is executed once when entering the SWOOP-session. The SWOOP-core is, with the communicated information, able to interleave access and execute dynamically depending on available hardware resources and resource usage of the software (such as number of available registers). This abstraction of execution order from the underlying architecture allows for executing code in both in-program-order and out-of-program-order depending on the circumstances.

The revised SWOOP architecture provides:

• a lightweight register remapping approach that enables separation of an access phase and its corresponding execute phase. This allows for an out-of-program-order execution of access- and execute phases, increasing MLP and ILP. Context remapping ensures registers written in access is properly forwarded to the correct execute phase, independent of any other
access phase executed in-between. A set of registers written to by an access phase, stored for the future corresponding execute phase, is called a (SWOOP) Context.

- a simple way for the software to communicate with the hardware, allowing the hardware to properly decide the execution path to be taken for each decoupled access-execute section in a program. The hardware aims to increase MLP by executing as many access phases in series as possible and ILP by executing as many execute phases in series as possible depending on available resources. The communication between software- and hardware consists of informing the hardware of access- and execute phases as well as resource usage in these (see Sections 3.3.3 and 4.3.3). Additionally the hardware is able to inform the software to allocate the appropriate amount of resources (for example stack space) for each SWOOP-session (see Section 3.3.4).

1.1 Goals

There are two main goals with this thesis. The first is to enable automated communication between the hardware (simulator) and software (compiler) – with focus on the hardware side (HW-SWOOP) – in order to more accurately model the interactions between hardware and software. This is in contrast to the previous version of SWOOP (see Section 2.5) [4], where each benchmark had to be manually tailored after the SWOOP compiler transformations. This goal consists of evaluating an appropriate method of communication between hardware and software as well as properly setup the communication.

The previous implementation contains a feature called Check-Miss which is invoked before each hardware-unroll and decides whether to continue unrolling or not (described in Section 2.5). This feature was implemented using a small cache to keep track of the information needed to make this decision. The second goal is to reduce this complexity and the need for an extra cache by always unrolling as far as possible instead of performing the Check-Miss after each unroll.

The automated software implementation is then to be evaluated by comparing the resulting work with some of the latest hardware implementations, such as in-order and out-of-order cores, using custom microbenchmarks and a benchmark from the SPEC CPU2006 [5] benchmark suite.

2 Background

The conventional execution of a program is the so-called in-program-order execution, where the program is executed in the order which it is written. In-Order cores execute in this manner and employ stall-on-use, which means that if the program wants to consume data that is not yet available (currently being fetched from memory) the core will simply halt execution and wait until the data is available. There are also In-Order cores that employ stall-on-miss where they instead stall already at a cache-miss instead of waiting until use of the data. For long-latency loads, when data is not available in the cache, the core may have to wait a very long time (see Figure 1, conventional execution). InO cores are,
however, very energy efficient in regard to computation and are most commonly found in embedded devices.

One way to tackle this stall-on-use problem is to overlap multiple memory accesses (memory level parallelism, MLP) and hide the latency with useful computation by reordering instructions (instruction level parallelism, ILP). There are multiple software- and hardware techniques that exploit this, of which some are described in this section.

2.1 Decoupled access-execute (DAE)
A software technique called decoupled access-execute (DAE) [6] reorders the code during compilation, where the target region of the program is transformed into an access phase and an execute phase. The access phase contains all loads and their requirements, making it heavily memory-bound. The execute phase consumes the data loaded by access and contains all computation, with the goal to make it heavily compute-bound. By issuing multiple load instructions in this way, multiple long-latency loads may be in flight at the same time (MLP) and the program has to wait for – at most – one long-latency load before the execute phase is able to execute (see Figure 1).

2.2 Out-of-Order (OoO)
Out-of-Order (OoO) execution is a technique where independent instructions are interleaved with loads and their use. This technique exists as a type of core (hardware) that is able to dynamically re-order program instructions to hide memory latency (see Figure 1). While really good at hiding many types of cache misses, it comes at a cost in design complexity, area budget, and energy efficiency – since the additional hardware resources needed to support dynamically analyzing and reordering instructions are extensive and power hungry.

2.3 Hardware loops
Hardware looping is a technique that can reduce the number of instructions needed in a loop body, by allowing the hardware to perform the looping of code instead of the software. This is applicable where the number of iterations in a loop is known beforehand and allows the removal of the branch instruction and the loop variables in a software loop, effectively removing the branch cost as well as the cost of an instruction. When paired with other techniques such as post-incrementing load/store, this technique can significantly reduce the number of instructions in a loop – increasing performance and energy efficiency. [7, p20-21]

2.4 Loop unrolling
Loop unrolling is a simple but highly effective compiler technique, used to increase ILP. The compiler is able to unroll loops a small amount of times to help with scheduling code with increased ILP [8, p735]. With a simple for-loop as an example:
Long-latency load handling using different techniques. Derived from [4, fig.1].

```plaintext
for(i = 0; i < N; i++) {
    LoopBody(i);
}
```

the compiler can unroll the loop to the following:
where the first (unrolled) loop is able to fill the processor pipeline with more instructions before reaching a branch-instruction, increasing the ILP of the loop. The second loop is added to catch all cases where \( 4 \nmid N \), that is \( N \) is not a multiple of 4. *The example was taken from [8, p735].*

2.5 Prior work

SWOOP (Software Out-of-Order Processing) is a novel software/hardware co-design that takes advantage of both out-of-program-order and DAE in order to exceed conventional hardware limits. By harnessing these techniques, it is able to hide memory latency while achieving both MLP and ILP and still maintain, or even increase, the energy efficiency of an InO-core. The main target of the SWOOP core is embedded- or mobile devices, where energy efficiency is very important – especially for devices with limited energy budget such as batteries, or power budget such as thermal constraints.

SWOOP is, like DAE, built upon access- and execute phases, where target regions in the code are transformed into a memory-bound region and a compute-bound region. The hardware relies on the software to indicate such regions, for which a pair of an access phase and an execute phase is called a *SWOOP-session*. This allows the SWOOP core to dynamically interleave access and execute within a single thread where memory- and control dependencies are entangled such that it is difficult to use traditional prefetching or parallelization (see Figure 1). [4]

The architectural support, essential for efficiency, of this co-design provides:

- *Context register remapping*, a lightweight register remapping approach that enables separation of an access phase and its corresponding execute phase. This is what allows for an out-of-program-order execution of access-and execute phases, increasing MLP and ILP. It ensures registers written in access are properly forwarded to the correct execution phase, independent of any other access phase executed in-between. A set of registers written to by an access phase, stored for the future corresponding execute phase, is called a (SWOOP) *Context*.

- *Check-miss* is a simple yet efficient mechanism for timely control flow changes to alternative execution paths. It alerts the core of upcoming long-latency loads allowing the core to adapt and prevent memory stalls. This feature only exists in this previous work on SWOOP, whereas the version described by this thesis uses a different (simplified) approach to achieve control-flow changes in SWOOP described in Section 3.2.
Support for *early commit of loads* (ECL) [9], which allows the access phase loads to commit before their data is returned from memory – thus the access phase can commit without blocking.

### 2.6 Tools- and frameworks

#### 2.6.1 Sniper Multi-Core Simulator

Sniper Multi-Core Simulator forms the base for the SWOOP hardware simulator. It is a high-speed, hardware-validated x86 simulator that both allows for fast simulation and trading simulation speed for accuracy, enabling flexible simulation options (for example only simulating regions of interest in detail). Although Sniper allows for multi-core simulation, only single-core architectures are used in evaluation since SWOOP focuses on single threads.

The simulator is built as two major parts, the front-end and the back-end. The back-end is responsible for simulating the effects (such as instruction cost, branch prediction, etc.) of program instructions on a specified microarchitecture. The front-end is responsible for instrumenting the executing program, for example by simulating the computational effects of instructions not available in the host architecture. [10, 11]

#### 2.6.2 Intel Pin

Intel Pin [12] is a tool for instrumenting executables as they are executing. The use ranges from static analysis of executables to dynamic analysis of a running executable to altering the execution path of the executable. This is what the simulator front-end and the majority of the SWOOP hardware is built with. It allows instrumentation of basic blocks, routines and instructions by allowing for attaching callback-functions before or after the target and when the function is called it can alter the instruction stream before it is passed to the host processor.

#### 2.6.3 McPAT

McPAT is a tool for calculating the power consumption of a simulated program and is used to generate the energy statistics in this thesis [13].

#### 2.6.4 LLVM/Clang

LLVM (Low Level Virtual Machine) and Clang, together with LLVM-passes for SWOOP, are used to compile the SWOOP-enabled binaries [14].

### 3 Theory

The SWOOP hardware/software introduces not only a set of new concepts, but also targeted enhancements of the x86 InO microarchitecture essential for efficiency.

#### 3.1 SWOOP software model

The SWOOP software model consists of transforming a target region (a target loop) in a program much like DAE, by separating loads and execution. SWOOP
does so by hoisting loads and their requirements, creating a memory-bound access phase, and a compute-bound execute phase which consumes data from the access phase. Additionally the SWOOP software model adds a setup phase that is placed before the access phase and serves to communicate with the hardware, making it aware of the resource requirements for the upcoming access and execute.

The software model is illustrated in Figure 2.

3.1.1 SWOOP-sessions

The transformed code-region, consisting of a setup phase and a software-loop containing an access- and execute phase is referred to as a SWOOP-session.
Figure 2: The SWOOP Execution model. In this example, the main loop is to execute 4 iterations, the hardware is able to store 3 contexts resulting in a total of 6 hardware-unrolls. The hardware will unroll the access phase as many times as possible as it does not allow for early exit in this phase, since it does not perform any calculations here – including loop conditionals. When unrolling the execute phase, however, the software is allowed to exit early (when the software loop is finished) as the loop conditionals are evaluated during this phase. Derived from [4, fig.2]
3.2 SWOOP execution model

The SWOOP execution model aims to execute as many consecutive access phases as it can before executing their respective execute phases. This is to increase MLP while in the access phase, hide memory latency before executing the execute phase and increase ILP overall for a SWOOP-session.

Up until SWOOP detects a SWOOP-session, it executes as a normal InO core without any special features. When SWOOP receives the setup phase instruction (the first instruction of a SWOOP-session), it is informed by the software that it has entered a SWOOP-session and it is also informed about the resource use in the upcoming access phase. SWOOP then calculates, based on the upcoming resource usage, how many SWOOP Contexts it can create which in turn determines how many consecutive access phases will be executed. It then allows the program to continue as usual until it detects it is entering the access phase of the SWOOP-session.

When in the access phase   SWOOP keeps track of the loads performed and stores all written registers to a Context. When it detects the end of the access phase, SWOOP has two possible paths to follow. Which path it takes depends on whether it can create an additional Context or not.

If SWOOP can create an additional Context: 
In this case, SWOOP increases its Context counter – effectively saving the just-created Context and starting a new – then executes an additional access phase to populate the additional Context.

If SWOOP can not create more Contexts: 
In this case, SWOOP resets its Context counter – effectively saving the just-created Context and selecting the first (oldest) one – then allows the program to continue to the execute phase.

When in the execute phase   SWOOP lets the program execute as normal and does not intervene until it detects the end of the execute phase. When at the end of the execute phase, SWOOP determines whether the software loop is trying to execute another iteration or if it is finished looping (see Section 4.3.2) as well as whether there are any SWOOP Contexts left to be consumed.

If the software loop is not finished and there are Contexts left: 
In this case, SWOOP merges in the next stored Context, effectively transparently updating the registers as if an access phase had just executed such that the software is not able to tell it, in fact, did not. SWOOP then proceeds to execute another (corresponding) execute phase.

If the software loop is not finished, but there are no Contexts left: 
In this case, SWOOP allows the program to continue as normal. When the software loop jumps back to the beginning of the loop body, it again enters the access phase and the cycle repeats.

The software loop is finished and there are no Contexts left: 
In this case, SWOOP allows the program to continue executing as normal –
leaving the SWOOP region executing as a normal InO-core again.

The software loop is finished and there are Contexts left:
In this case, SWOOP has executed additional access phases than execute phases resulting in false Contexts (see Section 4.3.4), that is Contexts that were not consumed by any execute phase. The reason for this state lies in how this version of the hardware implements the alternate execution path, which is further described in Sections 3.3.2 and 4.3.

When this case occurs, SWOOP simply clears all remaining Contexts before allowing the program to continue execution as normal – leaving the SWOOP region executing in-program-order again.

3.2.1 SWOOP Contexts

A Context consists of an access and execute phase in the form of context register remapping. Communication between an access- and execute phase is first and foremost through the register file. Context remapping is further described in Section 3.3.1.

3.2.2 Difference from prior work

As mentioned in Section 2.5, the prior version of SWOOP utilized a feature called Check-miss to determine whether or not to redirect execution flow while within a SWOOP-session. This feature has been removed in favor of always executing as many access/execute pairs as possible depending on available resources (such as number of physical registers available), in order to implement the alternate execution path (alternate to In-Order execution). The SWOOP-mode of the core is, however, only active when within a SWOOP-session – triggered by the setup phase – as mentioned in section 3.2 where it otherwise runs in normal (In-Order) mode.

3.3 Architectural Support

In order for SWOOP to function, it requires architectural support that consists not only of functional- but also efficient enhancements of the microarchitecture. The SWOOP core is based on a in-order, stall-on-use, low-power mobile-class core and comes with the following enhancements:

3.3.1 Context register remapping

SWOOP employs register remapping based on execution contexts, that exposes additional registers to the software. The aim is to use register-file communication between access and execute phases to maintain as much of the performance in the optimized code as possible. This remapping scheme is also efficient as it only needs to be active while in a SWOOP-session in the alternate execution path.

Each architectural register is remapped only once when it is first written in an access phase, while no additional remapping is performed in execute. As no remapping is needed when outside of a SWOOP-session and no additional remapping is required within an access or execute, or even between an access and execute in the same Context, this remapping scheme requires fewer updates
than in conventional OoO-cores. This is mainly due to that OoO-cores need to always enable register renaming in order to allow for dynamically reordering instructions, whereas SWOOP only need to enable Context remapping when within a SWOOP-session.

This context register remapping is enabled by an additional pipeline stage, resulting in a one cycle longer branch penalty. [4]

3.3.2 Hardware-unrolling of access and execute phases

In order to allow the alternate execution path, SWOOP employs loop unrolling (Section 2.4) together with hardware loops (Section 2.3) targeting the access and execute individually. The hardware takes note of where the access or execute phase begins. When an access or execute phase reaches its end, the hardware makes a decision according to the SWOOP execution model (Section 3.2) as to whether it should set the instruction pointer to the beginning of the current access or execute phase, or allow the software to continue execution of the program.

3.3.3 Software/Hardware communication

In order for the hardware to know how many access and execute phases it is able to run consecutively (hardware-unroll), it has to be informed by the software about how much resources will be used in the access phase, where the access phase begins, where the access phase ends and the execute phase begins (the border between access- and execute), and where the execute phase ends.

In order to inform the hardware of the resource usage in the access phase, i.e. how many registers will the access phase write to, a setup region is used in addition to access and execute. This region is located just before the software loop (before the access phase) and typically consists of a single instruction that serves two purposes. First and foremost, it informs the hardware of the number of registers used in the access phase such that it can properly decide the alternate execution path depending on available resources (in other words how many times it should unroll access and execute). However, it also serves to inform the software (or possibly the hardware, see Section 7.5) of how much extra stack-space needs to be allocated for spilled- and temporary variables if existing (see Section 3.3.4).

In addition to the setup phase, the hardware needs to know the address bounds of the SWOOP-session within the executable. Without this information, the hardware is not able to determine what instruction address it should jump to in order to unroll access and execute. These are single instructions located at the borders of the access and execute phase. access_begin marks where the access phase begins, execute_end marks where the execute phase (and also the software-loop) ends. Additionally the access- and execute phase share the execute_begin marker as an indication of both the end of access and beginning of execute. As these instructions are used by the front-end of the SWOOP-core, they have little impact on performance since they are handled by the front-end. In real x86-hardware, one could use prefix instructions to annotate start and end regions of the code. These additional x86-prefixes would be straight-forward to implement, adding only minimal size to the binary. An alternative to using individual marker-instructions within the hardware-unrolled loops, it could be
possible to hoist these instructions to the setup phase, invoking them only once per execution of the SWOOP-session. This could, however, complicate the communication- and setup phase for cases such as when there are multiple exit-points from an access phase, as multiple addresses would need to be mapped for a single access phase. This would also require additional resources such as a small cache or SWOOP-specific lookup-buffer for keeping track of which instruction addresses comprise the bounds of the different phases, thus probably impacting energy efficiency to some degree (see Section 7.6).

### 3.3.4 Stack interface

As the hardware decides how many unrolls is possible, depending on available resources, the stack memory needed may differ. If the hardware-unrolls $n$ times, the stack entry need to accommodate data from all $n$ access phases in addition to callee save and execute phase data – as illustrated in Figure 3. This is because the access phases are not allowed to overwrite one another. Additional memory for this extra stack utilization can, for example, be allocated using `alloca` (in software) by having the set-up instruction return the number of unrolls to be performed.

As the program moves to the execute phase, it will use $A0$ with $E0$ as the stack for the current context. As the execution-phase writes to the execution-phase area of the stack, after $E0$ is finished the execution-phase area of the stack will have been updated to that of $E1$. After $E0$ is finished, the offset for the access phase stack is modified such that it points to $A1$ and the context in regard to the stack is now ready to execute $A1/E1$.

![Stack frame](image)

Figure 3: Layout of the stack for three hardware unrolls, where the necessary stack space for the three access phases has been allocated. The allocated space for the execute phase does not need any special handling, as it is updated directly by the execute phase – thus can be re-used in-place. Which access phase stack area is used for each unroll works like a sliding window, and is denoted by the blue color.

This is possible due to that writes to the access phase area of the stack by the execute phase would break the constraints of the software-side of SWOOP. If this was allowed, the stack would also need to be reduced for each execution-phase as writes would be interleaved. This would probably reduce performance significantly as more memory-operations would be necessary.
3.3.5 Early commit of loads (ECL)

Loads initiated in the access phases are allowed to commit before data has been returned from the memory architecture, but only as soon as it is certain the load cannot cause an exception. This means the TLB (Translation Lookaside Buffer) access has been made and the loaded address has been verified.

ECL is needed to avoid excess stalling as SWOOP’s out-of-program-order execution of access phases is able to execute a vast number of instructions.

4 Implementation

The communication method decided upon- and implemented uses META-files to pass information between the software- and hardware, as it allows for easy prototyping- and modification depending on the requirements discovered during implementation and testing. One embodiment of this implementation would use x86 prefix instructions to communicate Swoop state with a low overhead.

The compiled executable is then to be launched in the modified SWOOP-Sniper simulator. As the simulator is a hybrid functional simulator, the front-end and back-end are launched separately and then connected via a named pipe through which they communicate. This is managed by a launcher called run-sniper, which is also responsible for parsing the META-file which is translated into launch-parameters for the front-end.

The SWOOP execution model has been implemented as a front-end to the Sniper simulator and is based on the Intel Pin framework for instrumenting running executables. The front-end is responsible for invoking the alternate execution path of SWOOP. In addition to the SWOOP front-end, a SWOOP static analyzer is available for analyzing the maximum possible stack usage of SWOOP access regions. The analyzer is strictly used for gathering statistics and is not implemented as an architectural support feature.

The Sniper back-end, responsible for simulating the effects of instructions on the configured microarchitecture (such as instruction cost, branch prediction, access latency and more), has been slightly modified as to allow the SWOOP front-end to query the hardware for the number of physical SWOOP-registers. This is used during the first setup-region as to allow proper calculation of unroll trip-count for each upcoming access- and execute phase.

The communication between the front-end and back-end is based on the SIFT protocol, which comes as part of the Sniper Multi-core simulator and is the default communication method between front-end and back-end. The SWOOP front-end is thus implemented as part of the SIFT-Recorder Pin Tool of the simulator front-end and the simulator back-end is equipped with a SIFT-reader to interpret the traces sent via the named pipe.

For a high-level overview of the implementation, see Figure 4.

4.1 Implementing SWOOP Contexts

The Context remapping is implemented using a context remapping vector and context renaming FIFO. The FIFO contains mappings from the architectural registers to physical registers and the vector contains as many bits as the maximum supported Contexts.
During normal execution, register remapping is not needed as the program is executed in-order.

During SWOOP alternate path execution, the core enables remapping and in regard to Contexts follow these rules: (1) For each new access, the context is incremented (CTX++) and the corresponding bits in the vector are set to 0. At the first write to an architectural register during the access phase, the bit corresponding to the register is set to 1. This prevents any future writes to the same register during the same Context to be remapped (each register written to is only remapped once per access phase). The register name is then pushed onto the FIFO, becoming the new head of the queue. The head now becomes the new effective architectural to physical map and will be in use until it gets remapped in a future access or execute phase. (2) When entering the first execute phase \(E_0\) it expects the original mapping of its corresponding access phase, \(A_0\) (the oldest register remapping in the FIFO, thus the first to be popped). The core then returns to the first context by resetting the context counter and changes the effective map from the youngest (head) to oldest (tail). (3) For each new execute phase, for each corresponding bit in the remapping vector that has been set a physical register is popped from the FIFO and the new register at the tail becomes the effective map. The Context is thus transparently merged with the context of the previous access-execute pair, just as if it had executed in its original program order. (4) When the core detects the software loop wants to exit, if there still are Contexts left that has not been consumed by an execute
phase, it discards (clears) these contexts and invalidates all instructions issued as with a branch misprediction.

The Context register remapping has only been slightly modified from its original design in the previous SWOOP implementation [4].

4.2 Software/Hardware communication

In this implementation META-files are used to mark regions within the target code and is used as a method for testing. In real x86-hardware, one could instead use prefix instructions to annotate start and end regions of the code. These additional x86-prefixes would be straight-forward to implement, adding only minimal size to the binary.

Using META-files, however, allows for easy configuration- and fast prototyping, as testing the boundaries and edge-cases of the hardware simulation is as simple as modifying the META-file. For example testing how the hardware handles inaccurate number of used registers (fail modes). It is also possible to simulate the cost of these instructions afterwards, by injecting additional instructions (such as nops) into the instruction stream for accurate simulation. This is more dynamic than using actual pseudo-instructions, as the cost of these instructions can be modified at a later time by substitution- or injection of multiple instructions. In real x86-hardware, one could use a prefix instruction to annotate start- and end regions of the code.

The meta-data files are implemented in JSON format as it is easy to define named collections in the form of objects, and as python – which the launcher scripts are written in – natively support this format [15].

Each META-file consists of a META-object, which in turn can contain multiple objects representing SWOOP-sessions. These SWOOP-session objects are indexed by the instruction address pointing to the instruction identified as the setup phase. This allows for easily identifying which SWOOP-session is defined as it correlates to the target region of the code. Each session-object contains all information for the SWOOP-session, such as number of registers used in access (as part of the setup phase), phase-bounds for start of the access phase (access_begin), end of execute (execute_end) and the border-marker where access ends and execute begins (execute_begin).

Additionally it is possible to specify that the instruction at the address of a given region is to be deleted (removed from the instruction stream). This is useful for when custom pseudo-instructions are used for special purposes in the simulator and are not originally part of the code. One such case would be for signaling regions of interest, where the simulation only need to be simulated in detail for a specific portion of the execution.

The meta-data files are parsed by the python launcher-script and converted into appropriately formatted arguments for the simulator.

The SWOOP front-end then parses the appropriately formatted arguments into an unordered map which is used as a lookup-table during instrumentation of the execution via the Pin Tool. An unordered map, which in C++11 is implemented as a hashmap, was chosen as it practically has a constant lookup-time \(\Theta(1)\) in contrast to using a vector- or array, which have a linear lookup-time of \(\Theta(n)\).

During execution the Pin Tool enters an instrumentation phase, which occurs long before the instructions are to be executed and the instruction flow
is forwarded to the processor, that allows for attaching callback functions to specific instructions as well as altering the instruction stream by for example removing instructions. It is during this phase the lookup-table is used to determine if any of the simulated SWOOP architectural features are to be invoked before or after a specific instruction. For example, if the instruction that is currently instrumented matches the start of a setup phase, a call to a subroutine for readying the (SWOOP) hardware is inserted before the execution of the instruction.

The reason for using Pin Tool is that the hardware features need to be detached from the software to be executed, both in order to behave accurately for simulation as well as allowing execution of multiple different programs with different settings. If the compiler instead would prepare the software to call these functions, it would introduce additional instructions to be executed – most likely affecting the performance- and execution of the program. Adding additional instructions would also increase resource usage and register pressure not seen in the non-swoop version of the program, further misrepresenting the actual execution of the software as well as affecting the register mapping performed by the compiler. Thus by using Pin Tool to transparently invoke these callback-functions, the software and its performance/execution is kept as accurate as possible.

For a full description of the META-format, see Appendix B.

4.3 The SWOOP alternate execution path in hardware

The SWOOP alternate execution path is implemented as a hardware loop-unroller, where the hardware will – with respect to available resources – force the software to execute as many consecutive access phases as possible before executing their corresponding execute phases. When the hardware enters a SWOOP-session, it goes through three main SWOOP-states: setup, access, and execute. These states and the execution flow are illustrated in Figure 5.
Figure 5: The SWOOP alternate execution path as seen by the SWOOP-core, compared to an InO-core execution path.
The SWOOP setup state is where the software informs the hardware about the maximum potential resource usage in the upcoming access and execute phases. In this state, the hardware calculates how many SWOOP Contexts it is able to hold for the current SWOOP-session. This number may vary between SWOOP-sessions, as different sessions may have different amount of resource usage.

The SWOOP access state is when the executing software is within the code-region marked as the access phase. The hardware enters this state when it detects the `access_begin` marker. When in this state, the hardware will Context remap all registers written to as well as keeps track of how many Contexts it has created. This is to later be able to pair the access with its corresponding execute. Additionally when in the access state, the hardware intercepts all segfaults (see Section 4.3.4).

When the hardware detects the border-marker between access and execute (`execute_begin`), it will look at how many access phases have been executed (how many Context have been created) and if it has created the maximum number of possible Contexts, it allows the software to continue into the code-region marked as execute. If the hardware is still able to create more contexts, however, it saves the current Context and executes another access phase – forcing the software to jump back to the first instruction of access.

The reason the hardware executes access phases until it no longer can accommodate new Contexts is because the access phase is designed to be heavily memory-bound by hoisting the loads of a loop body. This means calculations such as software loop constraints are not available in an access phase on its own, only when paired with its corresponding execute phase are these calculated values available. This means that based on the access phase alone, it is impossible to determine whether the software loop is finished looping. Adding such calculations to the access phase can quickly make it compute-bound instead of memory-bound, resulting in loss of MLP.

The SWOOP execute state is when the software has entered the code-region marked execute, after the hardware has finished executing access phases and allowed the software to continue. When entering this phase, the hardware resets the context counter as to start with the oldest Context (the Context of the first access phase executed) and remaps the registers to this Context. After doing so, the execute phase is allowed to execute until the hardware detects the end-marker of the execute phase (`execute_end`). This marker is located at the end of the software loop and is the instruction that determines if the software is to continue executing the software loop or if the software loop is finished. At this point, there are four possible paths for the hardware to choose from. Which path to take is determined by if the software loop is finished or not, as well as if there are any contexts left (that is Contexts not consumed by any execute phases):

The software loop is not finished and there are Contexts left:
When in this case, the hardware merges the next stored Context, effectively writing the values loaded in the access phase to the registers used by the execute phase. This is possible since, in the original execution path, access would
have written to these registers after execute. Thus, SWOOP makes it appear to the software as if it is in the normal execution path. When the Context has been merged, the hardware executes another execute phase – forcing the software to jump back to the first instruction of the execute phase.

The software loop is not finished, but there are no Contexts left:
In this case, the hardware enters the (normal) default state and allows the software to continue and as the software loop returns to its beginning (the first instruction of the access phase), the hardware will again detect the access_begin marker and enter the access state once again.

The software loop is finished and there are no Contexts left:
If the software loop is finished and there are no Contexts left, it means that the number of software loop iterations was a multiple of the number of Contexts that the hardware was able to create for the SWOOP-session. In this case, the hardware enters the (normal) default state and allows the software to continue execution of the rest of the program.

The software loop is finished and there are Contexts left:
In this case, the number of software loop iterations was not a multiple of the number of Contexts that the hardware was able to create for the SWOOP-session. This means more access phases were executed than execute phases and in turn not all Contexts have been consumed. When the hardware detects this, it first discards all remaining additional contexts.

This alternate execution path comes with some issues that need to be addressed, though, such as loss of registers (4.3.3), false Contexts (4.3.4) and how to handle writes to memory in access (4.3.5).

4.3.1 Determining unroll factor
The number of consecutive access or execute phases for the alternate execution path is calculated using the number of registers written to in the access phase, provided by the software, and the total number of available SWOOP-registers. The number of unrolls (additional phase-executions) to be performed, \( \gamma(n) \), is defined as:

\[
\gamma(n) = \begin{cases} 
0 & \text{if } R_{\text{SWOOP}} < n \\
\lfloor \frac{R_{\text{SWOOP}}}{n} \rfloor & \text{if } R_{\text{SWOOP}} \geq n
\end{cases}
\]

where \( n \) is the maximum possible number of registers used in one access phase (reported by software in the setup phase) and \( R_{\text{SWOOP}} \) is the number of available physical SWOOP registers in addition to the x86 registers (see Section 4.3.3). In this case, the SWOOP-core would perform additional unrolls as soon as one- or more Contexts fit in the additional SWOOP-registers.

4.3.2 Hardware unrolling
Hardware unrolling is implemented using both two unroll bits, signaling to the core whether it is currently in normal execution or in an unrollable phase, and two loop registers for storing an instruction-pointer to the beginning of the
hardware loop (in this case the first address of access and execute). The loop register is used to store which instruction address the unrolling should jump back to and the unroll bits is to not have to re-store the instruction-pointers each time we see the beginning address of access or execute in the current SWOOP-session. As with the Context register remapping, the key here is that the hardware unrolling is not needed when executing normally — only when executing the alternate path.

When the core detects it has entered (the first instruction of) an access or execute phase, it stores the instruction pointer in a loop register and sets the unroll bit.

When the core detects it is at the end of a phase, it makes a decision whether to unroll the phase or not according to the execution model (Sections 3.3.2 and 4.3), where it follows three rules: (1) If the core decides to unroll further, it simply sets the instruction pointer to the address stored in the corresponding loop register. (2) If the core decides to not unroll, it simply allows the software to continue execution. (3) If the core detects the end of the SWOOP-session, it sets the unroll bits to 0 to allow future unrolling, then allows the program to continue execution as normal.

The reason for storing both the access and execute phase instruction address is to enable detecting whether or not the software-loop is finished or not. This can be detected by looking at the address to which the software wants to jump at the end of the execute phase. If it matches the stored instruction address for unrolling the access phase, the software-loop is not finished.

4.3.3 SWOOP physical registers

The swoop implementation utilizes additional physical SWOOP registers (additional to the x86 registers) to store SWOOP-contexts created during the alternate path execution. These are implemented as configuration options in Sniper to allow easy modification to available additional registers, effectively modifying the unroll-amount possible without re-compiling the software or simulator. The simulator, with information gained about the SWOOP-session via the setup phase, uses this value to calculate how many additional contexts it can store in the additional registers. As the unroll-count depends on the register usage in the access phase as well as the number of available SWOOP-registers, there are situations where some of these additional registers become unusable.

The first situation is where the number of registers used in access, \( R_{access} \), is greater than the number of available SWOOP-registers, \( R_{SWOOP} \). This results in \( R_{SWOOP} \) numbers of unusable registers, but with a reasonable minimum number of additional SWOOP-registers there should always be room for at least one context.

The second situation is when \( R_{SWOOP} > R_{access} \) and is not a multiple of \( R_{access} \), resulting in \( R_{SWOOP} - (\gamma(R_{access}) \cdot R_{access}) \) unusable registers for the SWOOP-session.

4.3.4 False contexts

False contexts is one of the explicit differences between this version of SWOOP and the version of the prior work. These false contexts are a by-product of completely removing the Check-miss feature in favor of forced hardware-unrolling,
whereas the version of the prior work could circumvent these by using Check-miss.

As the hardware forcibly unrolls the access phase as many times as possible, with no regard for the software-loop constraints, it is possible to execute additional access phases than the original execution path would. This creates *false contexts*, which are SWOOP-contexts with no corresponding execute phase. As such, these false contexts do not only contain unreliable data but will also not be consumed by an execute phase and need to be specially handled. The reason for allowing false contexts to be created is that the access phase should minimize computation to maximize MLP. For example, if the loop constraint is based on the sum calculated in the loop, all calculation in the execute phase would need to be hoisted into the access phase itself and then the access and execute would no longer be decoupled. However, as the calculation is present in the execute phase it is possible to allow the execute phase to exit when the original software-loop wants since the loop constraints are available. When the hardware detects that the loop is finished, it can simply clear all remaining contexts as they are not going to be consumed by any execute phase.

Additionally, since the original execution path would not have executed these access phases, there is a risk the access phase reads memory it is not allowed, causing a segfault. Because of this, it is assumed that segfaults occurring within the access phase are caused by the hardware unrolling and not the software itself and thus must be intercepted- and handled/suppressed by the hardware. The implementation simply logs all handled segfaults and zeros the register to which the load was trying to write. This does not mean that the current hardware prevents segfaults from occurring, only that it quenches all segfaults originating from an access phase.

There is, however, a chance the segfault was caused by the software, but when in an access phase it is not currently possible to determine whether the program segfaulted because of the hardware-unrolling or a software-bug. This could be solved by propagating (or delaying) the captured segfaults until the execute phase by introducing an additional flag-bit int the SWOOP-registers. This flag-bit is used to indicate whether the load-instruction writing to the register segfaulted or not such that when an access phase load segfaults, the hardware simply sets this flag-bit and quenches the segfault. If the execute phase later tries to access this register, the hardware can detect that the load segfaulted and signal the program. This feature is necessary, and a requirement for precise exceptions, when implementing the real hardware. This is suggested as a possible future work in Section 7.3.

### 4.3.5 Handling of writes in access

In addition to context remapping, SWOOP also needs to take care of all writes to memory during the access phase. The software is, however, only allowed to write to the stack and specifically only the portion of the stack explicitly allocated for the access phase, in accordance with the stack interface in Section 3.3.4.

This could be implemented by having the hardware inform the software about the number of access phases to be executed in order to allocate the needed stack space via, for example, *alloca*. The software could then inform the hardware about the size of one access phase on the stack via an additional
setup phase instruction. When in the access or execute phase, as the hardware
is aware of which access or execute phase is currently executing, it can simply
modify the stack pointer accordingly – utilizing the supplied size of an access
phase stack area. Additionally, in order to allow the software to access the
execute phase stack area when in the execute phase, it can simply compare the
stack pointer offset to the supplied access phase stack size. If the offset is greater
or equal to the size of a single access phase on the stack, the software is trying
to access the execute phase on the stack and the hardware can modify the stack
pointer accordingly.

The emulated hardware does not currently allocate this extra stack-space
or modify the stack pointer directly, but rather utilizes data structures to store
windows of the stack transparently in the simulator front-end. The simulator
looks at every memory write that takes place in the access phase and tries to
determine whether it is a legitimate write to the stack or if it tries to write to
some other part of the memory. If the write is determined to be outside the
allowed space, the simulator will halt with an error as it is highly likely any
further execution will be incorrect.

If the write is to within the allotted space, the simulator records the memory
write to the corresponding window and allows the emulated hardware to con-
tinue execution. When the hardware is finished unrolling the access phase, the
last executed access phase will have written to the stack. To simulate the sliding-
window described in Section 3.3.4, the simulator simply re-plays the recorded
memory writes of the access phases for the corresponding execute phases.

4.4 Stack usage analysis

One of the interesting statistics to collect is stack allocation of the access phase
and, in extension, the extra stack allocation due to hardware unrolling. This
analysis of the access phase presents a slightly different problem, as the access
phase may contain branches that jump past some stack access instructions –
thus making it hard (or impossible) to analyze stack usage dynamically as not
all stack writes may be executed.

This feature is not implemented in hardware, as it is currently only of statis-
tical interest. This information could, however, possibly be used for historical-
or heuristic decision making in a future version of SWOOP.

4.4.1 Static analysis in Pin

Intel’s Pin, which the SWOOP front-end is built upon, contains a module
for statically analyzing the executable images loaded to memory before they
execute. Using this interface, the front-end statically analyzes the image of
the main-executable, looking for any existing SWOOP-regions as indicated by
the supplied META-file. When the analyzer locates an address marked as
access-begin, it starts looking for each memory write instruction whose tar-
get address is relative to the stack- or frame-pointer. These memory-writes are
considered stack-writes, and for each stack-write the analyzer stores the address
written to, number of stack-writes and size of the write. This is then stored as
an easily accessible map for quick look-up.

This static analysis comes with a caveat, though, as it may give misleading
results if executing code compiled without a frame-pointer, as the frame-pointer
register then is used as a general register – possibly falsely triggering instructions as stack writes when they in fact are not.

5 Evaluation

To evaluate this work, the Sniper Multi-Core Simulator is used [10, 11]. The cycle-level core model is modified to support the SWOOP processor and is based on the modified core model from the previous implementation of SWOOP [4]. Power- and energy estimates are calculated with McPAT version 1.3 in 28 nm. The microarchitecture details are listed in Table 1. For the baselines, two efficient processor implementations are used. For the In-Order cores a low-power mobile-class core is used and for the Out-of-Order core, a low-power out-of-order mobile-class core is used. The SWOOP-core is also based on the same core as the In-Order core, but with additional SWOOP-features.

A number of potential solutions in performance- and energy efficiency is compared across these cores, with some slight modifications to settings such as number of MSHR\(^1\) (Miss Status Handling Register) entries and physical SWOOP-registers. This is to evaluate how the SWOOP-core performs under various conditions.

\(\text{InO}\) targets a low-power mobile-class core.

\(\text{OoO}\) targets a low-power out-of-order mobile-class core, which hides long-latency loads by dynamically reordering instructions for out-of-order execution.

\(\text{SWOOP}\) is described in sections 3 and 4. SWOOP has one additional pipeline stage to support context register remapping during execution, described in section 3.3.1, which results in a one cycle longer branch penalty.

<table>
<thead>
<tr>
<th>Core</th>
<th>InO</th>
<th>SWOOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>u-Arch</td>
<td>1.5 GHz, 2-way superscalar</td>
<td>*</td>
</tr>
<tr>
<td>ROB</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>RS</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>Phys. Reg</td>
<td>32</td>
<td>32-128(^2)</td>
</tr>
<tr>
<td>Br. Penalty</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>L1-I cache:</td>
<td>32 KB, 8-way LRU</td>
<td>32 KB, 8-way LRU, 4-cycle, 8 MSHRs(^3)</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256 KB, 8-way LRU, 8 cycle</td>
<td></td>
</tr>
<tr>
<td>L3 cache</td>
<td>4 MB, 32-way LRU, 30 cycle</td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td>7.6 GB/s, 45 ns access latency</td>
<td></td>
</tr>
<tr>
<td>Prefetcher</td>
<td>stride-based, L2, 16 streams</td>
<td></td>
</tr>
<tr>
<td>Technology</td>
<td>28 nm</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Simulated single-core microarchitecture settings.

\(^{(*)}\) The OoO is 3-way superscalar at 1.5 GHz.

---

\(^1\) The MSHR keeps track of outstanding loads, effectively determining how many in-flight long latency loads are possible before the core stalls (thus directly affects the achievable MLP in a core).

\(^2\) Varies between 0/16/32/64/96 additional (SWOOP) registers. Set to 96 (64 additional registers) if not otherwise mentioned.

\(^3\) May vary with some tests.
5.1 Benchmarks

\texttt{Bench.longlatency} is a synthetic microbenchmark used to test (close to) the best-case scenario for SWOOP. It is designed to always generate a miss in the LLC (Last Level Cache), forcing the program to access the memory system. The data set to be accessed is an array of cache-line size entries, selected to not fit in the LLC. A smaller, secondary array containing unique random indexes of the data set array is used to randomly access the larger data set. It is generated so that the same entry is never loaded twice (as it could still be in cache) and the probability of having loaded the entry by data-locality is slim to none. This benchmark basically consists of the target loop, and so it is only measured in detail once.

Statistics on the SWOOP-session for this benchmark are available in Table 2.

\texttt{Bench.lineararray} is a synthetic microbenchmark used to test (close to) the worst-case scenario for SWOOP, where the data is almost always present in the cache and in a predictable order – not exposing much memory level parallelism due to lack of long-latency loads. The data consists of an array of integers, where each iteration an element of the array is accessed in index-order. This benchmark basically consists of the target loop, and so it is only measured in detail once.

Statistics on the SWOOP-session for this benchmark are available in Table 2.

\texttt{429.mcf} is a benchmark from the SPEC CPU2006 benchmark suite. It is based on MCF, a program used for single-depot vehicle scheduling in public mass transportation \cite{5}. A target loop has been identified as having potential for increased performance and energy-efficiency by use of SWOOP. In this benchmark, the target loop is executed multiple times in order to get an average measurement in this real life benchmark. This means both the target loop as well as the code executed in-between loops are measured.

Statistics on the SWOOP-session for this benchmark are available in Table 2.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
SWOOP-registers & \texttt{lineararray} & \texttt{longlatency} & \texttt{429.mcf} \\
\hline
Instructions in access & 11 & 13 & 4 \\
Instructions in execute & 9 & 9 & 8 \\
Registers written in access & 7 & 5 & 4 \\
Unroll-count & 10 & 14 & 12 & 19 & 17 & 25 \\
Access-phases & 1000000 & 1000006 & 1000012 & 1000000 & 489294 & 479700 \\
Execute-phases & 1000000 & 1000000 & 1000000 & 1000000 & 479332 & 479332 \\
False contexts & 0 & 6 & 12 & 0 & 9962 & 368 \\
\hline
\end{tabular}
\caption{SWOOP-session statistics for benchmarks, showing default (64)- and maximum tested (96) SWOOP-registers.}
\end{table}

5.2 Results

The results were gathered from multiple benchmarks, ranging from custom microbenchmarks for testing the extremes to a real-world applicable benchmark from the SPEC CPU2006 \cite{5}.
5.2.1 Performance

The performance results are shown in Figure 6.

Looking at the best case scenario (μBench.longlatency), SWOOP is able to significantly out-perform both the InO- and OoO-core. SWOOP is able to reach 3.7x the performance of the InO-core, while the OoO-core only reaches 1.7x speedup. This is most likely because the number of ROB-entries of the OoO-core limits its instruction stream visibility, limiting its ability to re-order instructions and therefore limiting its ability to find additional MLP. Meanwhile, SWOOP’s visibility is provided with compiler annotations which allows for an alternative execution path without requiring additional execution state.

Looking at the worst-case scenario (μBench.lineararray), OoO is able to take the lead over both SWOOP and InO. SWOOP is, however, only marginally behind OoO (slower by only 5.5%) and is still able to increase performance beyond the InO-core. That OoO is marginally faster is most likely due to its ability to re-order instructions individually instead of being limited to pre-defined regions of code as seen in SWOOP. That said, SWOOP is still able to find some level of MLP that can be exposed for a 1.08x speedup (in comparison to 1.14x by OoO) which most likely comes from the occasional LLC-load or long latency load.

When looking at 429.mcf, OoO takes a greater lead over SWOOP with 2.16x speedup versus SWOOP’s 1.65x. This is, again, most likely because the OoO-core is not limited to re-order only pre-defined regions of code but is able to re-order instructions individually. Additionally, the OoO-core is not only able to re-order instructions within the target loop but also in-between calls to the target loop. That said, SWOOP is able to expose MLP in the target loop resulting in a 1.65x speedup – on average – for the benchmark when constrained to only re-ordering the target loop. This is more than half of the speedup gained from using an OoO-core and in comparison highly limited resources for optimization.

![Figure 6: Speedup comparison across benchmarks using the default microarchitecture configuration, both with- and without prefetcher enabled. SWOOP is configured with 64 SWOOP-registers. Results are normalized to InO core with prefetcher enabled.](image)

When disabling prefetchers for each core-model, the general relation between
cores stay the same with the exception of \( \mu \)Bench.lineararray. For the worst-case scenario (\( \mu \)Bench.lineararray) SWOOP is actually able to out-perform OoO ever so slightly as it is not affected nearly as much as the InO-core or OoO-core. For \( \mu \)Bench.longlatency, the decrease in performance for SWOOP is 19% in comparison with 13% for the other cores. This is most likely because SWOOP relies heavily on the memory loads to increase performance as well as that SWOOP is so much faster in this case that the impact may be more prominent. Looking at 429.mcf, both OoO and SWOOP behave the same with a 34% decrease in performance.

5.2.2 Energy

The energy efficiency results are shown in Figure 7.

For each benchmark, SWOOP is able to out-perform both the InO- and OoO-core in energy efficiency – even when only improving performance marginally.

For the best-case (\( \mu \)Bench.longlatency) SWOOP is able to reduce energy use by 59%, compared to the InO-core. It is also able to out-perform the OoO-core by a whole 55%. Thus, given a program that is heavily memory bound SWOOP is able to greatly increase energy efficiency. Even though the OoO-core is able to increase performance, it is only able to increase energy efficiency by 4%. This is most likely the penalty of the more complex- and power hungry features of the OoO-core, which is even more evident in the worst-case benchmark.

Looking at the worst-case (\( \mu \)Bench.lineararray) SWOOP is able to increase energy efficiency even though it is not able to greatly increase performance. With a speedup of only 8% it is able to increase energy efficiency by 16% compared to the InO-core. This is probably due to the low cost of the additional features of the SWOOP-core architectural support. Looking at the OoO-core, however, the penalty of the additional hardware support for re-ordering instructions is clearly visible, as it uses 33% more energy than the InO-core. In this case, the OoO-core is not able to increase performance enough to overcome its energy cost.

When it comes to 429.mcf, SWOOP is again able to out-perform the InO- and OoO-core by 33% and 15% respectively. This is even though it is only able to increase performance by half that of the OoO-core. Again, this is most likely the cost of the additional hardware support of the OoO-core showing, whereas the additional cost of the SWOOP-core is low enough to greatly increase energy efficiency in comparison.

With disabled prefetchers, the cores follow the same trend as for the performance impact where the general relation between core-models stay the same, which can be seen in Figure 7. For both microbenchmarks, disabling prefetchers has the greatest impact on the OoO-core with 8.2% higher energy consumption for \( \mu \)Bench.longlatency and 25% for \( \mu \)Bench.lineararray. SWOOP receives the lowest impact for \( \mu \)Bench.longlatency at 6.5% but is in the middle for \( \mu \)Bench.lineararray at 20%. Looking at 429.mcf, however, SWOOP is affected the most when disabling prefetchers with an energy efficiency decrease of 39% but is closely followed by OoO at 36%.
Figure 7: Energy efficiency comparison across benchmarks using the default microarchitecture configuration, both with- and without prefetcher enabled. SWOOP is configured with 64 SWOOP-registers. Results are normalized to InO core with prefetcher enabled.

5.2.3 Sweeping hardware and software parameters

In addition to running the benchmarks with the default MSHR-size of 8, a sweeping benchmark was used to determine how the MSHR-size affected the results. These results are available in Appendix A, Figures 10 and 11, as well as the heatmap tables 3, 4, 5, 6, 7 and 8.

The MSHR is what limits how many outstanding long latency loads the core is able to handle, after which the core will stall – thus directly affecting the achievable MLP. This means there should be a clearly visible difference between the MSHR-sizes until it surpasses the maximum possible MLP of the program. For $\mu$Bench.longlatency the MLP should theoretically be directly tied to the size of the MSHR and the number of unrolls performed by the SWOOP-core, as its access phase will always generate a long latency load.

If the program is highly compute-bound or contains little to no long latency loads, the MSHR-size should not have a significant effect on performance as the outstanding loads will be very low. In the case of $\mu$Bench.lineararray the MSHR-size should still affect it to some extent, as SWOOP is able to find enough MLP to increase performance.

The results show that the size of the MSHR does have an effect on the achievable performance, as the heatmaps indicate that SWOOP is able to achieve more MLP than the other cores – depending on the number of available SWOOP-registers. SWOOP-96 is able to peak above 16 MSHRs where OoO peaks already at 8 MSHRs for both 429.mcf and $\mu$Bench.longlatency. This shows that SWOOP is able to increase MLP in comparison with InO and OoO, which should translate to a greater speedup with more MSHRs.

However, looking at the resulting performance and energy efficiency in Figures 10 and 11, there is no significant increase in either performance or energy efficiency. For SWOOP-96, which achieves the highest MLP of the SWOOP-cores, we see only a slight increase in performance for one of the benchmarks with a 3.6% increased speedup at 400% MSHR-size. This suggests there is some other component in the hardware that limits the achievable performance, as 32
MSHRs are able to handle the vast number of memory requests issued by the SWOOP-96 core.

The heatmaps supplied for each benchmark consists of one heatmap representing the total MSHR-latency (in femtoseconds) and one representing the normalized speedup, as an alternate visualization of these results. In the heatmaps over total MSHR-latency, the presence of MSHR-latency should indicate that there is more MLP available than the MSHR is able to handle at one time, meaning the MSHR should limit performance. A total MSHR-latency of 0 should thus indicate that the MSHR is able to handle the available MLP – thus not limit performance.

### \( \mu \text{Bench.longlatency} \)

<table>
<thead>
<tr>
<th>Core-model</th>
<th>MSHR-4</th>
<th>MSHR-8</th>
<th>MSHR-10</th>
<th>MSHR-16</th>
<th>MSHR-32</th>
</tr>
</thead>
<tbody>
<tr>
<td>InO</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OoO</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SWOOP-nounroll</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SWOOP-16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SWOOP-32</td>
<td>7.05 \times 10^{12}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SWOOP-64</td>
<td>2.28 \times 10^{13}</td>
<td>2.17 \times 10^{12}</td>
<td>3.33 \times 10^{11}</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SWOOP-96</td>
<td>4.3 \times 10^{13}</td>
<td>8.45 \times 10^{12}</td>
<td>3.04 \times 10^{11}</td>
<td>1.95 \times 10^{10}</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3: Heatmap over total MSHR latency in femtoseconds for L1-DCache for \( \mu \text{Bench.longlatency} \). A total latency of 0 indicates the MSHR is able to handle the available MLP and should not limit performance.

### \( \mu \text{Bench.lineararray} \)

<table>
<thead>
<tr>
<th>Core-model</th>
<th>MSHR-4</th>
<th>MSHR-8</th>
<th>MSHR-10</th>
<th>MSHR-16</th>
<th>MSHR-32</th>
</tr>
</thead>
<tbody>
<tr>
<td>InO</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>OoO</td>
<td>1.7</td>
<td>1.7</td>
<td>1.7</td>
<td>1.7</td>
<td>1.7</td>
</tr>
<tr>
<td>SWOOP-nounroll</td>
<td>0.97</td>
<td>0.97</td>
<td>0.97</td>
<td>0.97</td>
<td>0.97</td>
</tr>
<tr>
<td>SWOOP-16</td>
<td>2.16</td>
<td>2.15</td>
<td>2.15</td>
<td>2.15</td>
<td>2.16</td>
</tr>
<tr>
<td>SWOOP-32</td>
<td>2.71</td>
<td>2.87</td>
<td>2.87</td>
<td>2.87</td>
<td>2.87</td>
</tr>
<tr>
<td>SWOOP-64</td>
<td>3.23</td>
<td>3.43</td>
<td>3.43</td>
<td>3.43</td>
<td>3.43</td>
</tr>
<tr>
<td>SWOOP-96</td>
<td>3.51</td>
<td>3.73</td>
<td>3.73</td>
<td>3.74</td>
<td>3.74</td>
</tr>
</tbody>
</table>

Table 4: Heatmap over speedup normalized to InO MSHR-8 for \( \mu \text{Bench.lineararray} \).

### \( \mu \text{Bench.lineararray} \)

<table>
<thead>
<tr>
<th>Core-model</th>
<th>MSHR-4</th>
<th>MSHR-8</th>
<th>MSHR-10</th>
<th>MSHR-16</th>
<th>MSHR-32</th>
</tr>
</thead>
<tbody>
<tr>
<td>InO</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OoO</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SWOOP-nounroll</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SWOOP-16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SWOOP-32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SWOOP-64</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SWOOP-96</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5: Heatmap over total MSHR latency in femtoseconds for L1-DCache for \( \mu \text{Bench.lineararray} \). A total latency of 0 indicates the MSHR is able to handle the available MLP and should not limit performance.
Table 6: Heatmap over speedup normalized to InO MSHR-8 for μBench.lineararray.

### 429.mcf

Table 7: Heatmap over total MSHR latency in femtoseconds for L1-DCache for 429.mcf. A total latency of 0 indicates the MSHR is able to handle the available MLP and should not limit performance.

Table 8: Heatmap over speedup normalized to InO MSHR-8 for 429.mcf.

5.2.4 SWOOP-registers versus performance- and energy

Another evaluation is how the number of SWOOP registers affect performance- and energy efficiency. For this evaluation, each benchmark was run with 16, 32, 64 and 96 SWOOP-registers as well as with unrolling disabled (nounroll) but with all other architectural support enabled. The nounroll benchmark serves to show the worst-case performance and energy efficiency of SWOOP, where the core is unable to use the alternate execution path.

In regard to performance, the SWOOP-core seems to be able to continue increasing performance beyond 96 additional registers for heavily memory-bound applications, as shown by the increase in performance for μBench.longlatency. However, the performance gained seem to increase logarithmically as for each increased value the performance gained is less than for the previous – as shown in Figure 8. Looking at the energy efficiency for this microbenchmark, the gain decreases faster than the increase in performance as the number of additional
registers and increase in performance start to balance out. Even though performance seem to possibly increase further, energy efficiency seem to almost plateau after 96 registers.

For the worst-case, μBench.linearray, and 429.mcf SWOOP seems to plateau at 64 additional registers with only a marginal increase of less than 1% from 32 registers. Energy efficiency for these benchmarks plateau quickly at 16 additional registers for the worst-case and 32 additional registers for 429.mcf.

![Graph](image)

Figure 8: Speedup comparison across benchmarks with sweeping number of SWOOP-registers. Results are normalized to InO core.

![Graph](image)

Figure 9: Energy efficiency comparison across benchmarks with sweeping number of SWOOP-registers. Results are normalized to InO core.

### 5.3 Discussion

We assumed going into benchmarking- and evaluating the SWOOP-core it would, for some number of SWOOP-registers, get near OoO-speedup for MLP-heavy programs. For the best case SWOOP out-performed both the OoO-core as well as my expectations and for the other benchmarks it performed in line with the assumptions. The probable cause as to why SWOOP is able to outperform OoO in the best case is that the number of ROB-entries of the OoO-core
limits its instruction stream visibility, limiting its ability to re-order instructions and therefore limiting its ability to find additional MLP. In contrast, SWOOPs visibility is provided with compiler annotations which allows for an alternative execution path without requiring additional execution state.

Additionally we hoped that SWOOP would approach or surpass the energy efficiency of the OoO-core for programs with high MLP and highly reorderable instructions. This has been shown by the longlatency microbenchmark (see Figures 6 and 7).

For programs with slim to no MLP, however, we expected the SWOOP-core would at worst case be marginally slower than the InO-core used as baseline. Thus it would follow that, at worst, the energy efficiency of the SWOOP-core would only be marginally worse than the InO-core – in contrast to the OoO-core which requires significant energy for the dynamic instruction analysis and instruction re-ordering. This, too, has been shown by the lineararray microbenchmark, which is designed to contain as few long latency loads as possible. Additionally, SWOOP was able to increase energy-efficiency and performance beyond that of the InO-core even in such a case (µBench.lineararray) – most likely due to the occasional long latency load. This can be seen in Figures 6 and 7.

For 429.mcf, SWOOP was able to slightly improve performance but is still able to greatly increase energy-efficiency. Comparing these values with the result of the microbenchmarks it seems the benchmark is not exposing too much MLP, thus not allowing SWOOP to increase performance beyond the OoO-core. Looking at the performance of the OoO-core, it only supersedes SWOOP by 23%. This is probably because the OoO-core is able to re-order instructions in-between the SWOOP-sessions as well as in the access phase. Looking at energy efficiency, though, SWOOP out-performs both the InO- and OoO-core with 33% and 18% respectively.

Looking at 429.mcf and how it is compiled, it was clear that the algorithm did not allow for great MLP as it contains NULL-pointer checks and other control structures that prevent the compiler from creating a larger access phase. Currently the access phase consists of very few long latency load that is then directly used in the execute phase to determine whether or not to load data.

The results of the benchmarks are promising and follow-on steps are needed to fully evaluate the SWOOP-implementation for additional benchmarks. In order to fully evaluate the SWOOP-implementation more benchmarks are needed as the current data is inconclusive, in part due to tweaks needed on the compiler side and in part due to lack of benchmarks. For now the results are limited to two especially designed microbenchmarks and one real benchmark (429.mcf). If instead a whole suite of benchmarks were tested, it would be possible to form a better understanding of what types of applications SWOOP may improve as well as getting average data for SWOOP in a wide variety of applications. Additionally, it would be great with diversity in benchmarks as it would allow concluding whether SWOOP is especially efficient with certain types of problems. One group with potentially great conditions for SWOOP would be graph benchmarks / graph algorithms, as they are often heavily memory-bound.

5.3.1 Compilation of benchmarks

The compiler currently needs more tweaks, as there are multiple transformations of the code that the compiler is not currently able to handle. However, when
these transformations have been accounted for in the SWOOP compiler-passes, more benchmarks will become available and a greater data set will allow for better evaluation of the new SWOOP implementation.

5.3.2 Sweeping hardware and software parameters

In the current implementation of SWOOP we see that it is able to increase MLP, which is reflected by the supplied heatmaps over total MSHR-latency where OoO peaks already at 8 MSHRs and SWOOP-96 is able to peak above 16 MSHRs. For SWOOP-96, a number of 32 MSHRs are able to keep up with the outstanding misses, which should be reflected in the performance and energy efficiency compared to using only 8 MSHRs. What we see, however, is only a marginal increase in speedup at – for the best case – 3.6% with a 400% increase in number of MSHRs.

A probable reason for not seeing an increase in performance and energy efficiency, even with enough MSHRs to keep up with the vast number of long latency loads, is that some other part of the hardware limits performance. This should be investigated further, as the possibilities of SWOOP are still not fully achieved.

5.3.3 Stack usage analysis

The benchmarks used has so far not contained any stack usage for temporary- or spilled variables in the access phase, thus the impact on the stack by hardware unrolling access is still unknown – though the control/statistics structures are available for collecting this data.

What we would like to see is how much extra stack is utilized on average, as to determine what its implications may be.

6 Conclusion

SWOOP seemingly deliver on its promise of increased energy efficiency and the same- or even higher performance than conventional In-Order cores, thus is a viable option for embedded- or mobile devices where energy efficiency is important and performance cannot be compromised.

That said, SWOOP is especially effective for heavily memory-bound applications, but as long as the application has memory-bound regions of code wherein SWOOP can exploit the underlying MLP it is able to both increase performance and energy efficiency. Additionally, for heavily compute-bound applications SWOOP will affect neither performance nor energy efficiency visibly negatively as it allows for normal In-Order execution, thus the worst case will result in normal In-Order performance- and energy efficiency. In the worst-case microbenchmark, SWOOP is even able to improve performance and energy efficiency when there is almost no MLP available.

There does, however, seem to be some tweaks needed on the compiler-side of SWOOP in order to successfully handle the possible code transformations, as was evident when most benchmarks were not possible to compile – where the compiler transformed the code in such a way that the subsequent compiler passes were unable to continue compilation. Additionally, the limiting factor within the hardware should be investigated further as there is a possibility of
achieving an even greater performance and energy efficiency as SWOOP is able to achieve a far greater MLP than the other cores – which currently is not reflected in the achieved performance or energy efficiency.

The results are very promising, but in its current state results are inconclusive as to whether SWOOP is able to out-perform conventional hardware on average, as there is a lack of data. Before such a conclusion can be made, more benchmarks need to be tested in order to gather data on a variety of applications and algorithms – such as testing full suites of benchmarks. With such data it would also be possible to conclude whether SWOOP is especially effective for a specific group of applications, such as graph-based algorithms. Even so, SWOOP seems to be a good option for energy efficiency without compromising performance.

7 Future work

During the work on this thesis, a lot of interesting questions came to light that may be worth exploring in future works. A select few are here listed.

7.1 Dynamically enabling- or disabling register remapping using historical information

Since register remapping uses energy and hardware-unrolling is not always applicable, such as for applications with low MLP (see \(\mu\)Bench.lineararray), the unrolling- and remapping may use more energy than necessary. To counter this, SWOOP could store historical data to determine whether or not unrolling achieves MLP and disable unrolling below a certain threshold – possibly saving a lot of energy. This would be an alternative to the Check-miss feature of the previous incarnation of SWOOP.

7.2 Variable trip-counts for hardware-unrolling

The current implementation uses fixed trip-counts when hardware-unrolling, depending on information supplied to the hardware by the software to be executed. A possible future work would be to utilize variable trip-counts instead, allowing the hardware to dynamically alter the trip-count depending on the current register usage of the executing software. This could prove effective for applications whose register usage is highly dependent on input data, as the fixed trip-counts are always determined using the maximum possible usage.

7.3 Propagation of segfaults to the execute phase

As the current version suppresses segfaults that occur during the access phase, there is a risk of suppressing segfaults caused by the executing program and not the hardware unrolling. A future work could be to propagate these segfaults to the execute phase. If the software tries to use data in a register that caused a segfault, the hardware could signal a segfault. This could for example be implemented as flag-bits in the SWOOP-registers, signaling to the hardware that the data in the register is not valid.
7.4 Instruction-based communication between HW/SW

A smaller addition to the current version of SWOOP would be to enable communication between hardware- and software not through the use of META-files, but through actual (custom) instructions. This does not necessarily impact the accuracy of the simulation, but may serve to reduce the complexity in the simulator front-end and perhaps the compiler as well. This addition would then include modifications to the software side of the co-design.

7.5 Hardware-implemented stack allocation for SWOOP-contexts

As mentioned in section 3.3.4 a solution for allocating the necessary stack-space for each access phase could be solved by returning the number of unrolls to be performed with the setup-instruction and have the software allocate (using 	exttt{alloca} or alike) this stack space. A future work could, however, look at what possibilities exist for enabling this in hardware – looking at what impact it has on performance and energy, as well as how the different (SW/HW) approaches affect register pressure.

7.6 Implement software/hardware communication as a single setup phase

An alternative to individual marker-instructions for communicating phase-bounds to the hardware would be to expand the setup phase to multiple instructions that serve to inform the hardware of the instruction addresses comprising the bounds of the phases. This would require additional hardware-resources such as a small cache or SWOOP-specific lookup-buffer for keeping track of which instruction addresses are bounds of the current SWOOP-session. This work could thus look at the performance impact of using individual marker-instructions within the hardware-unrolled loop versus hoisting these instructions to the setup phase with additional hardware, as well as impact on energy efficiency.
References


Appendix A  Plots sweeping MSHR-size

A.1  Performance

Figure10: Speedup comparison across MSHR-sizes and SWOOP-register count. Normalized to InO MSHR 8.
A.2 Energy efficiency

Figure 11: Comparison of energy-efficiency across MSHR-sizes and SWOOP-register count. Normalized to InO MSHR 8.
Appendix B  SWOOP META-format

The META-file format has been designed to be as flexible as possible, whilst still enforcing the constraints imposed by SWOOP and the simulator. In real x86-hardware, one could use a prefix instruction to annotate start- and end regions of the code. In our implementation we use META-files as a method for testing, but additional x86-prefixes would be straightforward to implement and will add only minimal size to the binary. The use of META-files is in part to simplify prototyping and testing by allowing multiple input formats for addresses as the representation of addresses may vary from tool to tool and the requirements of the format changed during our research.

B.1 META-file structure

The general structure of a META-file is as follows:

```
|--- META object
    |--- SWOOP-session object
    |    |--- registers       (Integer)
    |    |--- access_begin   (Address or Address object)
    |    |--- [access_end]    (Address or Address object) (Reserved)
    |    |--- execute_begin  (Address or Address object)
    |    |--- execute_end    (Address or Address object)
    |    |___ [instruction removal list]
    |    |    |--- Address
    |    |    |___ [Address]
    |___ [SWOOP-session object]
```

Figure 12: META-file structure. Elements within brackets [ ] are optional.

B.1.1 META object

The META object is an unnamed object which functions as a wrapper for the whole META-file. It contains SWOOP-session objects which are labeled with the address of the setup phase instruction for the SWOOP-session. The label has to be a string, as to not break the JSON-format, but other than that it is of the Address type.

B.1.2 SWOOP-session object

A SWOOP-session object is an object directly added to the META object, containing the following elements:
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>int</td>
<td>Yes</td>
<td>Number of registers written to in access.</td>
</tr>
<tr>
<td>access_begin</td>
<td>Address or Address object</td>
<td>Yes</td>
<td>Address of first instruction by the access phase.</td>
</tr>
<tr>
<td>access_end</td>
<td>Address or Address object</td>
<td>No</td>
<td>Reserved</td>
</tr>
<tr>
<td>execute_begin</td>
<td>Address or Address object</td>
<td>Yes</td>
<td>Address of first instruction in execute.</td>
</tr>
<tr>
<td>execute_end</td>
<td>Address or Address object</td>
<td>Yes</td>
<td>Address of last instruction in execute.</td>
</tr>
<tr>
<td>delete_ins</td>
<td>List of Addresses</td>
<td>No</td>
<td>Additional addresses to delete from instruction stream.</td>
</tr>
</tbody>
</table>

B.1.3 List of addresses

A list of addresses simply consists of one- or more Addresses.

B.1.4 Address object

An Address object consists of two elements:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>Address</td>
<td>Yes</td>
<td>The address</td>
</tr>
<tr>
<td>delete_ins</td>
<td>Boolean</td>
<td>Yes</td>
<td>Delete the instruction at address from instruction stream.</td>
</tr>
</tbody>
</table>

B.1.5 Address

An address may be expressed in multiple forms, as to accommodate output from multiple tools:

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>122145670</td>
</tr>
<tr>
<td>numerical (string)</td>
<td>&quot;122145670&quot;</td>
</tr>
<tr>
<td>hexadecimal (string)</td>
<td>&quot;747cb86&quot;</td>
</tr>
<tr>
<td>hexadecimal with prefix (string)</td>
<td>&quot;0x747cb86&quot;</td>
</tr>
</tbody>
</table>

B.2 Example META-file

Below is an example META-file using all possible configurations of options:
B.3 Argument output of example file

Below is the resulting arguments from the example-file in section B.2. Note that the arguments have been re-arranged to line-by-line with added comments for clarity, however the order in which the arguments appear is correct.