OptiWISE: Combining Sampling and Instrumentation for Granular CPI Analysis

Yuxin Guo[§]
University of Cambridge, UK
yg413@cl.cam.ac.uk

Alex W. Chadwick[§]
University of Cambridge, UK
alex.chadwick@cl.cam.ac.uk

Márton Erdős
University of Cambridge, UK
marton.erdos@cl.cam.ac.uk

Utpal Bora
University of Cambridge, UK
ub230@cl.cam.ac.uk

Ilias Vougioukas
Arm, USA
ilias.vougioukas@arm.com

Giacomo Gabrielli
Arm, UK
giacomo.gabrielli@arm.com

Timothy M. Jones
University of Cambridge, UK
timothy.jones@cl.cam.ac.uk

***Abstract*—Despite decades of improvement in compiler technology, it remains necessary to profile applications to improve performance. Existing profiling tools typically either sample hardware performance counters or instrument the program with extra instructions to analyze its execution. Both techniques are valuable with different strengths and weaknesses, but do not always correctly identify optimization opportunities.**

We present OptiWISE, a profiling tool that runs the program twice, once with low-overhead sampling to accurately measure performance, and once with instrumentation to accurately capture control flow and execution counts. OptiWISE then combines this information to give a highly detailed per-instruction CPI metric by computing the ratio of samples to execution counts, as well as aggregated information such as costs per loop, source-code line, or function.

We evaluate OptiWISE to show it has an overhead of 8.1× geomean, and 57× worst case on SPEC CPU2017 benchmarks. Using OptiWISE, we present case studies of optimizing selected SPEC benchmarks on a modern x86 server processor. The per-instruction CPI metrics quickly reveal problems such as costly mispredicted branches and cache misses, which we use to manually optimize for effective performance improvements.

I. Introduction

In the quest for greater performance, application developers and compilers make use of a variety of techniques to optimize code. Of these, profiling is a key enabler for program understanding, in which an application is analyzed during execution to extract information about its run-time behavior. All major compilers contain support for profiling, either for generating profile information [1], [2] or for using it in their optimization decisions [3]–[5]. For developers, there are a number of stand-alone tools to generate profile information [6]–[9], giving insights for source-code transformation.

Previous approaches to profiling can be divided into two broad categories. *Sampling-based* profiling [6], [10]–[15] interrupts program execution to read architectural performance counters, such as the number of CPU cycles and the number of cache misses since the last interrupt. These interrupts can be periodic, random, or triggered by microarchitectural events in the processor, such as cache misses. This breaks execution up into a number of samples, each associated with the program counter at the point that the interrupt occurs. With periodic or random sampling, the number of samples an instruction has is assumed to be proportional to the time used to execute the instruction by the processor [10], [15] (i.e. more samples means more time is spent on that instruction). Sampling-based techniques are typically very low overhead and thus accurately measure real performance, but the results can be subtle and hard to understand, especially on out-of-order processors. In contrast, *instrumentation-based* profiling [16]–[20] inserts extra instructions into the program to count occurrences of fine-grained events during execution, such as the number of times an instruction, basic block, or edge in the control flow graph (CFG) is executed, or the amount of elapsed time between instrumentation points. Instrumentation techniques can thus be deterministic and potentially more fine-grained, but depending on the number of instructions inserted, can be disruptive to the program, potentially changing the performance characteristics. If so, any timing information collected will not necessarily reflect real application performance.

While information coming from the two profiling approaches is useful, as figure 1 demonstrates, looking at the results of sampling or instrumentation individually does not necessarily reveal optimization opportunities. Ideally we would like the CPI (cycles per instruction) or IPC (instructions per cycle) of a fine-grained part of a program, since this indicates how efficiently the pipeline is utilized, but this can not be obtained from either approach alone. Intuitively speaking, instructions whose latency per execution cannot be hidden by the pipeline of modern out-of-order processors (i.e. instructions with high CPI), and that are also executed a sufficiently large number of times, present the most promising opportunities for optimization. Sampling shows when an instruction spends a relatively large amount

[§]These authors contributed equally to this paper.

	Instruction	Samples	OptiWISE Executions	OptiWISE CPI
00	**sub** $0x2770d4ee,%eax	574	256×10^6	0.91
04	**xor** $0x7aa3411f,%eax	618	256×10^6	0.97
0f	**sub** $0x1,%edx	0	256×10^6	0.00
12	**jne** 00	65	256×10^6	0.10
14	**mov** %eax,%edx	0	1×10^6	0.00
16	**add** $0x1,%ecx	0	1×10^6	0.00
19	**and** $0x1fffffff,%edx	1	1×10^6	0.40
1f	**xor** (%rsi,%rdx,4),%eax	247	1×10^6	98.97
22	**cmp** %ecx,%ebx	0	1×10^6	0.00
24	**mov** $0x100,%edx	0	1×10^6	0.00
29	**jne** 00	0	1×10^6	0.00

OptiWISE loop analysis

Loop	Invocations	Avg. Iter.	Time (self)	IPC (self)
00←29	1	1×10^6	100% (17%)	1.70 (0.07)
00←12	1×10^6	256	83% (83%)	2.02 (2.02)

Fig. 1: An x86 micro-benchmark annotated with the number of samples and execution count measured by sampling-based and instrumentation-based profiling tools on a real system. Prior schemes identify instructions `00` and `04` but these are not good optimization opportunities for this application; instead the load (part of `1f`) is the best since it incurs a frequent cache miss. This is revealed if we compute the CPI for each instruction by combining the outputs of the two tools, the technique used by OptiWISE. OptiWISE's loop analysis helps to quickly find interesting regions when looking at large benchmarks.

of time in execution, but the reason behind this is unclear. It could be that the instruction is expensive, or it is possible that this instruction is cheap, but executed often, and hence frequently interrupted for sampling. In other words, sampling approximates the number of CPU cycles spent on an instruction but gives no information about how important this value actually is. On the other hand, instrumentation-based profiling is able to obtain the execution count of each instruction, but this does not indicate if an instruction is worth optimizing either, as its latency per execution may be easily hidden by the pipeline.

Sampling- or instrumentation-based profiling alone cannot provide developers with the cost per execution of an instruction (or other fine-grained code segment), a metric that is often quantified by the instruction's CPI, hence failing to directly show key opportunities for low-level optimization. To address this, we present OptiWISE, which intelligently combines sampling and instrumentation together to compute CPI. We estimate the number of cycles per execution of every instruction in the program, without the need for recompilation or extra analysis of the binary. OptiWISE builds on existing tools, running the application twice to estimate the total number of cycles taken to execute each instruction and to obtain instruction execution counts and the control flow graph. These data are then used to compute CPI values for each instruction and hence identify where the out-of-order pipeline of modern processors stalls, providing direct insights for optimization. OptiWISE supports both x86 and AArch64 architectures in Linux systems without extra compiler support (i.e. only the executable binary is required).

OptiWISE makes the optimization process straightforward

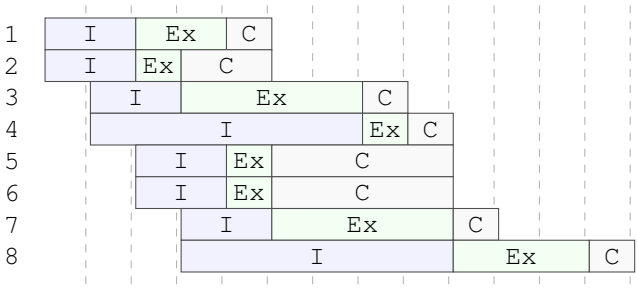

Fig. 2: Timeline of an example dynamic execution of an instruction sequence in an out-of-order superscalar processor. Instructions 2, 5 and 6 are never the oldest instruction in the pipeline on any cycle, so cannot be sampled by perf.

by aggregating instruction-, block-, loop-, line-, and function-level information, and annotating these with the source-code line number gathered from debugging information. We believe loop information is particularly useful and reveals inherently good candidates for optimization: loops execute repeatedly, so small improvements in each iteration can make big differences to performance, especially for loops with high iteration counts.

To demonstrate the benefits of OptiWISE, we present a case study using SPEC CPU2017 [21] benchmarks, showing several optimization opportunities quickly identified by OptiWISE used by a human expert previously unfamiliar with the benchmarks. These give up to 10% speedup with small source-code modifications on a real machine. In summary, our contributions in this paper are:

- A novel profiling approach that combines sampling-based and instrumentation-based profiling together to accurately estimate CPI of instructions and sets of instructions at varying granularities (e.g. loops) on modern high-performance machines;
- A heuristic to reconstruct approximate loop structure from a (possibly complex) CFG, even if nested loops share the same header;
- A case study on SPEC CPU2017 that demonstrates the optimization opportunities found by OptiWISE for real server processors.

II. Background

We discuss techniques related to OptiWISE, and the strengths and limitations of previous approaches.

A. Sampling-based profiling

Modern high-performance commercial application processors almost exclusively use the out-of-order superscalar paradigm. These feature an in-order front-end where instructions are fetched in predicted execution order and issued to an out-of-order back-end. Multiple instructions from the stream can be executed in the back-end at the same time. Ultimately, such processors have some in-order 'complete' or 'commit' queue, which is typically where exceptions or interrupts are handled.

When sampling is applied to these processors, some aspects of the processors' state will be captured stochastically. A typical

technique (used by perf [6]) is to use periodic interrupts to sample the architectural program counter. Since interrupts are handled in the complete queue, this means that the architectural program counter being sampled is, in reality, the address of the instruction at the head of the complete queue. This leads to peculiar or misleading results; for example, some addresses may never be sampled because they are never at the head of the complete queue since they always commit in the same cycle as a previous instruction.

Figure 2 illustrates such a phenomenon occurring in an example program execution. Each row represents a single instruction's progress through the pipeline stages `Issue`, `Execute`, and `Complete`. Randomly sampling the address of the next instruction to complete at one of the 13 vertical time steps illustrated leads to instruction 1 being chosen with probability $\frac{5}{13}$, instructions 3 and 8 with probability $\frac{3}{13}$, and instructions 4 and 7 with probability $\frac{1}{13}$. Other instructions cannot be sampled at all.

Alternative schemes may sample at other parts of the pipeline, but these will also be subject to quirks. For example, sampling components such as the front-end will lead to speculatively fetched instructions being sampled, which never actually execute.

Worse still, a sample may not even be assigned to the instruction that caused the pipeline stall, which is known as 'skid' [10], [14], [22]. To increase accuracy there are many proposals for applying hardware support to attribute samples to the correct instructions [10], [14]. However these schemes are not available in commercial hardware today, and so current sampling-based profiling tools do exhibit quirks in practice.

Another common sampling-based profiling technique is to trigger samples by overflow of counters associated with microarchitectural events such as cache misses. In this case the number of samples associated with an instruction will be proportional to the number of events associated with it. This too has many quirks on modern out-of-order processors, and typically more pronounced skid. Since we use periodic sampling in this work, we do not discuss the overflow of counters further.

B. Instrumentation-based profiling

Instrumentation-based profiling techniques insert monitoring code into the program to count the occurrences of events when executing the program, or to read timers at specific points in the program, helping to understand program behavior. A key design decision with instrumentation-based techniques is where to insert the instrumentation. The more instrumentation that is inserted, the more precise data can be collected, but the more the application performance will be affected, meaning performance measurements become unrepresentative.

For counting execution of instructions, there are generally three basic types of technique [18]: vertex profiling measures how many times a basic block executes [23], edge profiling counts how many times a control flow graph (CFG) edge is taken [16], and path profiling checks how many times a specific path in the CFG occurs (i.e. across multiple branch transitions) [17]. Instrumentation can be placed in basic blocks or on CFG edges. To minimize the overhead of profiling, previous works try to insert as little instrumentation as possible without losing accuracy [16], [17] but need the CFG of the application as a prerequisite for inserting instrumentation.

Despite being able to discover detailed program behavior and features—like the execution count of every instruction, the control flow pattern, and the hot paths of the program—execution-counting profiling techniques alone do not show the execution time of these segments.

There are plenty of existing tools for dynamic code/binary instrumentation, such as DynamoRIO [24] and Pin [25]. These enable instrumentation to be inserted into existing binaries in memory at run-time, without the need for recompilation. We use DynamoRIO in this work to perform instrumentation-based profiling because of its efficiency, support for various architectures, and ease of use. A key difference between our implementation and previous work is that we do not require the CFG of the program before instrumenting, but still keep the overhead to an acceptable level.

C. Detecting loops in a CFG

We use dominance analysis to find loops in the CFG of a program. Previous work [26]–[28] uses the following definitions for such analysis:

Dominance: in a CFG with a source node (e.g. function entry), a node *m* dominates *n* iff every path from the source to *n* goes through *m* before it reaches *n*.

Back edge: an edge in a CFG whose head dominates its tail.

Loop: each back edge defines a loop, whose loop header is the target of the back edge and loop tail is the source of the back edge. The loop body is defined as the nodes that can reach the loop tail without passing through the loop header.

However, this definition of a loop does not necessarily match programmer intuition. A specific case we observed in experiments is that multiple back edges may have the same target (i.e. loops associated with these back edges share the same loop header). In practice, these could either be different control paths of the same big loop (e.g. conditional or `continue` statement) or nested loops. We propose a heuristic to decide whether to merge or split loops that share the same loop header. As a result, we can closely estimate the true loop structure.

III. The OptiWISE Approach

Assuming that a program's execution is interrupted uniformly at random, then for any given set of program addresses A, the expected number $E(S_A)$ of samples observed with a program counter in A is equal to the execution count of the instructions at those addresses, N_A, multiplied by the average time these instructions spend eligible to be sampled per execution, T_A, multiplied by the sampling frequency, f. Here, A is a set of addresses that could correspond to a given function, loop, basic block, source-code line, or even just a single instruction's address.

$$E(S_A) = N_A \times T_A \times f$$

Among the four terms above, we are interested in T_A, as this value helps identify whether the processor pipeline is stalled and thus provides an opportunity for optimization. Using instrumentation tools to profile the application, we obtain $N_{\{a\}}$ for each individual address a in the program, and hence can sum the values to produce N_A for any set of addresses A. From a sampling-based profiling tool such as perf with a sampling frequency f, we can obtain a corresponding sample of the random variable $S_{\{a\}}$, and similarly sum these to produce S_A. By choosing f to be sufficiently high for accuracy, we can compute an estimate of T_A.

On simple scalar in-order processors, where we can assume that all instructions are executed in order and only one instruction can be executed at a time, for a single instruction at address a we may expect that $T_{\{a\}}$ is the average execution time of this instruction. But on modern out-of-order superscalar processors, depending on the sampling method, this may not be the case. Instructions executed by these processors can overlap with each other considerably (illustrated by figure 2 in section II-A). Meanwhile, the processor could assign samples to incorrect instructions due to the complex behavior of modern out-of-order superscalar processors, as discussed in section II-A. Therefore, the samples may not be attributed to the instruction that the processor truly stalled on, which means that the value $T_{\{a\}}$ is not necessarily the execution time of the instruction at address a on such processors.

In practice, we can heuristically estimate the execution time of instructions on out-of-order processors by two approaches:

1) We observe that $T_{\{a\}}$ is mostly determined by the behavior of the previous dynamic instruction of a, not the instruction at a itself, based on experiments in several real out-of-order processors from different vendors with perf (further details in section V-B). So attributing the samples on one instruction to its predecessor is a straightforward but powerful way to increase instruction-level profiling accuracy. We found that processors with Intel PEBS [11] support automatically handle this issue by assigning samples to the correct instructions.
2) OPTIWISE provides loop-level statistics, which consider more instructions in the set of addresses A. Prior work has shown that aggregating samples to coarser-grained segments like basic blocks and functions significantly increases the accuracy of sampling-based profiling (average error rate decreases from ∼60% to 29.9% and 9.1% for basic blocks and functions respectively) [10].

IV. IMPLEMENTATION

OPTIWISE aims to provide information for each instruction, loop, and function of the application to be profiled, allowing users to determine the bottleneck of applications. This section discusses how OPTIWISE works and how it is implemented. We first give an overview of the tool and then describe the implementation details of the key functionalities.

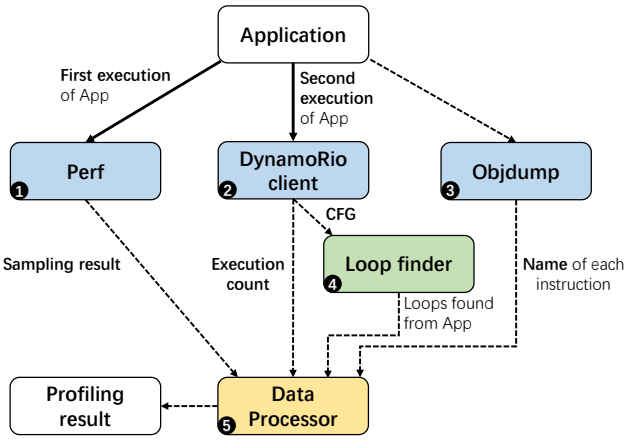

Fig. 3: Overview of OPTIWISE.

A. Overview

Figure 3 shows the tool's structure, which is composed of five components. Three of these components are developed based on existing tools: perf, DynamoRIO, and objdump, each of which has alternatives. The tool takes an arbitrary binary executable (i.e. the application to be profiled) compiled by an independent compiler and executes it twice.

In the first execution, perf (❶ in figure 3) is used to sample the application periodically, recording architectural performance counters and the PC when each interrupt is taken. Perf reports a wide range of hardware and software events in each sample but in this tool we just need three: the PC when each sample occurs, the number of user-mode CPU cycles elapsed since the last sample, and a call-stack trace at the sample point. The CFG of the application with the execution count of each edge is generated by another run with a DynamoRIO client (❷), which instruments the application to perform edge profiling with an acceptable overhead. Separately, we find the disassembly of each instruction and the functions/modules to which they belong by using `objdump` (❸) as a disassembler.

The output of these three components (❶ ❷ ❸) gives everything required to generate the final profiling result (e.g. CPI of an instruction, loop, or function). For security reasons, modern operating systems may randomly arrange the address space of processes, so the absolute address of a particular instruction may change in multiple executions of the application [29]. In the context of Linux, this is called address space layout randomization (ASLR). For this reason, OPTIWISE must aggregate data for each instruction based on its relative address, which is a unique pair of the module identifier and the offset from its base address, instead of the absolute address. The CFG of the application is fed into a loop finder program (❹ in figure 3), which identifies loops using the conventional approach based on dominance analysis. Then the data processing program (❺) reads, combines and processes the raw data, and assigns totals to each instruction and loop. OPTIWISE uses debugging information generated

by the compiler (if available) to heuristically determine the source-code line range corresponding to each loop.

B. Sampling the application

The first task for OPTIWISE is to profile the application by sampling to obtain the number of samples associated with each instruction, the number of user-mode CPU cycles since the last sample, and a stack trace for each sample (❶ in figure 3). The number of user-mode CPU cycles since the last sample is used to weight each sample in order to account for imperfect timing of sampling interrupts. This also helps in situations where unrelated applications or operating-system overheads consume some of the time in a sample period.

In addition, we also record the stack trace for each sample, which is used to generate the call graph, as discussed in section IV-D. Perf has several different mechanisms for capturing the stack at the sample point with different advantages and drawbacks for each; OPTIWISE is agnostic to the approach used. The simplest method is to use the frame pointer to unwind the call stack. However, this relies on the application not having had frame pointers optimized away. Alternatively, DWARF debugging information can be used to capture the call-stack, but this requires that debugging information be available and results in significantly larger traces as the sample requires post-processing.

C. Dynamic edge profiling

With only sampling data available it is hard to identify the fine-grained part of an application (e.g. a specific instruction) that limits performance. To address this we execute the application a second time using DynamoRIO to perform edge profiling on the program (❷ in figure 3). This process generates the CFG with edge frequencies for the application, which enables us to get the execution count for each instruction and identify interesting fine-grained code blocks such as loops.

DynamoRIO views the original application as a collection of basic blocks. It defines a basic block as a contiguous sequence of instructions with only one control-transfer operation, which terminates the block. Therefore, once an instruction is executed, the rest of the instructions in the same basic block will each execute, in sequence. However, this definition of a basic block contrasts with the standard compiler definition, since DynamoRIO allows an instruction to reside in multiple basic blocks (i.e. when a branch targets the middle of a contiguous sequence of instructions). Our CFG uses the standard compiler definition, therefore we must deal with this disparity. To do so we take the prefix of each DynamoRIO block that does not overlap with any other block then calculate the execution count of each CFG basic block by summing up the execution counts of all DynamoRIO blocks that overlap with it.

Dynamic edge profiling without knowing the CFG in advance makes it infeasible to place instrumentation in the edges of interest (which is popular for prior work [16]–[18]), since the position and number of CFG edges are unknown before executing the application. As a result, we add code to instrument edges within each basic block instead, as well as instrumentation to record how many times the blocks are executed (i.e. vertex profiling). The instrumentation code added to profile edges depends on the type of branch that ends the basic block, but all four versions use counters to record execution counts for a subset of the possible edges out of the block, as described in the following paragraphs.

Direct unconditional branch. This type of branch must jump to its target address once it is executed, so we simply allocate space for one edge-frequency counter and insert code to increment it every time the block is executed.

Direct conditional branch. This kind of instruction has two possible target addresses: the jump target address and a fall-through address. Here, immediately before the branch, b, we insert an additional conditional branch, c, that has the same condition as b but a different target address. Branch c targets branch b and jumps over the instrumentation code that increments a fall-through edge-frequency counter. The fall-through counter is only incremented when c is not taken, which means that b will also not be taken. We do not need to instrument the taken edge since we can easily determine its execution count based on the execution count of the basic block and the execution count of the fall-through edge.

Indirect branch. In this case the number of targets is unknown until the whole application has completed execution. This category of branch includes indirect jumps, function calls and returns. Our implementation uses the target address of the indirect branch as the key to a hash table of edge counters. Frequent modifications to data structures like this lead to considerable overhead, but indirect branches generally do not cover a high ratio of all control-transfer operations.

System call. By convention, a system call instruction should always return to the instruction after the one leading to the system call, and this was the case in all our experiments on both x86 and AArch64 Linux machines. Therefore the solution to a system call is similar to direct unconditional branches, but here the edge links to the next sequential block in the address space.

DynamoRIO can instrument the application in two ways: insert a *clean call* or insert *meta instructions*. A clean call allows writing the instrumentation code in C/C++, without worrying about interference between the inserted code and the original application. Although easy to use, this approach causes high overhead as it triggers a context switch every time a clean call is executed. Inserting meta instructions into an application means implementing the functionality at the assembly-code level, and manually maintaining the state of the original application (e.g. register states), which is efficient but harder to implement for complex tasks. In our DynamoRIO client only indirect branches need clean calls because we use a complex data structure (C++ `map`), while the other types can be handled by inserting meta instructions. Again, the total overhead of the clean call is limited due to the small portion of indirect branches in all control-transfer instructions.

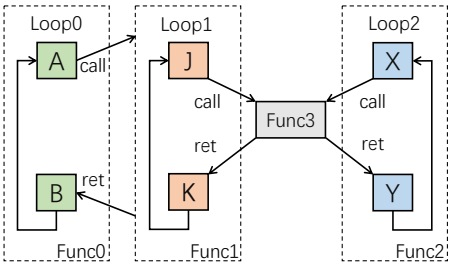

Fig. 4: CFG including three loops that call the same function.

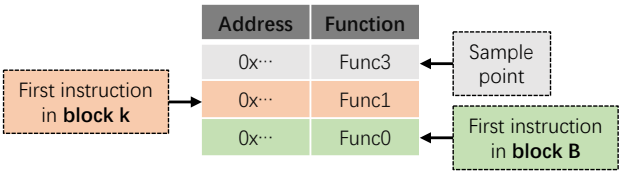

Fig. 5: Example of sample points with *call graph*.

D. Stack profiling

After generating the CFG with edge frequencies, we use dominance analysis to find loops (❹ in figure 3) and send the loop information to the data processing program (❺). For loops without function calls, it is straightforward to aggregate the profiling results by accumulating data corresponding to each instruction that belongs to the loop. However, assigning the correct profiling data to loops (or other sets of instructions) that include function calls is challenging, since functions can have multiple call sites. When a function is called from multiple different loops, we need to correctly subdivide profiling data (samples and execution counts) from the function among these loops to avoid double counting.

Figure 4 illustrates a scenario with two loops (*Loop1* in *Func1* and *Loop2* in *Func2*) calling the same function (*Func3*), and one outer loop (*Loop0* in *Func0*) calling *Func1*. Without additional information, we cannot determine how many samples are caused by the function calls in *Loop1* and *Loop2* respectively, as the control-flow pattern may differ between calls from different loops. Nested function calls further complicate matters. For instance, if another function (*Func4*) calls *Func1*, then the samples in *Loop1* (i.e. blocks *J* and *K*) and samples in *Func3* assigned to *Loop1* should be further subdivided into *Func0* and *Func4*. The same issue arises for other profiling data, such as execution counts: not all instructions in *Func3* have the same number of executions, making it impossible to accurately separate them between *Loop1* and *Loop2* without additional information.

One approach to approximately subdivide the profiling data between the different loops is to weight the statistics in a function based on the edge frequencies from function call instructions to this function. An approach similar to this is used by gprof [8]. Consider the example in figure 4. Assume that *Func3* is called 400 times in total, 300 times from *Loop1* and 100 times from *Loop2*. Then it is reasonable to approximately attribute $\frac{3}{4}$ of the samples in *Func3* to *Loop1* and $\frac{1}{4}$ of the samples in *Func3* to *Loop2*.

However, this method has two drawbacks. First, this estimate is not guaranteed to be accurate as it is unknown how a function behaves when it is called from different parts of the program. This is compounded in long and complex function-call chains as the inaccuracy in each estimate will aggregate. Second, we have observed several edge cases that should be specifically handled when using this approach, including function calls without using a *call* instruction (e.g. lookup in `.PLT`, the procedure linkage table), functions that throw exceptions and recursive function calls.

To attribute profiling data to the correct loop in an efficient and straightforward way, we gather additional call-stack information at run time, which allows us to attribute the correct profiling data to loops without introducing complicated algorithms in the data-processing program. We call this technique *stack profiling*. We discuss below how we attribute correct samples and edge-profiling results to loops.

Sample points. When sampling the application during the first execution, perf records *call graph* information, in the form of a stack trace for each sample. As illustrated in figure 5, this shows the call stack, with the sample point on top, and the outermost function on the bottom. The address for the sample point is the sample's PC value, while the address in each of the other rows shows the return address of the corresponding callee function (i.e. the function in the row above). For example, in figure 5, it is clear that this sample occurs in *Func3* should be assigned to *Loop1* and *Loop0*, instead of *Loop2* or any other functions calling *Func1*. The handling of nested loops and recursive functions is discussed below.

Execution counts. Similarly, we need to attribute edge-profiling results to the correct loop. To achieve this we maintain a stack of counters and record the number of instructions executed during each call to each function and all its callees. Specifically, we instrument the binary with inlined meta instructions as shown in algorithm 1. A table indexed by the address of function-call instructions (*callee_count_table*) is then used to record the number of instructions executed in callee functions at each call site. If a loop calls functions, the *callee_count_table* is used to find how many instructions are executed in the callee functions in this loop.

When there are nested loops, even across functions, the data is associated with all the loops within the nest, so that the total cost of each loop can be accumulated. In the example shown by figure 4, *Loop0* contains all statistics in *Loop1*. However, in the event of recursive functions, extra care is needed to avoid double counting the data within a loop, as the same loop may appear multiple times on the call stack. For this reason, we choose to only consider the most recent instance of each function on the call stack if it occurs multiple times. The whole process increases the overhead of the DynamoRIO client; if the user is only interested in instruction-level or basic-block-level properties then stack profiling can be turned off.

Algorithm 1. Instrumentation for stack profiling.

Annotation 1 In each basic block.

```
1: global_counter += block_size
```

Annotation 2 Before *call* instructions.

```
1: call_stack.push(call_instruction_addr)
2: counter_stack.push(global_counter)
3: global_counter ← 0
```

Annotation 3 Before *return* instructions.

```
1: call_addr ← call_stack.pop()
2: callee_count_table[call_addr] += global_counter
3: global_counter += counter_stack.pop()
```

E. Loop merging

In carrying out our case study on benchmarks like SPEC CPU2017 (see section VI), we found multiple loops that share the same head address (i.e. different back edges with the same target). Previous works that identify loops from the CFG or application binary either do not discuss this special case or simply regard them as different control flows of the same loop [20], [26]–[28]. However, we have determined that this pattern can occur with nested loops too (shown in figure 6). Therefore, we present a heuristic used to split potential nested loops sharing the same loop head and merge potentially different control flows of the same loop, to attempt to more closely match programmer intuition on the definition of a loop.

The heuristic follows a straightforward but practical rule: in a set of all loops sharing the same head address, a loop is nested if and only if it is the subset of some other loop in the set, and its back-edge frequency is at least T times larger than the sum of back-edge frequencies of all its supersets. We choose $T = 3$ based on qualitative experience in our case studies. As shown in algorithm 2, we iteratively identify and 'peel' the outermost (high-level program) loop. In each iteration, loops that are identified as nested are kept in inner_loops, while non-nested loops are removed and collected in current_loop. The set of blocks contained in any loops in current_loop is identified as the current outermost program loop. After outputting it, we iterate in order to process nested loop levels in the same way (only considering the loops remaining in inner_loops).

Consider the example in figure 6, which contains five back edges and hence five loops by default without applying the heuristic. Our heuristic identifies three individual loops among them (i.e. three of the five loops are merged). Loop X and Loop Y are recognized as inner loops as both of them are subsets of other loops and have back edge frequency 3 times larger than the sum of all their supersets. More specifically, our heuristic spends 3 iterations to deal with the loop in figure 6. Details of each `while` iterations are illustrated in table I.

This heuristic will not necessarily make decisions that match exactly with loops found in the source code. On the one hand, a true nested loop with a small back-edge frequency may be incorrectly merged with the outer loop. On the other hand, a frequently executed control-flow statement in a loop (e.g. a C/C++ `continue`) may be identified as a nested loop. However, for performance analysis the heuristic more closely follows the run-time behavior of loops on the processor and allows us to find the interesting loops. In particular, (1) if an inner loop does not have a larger iteration count than its outer loop, then regarding it as a control-flow path of the outer loop is not a big mistake; and (2) if a specific control-flow path of a loop is often taken, then identifying it as a nested loop makes sense.

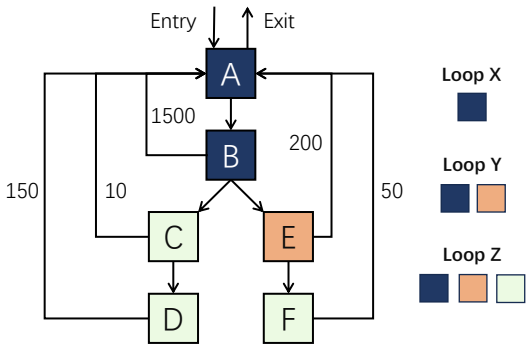

Fig. 6: Example of loops sharing the same loop head address that are processed by our heuristic. There are three loops identified from the five back edges by our heuristic: Loop Z is the outer loop that includes both Loops X and Y, whereas Loop Y includes X. Loop Z has three back edges whereas the others have just one each.

Iteration	current_loop	inner_loops	output
0	*Loop*{*A,B,C*}, *Loop*{*A,B,C,D*}, *Loop*{*A,B,E,F*}	*Loop*{*A,B*}, *Loop*{*A,B,E*}	*Loop Z*
1	*Loop*{*A,B,E*}	*Loop*{*A,B*}	*Loop Y, Loop Z*
2	*Loop*{*A,B*}	∅	*Loop X, Loop Y, Loop Z*

TABLE I: Content of variables after line 15 in algorithm 2 when handling the example in figure 6 with our heuristic.

F. Combining multiple runs

One key limitation of OPTIWISE is that the application must be run twice with different profiling techniques, and the results combined. This can cause issues with non-deterministic applications as the two runs may not have identical control flow. In particular, multi-threaded applications may behave very differently due to the timing differences in actions such as acquiring locks and spawning threads. The operating system may also introduce non-determinism in the results of system calls (e.g. `time`). In practice, we find that applications do not need to be entirely deterministic for the results of OPTIWISE to be useful; as long as the most frequently executed parts of

Algorithm 2. Handling loops sharing the same head address.

Require: same_head_loops
1: inner_loops ← sort_size_ascending(same_head_loops)
2: output ← ∅
3: T ← 3 ▷ Constant: relative edge frequency threshold
4: **do**
5: current_loop ← ∅
6: **for** i ∈ inner_loops **do**
7: freq_sum ← 0
8: **for** j ∈ inner_loops **do**
9: **if** i ⊂ j **then**
10: freq_sum += j.back_edge_freq
11: **if** freq_sum = 0
12: **or** T × freq_sum > i.back_edge_freq **then**
13: current_loop.push(i)
14: inner_loops.pop(i)
15: output.push(⋃ current_loop)
16: **while** inner_loops ≠ ∅ **end**

the program have similar control flow, the two runs will be statistically representative and the combined result meaningful.

On the other hand, the great advantage of using multiple runs is that the sampling run can be very representative of the true performance characteristics of the application, as the overhead of sampling-based profiling is typically very low (e.g. 2%). This, combined with the accuracy of instrumentation-based profiling, allows the combined results to have the advantages of both techniques, without many of the disadvantages of either. The requirement for multiple runs is thus a potentially reasonable trade-off depending on the nature of the program being profiled. The developer may often be able to expend some effort to make a deterministic version of the application for the purposes of profiling; for example, using a synthetic pseudo-random input rather than human input to make experiments repeatable, or profiling a single-threaded version of the application instead of a multi-threaded one.

V. Evaluation

Throughout this paper we use an Intel Xeon W-2195 system [30] (Ubuntu 20.04, 2.30GHz, 256GB memory, 1.1/18/24 MiB L1/L2/L3 cache) as the evaluation machine, and compile workloads with GCC v9.4.0 using `-O3 -march=native`, unless otherwise noted.

A. Performance

A key metric for any profiling tool is the time overhead compared with running the program without profiling. To evaluate OptiWISE we ran it on the benchmarks in the SPEC CPU2017 benchmark suite. Figure 7 shows the run time of OptiWISE on various benchmarks from the SPEC CPU2017 benchmark suite. Overall the geometric mean slowdown is 8.1× the base run time for both profiling runs. The analysis took on average 10 seconds, the worst case being 2 minutes for the *wrf* benchmark.

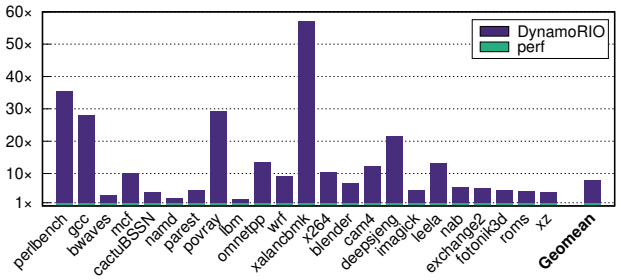

Fig. 7: Overheads of the profiling components of OptiWISE compared with an unprofiled run for SPEC CPU2017 benchmark programs with training inputs on the evaluation machine.

The perf sampling component is the fastest, which is to be expected given the low overhead of sampling-based profiling. It has a geometric mean slowdown of 1.01× compared with the baseline, and is typically the fastest part of OptiWISE. In terms of memory usage, perf consumes negligible run-time memory compared with the baseline. The file size of the profile data is around 160 KiB/s at a sampling frequency of 1,000 Hz in our experiments. Practically, we have found that the sampling frequency can be much lower for long-running programs with consistent behaviors, which reduces this space requirement proportionally.

The DynamoRIO client instrumentation is the most expensive part of OptiWISE, with a geometric mean overhead of 7.1×, but a worst case of 56× on one benchmark, *xalancbmk*. The overhead is higher in applications with a larger number of indirect branches. The run-time memory overhead is typically negligible, though in one case (*wrf*) it is as high as 2.1× the baseline. The memory cost should be proportional to the size of the program's code and control flow graph, so programs with many instructions and many indirect branches will have the worst overheads. The total size of the output files generated by the client was never more than 10 MiB in our experiments, and is proportional to the size of the control flow graph, rather than execution time, so we would not expect it to be significantly larger even in long-running programs.

The analysis component of OptiWISE includes the loop finder and the data processor. The time taken to run this component is determined by the size and complexity of the CFG of each function, rather than the execution time of the original program. For 17 of the 23 benchmarks, this analysis took under one minute, with the worst being *wrf* at 2 minutes. The memory usage of the analysis component is also proportional to the size and complexity of the CFG, and in our experiment was 1.3 GiB in the worst case, 615 MiB on average.

B. Sample attribution

As discussed in section II-A, perf (and other current software sampling-based profiling approaches) exhibit two main quirks when applied to current commercial out-of-order processors: overlapping executions in the pipeline, and miss-assigned samples. It is the second issue that causes the most difficulty in identifying the instruction to be optimized as this issue could

Instruction	Samples	
`mov %esi,(%r14,%rdx,4)`	86	Long-latency store
`add %ecx,%exi`	39318	Commit group start
`xor %ecx,%esi`	1	
`add %ecx,%exi`	0	
`xor %ecx,%esi`	0	
`add %ecx,%exi`	100	Commit group start
`xor %ecx,%esi`	0	
`add %ecx,%exi`	0	
`xor %ecx,%esi`	0	
`add %ecx,%exi`	87	Commit group start
`xor %ecx,%esi`	0	
`add %ecx,%exi`	0	
`xor %ecx,%esi`	0	

Fig. 8: Micro-benchmark with a slow store instruction followed by independent arithmetic instructions on x86 on the evaluation machine.

Instruction	Samples	
`udiv w10,w9,w2`	122	Long-latency division
`orr w2,w10,#0xffffffff`	202	
`add w9,w2,w9`	320	
`eor w9,w2,w9`	242	
`add w9,w2,w9`	143	
`eor w9,w2,w9`	240	
...	...	42 simple instructions
`add w9,w2,w9`	1114	Head of ROB at interrupt if `udiv` is executing

Fig. 9: Micro-benchmark with a slow division instruction followed by dependent arithmetic instructions on AArch64 profiled with perf on a Neoverse N1 processor [31].

generate a misleading profiling result that gives the wrong cost to instructions.

We report on the two issues here based on several experiments on Intel x86 and Arm AArch64 machines. We make the empirical observation that on these machines *perf assigns samples to the instruction after a long-latency instruction*. For example, the instruction after a frequent-cache-miss memory instruction will have many samples associated with it as a result of being after a long-latency instruction. Our assumption for why this occurs is that the timer interrupt gets scheduled to the next instruction boundary, which is after the long-latency instruction. When the interrupt is taken the PC points to the next instruction. With this observation we therefore heuristically assign the expected cost of a sample to the predecessor in execution order, which mitigates some of the profiling inaccuracies caused by sampling on these processors.

Meanwhile, we also find that the Intel PEBS [11] feature has a correction for those naturally expensive instructions such as cache-miss memory accesses or complex arithmetic operations. For example, the result shown in figure 1 is automatically generated on a processor with Intel PEBS support. Although Intel PEBS automatically assigns the majority of samples to the correct instructions, it does not handle mispredicted branches, whose samples are sometimes assigned to the branches themselves, and sometimes to their target instructions. Therefore, heuristic adjustment is still required for these mispredicted branches and machines without Intel PEBS support.

x86: We ran experiments on the evaluation machine without Intel PEBS. Figure 8 shows an example program including the assembly code and the number of samples assigned by perf. In this program, the `mov` operation (indicated in red) is a very slow memory access due to frequent cache misses. The instruction immediately afterward consequently shows a much higher sample count, due to occupying the oldest position in the reorder buffer more often (indicated in orange). Meanwhile, the following arithmetic instructions will normally all execute out of order whilst the store is taking place, and therefore are rarely sampled. Once the `mov` is complete, every 4 instructions (separated by green lines) are committed from the reorder buffer due to the maximum commit rate of 4 operations per cycle in this processor. Hence, the first of these 4 instructions (indicated in blue) occupies the oldest position in the reorder buffer for roughly one cycle, and is thus sampled with moderate frequency.

AArch64: We ran experiments on a system based on Neoverse N1 cores [31]. Whilst observing very similar anomalies to the x86 machine, one observed difference is that an operation that is dispatched and cannot abort is likely immediately removed from the reorder buffer, even if it may not yet have executed, due to a data dependence for example. Operations can be aborted if they may fail (e.g. a load or store instruction) or they are speculative (they follow a conditional branch that has not yet committed). In practice, this means that long sequences of instructions that never abort yield strange sampling behavior. For example figure 9 shows a slow `udiv` instruction (indicated in red) followed by a series of arithmetic operations that cannot abort. This results in frequent samples on an instruction much later (indicated in orange), specifically 48 instructions afterward. We infer that 48 is the number of arithmetic operations that this processor can issue before back-pressure occurs, so we conclude that this must therefore be the instruction at the head of the reorder buffer when the interrupt occurs when the `udiv` is executing.

OptiWISE does not attempt to address these sampling quirks in its output. We leave it to future work to attempt to address these automatically.

VI. Case Studies

Whilst performing extensive analysis and optimization is beyond the scope of this paper, we showcase OptiWISE's potential to find optimization opportunities using some examples. One of the authors identified these examples over the course of three workdays. They randomly picked four benchmarks from SPEC CPU2017, then manually optimized their performance over the 'train' (training/profiling) input set with the help of OptiWISE's output, and finally evaluated improvements over the 'ref' (reference/evaluation) input set on the evaluation machine. The author was previously unfamiliar with the codebase of these benchmarks, but had experience optimizing for modern CPUs. Of the four benchmarks investigated, three were successfully optimized.

A. 505.mcf

Mcf is a vehicle routing benchmark written in C. OPTIWISE suggests that 61.1% of the execution time of the benchmark is in a function called `spec_qsort` and its callees. This function is an implementation of the quick-sort algorithm and is behaviorally identical to the C standard-library function `qsort`. Hence, `spec_qsort` calls to a comparator function handed in as an argument. In 92% of cases, the comparator function is `cost_compare` (assembly code shown in figure 10), which accounts for 23.7% of the benchmark execution time alone.

From the OPTIWISE output we can infer that the jump instructions are very expensive and the instructions after are not, which suggests that branch-free code would be preferable here and so the code was optimized to this effect by rewriting the C source code, removing `if` statements and using a ternary conditional operator to select a return value (`return a?b:c`). The compiler is able to generate branch-free code using the `cmov` conditional move instruction. A similar function, `arc_compare` was also optimized identically. Another problem suggested by OPTIWISE is that the `spec_qsort` function contains a divide operation with a CPI of 38.12. The second operand of this division is always the same so the code was optimized to calculate the fixed-point inverse of the element size, and use a multiplication and bit shift operation to compute the same result as the division.

A third optimization suggested by OPTIWISE is loop unrolling. In the function `primal_bea_mpp` there is a loop containing 13.6% of the benchmark execution time. This loop has an average of 18.6 instructions per iteration and nearly 4,000 iterations per invocation. These statistics make it a good candidate for loop unrolling. Unrolling improves performance by reducing the number of times the loop condition is checked, and experimentally an unrolling factor of 4 was found to be most profitable.

Overall, these three optimizations improved the performance of the benchmark by 12% when applied to the 'ref' input.

B. 531.deepsjeng

Deepsjeng is a chess-playing benchmark written in C++. OPTIWISE shows it has a relatively flat profile, with time split across many functions where none is clearly dominant. However, notably a function called `ProbeTT` accounts for 16.7% of execution time, but unlike other functions has a terrible IPC of just 0.16, suggesting it as an optimization target. This function performs a lookup into a huge hash-map data structure. OPTIWISE reports one instruction alone accounts for 81% of its execution time: a load instruction that retrieves the stored value from the data structure, with an estimated CPI of 279. It therefore strongly suggests that this load regularly misses in all of the processor's caches and the cache miss is not mitigated by ILP. Furthermore, it also suggests that it would be worth adding even a substantial number of extra instructions if doing so could eliminate this cache miss. To this end, the program was optimized by adding prefetch instructions far in advance of the load. Computing the load address requires a substantial hash computation, on the order of dozens of instructions, but the CPI value of 279 gives confidence that this is justified in this case. The prefetch also has to be performed significantly earlier in the code, before it is certain that the `ProbeTT` function will even be called, meaning there will likely be incorrect prefetches at run time. The code was further optimized by removing a divide instruction from this hash computation, as the second operand of the division was constant throughout a given run of the application.

Overall these changes resulted in a 6.8% performance improvement when applied to the 'ref' input.

	Instruction	Samples	Executions	CPI
00	`mov (%rsi),%rdx`	510	3.241×10^9	0.62
03	`mov (%rdi),%rcx`	394	3.241×10^9	0.48
06	`mov 0x10(%rdx),%rsi`	915	3.241×10^9	1.12
0a	`mov $0x1,%eax`	25	3.241×10^9	0.03
0f	`cmp %rsi,0x10(%rcx)`	1480	3.241×10^9	1.81
13	`jl 30`	1004	3.241×10^9	1.23
15	`mov $0xffffffff,%eax`	1473	2.151×10^9	2.71
1a	`jg 30`	343	2.151×10^9	0.63
1c	`mov (%rdx),%rax`	958	1.134×10^9	3.34
1f	`mov (%rcx),%rcx`	179	1.134×10^9	0.62
22	`mov (%rax),%eax`	580	1.134×10^9	2.03
24	`cmp %eax,(%rcx)`	1264	1.134×10^9	4.41
26	`setg %al`	216	1.134×10^9	0.75
29	`movzbl %al,%eax`	259	1.134×10^9	0.90
2c	`lea -1(%rax,%rax,1),%eax`	690	1.134×10^9	2.41
30	`ret`	4060	3.241×10^9	4.96

Fig. 10: x86 assembly code for the `cost_compare` function in SPEC CPU2017's *mcf* benchmark running with the 'train' input annotated with per instruction CPI as generated by OPTIWISE. In this case, Intel PEBS was enabled, but sampling quirks discussed in section V-B affect the sampling of branches. Consequently, the highlighted costs are associated with the two conditional branch instructions (`13` and `1a`) suggesting that these branches are expensive instructions.

C. 603.bwaves

Bwaves is an explosion simulation benchmark written in FORTRAN. OPTIWISE suggested that a significant amount of the program's execution time was spent in a series of floating-point divide instructions. These were in a loop, dividing by what ultimately amounted to a constant, but the compiler had not optimized this because the `-ffast-math` flag had not been passed, and so this optimization could cause numerical instability. In this case though, a programmer can justify that precomputing the inverse of the division will not significantly impact numerical error, and indeed the result of the benchmark remained within the tolerance that SPEC allows (in fact, the result became closer to the reference answer). This resulted in a modest 2% speedup overall when applied to the 'ref' input.

VII. RELATED WORK

A. Sampling

Besides perf [6], there are other software tools [32], [33] that profile applications by sampling. However, they lose accuracy in modern out-of-order machines due to the reasons discussed in section II-A. To mitigate the inaccurate profiling result,

hardware-supported profiling has been proposed [10], [14]. TIP [10], state-of-the-art in this area, implements hardware modifications to enable highly accurate profiling in modern out-of-order processors. Taming hardware [34] constructs execution counts sufficient for PGO using only performance counters on superscalar out-of-order machines. It discusses a number of interesting aspects for handling the inaccuracies in PMUs. These hardware profiling schemes are orthogonal and complementary to our approach: they aim to provide a solution for attributing samples to the correct instructions.

Continuous profiling [15] also computes the CPI/IPC for instructions using execution counts and samples. The key difference here is that they estimate the execution counts instead of measuring them. The scheme finds all instructions that must (statically) have the same execution count, then uses heuristics to identify which of these might have CPI 1 and thus obtain the count. This is more suitable for in-order cores than today's high-performance out-of-order machines.

B. *Profiling loops*

Many previous loop profiling works have been developed for branch prediction and hence improving ILP. Kobayashi et al. [35] present an algorithm to find loops and analyze their properties (e.g. the number of distinct instructions in a loop and the proportion of instructions executed within loops). De Alba et al. have also improved path prediction inside loops [36], [37] and enabled dynamic loop unrolling [38].

Misailovic et al. [39] profile the performance and quality of service of applications after loop perforation (i.e. reducing the iterations executed by loops), in order to find opportunities for optimization with small quality-of-service losses. One of the most related works to OPTIWISE is from Moseley et al. [20], which applies both sampling- and instrumentation-based profiling tools to profile loops. The key difference between this work and ours is that they do not combine sampling and instrumentation together but develop two alternative approaches (one based on instrumentation while another based on sampling) and hence are unable to provide relative cost (i.e. CPI / IPC) of instructions and loops.

VIII. CONCLUSIONS

We have presented OPTIWISE, a tool for x86 and AArch64 using a novel profiling technique that combines sampling and instrumentation together to perform CPI and IPC analysis on fine-grained parts of a program. The profiled application is executed twice for sampling and instrumentation with average overheads of 8.1×. We have also performed a case study of the tool by optimizing benchmarks from SPEC CPU2017, demonstrating non-negligible speedups with modest source-code changes in a short amount of time.

ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant reference EP/W00576X/1, and Arm. Additional data related to this publication is available in the repository at http://doi.org/10.17863/CAM.104277. OPTIWISE is available at https://github.com/CompArchCam/optiwise.git.

APPENDIX

A. Abstract

Our artifact provides OPTIWISE binaries and source code, and scripts used to reproduce the results shown in the paper (figures 1 and 10 for the case study and figure 7 for tool overhead). In addition, we provide several micro-benchmarks used to evaluate the accuracy of OPTIWISE.

Our binaries are for x86-64 machines. Machines with Intel PEBS support will have more accurate profiling results (we recommend Intel Xeon W-2195 or similar systems to get results similar to ours). The OPTIWISE source code provided in the artifact can also be compiled for AArch64 machines, if required.

Our artifact supports Linux systems and is based on perf, objdump, and DynamoRIO. To reproduce the figure 7 experiment, Python3 and gnuplot are required. Our artifact includes the recommended version of DyanmoRIO by default so there is no need to manually install it.

B. Artifact check-list (meta-information)

- **Algorithm:** Profiling.
- **Program:** Several micro-benchmarks included in the artifact; SPEC2017 required but not provided.
- **Compilation:** CMake 3.5 or above and C++ 11 for OPTIWISE (not necessary as binaries are provided). GCC 9.4 or above and GFortran for SPEC2017.
- **Binary:** Binaries are included and support Linux (we recommend Ubuntu 20.04 or 22.04) for x86-64.
- **Run-time environment:** Provided binaries are for Linux (we recommend Ubuntu 20.04 or 22.04). Software dependencies include perf (whose version is related to the Linux kernel), objdump 2.38 or above, DynamoRIO (included in our artifact), Python3, and gnuplot. Root access is not mandatory but highly recommended.
- **Hardware:** Binaries provided only support x86-64 machines, but we provide source code that can be compiled for both x86-64 and AArch64. We recommend Intel processors with PEBS support (we use Intel Xeon W-2195 in our experiments).
- **Run-time state:** OPTIWISE is a profiling tool, so the results are very sensitive to the run-time state. However, the artifact does not aim to precisely reproduce the results in the paper but gives similar trends, so the run-time state does not need to be precisely the same as in our setup.
- **Execution:** No specific conditions are required. The experiment generally needs two hours but can be significantly longer if the machine has much fewer than 20 cores.
- **Metrics:** The overhead of OPTIWISE (i.e. execution time) and accuracy of our profiling results.
- **Output:** The artifact aims to generate the data for figures 1, 7 and 10 in the paper.
- **Experiments:** Scripts are provided to run experiments.
- **How much disk space is required (approximately)?:** 100MB for our artifact. Around 10GB for the SPEC CPU2017 benchmarks (which need to be obtained separately).
- **How much time is needed to prepare workflow (approximately)?:** Less than 10 minutes to download dependent software and compile the tool. An hour to compile SPEC benchmarks.
- **How much time is needed to complete experiments (approximately)?:** Generally two hours, but significantly longer if the machine has much fewer than 20 cores.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT license.
- **Archived (provide DOI)?:** http://doi.org/10.17863/CAM.104277

C. Description

1) How delivered: The artifact DOI points to a zipped archive containing scripts for running experiments, source code and binaries of our tool. We also link the GitHub page of our source code: https://github.com/CompArchCam/optiwise.git.

2) Hardware dependencies: Both x86-64 and AArch64 machines are supported by OPTIWISE, but our binaries are just for x86-64. We recommend Intel processors with PEBS support like Intel Xeon W-2195 or similar systems for reproducing our experimental results.

3) Software dependencies: Binaries in our artifact support Linux (we recommend Ubuntu 20.04 and 22.04) and rely on perf (whose version is related to Linux kernel), objdump (version 2.38 or above), and DynamoRIO (version 9.0 or above, the recommended version included in artifact). In principle, lower versions for perf and objdump still work but may lead to unexpected warning information or bugs. Additionally, Python3 and gnuplot are needed for plotting the graph for figure 7, but they are not dependencies of the tool itself. To compile the SPEC2017 benchmark, GCC 9.4 or above and GFortran are required.

4) Data sets: No specific data sets are required.

D. Installation

Install the software dependencies of the artifact on Ubuntu:

```
$ sudo apt install gcc g++ gfortran gnuplot
↪  python3
```

perf is a dependency of OPTIWISE that must be installed, but on Ubuntu the package for this is not consistently named. We suggest first running:

```
$ apt list --installed | grep
↪  'linux-generic'
```

This will print `linux-generic/jammy-updates...` or `linux-generic-hwe-20.04/focal-updates...` or similar as output. perf should be available in the correspondingly named `linux-tools` package, so for example:

```
$ sudo apt install
↪  linux-tools-generic-hwe-20.04
```

Binaries of OPTIWISE are provided in our artifact and should be ready for use once the above dependencies are installed. The evaluators can also manually compile the tool from the source code.

Download and unzip our artifact, and then from within the extracted directory run:

```
$ tar xzf optiwise_v0_9_0_x86_64.tar.gz
$ cd optiwise_v0_9_0_x86_64
# add OPTIWISE to the command line
$ export PATH=$(pwd)/bin:$PATH
# check if OPTIWISE is ready
$ optiwise check
```

It will more than likely report that it can't work because of kernel permissions settings. Running:

```
$ sudo sysctl -w kernel.perf_event_paranoid=
↪  2
```

This enables access to performance counters by ordinary users on the system, which will likely make it work.

Otherwise, manually compile OPTIWISE from source code (within the directory of the artifact):

```
$ tar xzf optiwise_source_code_v0_9_0.tar.gz
$ cd ./optiwise
$ make
# add OPTIWISE to the command line
$ export PATH=$(pwd)/install_dir.$(uname -m
↪  )$/bin:$PATH
# check if OPTIWISE is ready
$ optiwise check
```

E. Experiment workflow

Our experiments are performed by profiling applications with the OPTIWISE tool and looking at the result. Ultimately, we use perf and DynamoRIO to run a program twice and analyze the output. This process is summarized by the script and can be executed by running `optiwise run -- bin_to_be_profiled`. It is worth mentioning that the program profiled by OPTIWISE should be an ELF file but not a script.

We provide scripts to reproduce our experimental results automatically, but the experiments can also be performed manually by executing OPTIWISE commands on specific binaries of workloads.

F. Evaluation and expected result

There are three experiments to be reproduced: reproducing the motivating example in figure 1, checking the overhead of OPTIWISE in figure 7, and reproducing the profiling results in figure 10.

In our artifact, there are three sub-folders for each experiment: `./figure1`, `./figure7`, and `./figure10`. Each folder includes a `README.md` file that describes how to run the experiment in detail.

Since OPTIWISE is a profiling tool, its output can be vastly different on various hardware platforms, so it is hard for us to give an exact number of variations in the result. In figure 1, the key observation is that the CPI of the load instruction is much higher than the others. As for tool overhead (figure 7), we would expect that the relative relationship between the execution time of the baseline and profiler (i.e. how many times OPTIWISE is slower than the original program) should not have too much variation in similar hardware systems to ours. As for figure 10, it is likely that totally different results will be obtained on different processors, as these numbers are highly microarchitecture-dependent.

G. Experiment customization

It is easy to tune the experiments by OPTIWISE. On the one hand, by simply changing the binary name profiled by `optiwise run`, a different workload can be profiled as long as it is an ELF file. On the other hand, OPTIWISE is able to run the different profiling stages separately by commands like `optiwise sample`, `optiwise analyze` (use `optiwise help` to see more details) so that the output of each individual step can be tested. For best results, we suggest profiled applications be compiled with debugging information (e.g. by `gcc -g`).

H. Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

References

[1] GHC, "The Glasgow Haskell Compiler," https://www.haskell.org/ghc/, 2023. 1

[2] OCaml, "Chapter 21 Profiling (ocamlprof)," https://v2.ocaml.org/manual/profil.html, 2023. 1

[3] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proceedings of the International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, San Jose, CA, USA, Mar 2004, pp. 75–88. [Online]. Available: https://doi.org/10.1109/CGO.2004.1281665 1

[4] GNU, "GCC, the GNU Compiler Collection," https://gcc.gnu.org/, 2023. 1

[5] rust, "The rustc book," https://doc.rust-lang.org/rustc/what-is-rustc.html. 1

[6] Linux, "perf: Linux profiling with performance counters," https://perf.wiki.kernel.org/index.php/Main_Page, 2023. 1, 3, 10

[7] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250746 1

[8] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "gprof: a Call Graph Execution Profiler," in *Proceedings of the SIGPLAN symposium on Compiler construction (CC)*, 1982. [Online]. Available: https://doi.org/10.1145/800230.806987 1, 6

[9] DynamoRIO, "Dr. Memory," https://drmemory.org/. 1

[10] B. Gottschall, L. Eeckhout, and M. Jahre, "TIP: Time-Proportional Instruction Profiling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2021. [Online]. Available: https://doi.org/10.1145/3466752.3480058 1, 3, 4, 11

[11] Intel, "Intel 64 and IA-32 Architectures Software Developer Manuals," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2021. 1, 4, 9

[12] M. Tancreti, M. S. Hossain, S. Bagchi, and V. Raghunathan, "AVEKSHA: A Hardware-Software Approach for Non-intrusive Tracing and Profiling of Wireless Embedded Systems," in *Proceedings of the Conference on Embedded Networked Sensor Systems (SenSys)*, 2011. [Online]. Available: https://doi.org/10.1145/2070942.2070972 1

[13] A. Djupdal, B. Gottschall, F. Ghasemi, and M. Jahre, *Lynsyn and LynsynLite: The STHEM Power Measurement Units*. Springer International Publishing, 2021, pp. 93–114. [Online]. Available: https://doi.org/10.1007/978-3-030-53532-2_6 1

[14] P. J. Drongowski, "Instruction-based sampling: A new performance analysis technique for amd family 10h processors," *Advanced Micro Devices*, 2007. 1, 3, 11

[15] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous Profiling: Where Have All the Cycles Gone?" *ACM Transactions on Computer Systems*, vol. 15, no. 4, 1997. [Online]. Available: https://doi.org/10.1145/265924.265925 1, 11

[16] T. Ball and J. R. Larus, "Optimally Profiling and Tracing Programs," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 4, 1994. [Online]. Available: https://doi.org/10.1145/183432.183527 1, 3, 5

[17] ——, "Efficient Path Profiling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1996. [Online]. Available: https://doi.org/10.1109/MICRO.1996.566449 1, 3, 5

[18] T. Ball, P. Mataga, and M. Sagiv, "Edge Profiling versus Path Profiling: The Showdown," in *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1998. [Online]. Available: https://doi.org/10.1145/268946.268958 1, 3, 5

[19] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind, "Vertical Profiling: Understanding the Behavior of Object-Oriented Applications," in *Proceedings of the ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, 2004. [Online]. Available: https://doi.org/10.1145/1028976.1028998 1

[20] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri, "Identifying Potential Parallelism via Loop-centric Profiling," in *Proceedings of the International Conference on Computing Frontiers (CF)*, 2007. [Online]. Available: https://doi.org/10.1145/1242531.1242554 1, 7, 11

[21] J. Bucek, K. Lange, and J. von Kistowski, "SPEC CPU2017: Next-Generation Compute Benchmark," in *Proceedings of the International Conference on Performance Engineering (ICPE)*, 2018. [Online]. Available: https://doi.org/10.1145/3185768.3185771 2

[22] H. Xu, Q. Wang, S. Song, L. K. John, and X. Liu, "Can We Trust Profiling Results? Understanding and Fixing the Inaccuracy in Modern Profilers," in *Proceedings of the ACM International Conference on Supercomputing ICS*, 2019. [Online]. Available: https://doi.org/10.1145/3330345.3330371 3

[23] D. E. Knuth and F. R. Stevenson, "OPTIMAL MEASUREMENT POINTS FOR PROGRAM FREQUENCY COUNTS," *BIT Numerical Mathematics*, vol. 13, no. 3, 1973. [Online]. Available: https://doi.org/10.1007/BF01951942 3

[24] D. Bruening, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004. [Online]. Available: https://hdl.handle.net/1721.1/30160 3

[25] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," 2005. [Online]. Available: https://doi.org/10.1145/1065010.1065034 3

[26] T. Wei, J. Mao, W. Zou, and Y. Chen, "A New Algorithm for Identifying Loops in Decompilation," in *Proceedings of the International Static Analysis Symposium (SAS)*, 2007. [Online]. Available: https://doi.org/10.1007/978-3-540-74061-2_11 3, 7

[27] G. Ramalingam, "Identifying Loops In Almost Linear Time," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 2, 1999. [Online]. Available: https://doi.org/10.1145/316686.316687 3, 7

[28] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee, "Identifying Loops Using DJ Graphs," *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 6, 1996. [Online]. Available: https://doi.org/10.1145/236114.236115 3, 7

[29] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits," in *Proceedings of of the USENIX Security Symposium (USENIX Security)*, 2003. 4

[30] Intel, "Intel® Xeon® W-2195 Processor," https://www.intel.com/content/www/us/en/products/sku/126793/intel-xeon-w2195-processor-24-75m-cache-2-30-ghz/specifications.html. 8

[31] A. Pellegrini, N. Stephens, M. Bruce, Y. Ishii, J. Pusdesris, A. Raja, C. Abernathy, J. Koppanalil, T. Ringe, A. Tummala, J. Jalal, M. Werkheiser, and A. Kona, "The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC," *IEEE Micro*, vol. 40, no. 2, pp. 53–62, 2020. [Online]. Available: https://doi.org/10.1109/MM.2020.2972222 9

[32] Google, "gperftools," https://github.com/gperftools/gperftools, 2020. 10

[33] NTNU, "PPerf," https://github.com/EECS-NTNU/pperf, 2020. 10

[34] D. Chen, N. Vachharajani, R. Hundt, X. Li, S. Eranian, W. Chen, and W. Zheng, "Taming Hardware Event Samples for Precise and Versatile Feedback Directed Optimizations," *IEEE Transactions on Computers*, vol. 62, no. 2, 2013. [Online]. Available: https://doi.org/10.1109/TC.2011.233 11

[35] M. Kobayashi, "Dynamic Characteristics of Loops," *IEEE Transactions on Computers*, vol. 100, no. 2, 1984. [Online]. Available: https://doi.org/10.1109/TC.1984.1676404 11

[36] M. R. de Alba and D. R. Kaeli, "Runtime Predictability of Loops," in *Proceedings of the International Workshop on Workload Characterization (WWC)*, 2001. [Online]. Available: https://doi.org/10.1109/WWC.2001.990748 11

[37] M. de Alba and D. Kaeli, "Path-based Hardware Loop Prediction," in *International Conference on Control, Virtual Instrumentation and Digital Systems*, 2002. 11

[38] M. R. de Alba and D. R. Kaeli, "Characterization and evaluation of hardware loop unrolling," in *Proc. of the First Boston Area Architecture Conference*, 2003. 11

[39] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of Service Profiling," in *Proceedings of the International Conference on Software Engineering (ICSE*, 2010. [Online]. Available: https://doi.org/10.1145/1806799.1806808 11