An Extensible and Flexible Methodology for Analyzing the Cache Performance of Hardware Designs

Lucas Klemmer Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
lucas.klemmer@jku.at, daniel.grosse@jku.at

***Abstract*—Caches are essential to achieve high performance in modern hardware designs as they bridge the performance gap between digital logic and memories. However, prior research for analyzing the cache performance does not support the designer during the cache implementation.**

In this paper, we present an extensible, automated, and flexible methodology for analyzing cache performance during HDL design. Our approach works by monitoring cache interfaces based on waveforms from simulators, formal tools, or logic analyzers. Both, the generic cache analysis algorithm and the analysis metrics are design agnostic and can be reused across designs and design configurations. We demonstrate that our methodology is applicable throughout all stages of the hardware development cycle from the first test, to debugging, all the way to multi-million cycle simulations.

I. Introduction

With growing demands on computing performance, industry and academia are shifting to application-specific designs [1]. This shift is complemented by the emergence of a range of sophisticated new *Hardware Description Languages* (HDLs) and hardware generators. Both facilitate the customization of a hardware design to meet application-specific requirements by allowing to easily generate thousands of combinations of *Intellectual Property* (IP) and their parameters. For example, in many *System-on-Chip* (SoC) designs, that are leveraging new HDLs and hardware generators [2], [3], the modification of a single HDL line is sufficient to switch the bus implementation, such as transitioning from AXI to Wishbone. Likewise, changing complete components and behaviors (e.g., peripherals, branch prediction, or caches) can be as simple as adding or removing a few lines in a configuration file.

As the disparity in speed between processors and main memory widened, *caches* became very crucial hardware components [4], [5]. Caches provide faster access to frequently used data and instructions, reducing the time processors spend waiting for information from slower main memory. They exploit the principles of temporal and spatial locality to store relevant data, optimizing data retrieval efficiency. Evaluating cache performance is paramount for system design due to its direct influence on memory hierarchy efficiency [6]. A meticulous analysis of cache behavior enables architects to fine-tune memory access patterns and enhance overall computational performance. Thus, many approaches have been developed, including for instance static cache analysis, cache simulation, analytical cache models, and utilization of performance counters (more details in Section II) to provide guidance in high-level architecture decisions. Important cache parameters strongly connected to the cache architecture include in particular cache size, type (e.g., direct mapped, set associative and fully associative), replacement policy, and writing strategies. However, when it comes to implementing a cache in an HDL, the cache performance has to be analyzed also at this level. Indeed, all the above-mentioned approaches do not target the HDL design process. In fact, evaluating performance metrics (such as cache hits, cache misses, conflict misses, ...) as well as relating the results to software is left to the HDL designer. This means, that they have to set up complex testbenches or extend the cache design with performance counters and "export" this information as the basis for the metric computation. This, however, is clearly not efficient and requires significant effort, in particular HDL code extensions as well as running simulations over and over again.

Contribution: In this paper, we present an **extensible, automated, and flexible methodology for analyzing cache performance of hardware designs**[1]. Our methodology eliminates the need for creating cache models and requires no in-depth knowledge about cache internals, as it derives all information by monitoring the cache's interfaces. It is based on the open-source *Waveform Analysis Language* (WAL)[2] [7], which is a *Domain-Specific Language* (DSL) for analyzing waveforms created by simulators, formal tools, and logic analyzers. WAL is a fully fledged programming language with first class support for hardware concepts (e.g., simulation time or design hierarchy) that allows writing generic analysis programs. These generic programs then require only very little "glue code" to bind them to concrete hardware designs, which makes them very flexible and widely reusable. The proposed cache analysis methodology consists of three components that leverage WAL to be as flexible as possible: The main component is the **generic cache analysis algorithm** that detects requests to the cache, responses, and request drops. This algorithm does not analyze any metric and is only detecting and forwarding events to the concrete analysis metrics.

Those **analysis metrics** are the next component of our

[1]Code available at https://github.com/ics-jku/wal-cache-analysis
[2]WAL is available open-source at https://github.com/ics-jku/wal

methodology. Analysis metrics describe either a single value (e.g., hit-rate, average delay) or a data aggregation function (e.g., collect the moving average during simulation to generate a chart). Each metric can define fully encapsulated local variables and can implement a few callback functions that are invoked by the main algorithm. All metrics are executed separately, don't interfere with others, and are defined using convenient macros. Our methodology contains a set of predefined metrics but adding new ones is also straightforward and requires no changes to the main algorithm.

The final component of the proposed methodology is the **glue code** that is used by the generic algorithm to detect events such as requests or responses. With much of the analysis abstracted away, this glue code usually consists of only a few lines of code per design, which specifies the involved signals or simple conditions on those signals.

We demonstrate that our methodology is applicable throughout all stages of the hardware development cycle from the first test, over debugging, all the way to analyzing multi-million cycle simulations. However, since our methodology does not depend on where the waveforms come from, it can also be applied in a post-silicon setting, for example with waveforms produced by logic analyzers. Regardless of the analysis' scope, the generic algorithm of our methodology typically remains the same, i.e., in rare cases, only minimal adjustments are necessary. Our approach allows analyzing more than just the most basic metrics such as hit/miss rates. For example, it makes it is possible to visualize cache behavior, or to combine the cache performance analysis with the software that is exercising the caches.

The rest of the paper is structured as follows. First, we discuss related work in Section II. Then, we provide a brief introduction into the WAL programming language in Section III. Next, Section IV introduces our cache analysis methodology and presents the individual components in detail. This includes the generic cache analysis algorithm (Section IV-A), the cache metrics (Section IV-B), the small amount of required design-specific code (Section IV-C), and an example of how all components can be combined to analyze a waveform (Section IV-D). Next, we evaluate our methodology in Section V and finally conclude this paper in Section VI.

II. Related Work

Numerous approaches have been developed to evaluate cache analysis problems from various perspectives. In essence, several directions can be distinguished.

Static cache analysis refers to a method of evaluating cache behavior without executing the actual program [8]. It involves inspecting the program's source code, intermediate representations, or binaries to make predictions about cache performance, leveraging abstract interpretation to derive the worst-case execution time and safe approximations [9]. As designers are interested in the average case execution behavior, combinations with symbolic execution have been developed, e.g., [10].

To evaluate new designs and to address the increasing design complexity, cache simulators have been proposed (e.g., [11]). A survey describing 28 cache simulators can be found in [12]. Cache simulators come in various types tailored for different analysis needs. Trace-driven simulators use pre-recorded memory access traces for evaluating cache behavior in specific program executions [13]. Execution-driven simulators simulate cache interactions in real-time during the execution of actual programs on the host platform, providing insights into dynamic scenarios. Full-system simulators simulate entire computer systems, including processors and peripherals, providing a comprehensive environment for evaluating cache behavior in a holistic system context (e.g., [14]). Since simulating caches can be time consuming hardware accelerated cache simulators have been proposed [15], [16] Further, cache simulators adapted to analyzing out-of-order processors [17], [18], [19].

Analytical simulators employ mathematical models to predict cache behavior theoretically, offering valuable insights without the need for program execution (see e.g., [20], [21]).

Finally, there are approaches leveraging performance counters (or more general performance monitoring units) [22] as well as respective APIs to access these hardware units [23]. In the same line of research, approaches using vendor-specific interfaces can be utilized to perform live cache inspection on hardware [24].

In contrast to the reviewed approaches, which mostly focus on design space exploration, our proposed methodology is complementary as it focuses on supporting the HDL design process. Our goal is to provide a generic and reusable solution which allows computing cache performance metrics on waveforms without the need to extend the HDL (e.g., with performance counters etc.) or complex testbench infrastructure. Further, since our methodology works directly on waveforms, and thus, so to say, on the ground truth, no cache model has to be created and kept up-to-date. This allows our model to be additionally employed in *Continuous Integration* (CI) scenarios for example to catch regressions.

III. Waveform Analysis Language

WAL [7], [25], [26], [27], [28], [29] is a programming language for debugging and analyzing waveforms created by hardware simulators or formal tools. In WAL, signals from waveforms can be used like variables, and simulation time and design hierarchy are fundamental parts of the language. Thus, accessing signals in WAL is similar to accessing variables, with the difference that the value returned depends on the loaded waveform and the time at which the signal is accessed.

As an example for WAL, we consider the problem of finding all time points at which a cache is requested. We assume that the cache has a simple handshaking interface with a `req` and an `ack` signal. This problem can be solved using the WAL expression: `(find (= req ack 1))`. WAL has a Lisp-inspired syntax in which expressions follow the form `(op a b c ...)`, where `op` is the name of a function and `a b c ...` are the whitespace-separated arguments to `op`. The `find` function evaluates the inner equivalence check expression

at every timestamp of the waveform and returns a list of all time points at which the expression evaluates to true. The value of the signals are automatically read from the waveform at the correct time by WAL.

Now, we can extend this problem to find all time points at which a cache hit occurs, i.e., at which a request is immediately acknowledged. A cache hit thus occurs when both `req` and `ack` are high, but `req` was low at the previous time point. This condition can be expressed in WAL with the following expression: `(find (&& (= req@-1 0) (= req ack 1)))`. The value of a signal at a previous time can be accessed using the `@` operator (e.g., `x@-1` evaluates to the value of `x` at the previous time point). For more information on WAL we refer to the WAL documentation at https://wal-lang.org.

IV. Cache Analysis Methodology

In this section, we present an overview of the components of our cache analysis methodology. The goal of our methodology is to provide an extensible and flexible way to perform cache analyses. In order to achieve this, our methodology is split in three components: (1) the generic analysis algorithm, (2) the analysis metric implementations, and (3) the design-specific glue code.

The first two components achieve the first goal of extensibility, since the analysis algorithm is generic and works across designs, and since new metrics can be easily added in a plug-and-play fashion without changes to the generic algorithm. Metrics can range form basic hit/miss rates to the generation of visualizations of the cache behavior over the simulation time. Of these parts all but the glue code can be completely reused across widely different designs.

An overview of all components of our methodology is shown in Fig. 1. All components left of the dotted line are reusable across designs and all design-specific components are shown on the right side of the dotted line. In the rest of this section, we present all of the three components in detail. The generic algorithm is introduced in Section IV-A. Section IV-B describes how performance metrics can be implemented, Section IV-C presents how the analysis is bound to a specific design using small amounts of glue code, and Section IV-D concludes this section by showcasing how the analysis is applied to a waveform.

A. Generic Cache Analysis Algorithm

The most abstract layer of our methodology is the generic cache analysis algorithm (top center in Fig. 1). From a high-level, the abstract idea of our methodology can be summarized as follows: The cache analysis runs over the waveform, trying to detect certain events (marked blue in Fig. 1). If an event is detected, a callback function is called for every selected metric (marked dark yellow in Fig. 1). These Callback functions then perform the computation of the metric (e.g., calculate the time between now and the last time the callback was called). They can store their (temporary) results in metric-specific variables (marked light orange in Fig. 1).

In more detail, this means that the first step of our cache analysis is performed by the user by selecting at which points of the waveform data is sampled. This defines the unit of a step forward or backward. For example, if no sampling condition is specified, the time point index is incremented whenever any signal in the waveform changes. However, if the waveform is sampled, for example only on rising clock edges, we can calculate the delay in clock cycles by subtracting the index at which the request was detected from the index at which the response occurred.

Next, the algorithm iterates over all sampled timestamps and checks if the cache is requested using the design-specific glue code. If this is the case, the algorithm stores the current time and moves forward until one of three conditions is satisfied. The first condition is a successful response by the cache (can be either a cache hit or a miss). In this case of a successful response, the algorithm calls the *on-response* callback functions of all registered analysis metrics. The second condition is satisfied when the request is dropped, for example due to a cache flush. In this case, the algorithm calls the *on-drop* callback functions of all registered analysis metrics. Finally, the last condition is satisfied if the end of the trace is reached. In this case, no callback functions are invoked, and the request is ignored. Now, the algorithm advances to the next sampled timestamp to repeat the same process.

After the end of the waveform is reached, the finalization callback functions of all registered analysis metrics are called and their results are collected in a list which is then returned as the result of the analysis algorithm.

B. Cache Metrics

The second most abstract layer of our methodology is the metrics. With the algorithm presented in Section IV-A, our methodology can traverse waveforms and detect events that are important for performing cache analyses. However, it does not compute any metric on its own. This task is exclusively handled by the metric implementations. By splitting the computation of metrics from the generic algorithm, it becomes easy to add or remove new metrics to the analysis without touching the main algorithm. This also allows distributing the cache analysis as a library, enabling users to easily add new (generic or design-specific) metrics. In Fig. 1, the generic metric is shown in the top-left corner.

New metrics can be defined using the *define-cache-metric* macro. Metrics can either be reusable across designs (cf. *cache/hit-rate* in Fig. 1) or design-specific. The general usage pattern of this macro is shown in Listing 1. Each metric can define local variables (Line 3) and local functions (Line 5). These variables and functions live in their own namespace and are only accessible to other functions and callbacks within the same metric. This means that all metrics are separated from another and can be evaluated independently.

Next, each metric can, but does not have to, define the special callbacks shown in the middle section of the *Metric* component in Fig. 1 (dark yellow). This is shown in Listing 1 on Lines 7-10. These callbacks form the core functionality

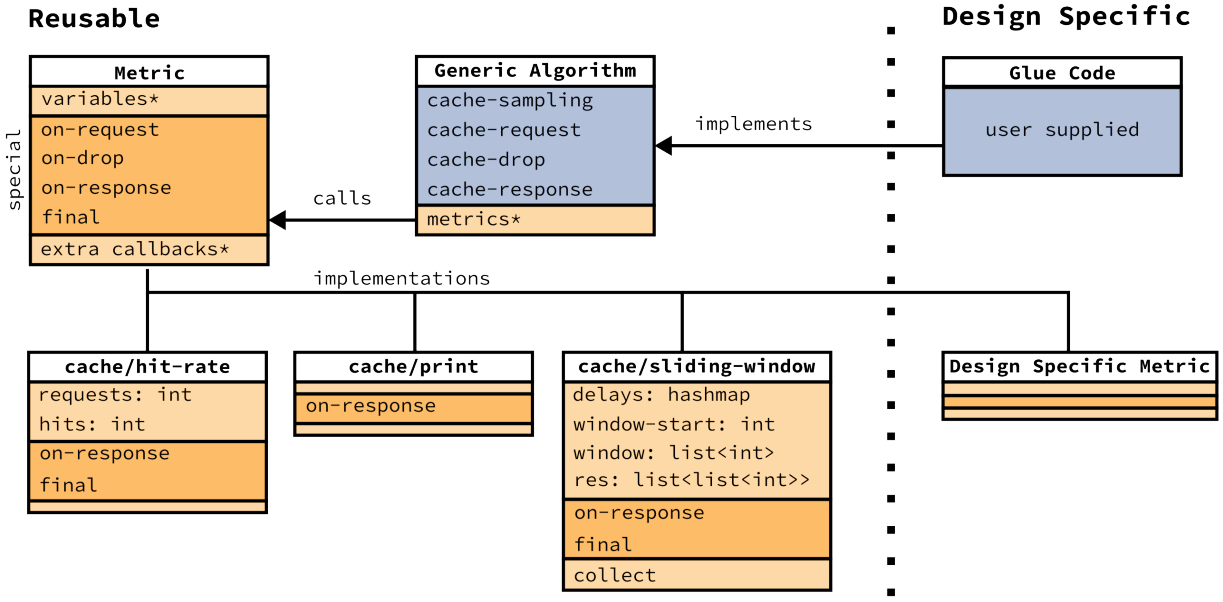

Fig. 1: Cache analysis methodology overview.

```
1  (define-cache-metric name
2    ;; Local Variables
3    (define var 0)
4    ;; Local Functions
5    (defun f-internal [] ...)
6    ;; Special Callbacks
7    (define-callback on-request [])
8    (define-callback on-response [req resp end])
9    (define-callback on-drop [req drop end])
10   (define-callback final [])
11   ;; Extra Callbacks
12   (define-callback cb1 [] ...))
```

Listing 1: Usage pattern for defining new analysis metrics.

```
1  (define-cache-metric cache/average-delay
2    (define responses 0)
3    (define acc-delay 0)
4
5    (define-callback on-response [req-t resp-t end]
6      (inc responses)
7      (set! acc-delay (+ acc-delay
8                         (- resp-t req-t))))
9
10   (define-callback final []
11     (if responses
12         (/ acc-delay responses)
13         "no responses")))
```

Listing 2: Implementation of the average delay metric.

required by most cache analyses. The *on-response* and *on-drop* callbacks receive as arguments the time at which the current request started, the time at which the response arrived or the drop occurred, and the upper-bound timestamp of the current analysis. This upper bound is useful for writing metrics that allow a parallel trace analysis. Finally, metrics can define extra callbacks (Line 12). These callbacks receive no arguments and get called on every sampled timestamp before the conditions for the special callbacks are evaluated.

Exemplary Metric Definition: Listing 2 shows the definition of the metric `cache/average-delay` which analyzes the average delay of cache responses. To perform the analysis, the metric implementation has to keep track of two values. First, the number of responses and second, the accumulated delay. Two variables are defined for this purpose on Line 2 and on Line 3. Next, the *on-response* callback function is defined on Lines 5-8. Whenever it is called, the number of responses is incremented (Line 6) and the observed delay for the current request is added to the accumulated delay counter variable (Line 8). The *final* callback function is defined on Lines 10-13. If the variable *responses* is not 0, the result is the accumulated delay divided by the number of responses, else an error message is returned.

C. Design-Specific Code

The final and lowest layer of our methodology is the glue code that implements the events required by the generic algorithm (c.f., Fig. 1) and the metrics. Since the two previous layers work on abstract events and callbacks, they can be reused across all designs. Most often, this leaves only the lowest layer with design-specific code, which often consists of only a few lines. All parts related to the glue code are highlighted in blue in Fig. 1.

In this section, we present the code required to apply our methodology to the VexRiscv processor [30]. VexRiscv is an open-source, pipelined, and highly parametrizable RISC-V processor. The glue code for VexRiscv's data cache is shown in Listing 3. To improve readability, we first define some aliases for the long signal names on Lines 1-11. Next, we

```
1  (alias clk dut.dataCache_1.clk)
2  (alias rst dut.dataCache_1.reset)
3  (alias valid
4    dut.dataCache_1.io_cpu_execute_isValid)
5  (alias stuck
6    dut.dataCache_1.io_cpu_execute_haltIt)
7  (alias miss
8    dut.dataCache_1.io_cpu_execute_refilling)
9  (alias flush_rdy
10   dut.dataCache_1.io_cpu_flush_ready)
11 (alias flush_vld
12   dut.dataCache_1.io_cpu_flush_valid)
13
14 (defmacro cache-sampling []
15   '(&& (rising clk)
16        (low rst)))
17
18 (defmacro cache-request []
19   '(&& (high valid)
20        (|| (! stuck) miss)
21        (! (high valid miss))@-1))
22
23 (defmacro cache-drop []
24   '(high flush_rdy flush_vld))
25
26 (defmacro cache-response []
27   '(&& (high valid) (low stuck miss)))
```

Listing 3: Glue code for the VexRiscv data cache.

implement the required events. As previously stated, the *cache-sampling* event specifies which time points are considered by the analysis. In combination with WAL's `sample-at` function, this event can be used to practically set the unit of the delay we are analyzing. In the case of VexRiscv, we sample at every timestamp at which the clock is rising and at which the reset signal is low. Requests are dropped if the *flush_ready* and *flush_valid*, signals are both high (Line 23-24)

Next, the *cache-request* is implemented on Lines 18-21. In the case of VexRiscv, a new request starts when the *valid* signal is high and the core is not stuck due to a cache miss (Line 20). Additionally, we have to check if we are currently in an ongoing request which has seen no response until now (Line 21). Finally, the *cache-response* event is implemented on Lines 26-27.

D. Running the Analysis on a Waveform

With the processor-specific glue code defined, the analysis can be run on a waveform as shown in Listing 4. First, on Line 1 the cache analysis library is imported. This makes the generic algorithm and a number of predefined metrics available. Next, the waveform is loaded into WAL on Line 2 and the timestamps at which signals should be sampled are set using WAL's `sample-at` function. A list of all metrics that should be analyzed is created on Line 7. Please note that *cache/hit-rate* and *cache/average-delay* metrics are included with the cache analysis library. Finally, the analysis is started by a call to the *analyze-cache* function, together with the list of callbacks and the time range that should be analyzed on Line 9. In Listing 4, the analysis is applied on the whole waveform from the first timestamp 0 to the last timestamp, which can

```
1  (use cache-analysis)
2  (load "waveform.fst")
3
4  (sample-at (find (cache-sampling)))
5
6  (define callbacks
7    (list cache/hit-rate cache/average-delay))
8
9  (define results (analyze-cache callbacks 0
       MAX-INDEX))
```

Listing 4: Running the analysis on a waveform.

be determined using the `MAX-INDEX` special variable available in WAL.

V. Experiments

In this section, we present experiments that show how our cache analysis methodology is applicable without changes to the automated analysis of various processor configurations (Section V-A), debug scenarios using visualizations (Section V-B), and large-scale (Section V-C) simulation analysis using parallelization.

A. Automated Configuration Analysis

Now we automatically analyze various configurations of the VexRiscv processor. We analyze both, data caches and instruction caches, for cache sizes ranging from 1024 bytes to 32768 bytes and with cache line sizes of 32 and 64 bytes. In the case of the data cache analysis, the instruction cache size is fixed to 8192 bytes with a line size of 32. For the instruction cache analysis, the data cache size is fixed to 8192 bytes with a line size of 32.

Both caches are connected to the same memory via a Wishbone interface. To get realistic delay results in our simulation testbench, we configured the memory to have a read and write delay of 20 cycles. To exercise the caches, we ran the *nettle-aes* benchmark of the Embench suit [31].

Data Cache: The results of four performance metrics of the data cache are shown in Table I. The first two columns list the cache size and cache line size, respectively. The next column lists the *Instructions per Cycle* (IPC) value for this simulation run. To compute the IPC value, we defined a new metric that utilizes an extra callback to count the number of executed instructions and the number of all clock cycles over the complete simulation time. The code of this IPC metric is shown in Listing 5. This IPC metric uses an extra callback named *ipc* that is invoked on every sampled timestamp. This callback counts the number of cycles and the number of executed instructions, which allows the IPC value computation in the *final* callback. We use the IPC value as a proxy to analyze how the cache impacts the overall performance of the processor.

The last three columns list the average read delay over all requests in clock cycles, the average read delay of cache misses in clock cycles, and the read hit rate as a percentage.

```
(define-cache-metric ipc
  (define instructions 0)
  (define valid-cycles 0)

  (define-callback ipc []
    (when (instr-done)
          (inc instructions))
    (inc valid-cycles))

  (define-callback final []
    (when valid-cycles
      (/ instructions valid-cycles))))
```

Listing 5: IPC Analysis Metric.

TABLE I: Data Cache Results.

Size	L. Size	IPC	Delay*	Miss Delay*	H. Rate (%)
1024	32	0.052	43.297	175.418	75.317
1024	64	0.028	85.178	351.457	75.764
8192	32	0.256	6.751	174.996	96.141
8192	64	0.216	9.908	351.325	97.179
32768	32	0.380	2.011	173.346	98.840
32768	64	0.377	2.290	349.592	99.345

* In clock cycles

Based on the hit rate shown in the last column, we can see that the nettle AES benchmark makes heavy use of the data caches. The two configurations with a cache size of 1024 each achieve a hit rate of 74%. Such a low hit rate combined with the high penalty of cache misses of 175 or 351 cycles leads to a very high average delay and, as a result, a low IPC value for those two configurations. With increasing cache and line sizes, the hit rate improves continuously until it is above 99%. The average delay of cache misses is coupled to the line size only, with higher line sizes resulting in a higher delay.

Instruction Cache: Table II lists the performance metrics of some configurations of VexRiscv's instruction cache. Since the IPC, delay, miss delay, and hit rate metrics are generic, they can be used without change for the instruction cache as well. Only the signal names in the glue code and some of the required event conditions had to be changed due to a slight difference in the cache's interface. In the case of the instruction cache, we varied the instruction cache size and instruction cache line size and fixed the data cache size to 8192 bytes and a line size of 32. Compared to the data cache, even smaller instruction cache sizes achieve a much higher hit rate than the data cache. Further, we see that the IPC stops increasing (and actually is the same for data cache size 8192 as in Table I) with instruction cache sizes larger than 8192. This indicates that, in this configuration, the nettle-aes benchmark is limited by the data cache.

Many components of the design influence the results from Table I and Table II. For example, the bus interface between the cache and the memory has an effect on the time it takes to fetch data from the memory into the cache. VexRiscv allows thousands of possible configurations. Analyzing all of them would be out of scope for this paper. However, evaluating them is only a matter of generating and simulating the configurations and running the same analysis on the resulting traces. Since the analysis is run on the waveform of the real hardware design, cycle accurate results are obtainable without access to an up-to-date cache model. This means that the analysis results would still be accurate even when the VexRiscv core itself is updated or extended.

TABLE II: Instruction Cache Results.

Size	L. Size	IPC	Delay*	Miss Delay*	H. Rate (%)
1024	32	0.194	1.524	201.618	0.992
1024	64	0.187	1.624	372.136	0.996
8192	32	0.256	0.196	198.179	0.999
8192	64	0.256	0.190	372.551	0.999
32768	32	0.256	0.196	198.179	0.999
32768	64	0.256	0.190	372.551	0.999

* In clock cycles

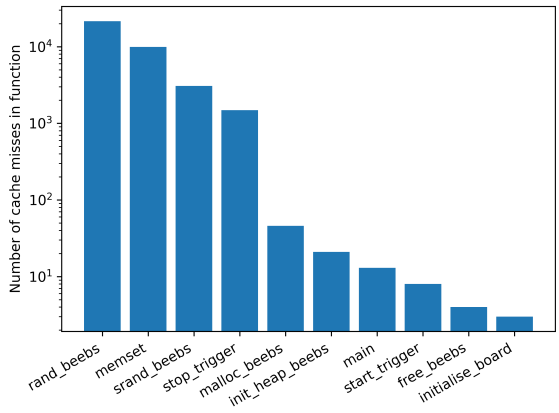

Fig. 2: Functions with the most instruction cache misses.

B. Visualization

Our cache analysis methodology is not only useful to analyze metrics that result in one "hard" answer, such as the hit/miss rate. Analysis metrics can also aggregate data for further processing, such as visualizations.

One example of this is shown in Fig. 2 where the ten functions with the most instruction cache misses of the Embench *tarfind* benchmark are shown in a bar chart. For this metric, we read the symbol table of the ELF file and created a WAL list of all functions with their names and their start and end addresses. The idea behind this metric is to check if the current response was a miss, and, if this is the case, to read the address that was requested. This address is then compared against the list of functions extracted from the ELF file, and a corresponding entry in a hash map is incremented. Finally, a Python function is called that draws the bar chart using the Python plotting library Matplotlib.

Another example of a visualization metric is shown in Fig. 3, which plots the average instruction cache delay

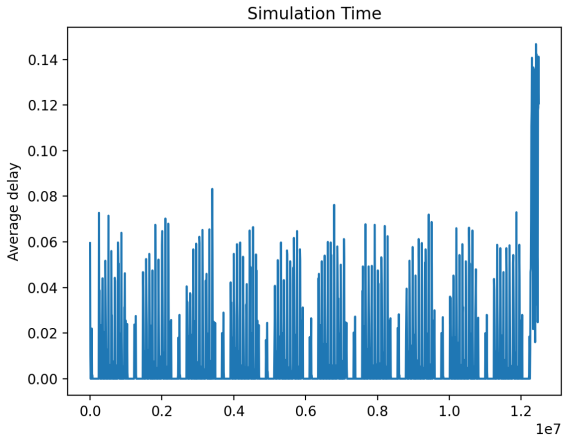

Fig. 3: Sliding window delay (in clock cycles) of IBEX's data cache during the simulation of the Coremark benchmark.

(in clock cycles) over the course of the simulation for the IBEX [32] (formerly known as Zero-riscy) processor. The data is plotted using Matplotlib and aggregated by a metric that moves a sliding window (of configurable width) over the waveform.

C. Large Scale Analysis

The results of simulating large designs with complex firmware are often simulation runs over millions of clock cycles. Analyzing waveforms of this size can become challenging when done in a purely sequential fashion. However, many analysis tasks can be computed on the same trace in parallel by dividing the trace into multiple slices. This is also supported by our methodology through a parameter which specifies the *end* index until which cache requests are handled. By combining this with the `step` function to set the index to some time point, we can analyze a slice of the overall waveform. Please note, that the waveform is not cut at the *end* index and that requests, that start within the slice but whose response occurs after the *end* index, are correctly handled. In many cases, this system allows a very natural way to parallelize metrics that often requires little to no changes to the normal metrics.

To show that our methodology scales to multi-million cycle simulations, we ran benchmarks for millions of cycles on two processors. We then analyzed the instruction cache of IBEX and the data cache of VexRiscv. The results are shown in Table III. To improve the runtimes of the analysis, we parallelized it using the Cuneiform language for large-scale data analysis [33]. Using Cuneiform, we created 64 tasks, each of which processes one slice of the waveform. After the tasks are started, Cuneiform handles synchronization and the passing of all slice results to the final combination function. This function takes and combines the partial results of all slices, for example by taking the average or the sum of all sub results.

All results in Table III were computed on a 64-Core AMD EPYC 7713 Processor with 128 threads and 256 GB RAM.

TABLE III: Parallel Analysis Results.

Core	Benchmark	Cycles	Hit Rate (%)	Runtime (s)
IBEX	coremark	6,380,921	97.3	180
VexRiscv	picojpeg	4,948,761	99.6	242
VexRiscv	qrduino	5,082,492	99.9	155
VexRiscv	huffbench	7,395,706	99.8	381
VexRiscv	tarfind	16,972,672	99.9	620
VexRiscv	wikisort	18,415,326	99.9	335

The VexRiscv configuration uses instruction and data caches of 8192 bytes, while the IBEX configuration uses an instruction cache of 1024 bytes size (IBEX has no data cache). For both, IBEX and VexRiscv, we used the same hit rate metric implementation as in Section V-A Additionally, we added a small WAL function that creates the start and end time points of the slices.

VI. Conclusions

In this paper, we presented an extensible and flexible cache analysis methodology leveraging the open-source *Waveform Analysis Language* (WAL). Our methodology is reusable and easily extendable to new designs. We have shown that our methodology works across all stages of the hardware development cycle in experiments ranging from automated analysis of processors using various cache configurations, over generating visual debug information to the analysis of multi-million cycle simulations. For this, the HDL designer only has to provide minimal glue code to connect their cache interface for the cache performance analysis. In future, work we plan to extend our analysis to more complex setups including processors with cache hierarchies.

Acknowledgments

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

[1] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, p. 48–60, Jan 2019.
[2] L. Truong and P. Hanrahan, "A golden age of hardware description languages: Applying programming language techniques to improve design productivity," in *Summit on Advances in Programming Languages*, 2019.
[3] M. Käyrä and T. D. Hämäläinen, "A survey on system-on-a-chip design using chisel hw construction language," in *IECON 2021*, 2021, pp. 1–6.
[4] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, p. 473–530, sep 1982.
[5] J. Handy, *The cache memory book*. Morgan Kaufmann, 1998.
[6] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2017.
[7] L. Klemmer and D. Große, "WAL: a novel waveform analysis language for advanced design understanding and debugging," in *ASP Design Automation Conf.*, 2022, pp. 358–364.
[8] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm, "Cache behavior prediction by abstract interpretation," in *International Static Analysis Symposium*, 1996, p. 52–66.
[9] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, may 2008.

[10] D.-H. Chu, J. Jaffar, and R. Maghareh, "Precise cache timing analysis via symbolic execution," in *Real-Time and Embedded Technology and Applications Symposium*, 2016, pp. 1–12.

[11] M. Shihabul Haque, J. Peddersen, A. Janapsatya, and S. Parameswaran, "Dew: A fast level 1 cache simulation approach for embedded processors with FIFO replacement policy," in *Design, Automation and Test in Europe*, 2010, pp. 496–501.

[12] H. Brais, R. Kalayappan, and P. R. Panda, "A survey of cache simulators," *ACM Comput. Surv.*, vol. 53, no. 1, Feb 2020.

[13] R. A. Uhlig and T. N. Mudge, "Trace-driven memory simulation: A survey," *ACM Comput. Surv.*, vol. 29, no. 2, p. 128–170, Jun 1997.

[14] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug 2011.

[15] J. Schneider, J. Peddersen, and S. Parameswaran, "Mashfifo: A hardware-based multiple cache simulator for rapid FIFO cache analysis," in *Design Automation Conf.*, 2014, p. 1–6.

[16] ——, "A scorchingly fast FPGA-based precise L1 LRU cache simulator," in *ASP Design Automation Conf.*, 2014, pp. 412–417.

[17] R. J. Douma, S. Altmeyer, and A. D. Pimentel, "Fast and precise cache performance estimation for out-of-order execution," in *Design, Automation and Test in Europe*, 2015, pp. 1132–1137.

[18] K. Ji, M. Ling, Q. Wang, L. Shi, and J. Pan, "AFEC: an analytical framework for evaluating cache performance in out-of-order processors," in *Design, Automation and Test in Europe*, 2017, pp. 55–60.

[19] K. Ji, M. Ling, Y. Zhang, and L. Shi, "An artificial neural network model of lru-cache misses on out-of-order embedded processors," *Microprocessors and Microsystems*, vol. 50, pp. 66–79, 2017.

[20] A. Ghosh and T. Givargis, "Cache optimization for embedded processor cores: an analytical approach," in *International Conference on Computer-Aided Design*, 2003, pp. 342–347.

[21] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, vol. 27, May 2009.

[22] J. Treibig, G. Hager, and G. Wellein, "Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering," in *Euro-Par 2012*, 2012, pp. 451–460.

[23] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, 1999.

[24] D. Tarapore, S. Roozkhosh, S. Brzozowski, and R. Mancuso, "Observing the invisible: Live cache inspection for high-performance embedded systems," *IEEE Trans. on Comp.*, vol. 71, no. 03, pp. 559–572, mar 2022.

[25] L. Klemmer and D. Große, "WAVING goodbye to manual waveform analysis in HDL design with WAL," *IEEE Transactions on Computer Aided Design of Circuits and Systems*, 2024, (accepted).

[26] ——, "Waveform-based performance analysis of RISC-V processors: late breaking results," in *Design Automation Conf.*, 2022, pp. 1404–1405.

[27] L. Klemmer, E. Jentzsch, and D. Große, "Programmable analysis of RISC-V processor simulations using WAL," in *Design and Verification Conference and Exhibition Europe*, 2022.

[28] F. Skarman, L. Klemmer, O. Gustafsson, and D. Große, "Enhancing compiler-driven HDL design with automatic waveform analysis," in *Forum on Specification and Design Languages*, 2023, pp. 1–8.

[29] L. Klemmer and D. Große, "Towards a highly interactive design-debug-verification cycle," in *ASP Design Automation Conf.*, 2024, pp. 692–697.

[30] "GitHub - VexRiscv: A FPGA friendly 32 bit RISC-V CPU implementation," https://github.com/SpinalHDL/VexRiscv.

[31] "Embench: A modern embedded benchmark suite," https://www.embench.org/, 2024.

[32] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? a comparison of ultra-low-power RISC-V cores for internet-of-things applications," in *International Symposium on Power and Timing Modeling, Optimization and Simulation*, 2017, pp. 1–8.

[33] J. Brandt, W. Reisig, and U. Leser, "Computation semantics of the functional scientific workflow language Cuneiform," *Journal of Functional Programming*, vol. 27, p. e22, 2017.