MARTA: Multi-configuration Assembly pRofiler and Toolkit for performance Analysis

Marcos Horro
Universidade da Coruña
CITIC
Spain
marcos.horro@udc.es

Louis-Noël Pouchet
Colorado State University
Department of Computer Science
USA
pouchet@colostate.edu

Gabriel Rodríguez
Universidade da Coruña
CITIC
Spain
gabriel.rodriguez@udc.es

Juan Touriño
Universidade da Coruña
CITIC
Spain
juan@udc.es

Abstract—Benchmarking to characterize specific software or hardware features is an error-prone, arduous and repetitive task. Designing a specialized experimental setup frequently requires writing new scripts or ad-hoc programs in order to properly exhibit interesting performance effects, using code changes and hardware events measurements. These artifacts may have limited reusability for subsequent experiments, since they are dependent on specific problems and, in some cases, platforms.

To improve productivity and reproducibility of such experiments, which are often investigative in nature, we introduce MARTA: a fully customizable toolkit that aims to increase productivity by generating benchmark templates, compiling them, and profiling the regions of interest (RoI) specified using hardware events, and performing static code analysis. MARTA can also be applied on existing code regions of interest, it only requires to write a simple configuration file.

In an orthogonal dimension, the system is able to run various statistical analyses on the measurements collected. MARTA uses data mining and machine learning or AI-based techniques for classification and regression, automatically extracting the features of the experimental setup which have the most impact on performance or whichever other metric of interest, given a large set of experiments and dimensions to consider. These post-processing tasks are valuable for deriving knowledge from experiments and are not included in most profiling tools.

We also provide a set of cases of study to illustrate the ability of MARTA to conveniently create a reliable and reproducible setup for high-performance computing experiments, investigating three vastly different performance effects on modern processors.

Index Terms—profiling, benchmarking, micro-benchmarking, performance analysis, compiler characterization

I. Introduction

Performance analysis is required in any discipline of computer engineering, for both hardware and software effects. From real-time systems, where it is needed to assess the latencies of instructions, to high-performance computing, where codes need to be highly optimized and tuned to the execution infrastructure to extract the maximum possible computing throughput, profiling is crucial for characterizing systems, either in a holistic manner or at a fine-grain level [1, 2].

In order to assess the validity of an experiment, measurements are performed in a platform under a set of conditions. For this reason, measuring events in a system requires the setup of a controlled environment in order to ensure reproducibility and repeatability for a set of experiments, e.g., setting the scaling governors of processors, setting maximum frequencies of clocks, the memory page size used, core affinities, etc. Additionally, passing arguments, macros and other variables to programs is also an error prone task, leading to false positives or other undesired and undefined behaviors within the experimental setup. Checking and setting all these conditions and parameters is usually done manually by users (or semi-automatically using ad hoc scripts).

Orthogonally, when profiling a specific region of code (possibly included in a full application), there could be different dimensions or parameters of interest for the user to consider and configure, e.g., size of rows and columns for a matrix-vector multiplication kernel, data precision (32-bits vs. 64-bits), number of repetitions and step of a loop, access stride, array padding, etc. Analyzing which of these dimensions has the most significant impact in terms of performance can be automated in a similar way, avoiding manual and repetitive post-processing tasks.

In this paper we present a toolkit for automating the experimental setup configuration, compilation, execution and collection of data (static and dynamic) for a code region. In addition, using data mining and machine learning or AI-based techniques for classification such as decision trees and random forests, it generates categories for analyzing the influence of parameters and input arguments in the execution of a code. We make the following contributions:

- The design and implementation of MARTA[1], a framework that fully automates experimental setup, compilation, execution and performance data collection, typically for a computational kernel. The system supports the analysis of post-mortem execution data, the static analysis of binaries through LLVM-MCA, and the automated inspection of compilation logs and optimization reports.
- We automate the specialization of template codes and header files including C/C++ macros to quickly create micro-benchmark versions, controlling features such as allocation strides or array padding, or enabling or disabling compiler optimizations such as dead code elimination or loop jamming that interfere with the correct instrumentation of the region of interest.

---

[1]MARTA is open source BSD-licensed software, available at https://github.com/UDC-GAC/MARTA.

- We present case studies that display MARTA in action, showing how to quickly generate a space of program variants, run them, and use MARTA to investigate the structure of the data collected: decision trees are learned to visualize sub-spaces of interest in the data, by building compact predictors (decision tree, k-means, etc.) for the data analyzed.

The paper is organized as follows: Section II describes the architecture of the presented framework. Section III overviews key aspects of the methodology for reproducibility. Section IV presents a set of case studies that exemplify potential uses of the tool. Section V discusses the strengths, capabilities and limitations of the proposed approach. Section VI presents related work on profiling and monitoring tools, and Section VII concludes the paper.

II. MARTA: SYSTEM'S ARCHITECTURE

The proposed framework is composed of two main modules: the Profiler and the Analyzer. The Profiler is in charge of the compilation, execution and collection of data. The Analyzer inspects the data provided by the Profiler, applying data mining and machine learning or AI-based techniques. Both modules are primarily written in Python 3, C, and Makefile language. The framework works on any operating system, but the compilation and execution facilities are designed specifically for POSIX-compliant systems. MARTA can run on any architecture, the only limitation being the naming of hardware events, specified through configuration files. The high-level architecture of the system is depicted in Figure 1. The two components of the system, Profiler and Analyzer, are independent between them, and can operate autonomously, as they only interface through CSV files containing profiling data. In the rest of the section we describe both modules in detail.

A. Profiler

The Profiler module is designed for parsing the configuration files, compiling all the binary versions specified in them, and running the generated binaries, collecting execution data. The strength of this module lies in its ability to generate as many different executable versions as necessary, as defined by the Cartesian product of the sets of different options in the configuration, e.g., compile-time options (e.g., whether to enable or disable particular optimizations), program inputs, or program features (e.g., -D flags enabling different code paths). The generation of different program versions, which is often a bottleneck in micro-architectural exploration, can be done in parallel. In order to achieve maximum reliability, the Profiler integrates with several different tested-and-true software packages such as the PolyBench/C library [3], using their low-level configuration and measuring capabilities. The upper part of Figure 1 details the design of this module. The Profiler receives two inputs:

- **Configuration file**: a structured YAML file containing all parameters related to compilation (e.g. -D flags, compilers and their flags, etc.), execution (e.g. threads to launch and their affinity, number of repetitions, maximum
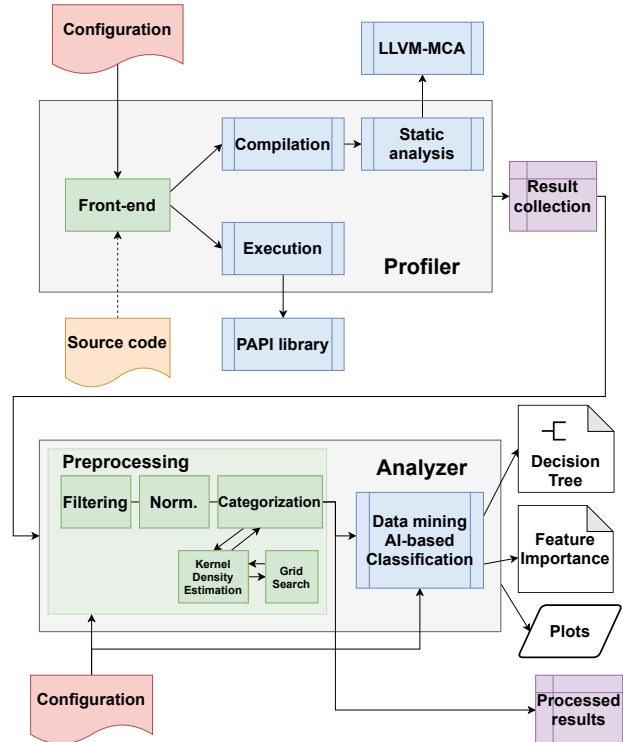

Figure 1: High-level architecture of the toolkit, composed of the Profiler and the Analyzer modules, which operate independently.

deviation in measurements, etc.), and data collection (e.g. output format, dimensions to include, static code analysis parameters, etc.). For convenience, some of these parameters can be overwritten by using CLI arguments.

- **Source code/application**: typically a C/C++ program whose execution prints in standard output values collected from hardware counters, as well as the execution time and values reported by the Time Stamp Counter (TSC). The system helps produce this output format by including a set of functions and macros at runtime.

In order to instrument binaries, MARTA follows the steps detailed in Algorithm 1. The `execute` function used in that approach is disclosed in Algorithm 2. The output generated by all the executions in the experimental set is encoded into a CSV file, which is passed as input to the Analyzer module.

B. Analyzer

The Analyzer integrated in the tool is meant for processing raw data, typically the output of the Profiler, and mining knowledge from these data, primarily through the use of scikit-learn [4]. It can also generate relational plots given a set of dimensions of interest. The inputs to this module are:

- **Configuration file**: a structured YAML file specifying data wrangling parameters (including filtering, normalization and categorization) as well as classification and

Algorithm 1: MARTA generates different binary versions for collecting the execution time, the TSC value and the PAPI events. Each version is executed `nexec` times. Values that deviate farther than a user-specified threshold from the mean are discarded as outliers.

Input: Executable $binary$, boolean $discard\_outliers$, float $threshold$
Result: Dictionary of values
values = {};
for *type in [TSC, time, PAPI_counters]* **do**
    data = [];
    execute_preamble_commands();
    for $i \leftarrow 0$ **to** $nexec$ **do**
        | data.append(`execute`(type, binary));
    end
    execute_finalize_commands();
    if *discard_outliers* **then**
        | condition = (abs(data - data.mean()) <=
        | threshold * data.std());
        | data = data.select(condition);
    avg_value = sum(data) / nexec;
    values[type] = avg_value;
end
return values;

Algorithm 2: High-level approach of the `execute` function: warm up if specified, and then instruments a number of times the specified region of code.

Input: BenchmarkType $type$, Executable $binary$
Result: float
if *hot_cache* **then**
    for $i \leftarrow 0$ **to** $warmup$ **do**
        | run_code($type$, $binary$);
    end
v0 = measure($type$);
for $i \leftarrow 0$ **to** $steps$ **do**
    | run_code($type$, $binary$);
end
v1 = measure($type$);
return (v1-v0) / $steps$;

---

plotting parameters. For classification customization, all parameters follow the same naming or API as in scikit-learn.

- **Input data**: CSV file labeled according to the dimensions of interest described in the configuration file. The Profiler output is fully compatible with this description, but any CSV file can be used as input to this module.

The preprocessing stage is needed for the classification algorithms to only consider the dimensions of interest, and to know the categories into which data will be classified. Preprocessing is performed as follows:

- **Filtering**: based on the dimensions of interest in the problem, e.g., select columns containing a specific set of values, a range, a concrete value and discard the rest, etc.
- **Normalization**: values of interest can be normalized using min-max or z-score techniques.
- **Categorization**: dimensions of interest are typically continuous values, e.g., the performance in GFLOPS, or the average of the TSC values reported. The system is able to discretize these values into a collection of bins or categories. This can be configured statically, by describing the number of categories to create in the interval using a constant step, or dynamically, using kernel density estimation (KDE) for guessing the optimal number of categories to generate, as well as their boundaries. For the hyperparameter tuning in KDE grid search is used, Silverman's rule of thumb for normal distributions [5] and the Improved Sheather-Jones algorithm [6] for multimodal distributions.

The system randomly splits input data into training and testing subsets, following the Pareto principle or 80/20 rule of thumb, for training classifiers. Currently, the Analyzer implements a decision tree and a random forest classifier. The first one is meant to classify target categories depending on the dimensions of interest specified, and the second one to measure their importance. Adding other classifiers such as SVM, k-means, or K-neighbors is trivial thanks to scikit-learn's homogeneous API. The final output produced by the system is composed of the following optional elements:

- **Classification knowledge**: the system outputs the generated classification model as a decision tree. It also shows the accuracy and the confusion matrix for the model. It is possible to employ `dtreeviz` [7] for improving the visualization of the decision tree.
- **Feature importance analysis**: by applying a random forest classifier, the system is able to extract the impurity-based feature importance. This is computed as the total reduction of the criterion brought by that feature. The system performs feature importance analysis using Mean Decrease Impurity (MDI), which counts the times a feature is used to split a node, weighted by the number of samples it splits.
- **Plots**: it is possible to configure the plotting of different types of graphs: scatter plots, KDE plots, etc.
- **Processed results**: a CSV file, similar to the input, containing the results of these processing steps.

III. MEASUREMENT METHODOLOGY

We briefly overview key features of our measurement methodology. As we particularly target small running times for regions of interest (that is, micro-benchmarking), we pay special attention to ensure reproducibility by implementing the following principles: (a) the machine shall be configured in a state that can be reproduced (e.g., fixed frequency, thread pinning to cores, aligned memory allocation); (b) each experiment shall be repeated multiple times until a satisfactory confidence on each measurement is reached; and (c) the measurement

approach shall be as insensitive to the execution context as possible. We highlight our solutions, inspired by good practice in the field and in particular from the PolyBench/C [3] testing harness system.

We remind the reader that conducting experiments correctly so that one can trust the outcome is a particularly difficult and error-prone process in computer science, due to the immense variability induced by the execution context and the multiple ways to implement a program that performs a given computation. The reader is highly encouraged to revisit the guide by Blackburn et al. [8] on how to properly assess experimental evaluations, as MARTA can only go as far as automating tasks, but the user remains responsible for correctly setting up the experiments. This includes which knob(s) of the experimental setup are fixed or left free such as disabling turbo boost.

A. Machine Configuration

We offer various knobs to control the system that will execute the programs. This includes: (a) disabling turbo boost (via MSR); (b) fixing CPU frequency (for Linux systems only); (c) pinning threads to particular cores (using OpenMP environment variables or the `taskset` command, and also using the system calls provided by the toolkit directives); and (d) using an uninterrupted process scheduler (for Linux systems, the FIFO scheduler). Note that most of these knobs require administrator privileges on the host machine. Turning on all these features would ensure that in between two experiments the effects from the operating system's decisions are mitigated: the frequency is fixed for a concrete core (or set of cores) with the proper thread affinity set, allowing to relate cycles to wall clock time easily and systematically for the whole experiment duration. As an illustrative example, running a DGEMM computation may see a variability of over 20% in terms of cycles between two runs of the exact same software on our testing setup, while this variability reduces to less than 1% with the setup fixed by MARTA.

B. Repeating Runs

A possible approach to increase the confidence in the measured values is to repeat the same experiment multiple times to characterize the variation between runs, and determine whether this variation is acceptable. This is a central aspect of reproducibility. MARTA lets the user determine what is the acceptable variability threshold, which depends on the stability of the host machine and how it was configured. It also depends on the data distribution: the variability between runs should be, at least, an order of magnitude lesser than that of the effects that are to be measured.

In MARTA, the default experimental setup is to re-run the same experiment X times, remove the largest and smallest measures from the set (keeping X-2 samples), compute the arithmetic mean of the X-2 samples, and compute the deviation between each sample and the arithmetic mean. If one sample exceeds a threshold T then the whole experiment (the multiple runs of the same program) is discarded, and needs to be repeated. In our tests, we found that X=5 and T=2% are reasonable values for experimental validation (detailed in Section IV).

C. Measuring on the CPU

It is key to ensure that we measure events while understanding their sensitivity to the experimental setup. For example, some hardware counters measure elapsed time (e.g. `CPU_CLK_UNHALTED.REF_P`) while some others are insensitive to frequency and measure elapsed active cycles (e.g. `CPU_CLK_UNHALTED.THREAD_P`). Accurately measuring performance typically involves accurately measuring time. MARTA offers both frequency-sensitive and frequency-independent measurements of time. The number of hardware counters available may be in the hundreds. We preselected in MARTA relevant counters for measuring time, but the user may include other counters to collect data such as data traffic, branch utilization, etc..

Typically processors do not allow to measure more than a handful of counters in the same run in an exact manner. Sampling of the counter value may be implemented instead, and some pairs of counters simply cannot be measured at the same time. To avoid any issue with PAPI counter multiplexing, MARTA performs one experiment per counter to be monitored (exact value, no sampling), running the program with only that counter and the timestamp counter being monitored. For each counter, multiple runs are launched and variability is assessed as described above.

IV. EVALUATION: CASE STUDIES

In order to illustrate the performance and capabilities of the tool, this section describes three case studies which consider three different research questions. Each of them contains the description of the space to explore, the motivation for using MARTA, and the use of the tool in order to answer the research question. Note that all the plots in this section have been automatically generated by the framework, directly using the output of the Profiler, together with a configuration file, as the input to the Analyzer.

A. Micro-benchmarking gather instructions

> **RQ1.-** *How does the performance of SIMD gather instructions vary with respect to the number of cache lines and elements retrieved, both for 128-bit and 256-bit registers, assuming cold cache, and for Intel Cascade Lake and AMD Zen3?*

Definition of the exploration space: Gather is a complex x86 macro-instruction introduced in AVX2 for loading random data points given a starting address and a set of indices. This instruction has been reported to deliver variable latencies [9], depending on the source and destination operands. Hofmann et al. [10] demonstrate the performance variability of the gather instruction for the Intel Knights Corner and Intel Haswell architectures, depending on the number of elements fetched by the gather instruction from a cache line.

```
1  #include "marta_wrapper.h"
2  #include <immintrin.h>
3  void gather_kernel(float *restrict x) {
4      __m256i index =
5              _mm256_set_epi32(IDX7, IDX6, IDX5,
6                               IDX4, IDX3, IDX2,
7                               IDX1, IDX0);
8      __m256 tmp = _mm256_i32gather_ps(x, index, 4);
9      DO_NOT_TOUCH(tmp);
10     DO_NOT_TOUCH(index);
11 }
12 MARTA_BENCHMARK_BEGIN;
13 POLYBENCH_1D_ARRAY_DECL(x, float, N);
14 init_1darray(POLYBENCH_ARRAY(x));
15 MARTA_FLUSH_CACHE;
16 PROFILE_FUNCTION(gather_kernel(POLYBENCH_ARRAY(x) +
       OFFSET));
17 MARTA_AVOID_DCE(x);
18 MARTA_BENCHMARK_END;
```

Figure 2: Input code for micro-benchmarking the gather FP instruction. The configuration file for this benchmark will declare all possible values for the IDXx variables.

Differently, the present experiment explores the impact of the number of cache lines touched by a gather instruction but considering a cold cache, i.e., when data fills come from main memory. We actively avoid any cache prefetching impact, and analyze the real cost of gathering random data elements from main memory.

MARTA in action: The source code employed to explore this search space using MARTA is detailed in Figure 2. The DO_NOT_TOUCH(var) directives avoid any compiler optimization on variable var, e.g., dead code elimination. The assembly code generated for this input is shown in Figure 3. As it can be seen, the instrumentation overhead is minimal. Similarly, any assembly code can be plugged directly into the input source passed to MARTA for compilation and execution.

As an example of our configuration, the possible values for the IDX# variables for gathering 8 elements are:

- IDX0: [0]
- IDX1: [1, 8, 16]
- IDX2: [2, 9, 32]
- IDX3: [3, 10, 48]
- IDX4: [4, 11, 64]
- IDX5: [5, 12, 80]
- IDX6: [6, 13, 96]
- IDX7: [7, 14, 112]

The Cartesian product of these lists of variables generates a space of more than 2K elements, including all combinations of these gather instructions touching any number of cache lines from 1 to 8. The same reasoning is extended for the remaining gather experiments, varying from 2 to 8 data points or elements. In total, we generate more than 3K combinations for each platform. The total execution time for all these experiments is roughly three hours on each machine, including the compilation process.

Evaluation: Our analysis focuses on two factors: 1) the impact of the number of distinct cache lines on the gather instruction, and 2) the difference in performance between two of the latest Intel and AMD architectures: a Xeon Silver

```
1      ...
2      vmovaps     ymm1, YMMWORD PTR [rsp]
3      vmovdqa     ymm2, YMMWORD PTR .LC1[rip]
4      call        polybench_start_timer@PLT
5      test        eax, eax
6  begin_loop:
7      vmovaps     ymm3, ymm1
8      vgatherdps  ymm0, DWORD PTR [rax+ymm2*4], ymm3
9      add         rax, 262144
10     cmp         rbx, rax
11     jne         begin_loop
12     call        polybench_stop_timer@PLT
13     ...
```

Figure 3: Example of 256-bit-register assembly code generated for this experiment: rax holds and offset to avoid data reuse, ymm2 is used to compute the indices, and ymm3 holds the mask (e.g. when gathering less than 8 elements using 256 bits).
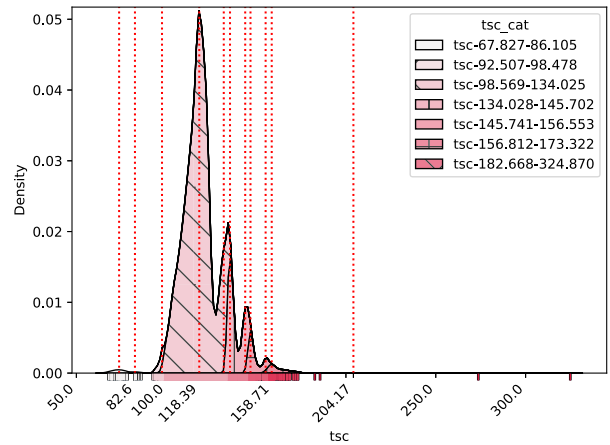

Figure 4: Distribution plot for gather with respect to its performance in terms of TSC cycles (log scale). Vertical dashed lines mark the peak centroids of each category found.

4126 and a Ryzen9 5950X, featuring Cascade Lake and Zen3 architectures, respectively. The target performance metric is the number of TSC cycles, in order to be frequency agnostic.

MARTA generates categories based on the density distribution of the values obtained. Figure 4 illustrates the resulting distribution plot, showing the different centroids and the categories induced by MARTA according to the kernel density estimation (KDE) approach. The legend in this figure describes the categories generated for the TSC values. Even though some of them are not visible in the figure due to their order of magnitude, the system helps to locate them by displaying the peak centroids in the distribution.

Based on this model, the system builds a decision tree as shown in Figure 5, with an accuracy rate of $\approx 91\%$. The model uses the number of cache lines touched by the instruction (N_CL), the vector width (vec_width, 0 for 128 bits and 1 for 256 bits), and the host platform (arch, 0 for AMD and 1 for Intel). The structure very clearly gives the intuition that
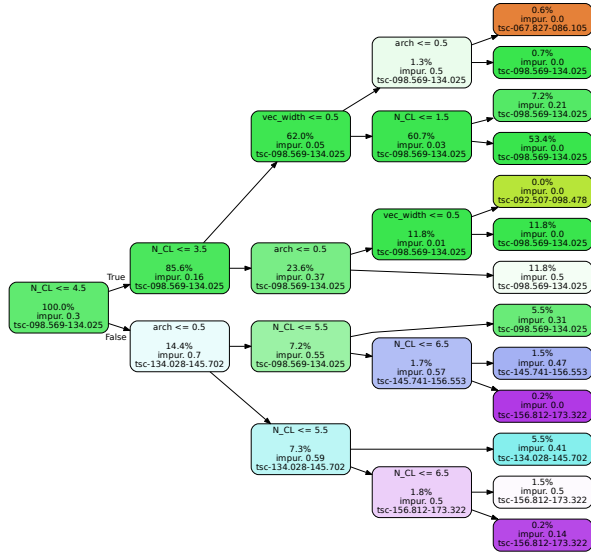
Figure 5: Decision tree for predicting the performance of gather instruction based on the categories generated by the system. `N_CL` is the number of cache lines touched by the program, `arch` is 0 for AMD Zen3 and 1 for Intel Cascade Lake. `vec_width` is 0 for 128-bit vectors, and 1 for 256-bit vectors. Nodes in lighter colors represent a higher impurity degree, which is not desirable.

the degradation in performance is related to the increase in the total number of cache lines touched by the gather instruction. However, this simple model can also discover some other hidden architectural effects, e.g., following the decision tree, our model is able to detect that the AMD Zen3 performs better when the number of cache lines touched is 4 when using 128-bit width vectors. This behavior is not present in the Intel machine.

This decision tree also serves to investigate why the predictor missclassifies certain points. In this specific case, after manual exploration, we found out that most errors are attributable to fuzzy categorical boundaries and natural measurement noise.

In this case we focus on decision trees, as they allow to visualize a partitioning of the space in a manner that is intuitively interpretable by the user. Other techniques such as linear regression might provide lower RMSE, but they are also typically much less intuitive and make knowledge transfer harder than, e.g., a small decision tree as built by MARTA. On the other hand, the feature importance analysis (MDI) reports a high difference between the number of cache lines touched, the architecture, and vector width: 0.78 against 0.18 and 0.04, respectively.

Answering our research question, the performance of gather operations is clearly dependent on the number of cache lines used. The degradation is remarkable when increasing the

number of cache lines touched by the instruction. On Intel Cascade Lake there is no influence on performance of the vector width or the use of masks. There is, however, a noticeable and interesting difference when using the 128-bit width version on AMD Zen3.

B. Empirical throughput of FMA instructions

RQ2.- *How many independent FMA instructions can be executed in just one cycle, assuming hot cache, regardless of data type, vector size and ISA?*

Definition of the exploration space: FMA instructions perform fused multiply-add operations and were introduced as extensions of the 128- and 256-bit SIMD instructions in x86. There were some divergences between Intel's and AMD's implementation at the beginning, but modern architectures, starting from Haswell on Intel and Zen2 on AMD, implement the FMA3 ISA. All instructions available in this ISA have the form of $d = a \times b + c$, where d must be the same register as either a, b or c. As such, there are different variants of these same operations, for instance, `vfmadd132ps` and `vfmadd213ps`, which vary the operands chosen for the multiplication and the destination operand. These instructions have dedicated resources in the pipeline, typically FMA units. However, these units share ports in the pipeline with other architectural units such as the division, integer (e.g. ALU, jump, load effective address, etc.), or shift units. In this case we want to get the actual throughput of any FMA instruction for a given platform, regardless of data type, vector width, or interferences with any other instruction.

MARTA in action: This experiment requires micro-benchmarking, and MARTA includes a specific configuration for this purpose. It also requires hot cache conditions in order to get the maximum throughput of consecutive and independent FMA instructions. We consider two or more FMA instructions to be independent iff there is no data dependence among them. MARTA is able to automatically generate the C code required for benchmarking a list of assembly instructions. It can also generate all the possible permutations of the subsets of this instruction list. This is useful if our experiment requires to consider all possible orderings of the instructions to measure.

For this purpose, we specify the list of assembly instructions to benchmark in a configuration file, as described in Figure 6, or using the CLI, e.g., `marta_profiler perf --asm "vfmadd213ps %xmm2, %xmm1, %xmm0"`. MARTA is also in charge of unrolling these instructions, for reproducibility reasons, and executing warm-up iterations. All these parameters are also configurable during runtime.

Extending the example in Figure 6, MARTA makes it straightforward to write more benchmarks but changing the registers (i.e. vector width) and the data type (`ps` suffix in the code). MARTA automates the generation of these combinations according to the number of consecutive independent instructions we want to issue, from only the first instruction up to all of them.

```
1  asm_body:
2      [
3          "vfmadd213ps   %xmm11,  %xmm10,  %xmm0",
4          "vfmadd213ps   %xmm11,  %xmm10,  %xmm1",
5          "vfmadd213ps   %xmm11,  %xmm10,  %xmm2",
6          "vfmadd213ps   %xmm11,  %xmm10,  %xmm3",
7          "vfmadd213ps   %xmm11,  %xmm10,  %xmm4",
8          "vfmadd213ps   %xmm11,  %xmm10,  %xmm5",
9          "vfmadd213ps   %xmm11,  %xmm10,  %xmm6",
10         "vfmadd213ps   %xmm11,  %xmm10,  %xmm7",
11         "vfmadd213ps   %xmm11,  %xmm10,  %xmm8",
12         "vfmadd213ps   %xmm11,  %xmm10,  %xmm9",
13     ]
```

Figure 6: List of instructions to benchmark for getting the FMA throughput in AT&T format, using 128-bit vectors for single precision.

Evaluation: In this experiment we ran a set of benchmarks varying the following features: 1) number of independent FMA instructions executed contiguously (from 1 to 10), 2) vector width (128 bits, 256 bits, and 512 bits, if available), and 3) data type (single and double precision). A total of 60 benchmarks are generated. We ran these experiments on three different machines: Intel Xeon Silver 4216 (Cascade Lake), Intel Xeon Gold 5220R (Cascade Lake), and AMD Ryzen9 5950X (Zen3). The results are shown in Figure 7 in the form of a line plot, colored by the configuration (data type and vector width), and with the line style according to the architecture used. The figure clearly shows the saturation
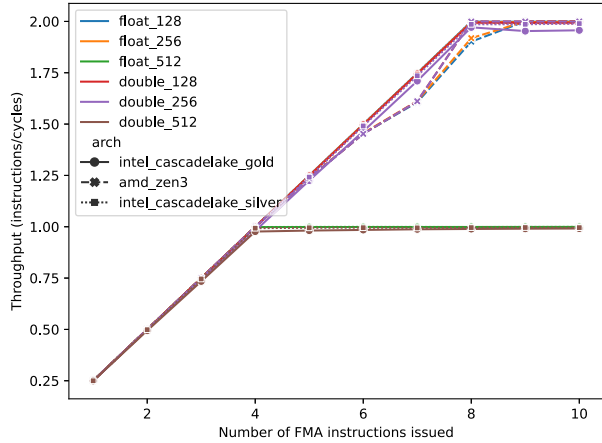

Figure 7: Line plot generated by MARTA according to the number of independent FMA instructions issued and the reciprocal throughput obtained, computed as the number of instructions executed divided by the number of cycles. Some data overlap, but we have preferred to not edit the figure in the interest of preserving the original MARTA output. The figure shows how configurations using 512-bit vectors (`float_512` and `double_512`) are the only ones deviating from the norm.

points for these architectures. *Conducting such experiment can validate, or even replace, the manufacturer documentation on the throughput of specific instructions*.

We observe under which scenario both AMD and Intel

machines allow 2 FMAs to be executed in a single cycle, independently of their vector width. *It requires to have at least 8 independent FMAs in the loop body to achieve a throughput of 2 FMAs per cycle*, as otherwise the throughput is reduced. This experiment highlights how one would fail to achieve a throughput of 2 FMAs per cycle with only two independent FMAs in flight. We suspect this is related to the 4-cycle latency of FMA instructions. For Intel machines using AVX-512, only one FMA can be issued per cycle. This indicates most likely the availability of a single AVX-512 FPU.

MARTA can generate a decision tree-based predictor for all architectures, as shown in Figure 8. This predictor, while naive, is able to extract the importance of the features, accurately categorizing all data points.
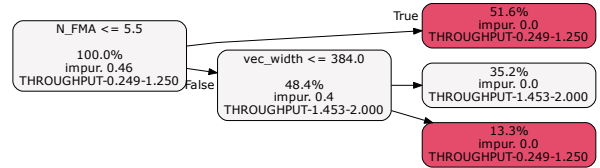

Figure 8: Simple predictor synthesized by MARTA according to the number of FMAs issued and the vector width of the instructions used.

To conclude our case study, we observed that both AMD Zen3 and Intel Cascade Lake have a maximum throughput of 2 FMAs per cycle using vectors of 128 and 256 bits, i.e., they can issue 2 FMAs in a single cycle, provided there are enough independent instructions in flight. AMD Zen3 does not feature AVX-512, while our Intel Cascade Lake processors feature a single AVX-512 FPU. From an architectural point of view, this is typically done by fusing both 256-bit units when issuing instructions using 512-bit vectors.

C. Influence of access pattern on memory bandwidth

RQ3.- *Consider a vector operation of the form $c(f(i)) = a(g(i)) * b(h(i))$. How does the memory bandwidth vary with the access patterns, influenced by the access functions f, g, and h?*

Definition of the exploration space: The performance of memory-bound benchmarks is limited by the main memory bandwidth. This bandwidth is affected by several factors, not all of them intuitive. Modern architectures are designed to efficiently access data in a streamlined fashion by implementing mechanisms such as software and hardware prefetching. In this section, we focus on the analysis a triad operation on a double-precision floating-point data type, studying the effect of changing the element access order.

Intuitively, a purely streamlined access, i.e., $f(i) = g(i) = h(i) = i$, will deliver the best possible bandwidth. But how is this peak bandwidth affected by strided accesses, or even random accesses, in one or more streams, such as could occur for example in Sparse Matrix-Vector Multiplication computations?

MARTA in action: We manually write a tuned version of the STREAM benchmark [11] implementing the proposed

```
1   __m256d regA1 = _mm256_load_pd(&a[data_a]);
2   __m256d regA2 = _mm256_load_pd(&a[data_a + 4]);
3   __m256d regB1 = _mm256_load_pd(&b[data_b]);
4   __m256d regB2 = _mm256_load_pd(&b[data_b + 4]);
5   __m256d regC1 = _mm256_mul_pd(regA1, regB1);
6   __m256d regC2 = _mm256_mul_pd(regA2, regB2);
7   _mm256_store_pd(&c[data_c], regC1);
8   _mm256_store_pd(&c[data_c+4], regC2);
```

Figure 9: AVX triad kernel used for measuring memory bandwidth.

triad operation directly using AVX intrinsics, to avoid compiler optimization interference in the measurements. The code is shown in Figure 9. The accessed elements of each array are determined by the access variables data_{a,b,c}. The values of these variables are the ones that drive the study, and determine whether the access to each stream is sequential, strided, or random. We set up the experiment so that all three streams and access variables are 64B-aligned (i.e., memory block-aligned). This means that strided and random accesses are not defined in terms of individual array elements, but of memory blocks. Once a block is selected, its eight contiguous double-precision elements are accessed (for a and b) or written (for c). We do this so that the total number of data accesses is invariable across different access patterns. Similarly, the total number of cache hits and misses will be invariable in the absence of prefetching, with the rare exception of the same block being selected twice in close succession for the random access experiments. The total number of iterations is equal to the total number of memory blocks in each array, STREAM_BLOCKS. The processor used for evaluation is an Intel Xeon Silver 4216 (Cascade Lake) and, for that reason, the size of each array is defined to be 16 Mi elements, i.e., 128 MiB or at least four times the total LLC size of 22 MiB, as recommended by the STREAM author. When accessing streams with a stride S, the benchmark accesses each block of each array exactly once as follows. During a first traversal, only blocks in positions $B \mid B \bmod S = 0$ are accessed. In a second traversal, blocks in positions $B \mid B \bmod S = 1$ are selected. The process continues until, in traversal $S - 1$, the last untouched blocks are accessed. This avoids unwanted cache reuse with large access strides.

We write the following benchmark versions: one with all sequential accesses which serves as a baseline; four strided versions, one with a stride on b only, one with a stride on c only, one with a stride on b and a, and one with a stride on all three streams; and four random versions in which rand() is used for each randomly accessed stream, in the same fashion as the strided access.

Evaluation: We use MARTA to automatically run 630 different microbenchmarks. Each of the 9 different code versions described above is run using 1, 2, 4, 8 and 16 cores. Each strided version is run with S values from 1 to 8 Ki. We build a decision tree that tries to predict the achievable bandwidth by each access pattern given its stride and number of execution threads. This is useful to bound the performance of different

classes of kernels, e.g., a Sparse Matrix-Vector Multiplication, which is similar to a triad in which one of the streams is accessed randomly. The decision tree shows remarkable impurity when classifying strided accesses. In order to study this, we first focus on analyzing the results obtained for a single thread. These are shown in Figure 10. The bandwidth achieved by fully sequential accesses is approximately 10 times smaller than the peak, at just 13.9 GB/s. Sequential and random accesses are not affected by the STRIDE parameter, and appear in this figure as bounds to the actual performance obtainable by the strided versions. The figure clearly shows how the bandwidth drops with the stride: it drops sharply for $S = \{2, 4, 8, 16, 32, 64\}$, to an average 9.2 GB/s for the case of strided b only. The clear reason in this case is the ineffectiveness of the next-line hardware prefetcher. There is another sharp drop starting at $S = 128$, to an average 4.1 GB/s, which is similar to the performance of accesses using rand(). This is ultimately an artifact caused by the single-threaded execution, as will be shown with multithreaded experiments.
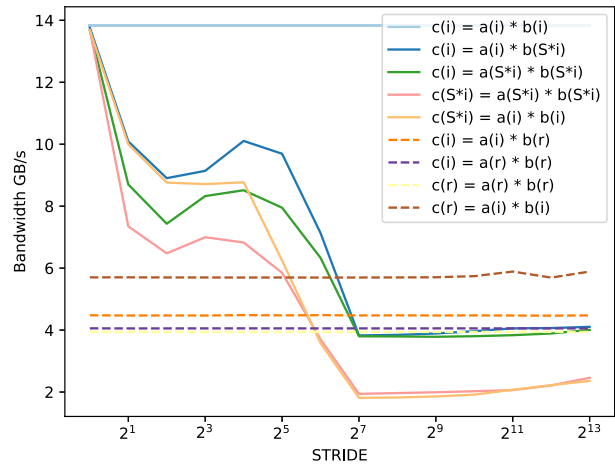

Figure 10: Bandwidth obtained for different access patterns using a single thread. Accesses labeled as x[i] represent sequential accesses, x[S*i] represent strided accesses, and x[r] represent random accesses.

We analyze the bandwidth evolution as the number of threads increases up to the 16 physical cores available in the processor. We presume that the very low bandwidth achieved for a single thread is caused by front-end stalls and, consequently, trivially parallelizing the triad computation using OpenMP should greatly increase the available bandwidth. The results are shown in Figure 11. We can see a clear increasing trend for all benchmark versions, except for those calling rand(). In this case, using multiple threads to access memory is harmful for performance, achieving a low peak bandwidth of only 0.4 GB/s for the version which accesses three random streams through calls to rand(). This low performance is caused by the enormous overhead introduced by the call to rand() as implemented in stdlib, as these versions emit, on average, 5x and 6x more memory loads

and stores, respectively. This effect demonstrates the limits of MARTA: the user remains responsible for understanding and explaining the experimental results. In this case, MARTA identifies a large increase in the number of issued instructions, pointing towards experimental setup effects and, ultimately, to computation bounds introduced by the random-number generator.
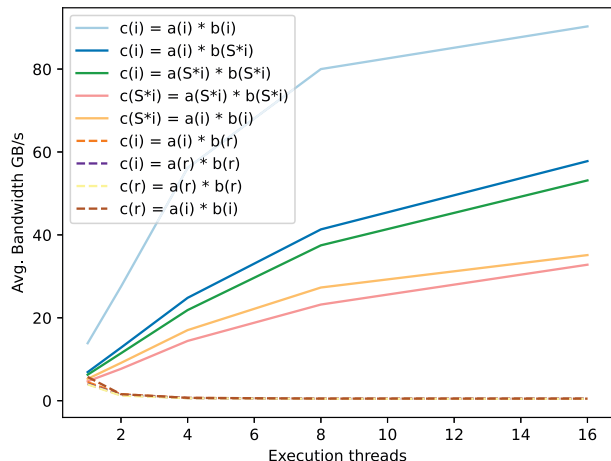

Figure 11: Multithreaded bandwidth for all strides for each thread count. Values shown are averages all strides for each thread count.

V. DISCUSSION

MARTA was conceived as a "push-button" system for profiling and performance analysis, basing its reliability on tested-and-true libraries and software. MARTA integrates the PolyBench/C library [3] for instrumenting codes using the PAPI library. It also relies on PolyBench/C directives for declaring and initializing n-dimensional arrays, as well as flushing the cache. The `pyperf` library is used for setting the processor frequency (and the turbo boost in Intel architectures). Data mining and machine learning algorithms primarily from scikit-learn are employed for performance data analysis. For the preprocessing stage, the Analyzer employs the `pandas`, `numpy`, and `KDEpy` Python libraries. Using this approach, MARTA benefits from the extensibility provided by integrating tested open source components, yet retains control over the profiling and performance analysis processes using a simple configuration file.

MARTA is not limited to any particular set of compilers/assemblers. The compilation/assembly steps are included as recipes in the configuration files. There is, however, a restricted set of automated analyses and profiling tools that are currently supported by the tool, including LLVM-MCA, PAPI counters, and SciKit-Learn. Some non-currently-supported technologies, which we plan to support in the future, include OSACA, RAPL, or ISAs different than x86. MARTA integrates external libraries and tools using high-level interfaces and avoiding ad-hoc code. New functionalities can be added by extending these interfaces to integrate each specific new tool or behavior.

MARTA currently supports and targets both single- and multi-threaded/core profiling tasks. Post-processing tasks have been optimized for data mining and basic ML classification, regression and clustering. MARTA does not currently implement deep learning algorithms as typically the small number of samples collected might not be sufficient for this type of AI techniques. However, our systematic export of data series to CSV allows seamless integration with any other data mining or deep learning framework.

VI. RELATED WORK

Many different tools and approaches for profiling applications and architectures have been developed, from open-source to commercial solutions, most of them are based on instrumentation using hardware counters. Accessing to the values contained in those counters typically requires reading Model Specific Registers (MSR) or Performance Monitor Counters (PMC). Profilers commonly use interfaces for accessing these hardware counters, such as `perf_events`. The API of these interfaces is usually complex and low level. Therefore, there are many different implementations depending on the operating system and platform. For Linux, `libpfm4` was developed as an almost zero-overhead solution for accessing these counters, and it is used in high-level interfaces such as PAPI. This library provides a flexible API for settings and programming events on these hardware counters. This alternative is intrusive, as the source code must be modified, but it provides accurate measurements (exact values are possible, sampling hardware counters is optional) with very low overhead, as it also employs `rdpmc`, if possible, for reading performance counters.

There are different alternatives for monitoring and collecting execution data from binaries such as `perf`, which is included in the Linux kernel, OProfile [12] or LIKWID [13], which also integrates a library for accessing hardware counters similar to PAPI. These are low-level tools based on dynamic instrumentation and, therefore, do not require source codes to be recompiled. TAU [14], Extrae [15], Vampir [16], Scalasca Trace Tools [17], HPCToolkit [1], and Intel oneAPI [2] (formerly Intel Parallel Studio XE) provide sophisticated profiling environments, producing detailed and complete trace execution analyses from binaries. This last one employs a top-down methodology [18], which is meant for spotting the bottlenecks in the pipeline from a hierarchical point of view. Abel and Reineke [19] propose a micro-benchmarking methodology based on measuring instruction latencies and reciprocal throughput for x86 architectures. The scope of this approach is meant for measuring instructions only, and not real regions of code. Downs [20] presents another micro-benchmarking methodology for the analysis of micro-architectural features based on a set of synthesized benchmarks and a framework for building benchmarks. This toolkit uses `libpfc` [21] (another library for reading PMU using `rdpmc` instructions), and is limited to x86 architectures. timemory [22] is a modular C++ toolkit for performance analysis and logging. It provides a simple interface for instrumenting programs avoiding low level

details of the instrumentation back-end and supports different programming languages. kerncraft [23] is a loop kernel analysis and performance modeling tool, which provides a framework for data reuse and cache analysis.

Our approach goes beyond the ones presented above by combining several of their abovementioned features, improving productivity and reproducibility with a simple interface for combining different parameters, which includes execution environment configuration. MARTA is lightweight and highly tunable, allowing to easily create large sets of data to analyze. This enables the analysis of very large exploration spaces for benchmarking via automated data mining techniques. We believe MARTA answers a need for practitioners who drive their work based on experimental data for relevant micro-benchmarks: MARTA automates the experimental setup to ease reproducibility and portability on different machines/setups, and importantly, facilitates the task of analyzing the data produced by building predictors for the data from the input feature values.

VII. Concluding Remarks

We presented MARTA, a fully implemented, open source, and highly configurable toolkit for performance analysis of programs. Productivity and reproducibility are improved with automated benchmark template generation from a simple configuration file, implementing a sound experimental setup that exploits hardware counters in the host platform when available. MARTA also integrates fine-grained directives for instrumenting and monitoring small regions of code, enabling micro-benchmarking analysis. An important aspect of MARTA is to facilitate performance analysis and debugging: the toolkit applies data mining and machine learning or AI-based techniques on the measurements, automatically extracting the features of the experimental setup which have the most impact on performance. These post-processing tasks are valuable for deriving knowledge from experiments, and are often not included in other profiling tools.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPC-Toolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[2] Intel. (n.d.) Intel oneAPI. https://software.intel.com/content/www/us/en/develop/tools/oneapi.html#gs.1ptgr0.

[3] L.-N. Pouchet and T. Yuki. (n.d) PolyBench/C 4.2.1. http://polybench.sourceforge.net.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[5] B. W. Silverman, *Density estimation for statistics and data analysis*. Chapman and Hall/CRC, 1986.

[6] Z. I. Botev, J. F. Grotowski, and D. P. Kroese, "Kernel density estimation via diffusion," *The annals of Statistics*, vol. 38, no. 5, pp. 2916–2957, 2010.

[7] T. Par, T. Lapusan, and P. Grover. (n.d.) dtreeviz : Decision Tree Visualization. https://github.com/parrt/dtreeviz.

[8] S. M. Blackburn, A. Diwan, M. Hauswirth, P. F. Sweeney, J. N. Amaral, T. Brecht, L. Bulej, C. Click, L. Eeckhout, S. Fischmeister *et al.*, "The truth, the whole truth, and nothing but the truth: A pragmatic guide to assessing empirical evaluations," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 38, no. 4, pp. 1–20, 2016.

[9] A. Abel and J. Reineke, "uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 673–686.

[10] J. Hofmann, J. Treibig, G. Hager, and G. Wellein, "Comparing the performance of different x86 simd instruction sets for a medical imaging application on modern multi- and manycore chips," in *Proceedings of the Workshop on Programming Models for SIMD/Vector Processing (WPMVP)*, 2014, p. 57–64.

[11] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, 1995.

[12] J. Levon. (n.d.) OProfile. https://oprofile.sourceforge.io/news/.

[13] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A Lightweight Performance-Oriented Tool Suite for X86 Multicore Environments," in *Proceedings of the 39th International Conference on Parallel Processing Workshops*, 2010, pp. 207–216.

[14] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[15] BSC Performance Tools. (n.d.) Extrae. https://tools.bsc.es/extrae.

[16] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis tool-set," in *Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, 2008, pp. 139–155.

[17] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[18] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 35–44.

[19] A. Abel and J. Reineke, "nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020, pp. 34–46.

[20] T. Downs. (n.d.) uarch-bench. https://github.com/travisdowns/uarch-bench.

[21] O. Bilaniuk. (n.d.) libpfc. https://github.com/obilaniu/libpfc.

[22] J. R. Madsen, M. G. Awan, H. Brunie, J. Deslippe, R. Gayatri, L. Oliker, Y. Wang, C. Yang, and S. Williams, "Timemory: modular performance analysis for HPC," in *Proceedings of the International Conference on High Performance Computing*, 2020, pp. 434–452.

[23] J. Hammer, J. Eitzinger, G. Hager, and G. Wellein, "Kerncraft: A tool for analytic performance modeling of loop kernels," in *Proceedings of the 10th International Workshop on Parallel Tools for High Performance Computing*, 2016, pp. 1–22.