Quantifying Throughput of Basic Blocks on ARM Microarchitectures by Static Code Analyzers: A Case Study on Kunpeng 920

Qingcai Jiang†, Shaojie Tan†, Zhenwei Cao†, Xiaoyu Hao†, Junshi Chen† and Hong An†
†School of Computer Science and Technology, University of Science and Technology of China, Hefei, China
Email: †{jqc, shaojiemike, caozhenwei, hxy2018}@mail.ustc.edu.cn, †{cjuns, han}@ustc.edu.cn

***Abstract*—The performance of high-performance computing (HPC) and other real-world applications is becoming unpredictable as the micro-architecture of the modern central processing unit (CPU) turns to be more and more complex. As a consequence, predicting the execution time of a code snippet is notoriously difficult. Basic block throughput predictor, also known as static code analyzer, provides a ubiquitous way to predict the execution time of a basic block, which includes assembly code segments without jump-related instructions. In this article, we build a workflow to faithfully run, collect and analyze basic blocks from real-world applications. Several static code analyzers are introduced, compared and optimized to show which one performs better on accuracy and other metrics on Kunpeng 920 processor. Under extensive experiments, we achieve state-of-the-art 86.7% accuracy in predicting the throughput of all basic blocks.**

***Index Terms*—High-performance architecture, performance modeling and simulation, static analysis, in-core architecture exploration, throughput prediction**

I. Introduction

Recent decades have witnessed the rapid trends of improvements in high-performance processing unit architecture, which is considered one of the most sophisticated and advanced man-made machines. To be specific, the prevalent micro-architecture of the modern high-performance central processing unit (CPU) used in desktop computer and server system are mostly still based on the von Neumann architecture [1], which contains three main parts: in-order issuing part as front-end, out-of-order (OOO) execution part as back end and a memory subsystem to handle data exchange. Moreover, there exists a lot of details in the micro-architectures to facilitate in launching and scheduling of the instructions, such as decoded stream buffer (DSB), move elimination, micro/macro-fusion and ones/zeroing idioms [2], to name a few. As a consequence, all these complex specifics translate to increasing difficulty in systematically describing, predicting and optimizing the performance of numerical and industrial applications running on such systems.

The performance model remains to date the method of choice to describe the characteristics of a certain CPU, which has wide-ranging implications in several fields. For example, it can provide software developers an insight into peak performance, memory traffic and execution time, therefore showing the potential improvements in compiler and code optimization. Also, it can provide hardware developers with possible bottlenecks in the physical design, thus balancing the trade-offs among energy, performance and cost.

According to their difference in interpretability, performance models can be formulated as three major types: ‘white-box’ performance modeling (a.k.a. mechanistic modeling), ‘black-box’ performance modeling (a.k.a. empirical modeling) and ‘grey-box’ performance modeling [3]. ‘White-box’ performance modeling is built from the first principle information of a certain CPU architecture to provide an intuitive trace of how an application is running. ‘Black-box’ performance modeling is based on statistical data and machine learning methods like curve fitting and neural networks [4] to automatically infer the performance of a code snippet without knowing the fundamental details on a CPU. ‘Grey-box’ performance modeling aims to bridge the gap between ‘white-box’ performance modeling’s accuracy and ‘black-box’ performance modeling’s efficiency which is built from insights in the underlying system but has a number of unknown parameters. ’White-box’ performance models at different granularities can give various views at the system level. For example, the most simple and coarse-grained ’white-box’ performance model is the Roofline model [5], which evaluates in-core floating performance and out-of-core memory transfer to estimate whether a given computing kernel will be computing-limited or memory-limited. At the most fine-grained level, simulators like gem5 [6] provide a system-level architecture as well as processor microarchitecture to run an application using an analytical and cycle-by-cycle way. There is also a middle-grained performance model called static code analyzer aiming to predict the steady-state throughput of basic blocks, i.e., the number of cycles a certain CPU takes to execute a code snippet (usually without jump-related instructions) in a steady state.

Static code analyzer, also known as basic block throughput predictor, which mainly focuses on simulating the in-core

Corresponding Authors: Hong An and Junshi Chen performance, offers great advances for compiler designers, performance engineers and chip architects. For instance, static code analyzers can show potential improvements for compiler designers on vectorization, instruction scheduling and logical-physical register mapping [7]–[9]. At the same time, static code analyzers can point out underlying bottlenecks for a given kernel for performance engineers to optimize the framework for their codes. As for chip architects, static code analyzers can give an analytical view of certain underlying problems, for instance, which execution port is the bottleneck of a given benchmark, and how to adjust the number of pipeline stages et al. Furthermore, chip architects can take advantage of these data to guide the design of next-generation chips.

In recent years, several static code analyzers have been proposed to partially address the aforementioned problems, both in 'white-box' and 'black-box' models, such as OSACA [10], [11], llvm-mca [12], IACA [13], CQA [14], Ithemal [15], DiffTune [16] and uiCA [17]. However, few of these tools are designed for Arm CPU architecture and AArch64 ISA, which are the fundamental basis of the Kunpeng 920 high-performance server processor [18]. Only llvm-mca provides native support for TaiShan v110 (TSV 110) micro-architecture of Kunpeng 920 processor [19], but our tests show that this implementation cannot provide the convincing accuracy we needed.

In this work, we provide a solution to predict basic blocks with much higher precision on the Kunpeng 920 processor by using the optimized version of llvm-mca, which opens the door for guiding the optimization of numerical computation-intensive kernels and providing analytical insights for compiler developers.

We summarize the contributions as follows:

- We collect the information for describing an instruction in the framework of basic block throughput predictor, including throughput, latency, port pressure, operands and micro-ops (uops), in a given Kunpeng 920 processor.
- We build an extensive benchmark from real-world applications and conventional evaluation tools, together with a runtime environment system to run a basic block and get the accurate corresponding throughput.
- After comparing several static code analyzers in the terms of accuracy and additional metrics, we determine llvm-mca as a continuously developing framework to predict the throughput of basic blocks. In addition, we figure out several issues that cause performance degradation in llvm-mca.
- Vast experiments are performed to evaluate the effectiveness of our methods, which show that our results exceed current tools in AARch64 architecture by a wide margin.
- We have open-sourced our work at Github[1][2] in the hope that our approach can provide insight for relevant work aiming at statically benchmarking the steady-state throughput of basic blocks on particular high-performance microarchitectures.

[1]https://github.com/qcjiang/llvm-project/tree/tsv110
[2]https://github.com/qcjiang/OSACA

II. Related Work

A. Different Types of Performance Models

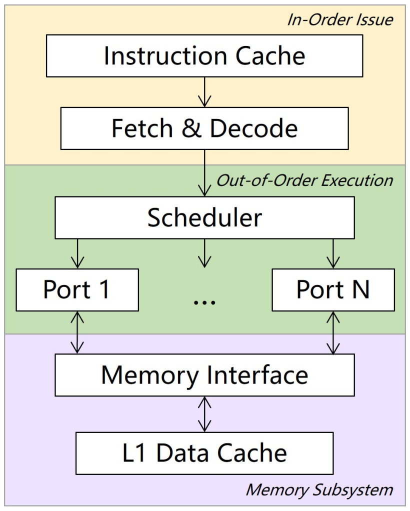

Fig. 1. A generic model of modern CPU design containing an in-order issue part, an out-of-order execution part and a memory subsystem.

Existing performance models can be roughly categorized into two types: simulation-based models and machine-learning-based models, subsequently, simulation-based models can be distinguished according to their particular granularity in different system levels.

At the higher level, which includes multi-core and memory systems, the most simple yet effective performance model tool is the Roofline model [5]. It defines operational intensity as the ratio of executed floating-point operations and the total traffic of bytes, therefore the Roofline model gives an upper bound for the feasible performance, which can be either compute bound or memory bound. To present a more specific performance, the Execution-Cache-Memory (ECM) [20] introduces different memory hierarchies to describe data transfer time according to the bandwidth of the corresponding cache level. Several high-performance applications like stencil [21], explicit ODE methods [22], Jacobi smoother kernel [23], SpMV and Lattice QCD [24] can be modeled, predicted and optimized precisely within the framework of the ECM model.

At the lower level, which focuses on single-core performance and excludes memory access, static code analyzers are able to provide compelling simulation analysis by giving an assumption of the throughput of an assembly instruction kernel. They are generally composed of an in-order front-end, an out-of-order backend and a memory subsystem like Figure 1. The Open Source Architecture Code Analyzer (OSACA) [10], [11] is an open-source basic block analyzer for some of the Intel, AMD and ARM micro-architectures. It provides fast throughput analysis and detection for critical path and loop-carried dependencies, but this coarse-grained model could not be modified to add specific micro-architectures like instruction decoder, reorder buffer and so on. LLVM Machine Code Analyzer (llvm-mca) [12] is a performance analysis tool that uses information available in LLVM [25] to statically measure the performance and some advanced features like bottleneck analysis and cycle-by-cycle view of basic blocks in a specific CPU. It supports many of the micro-architectures in X86 and ARM and is the only static code analyzer that provides a native implementation for Kunpeng 920 processor. Intel Architecture Code Analyzer (IACA) [13] is released in 2012 to provide static throughput and critical path analysis only for Intel micro-architectures from Haswell (HSW) to Skylake-X (SKX). It supports micro-ops (μops) level and cycle-by-cycle instruction simulation, hence enabling it to provide more accurate results than other 'white-box' analyzers such as OSACA and llvm-mca. Unfortunately, IACA is closed-source and announced its end of life in April 2019.

As for machine-learning-based models, Ithemal [15] is a 'black-box' basic block throughput predictor that developed from an LSTM–based deep neural network. It supports only Intel's Ivy Bridge, Haswell and Skylake micro-architecture and it outperforms IACA and llvm-mca by a large margin. To further investigate the interpretability of such a machine-learning-based model, DiffTune [16] is presented to learn the microarchitecture-specific parameters for the x86 simulation model from coarse-grained end-to-end measurements.

On the other side, full system simulators like gem5 [6], ZSim [26] et al. provide a detailed simulation of entire programs from the bottom layer of the processor, including front-end pipeline structure, memory hierarchies and multi-core architectures.

B. *Microbenchmarking*

To alleviate the burden of manually reverse engineering the corresponding parameters for static code analyzers, it is critical to develop an efficient way to do microbenchmarking.

Johannes Hofmann, et al., developed iBench [27], a tool designed to measure the instruction's latency and throughput. In order to obtain the execution cycle of instructions, they embed assembly instructions into a C program. The program is executed on a fixed CPU frequency so that the number of execution cycles can be calculated by multiplying execution time by CPU frequency. To measure throughput, which means execution cycles in a steady state for a dependency-free sequence of instructions, they create instruction sequences in which each instruction does not wait for any register or memory of previous instructions. And to measure latency, they create instruction sequences by repeating instructions with the same operands many times, so that each instruction can't start execution until the previous instructions finish execution.

Asmbench [28] is a framework proposed to deduce instruction information(latency and throughput) through run-time instruction benchmarking. The framework, depending on LLVM's C-API, does not use any architecture-specified performance counters. For automation and architecture-independency, they use LLVM's intermediate representation and backend instruction database "TableGen". To generate a benchmark, they create a long chain of instructions to measure latency, in which each instruction processes the output of preceding instruction input, and sequences of instructions with independent input and output to measure throughput. They describe resource conflicts by comparing the throughput values when the instruction pairs are run separately and when the instructions are run in combination.

III. Basic Block Throughput Prediction

In this section, we will introduce the basic concepts related to basic block throughput prediction, including fundamental definitions and assumptions and the port model we build for static code analyzers.

A. *Basic Definitions*

1) How to describe an instruction: It is important to provide formal and correct definitions of the information of each instruction for static code analyzers to generate accurate simulation results.

Listing 1. An example of fsub instruction information for Kunpeng 920 processor in OSACA code. The prefix entry in the operands field refers to the type of a register, i.e. scalar register, vector register, and the shape entry denotes the way a vector register can be accessed.

```
- name: fsub
  operands:
  - class: register
    prefix: v
    shape: s
  - class: register
    prefix: v
    shape: s
  - class: register
    prefix: v
    shape: s
  latency: 5.0
  port_pressure: [[1, '45']]
  throughput: 0.5
  uops: 1
```

- Operands. Operands of an instruction show what information is to be operated on. The most important general categories of operands are memory address, register and immediate.
- Latency. Latency is the number of cycles after which the data is available for another operation. It describes the execution time of instruction **with** dependency.
- Throughput. Throughput is the number of cycles after an issue that another instruction can begin execution. It describes the execution time of instruction **without** dependency.
- Numbers of uops. In modern CPU design, regardless of AArch64 or X86 micro-architecture, instructions are divided into one or more uops for execution. The number of uops denotes the number of these smallest operations of execution by the processor in an instruction.

- Port pressure. As depicted in Figure 1, in the out-of-order backend, uops are distributed in different ports. Port pressure describes which port will an instruction executes the corresponding uops.

2) How to define the throughput of a basic block: The throughput of a basic block is generally defined as ”The average number of cycles for executing a basic block repeatedly in a steady state”. However, different definitions of basic block lead to the different notation of throughput, for example, if basic blocks end up with a jump-related instruction, they are treated as executing the block in a way that the branch is always taken; for basic blocks that do not end with a jump-related instruction, they are treated as unrolling the basic block for plenty of times to make them executed in a steady state. In this work, we construct all basic blocks without a jump-related instruction for convenience, i.e. Listing. 2, and we remark that introducing a jump-related instruction at the end of the basic blocks will not affect the accuracy of our results.

Listing 2. An example of a basic block on Kunpeng 920

```
mov     x23, x26
lsl     x1, x1, #2
sub     x1, x1, x3
add     x1, x2, x1, lsl #3
str     x1, [sp, #0xa8]
```

B. Conventional Assumptions

The major capability of static code analyzers is to model the throughput and dependencies of an assembly code snippet statically, which means they cannot include the runtime information of the execution process, such as register value and the address a jump-related instruction is targeted for. To this end, several notoriously common assumptions are introduced to facilitate our simulations.

- All memory access hit L1 cache. Since the field of research in static code analyzers is limited to in-core analysis, we assume that all memory access will be executed optimally without cache or TLB misses.
- Steady-state execution. The execution of a basic block normally consists of a warm-up and wind-down phase. Since we assume a large number of unrolling iterations, they will not be taken into account for static code analyzers for simplicity.

IV. Methodology

A. Architecture of Kunpeng 920

Kunpeng 920 is a 64-bit ARM server microprocessor introduced by HiSilicon in early 2019. Fabricated by TSMC on a 7nm HPC process based on the TaiShan v110 microarchitecture, this chip incorporates 64 cores operating at 2.6 GHz with a TDP of 180 W. It is empowered with ARM v8.2 instruction set architecture (ISA) and NEON Advanced SIMD extension. We take it as an example to analyze the performance of static code analyzers on AArch64 architectures.

We first analyze the structure of back end pipeline of Kunpeng 920 based on existing knowledge and our results on the characteristics such as throughput and port usage of each instruction. As sketched in Figure 2, we find that each core in Kunpeng 920 is equipped with one port for integer type instructions, two ports for integer and jump type instructions, one port for multi-cycle instructions such as divide and sqrt, two ports for float and SIMD type instructions and two ports for memory load/store instructions.

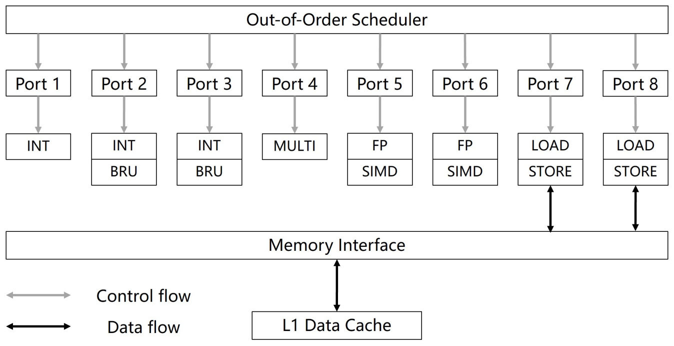

Fig. 2. Overview of the out-of-order execution back end and memory subsystem components of a Kunpeng 920 core.

B. Modeling Characteristics of Each Instruction

For obtaining an accurate static code analyzer, it is the cornerstone to get the required information such as latency, port pressure, throughput and number of uops in every single instruction by rule and line, we perform several measurement methods on Kunpeng 920 processor.

As introduced in Section II-B, we generate assembly benchmarks and data files for iBench and asmbench, respectively. These two tools enable us to run, measure and compare the **throughput** and **latency** result of each instruction with each other.

C. Implementating TSV 110 on OSACA

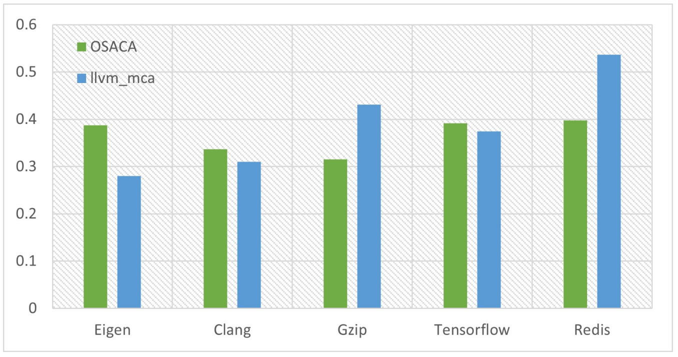

Fig. 3. Mean error rate of OSACA and baseline version of LLVM-MCA on 5 different workloads.

After obtaining the required parameters of each instruction, we only need to serialize the parameters to a configuration file as an input of the OSACA program. OSACA will give an assumption of the basic block's throughput by calculating the maximum throughput (TP) and loop-carried dependencies (LCD). To be specific, instead of building a complete front-back end model and running a cycle-by-cycle simulation, OSACA simply averages the throughput of each instruction to the ports it can be launched. For LCD calculation, OSACA will unroll the basic block twice and run a Dijkstra-like algorithm to find the longest weighted dependency chain between loops of the unrolled basic block as the LCD result. These designs bring about two defects:

- Since the OSACA framework does not provide a full pipeline model, we cannot add design details under the hood of modern CPUs such as move elimination, zero/one idioms, macro-fusion et al. Which is a key step for improving the accuracy in predicting the throughput of basic blocks.
- The term TP and LCD have their advantage type of basic block to provide better throughput assumptions. For basic blocks that do not have dependencies, since the processor will launch the instructions to the ports they belonged at an equal rate, the term TP can faithfully represent the throughput of each basic block, and the term LCD will be zero. For basic blocks that have dependencies between each pair of instructions, since the next instruction will wait for the previous instruction to complete, the term LCD can faithfully denote the chaining dependency of this basic block, thus providing an accurate assumption of the throughput of a given basic block, and the term TP will be relatively small. But for the basic blocks that have part of dependencies, both TP and LCD cannot faithfully represent the throughput of these basic blocks.

TABLE I
THE ASSEMBLER SYMBOL OF SHIFT SEGMENT IN THE ENCODING OF 'ADD (SHIFTED REGISTER)' INSTRUCTION

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

Due to the fact that in the framework of OSACA, we cannot add micro-architecture specifics, the accuracy of this tool is not able to be improved systematically. As a result, we choose llvm-mca as the continuously developing framework.

D. Implementating TSV 110 on LLVM-MCA

Although llvm-mca provides native support for Kunpeng 920, as we can see from Figure 3, its accuracy is not quite desirable. To further investigate the potential of llvm-mca, we adopt several ways to improve the accuracy of this tool.

1) Correcting the parameters of each instruction: First of all, we correct the wrong latency number of instructions, such as fdiv, fmla and fsqrt , in the original llvm-mca version. Later, we find a wide range of errors in the latency in various types of instructions. Simple instructions such as add, and, eor, orr and sub, can be divided into different types: extended register, immediate and shifted register. The instruction without shift is also categorized to the shifted register type, such as 'add x0, x1, x2', whose encoding format is depicted in Figure 4. However, the latency number of the instruction with shift and without shift make a difference, for instance, the latency of 'add x0, x1, x2' is 1 cycle thus the latency of 'add x0, x1, x2, lsl #3' is 2 cycles. The tool llvm-mca confuses these two types of instructions thus resulting in great error. We notice that the assembler symbol of the shift segment in the encoding of 'ADD (shifted register)' instruction has one operand reserved, as shown in Table I. To this end, we assign this reserved item to the instruction without shift, and this simple solution works perfectly with llvm-mca.

2) Move elimination: By doing extensive experiments, we find that Kunpeng 920 will perform move eliminations on the move instructions with register-register operands, such as 'mov x1, x2', by the renamer units. However, not all register-register move instructions will be eliminated by the processor. It is a common case that each move elimination will occupy one elimination slot in the processor. We discover that each core in Kunpeng 920 is equipped with one elimination slot, which means it can only perform one move elimination on each cycle. We implement move elimination feature on llvm-mca to provide better accuracy results.

V. RESULT AND ANALYSIS

In this section, we evaluate the effectiveness of our methods by validating the accuracy of the optimized llvm-mca tool on different metrics. We use a server machine with one Kunpeng 920 socket containing 96 cores for configuring and running experiments as well as collecting and analyzing the data. In addition, this machine has 188 GB of RAM and the OS id Ubuntu 20.04.5 with Linux kernel 5.4.0-128.

A. Evaluation metrics

1) MAPE: The mean absolute percentage error (MAPE), also known as mean absolute percentage deviation (MAPD), is a measure of prediction accuracy of a forecasting method in statistics. It usually expresses the accuracy as a ratio defined by the formula:

$$\mathrm{MAPE}(B) = \frac{1}{|B|} \cdot \sum_{(m,p)\in B} \frac{|m-p|}{m}. \tag{1}$$

In our experiments, B denotes the dataset of basic blocks from an application, m means the measured result of a basic block from BHive and p means the predicted result of a basic block from llvm-mca.

2) Kendall's tau: Kendall's tau is a statistic used to measure the ordinal association between two measured quantities, which has the following formula:

$$\tau(B) = \frac{(\mathrm{Nc}) - (\mathrm{Nd})}{\begin{pmatrix} |B| \\ 2 \end{pmatrix}}, \tag{2}$$

where Nc denotes the number of concordant pairs and Nd denotes the number of discordant pairs.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	0	1	0	1	1	shift		0	Rm					imm6						Rn					Rd				
	op	S																													

Fig. 4. The encoding of 'ADD (shifted register)' instruction in Armv8-A Instruction Set Architecture.

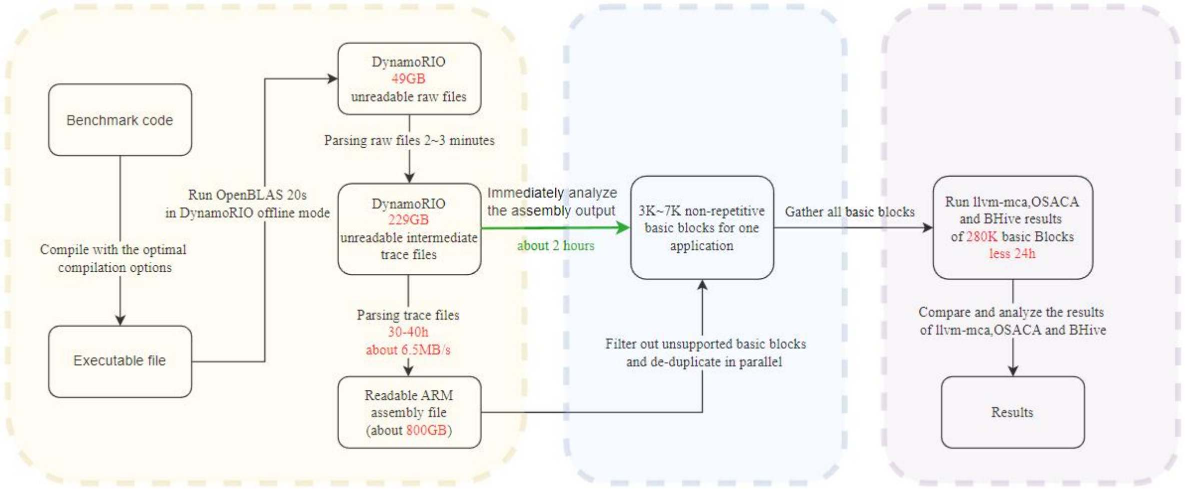
Benchmark code
Compile with the optimal compilation options
Executable file
Run OpenBLAS 20s in DynamoRIO offline mode
DynamoRIO 49GB unreadable raw files
Parsing raw files 2~3 minutes
DynamoRIO 229GB unreadable intermediate trace files
Parsing trace files 30-40h about 6.5MB/s
Readable ARM assembly file (about 800GB)
Immediately analyze the assembly output
about 2 hours
3K~7K non-repetitive basic blocks for one application
Filter out unsupported basic blocks and de-duplicate in parallel
Gather all basic blocks
Run llvm-mca,OSACA and BHive results of 280K basic Blocks less 24h
Compare and analyze the results of llvm-mca,OSACA and BHive
Results

Fig. 5. Workflow of executing, extracting, collecting and analyzing basic blocks, using OpenBLAS as an example.

B. Benchmarks: Basic Blocks from Real World Applications

TABLE II
SOURCE APPLICATIONS OF BENCHMARK BASIC BLOCKS

Application	Domain	# Basic Blocks
OpenBLAS	High Performance	6336
FFTW	High Performance	15616
LAPACK(dgetrf)	High Performance	6142
Eigen(MM+MV)	Scientific Computing	6127
Clang	Compiler	35598
Redis	Database	5716
GZip	Compression	3749
TensorFlow	Machine Learning	161450
Embree	Ray Tracing	7272
FFmpeg	Multimedia	33485
Total		**281491**

Table II shows the applications which we chose to collect basic blocks for benchmarking. We select these applications from a diverse range of domains to represent real-world workloads [29]. And we‘re more focused on high-performance applications, e.g., Fastest Fourier Transform in the West (FFTW) and Linear Algebra PACKage (LAPACK).

We used the highest Kenpeng 920 optimization settings (-O3, NEON SIMD, et al.) to compile all applications. It is particularly important to these High-performance applications like FFTW.

We implement the benchmarks building procedure in **three steps**:

First, we dynamically capture every assembly instruction executed by the process with the help of a runtime code manipulation system named DynamoRIO [30].

TABLE III
COMPARISON OF DIFFERENT APPLICATIONS LLVM-MCA RESULTS ON BHIVE

	llvm-mca-optimized		llvm-mca-baseline	
Application	**MAPE**	**Kendall**	**MAPE**	**Kendall**
OpenBLAS	7.52%	0.7189	7.82%	0.2180
OpenBLAS_level1_ddot	8.15%	0.7453	12.29%	0.3153
OpenBLAS_level2_dgemv	8.99%	0.7531	20.45%	0.3158
OpenBLAS_level3_zgemm	12.20%	0.7562	31.24%	0.3197
FFTW	12.93%	0.7396	30.86%	0.1851
LAPACK(dgetrf)	11.98%	0.7367	30.73%	0.3010
Eigen(MM)	15.31%	0.7071	143.05%	0.2406
Eigen(MV)	14.75%	0.7000	27.94%	0.2305
Clang	12.66%	0.7224	30.91%	0.2166
Redis	26.05%	0.5941	53.72%	0.1894
GZip	12.11%	0.5626	43.09%	0.2247
TensorFlow	9.33%	0.7022	37.42%	0.0712
Embree	12.71%	0.6442	45.16%	0.1932
FFmpeg	13.80%	0.6770	29.82%	0.2373
AVERAGE	**13.32%**	**0.6926**	**35.84%**	**0.2404**

We use the official benchmarking input to simulate the realistic execution, except for FFmpeg and Gzip. We tested Eigen on two sparse linear algebra workloads: sparse matrix-matrix multiplication (Eigen_MM) and sparse matrix-vector multiplication (Eigen_MV), as shown in Figure 6. We test LAPACK and OpenBLAS at the same level-3 matrix-matrix multiplication.

As shown in the yellow part of Figure 5, capturing raw assembly code is extremely time-consuming and storage-consuming. For example, collecting 20s OpenBLAS program data will need about 40 hours to analyze and 1 TB of storage to store the result.

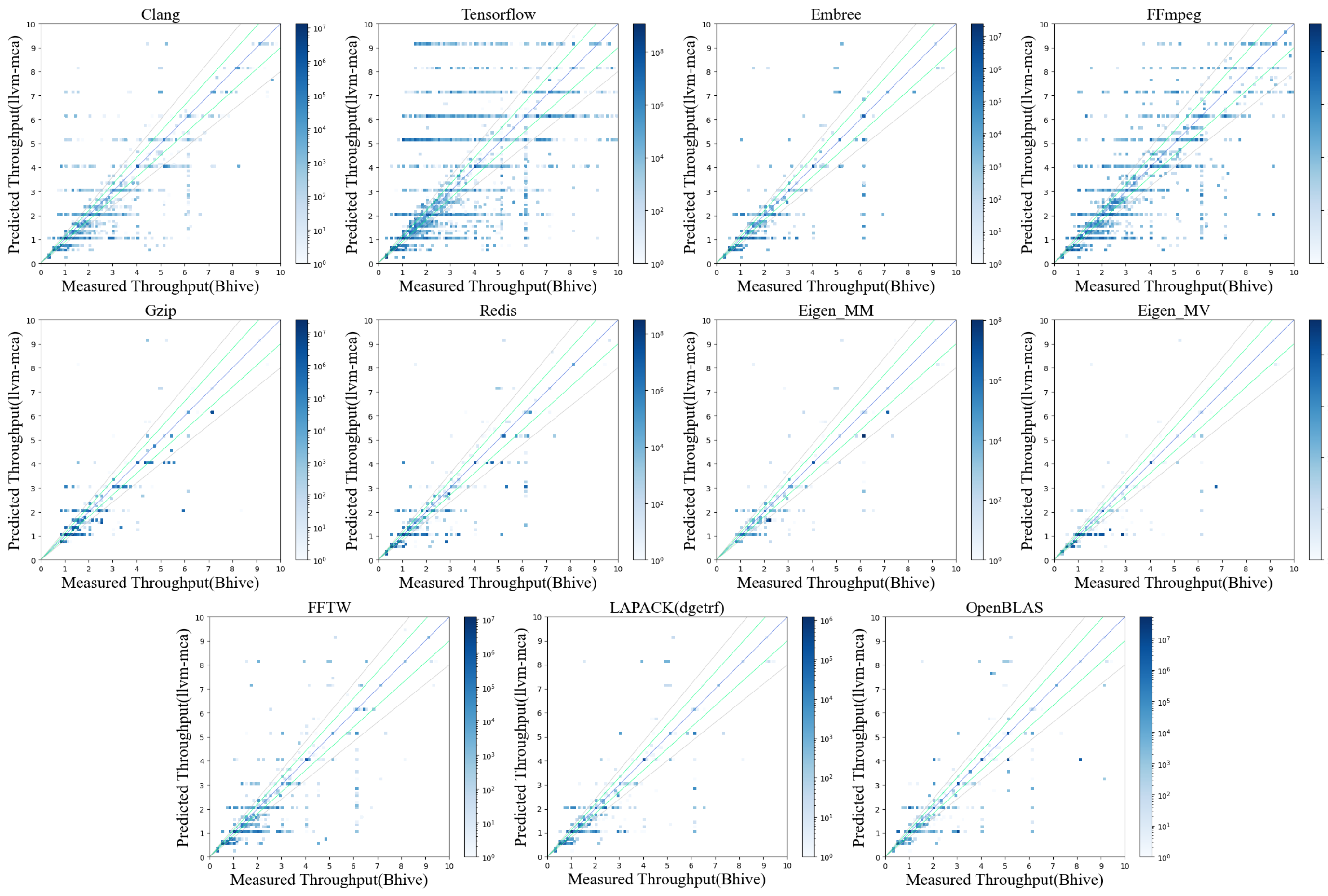

Fig. 6. Heatmaps of optimized version of llvm-mca for basic blocks with a measured throughput of less than 10 cycles/iteration on Kunpeng 920, the green dash line in each figure denotes 10% error rate and the purple dash line denotes 20% error rate.

In the **second** step, we use an in-time method of processing assembly code instead of saving all assembly code to erase huge storage consumption. We slice the assembly code into basic blocks with jump and system instructions and generate datasets by filtering basic blocks not supported by BHive and de-duplication.

We extracted a large number of diverse basic blocks in Clang/LLVM (compiler) and TensorFlow because they are written with sophisticated algorithms and data structures. In contrary to general purpose applications, highly optimized applications: FFTW, Eigen and OpenBLAS run handwritten assembly loops many times, hence the kernel basic blocks of loops will be repeatedly collected.

Third, we compute llvm-mca, OSACA results on the basic block dataset in one day in a parallel scheme. And compare it with the standard result of BHive. We record the detailed data in a readable Excel file, which will help researchers to reproduce the experimental results and further research. We have opensourced our basic block dataset, and it is available to download at Github[3].

[3]https://github.com/Kirrito-k423/BHive-Prediction-Compare

We evaluate llvm-mca in version 13.0.0 as a baseline for our experiments and compare the results with our optimized version, shown in Table III. Our optimized version provides more accurate predictions in all cases. The MAPE is much lower compared to the baseline version of llvm-mca, and Kendall's tau coefficient is also always higher than the baseline version of llvm-mca. In most cases, our optimized version outperforms the baseline version by a large margin.

We also draw heatmaps to compare our optimized llvm-mca and BHive result on Kunpeng 920 shown in Figure 6. We remark that our optimized version significantly fixed the large predicted throughput in Clang, Redis, and HPC applications and balanced the low predicted throughput in Tensorflow.

As presented in Table III, simulation results in 14 different applications show that our optimizations on llvm-mca translate to a 22.52% improvement on average MAPE. And our tool achieves 86.7% accuracy in predicting the throughput of all basic blocks, which offers a great advance for system developers to statically predict the throughput of basic blocks with higher credibility.

VI. Conclusion

In this work, we implement TSV 110 micro-architecture to two state-of-the-art static code analyzers, OSACA and llvm-mca. After comparing with OSACA tool in terms of accuracy and extendibility, we choose llvm-mca as a prototyping framework for further development. Also, we build a highly efficient framework to run, collect and post-process the basic blocks for validating our methods, along with an accurate runtime environment to obtain measured throughput as a benchmark. By exploiting the micro-architecture information of Kunpeng 920 processor and implementing them within llvm-mca, we achieve an 86.7% accuracy in predicting the throughput of all basic blocks, which is better than any other 'white-box' static code analyzers in AArch64 architecture.

Acknowledgment

We are thankful to the reviewers for evaluating this study and providing valuable feedback. This work is partially supported by the National Natural Science Foundation of China (Grant No. 62102389).

References

[1] J. Von Neumann, "First draft of a report on the edvac," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[2] A. Fog. The microarchitecture of intel, amd, and via cpus. [Online]. Available: https://www.agner.org/optimize/microarchitecture.pdf

[3] L. Eeckhout, "Computer architecture performance evaluation methods," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–145, 2010.

[4] J. E. Dayhoff, *Neural network architectures: an introduction*. Van Nostrand Reinhold Co., 1990.

[5] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[7] R. C. Lozano, M. Carlsson, F. Drejhammar, and C. Schulte, "Constraint-based register allocation and instruction scheduling," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2012, pp. 750–766.

[8] A. McGovern and J. Moss, "Scheduling straight-line code using reinforcement learning and rollouts," *Advances in neural information processing Systems*, vol. 11, 1998.

[9] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," *ACM sigplan notices*, vol. 38, no. 5, pp. 77–90, 2003.

[10] J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein, "Automated instruction stream throughput prediction for intel and amd microarchitectures," in *2018 IEEE/ACM performance modeling, benchmarking and simulation of high performance computer systems (PMBS)*. IEEE, 2018, pp. 121–131.

[11] J. Laukemann, J. Hammer, G. Hager, and G. Wellein, "Automatic throughput and critical path analysis of x86 and arm assembly kernels," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2019, pp. 1–6.

[12] llvm-mca — llvm machine code analyzer. [Online]. Available: https://llvm.org/docs/CommandGuide/llvm-mca.html

[13] Intel architecture code analyzer user's guide. [Online]. Available: https://software.intel.com/content/dam/develop/external/us/en/documents /intel-architecture-code-analyzer-3-0-users-guide-157552.pdf

[14] A. S. Charif-Rubial, E. Oseret, J. Noudohouenou, W. Jalby, and G. Lartigue, "Cqa: A code quality analyzer tool at binary level," in *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE, 2014, pp. 1–10.

[15] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *International Conference on machine learning*. PMLR, 2019, pp. 4505–4515.

[16] A. Renda, Y. Chen, C. Mendis, and M. Carbin, "Difftune: Optimizing cpu simulator parameters with learned differentiable surrogates," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 442–455.

[17] A. Abel and J. Reineke, "uica: Accurate throughput prediction of basic blocks on recent intel microarchitectures," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–14.

[18] Kunpeng 920 chipset. [Online]. Available: https://www.hisilicon.com/en/products/Kunpeng/Huawei-Kunpeng/Huawei-Kunpeng-920

[19] Taishan v110 - microarchitectures - hisilicon. [Online]. Available: https://en.wikichip.org/wiki/hisilicon/microarchitectures/taishan_v110

[20] J. Treibig and G. Hager, "Introducing a performance model for bandwidth-limited loop kernels," in *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2009, pp. 615–624.

[21] H. Stengel, J. Treibig, G. Hager, and G. Wellein, "Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 207–216.

[22] J. Seiferth, C. Alappat, M. Korch, and T. Rauber, "Applicability of the ecm performance model to explicit ode methods on current multi-core processors," in *International Conference on High Performance Computing*. Springer, 2018, pp. 163–183.

[23] S. Kronawitter and C. Lengauer, "Optimization of two jacobi smoother kernels by domain-specific program transformation," in *Proceedings of the 1st International Workshop on High-Performance Stencil Computations (HiStencils)*, 2014, pp. 75–80.

[24] C. Alappat, N. Meyer, J. Laukemann, T. Gruber, G. Hager, G. Wellein, and T. Wettig, "Execution-cache-memory modeling and performance tuning of sparse matrix-vector multiplication and lattice quantum chromodynamics on a64fx," *Concurrency and Computation: Practice and Experience*, p. e6512, 2021.

[25] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[26] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Computer architecture news*, vol. 41, no. 3, pp. 475–486, 2013.

[27] J. Hofmann, "ibench-instruction benchmarks, 2017," *URL https://github.com/RRZE-HPC/ibench*.

[28] J. Hammer, G. Hager, and G. Wellein, "Ooo instruction benchmarking framework on the back of dragons," *SC18 SRC Poster (in review)*, 2018.

[29] Y. Chen, A. Brahmakshatriya, C. Mendis, A. Renda, E. Atkinson, O. Sỳkora, S. Amarasinghe, and M. Carbin, "Bhive: A benchmark suite and measurement framework for validating x86-64 basic block performance models," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 167–177.

[30] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '03. USA: IEEE Computer Society, 2003, p. 265–275.