CCF Transactions on High Performance Computing
https://doi.org/10.1007/s42514-023-00160-0

REGULAR PAPER

Uncovering the performance bottleneck of modern HPC processor with static code analyzer: a case study on Kunpeng 920

Shaojie Tan[1] · Qingcai Jiang[1] · Zhenwei Cao[1] · Xiaoyu Hao[1] · Junshi Chen[1] · Hong An[1]

Received: 30 March 2023 / Accepted: 28 June 2023

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 is a crucial feature of the static code analyzer. It offers a ubiquitous method for predicting the execution time of a basic block. 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 a Kunpeng 920 processor. Through extensive experiments, we achieve state-of-the-art 86.7% accuracy in predicting the throughput of all basic blocks. Moreover, we showcase the potential applications of our optimized static code analyzer in two certain aspects: 1. Guiding the application's optimization through bottleneck analysis and 2. Exploiting the potential bottleneck of a CPU on a certain workload through fast hardware pre-evaluation.

Keywords High-performance computing · Micro-architecture exploration · Performance modeling and simulation · Static analysis · Throughput prediction

1 Introduction

Recent decades have witnessed the rapid trends of improvements in the architecture of high-performance processing units, which is considered one of the most sophisticated and advanced machines. In contemporary desktop and server systems, the prevailing microarchitecture of high-performance central processing unit (CPU) is typically comprised of three widely accepted main components: the frontend, responsible for in-order issuing; the backend, which executes instructions out-of-order (OOO); and a memory subsystem that facilitates efficient 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 (Fog 2022). 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 is still the most commonly used approach for describing the features of a specific CPU, and its significance extends to various 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

Shaojie Tan, Qingcai Jiang have contributed equally to this work.

✉ Junshi Chen
cjuns@ustc.edu.cn

✉ Hong An
han@ustc.edu.cn

Shaojie Tan
shaojiemike@mail.ustc.edu.cn

Qingcai Jiang
jqc@mail.ustc.edu.cn

Zhenwei Cao
caozhenwei@mail.ustc.edu.cn

Xiaoyu Hao
hxy2018@mail.ustc.edu.cn

[1] School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, Anhui, China

Published online: 15 September 2023

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 (Eeckhout 2010). '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 (Dayhoff 1990) 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 (Williams et al 2009), 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 (Binkert et al 2011) 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 analyzers play a significant role in predicting basic-block throughput, which is a crucial aspect of simulating in-core performance. The tool provides significant benefits for compiler designers, performance engineers, and chip architects, enabling them to make significant advancements in their respective domains. For instance, static code analyzers can show potential improvements for compiler designers on vectorization, instruction scheduling and logical-physical register mapping (Lozano et al 2012; McGovern and Moss 1998; Stephenson et al 2003). 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. Furthermore, chip architects can take advantage of these data to guide the design of next-generation chips.

Obviously, accuracy is the premier concern of a static code analyzer, and the deviation often misleads the optimization from a good performance (Pohl et al 2019, 2020; Mendis and Amarasinghe 2018; Huh and Tuck 2017) (details in 7.1). However, accurately predicting the throughput of a certain basic block remains a challenge for many problems. For example, how to describe the characteristics of each instruction in instruction set architecture (ISA)? How to add more details to an existing model? How to build a comprehensive benchmark and runtime system to faithfully evaluate the accuracy of a static code analyzer?

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 (Laukemann et al 2018, 2019), llvm-mca (llv 2022), IACA (iac 2017), CQA (Charif-Rubial et al 2014), Ithemal (Mendis et al 2019), DiffTune (Renda et al 2020) and uiCA (Abel and Reineke 2022). 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 (HiSilicon 2021). Only llvm-mca provides native support for TaiShan v110 (TSV 110) micro-architecture of Kunpeng 920 processor (Huawei 2022), but our tests show that this implementation cannot provide the convincing accuracy we needed. Therefore, it remains three following critical problems:

1. How to build a ubiquitous benchmark to faithfully evaluate the performance of a particular static code analyzer on Arm architecture?
2. Which static code analyzer provides the highest precision in predicting the throughput of a given basic block on TSV 110?
3. What are the fundamental barriers to the current state-of-the-art static code analyzer from getting an even better accuracy?
4. What is the potential application for static code analyzers?

In this work, we provide a solution to predict basic blocks with much higher precision on the Kunpeng 920 processor by establishing a workflow to benchmark the accuracy of static code analyzers and calibrating one of these static code analyzers with Kunpeng 920's micro-architecture specifics we discovered, which opens the door for guiding the optimization of numerical computation-intensive kernels and providing analytical insights for compiler developers.

Our preliminary work has been presented in Jiang et al (2022), which only includes a simple design without analysis and details of our framework. Compared to Jiang et al (2022), we now provide observations, implementation

details, and underlying application insight through extra experiments.

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 and resolve 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 apply the static code analyzer to a matrix–vector multiplication demonstration to showcase its ability to identify bottlenecks, reveal optimization opportunities. Additionally, we observe that this tool possesses an advantage over traditional performance model when guiding CPU design during the pre-evaluation stage.
- 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.

The subsequent sections of this paper are structured as follows. Section 2 provides an overview of relevant literature, including existing performance models and microbenchmarking tools. Section 3 introduces the fundamental concepts of basic block throughput prediction. In Sect. 4, we outline the methodologies employed in modeling Kunpeng 920 through the use of a static code analyzer. Section 5 offers a detailed account of how benchmark programs are constructed. The experimental results and corresponding analyses are presented in Sect. 6. We propose a demo for using a chosen static code analyzer to guide the optimization of a computation-intensive kernel in Sect. 7. Section 8 briefly discusses the disadvantages of current static code analyzers. Section 9 concludes our work with a proposal for future work.

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

[2] https://github.com/qcjiang/OSACA.

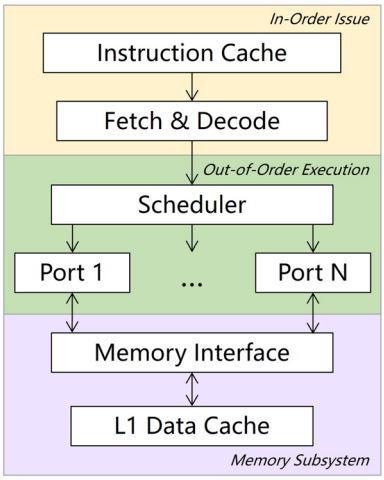

Fig. 1 A generic model of modern CPU design containing an in-order issue part, an out-of-order execution part and a memory subsystem

2 Related work

2.1 Different types of performance models

Existing performance models can be roughly categorized into two types: simulation-based models ('white-box' models) and machine-learning-based models ('black-box' models). Furthermore, simulation-based models can be further distinguished according to their particular granularity in different system levels.

At the higher system level, which includes multi-core and memory systems, the most simple yet effective performance model tool is the Roofline model (Williams et al 2009). 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) Treibig and Hager (2009) introduces different memory hierarchies to describe data transfer time according to the bandwidth of the corresponding cache level. Several high-performance applications like stencil (Stengel et al 2015), explicit ODE methods (Seiferth et al 2018), Jacobi smoother kernel (Kronawitter and Lengauer 2014), SpMV and Lattice QCD (Alappat et al 2021) can be modeled, predicted and optimized precisely within the framework of the ECM model.

At the lower system 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

Listing 1 An example of a dependency-free instruction sequence on Kunpeng 920 formeasuring throughput in iBench

```
add x3, x1, x2
add x4, x1, x2
add x5, x1, x2
add x6, x1, x2
add x7, x1, x2
```

Listing 1 An example of a dependency-free instruction sequence on Kunpeng 920 for measuring throughput in iBench

Listing 2 An example of an instruction sequence on Kunpeng 920 for measuring latencyin iBench

```
add x3, x1, x2
add x4, x1, x3
add x5, x1, x4
add x6, x1, x5
add x7, x1, x6
```

Listing 2 An example of an instruction sequence on Kunpeng 920 for measuring latency in iBench

subsystem like Fig. 1. The Open Source Architecture Code Analyzer (OSACA) (Laukemann et al 2018, 2019) 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) (llv 2022) is a performance analysis tool that uses information available in LLVM (Lattner and Adve 2004a) to statically measure the performance and some advanced features like bottleneck analysis and cycle-by-cycle pipeline 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) (iac 2017) 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. Code Quality Analyzer (CQA) tool Charif-Rubial et al (2014) is a loop-centric code quality analyzer that focuses on the performance of innermost loops. It provides an estimation of the number of cycles required for each iteration of a specific innermost loop, as well as the identification of architectural bottlenecks

As for machine-learning-based models, Ithemal (Mendis et al 2019) 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 (Renda et al 2020) 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 (Binkert et al 2011), ZSim (Sanchez and Kozyrakis 2013) etc. 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.

2.2 Microbenchmarking

To alleviate the burden of manual reverse engineering of parameters for static code analyzers, it is critical to develop an efficient method for microbenchmarking.

Johannes Hofmann, et al., developed iBench (Hofmann 2017), 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 as shown in Listing 1. 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 as sketched in Listing 2.

Asmbench Hammer et al (2018) is a framework proposed to deduce instruction information (latency and throughput) through runtime 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'. In order to generate a benchmark, the developers of Asmbench create a lengthy chain of instructions to measure

Listing 3 An example of fsub instruction information for Kunpeng 920 processor inOSACA 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 canbe 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
```

Listing 3 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.

latency. Each instruction processes the output of the preceding instruction's input. For measuring throughput, they utilize sequences of instructions with independent input and output. To identify resource conflicts, the tool compares the throughput values when instruction pairs are executed separately versus when they are executed in combination. This approach enables the tool to accurately identify and address any conflicts that may be present.

3 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.

3.1 Basic definitions

3.1.1 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.

- 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. As depicted in Listing 3, 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 Fig. 1, in the out-of-order backend, uops are dispatched in different ports. Port pressure describes which port will an instruction executes the corresponding uops.

3.1.2 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 instance, if a basic block contains a jump-related instruction at the end, it is typically assumed to be executed in a way that always takes the branch. On the other hand, if a basic block does not end with a jump-related instruction, it is often assumed to be unrolled multiple times in order to achieve a steady state. In this work, we construct all basic blocks without a jump-related instruction for convenience, i.e. Listing 4, and we remark that introducing a jump-related instruction at the end of the basic blocks will not affect the accuracy of our results.

3.2 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.

Listing 4 An example of a basic block on Kunpeng 920

results.

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

Listing 4 An example of a basic block on Kunpeng 920

- 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.

4 Methodology

4.1 Architecture of Kunpeng 920

Kunpeng 920 is a 64-bit ARM server microprocessor introduced by HiSilicon in early 2019. Fabricated by TSMC on a 7 nm 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 Fig. 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.

4.2 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, in a delineated and systematic manner. To this end, we have conducted multiple measurement methods on the Kunpeng 920 processor.

As introduced in Sect. 2.2, 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.

Obtaining accurate data on the *number of uops* and *port usage* for each instruction is a challenging task, particularly without access to prior knowledge such as Arm's official manual or Agner Fog's "Instruction Table" for X86 architecture (Fog 2022). While most instructions consist of a single uop and are easy to map to their corresponding port, certain instructions prove to be more complex. For example, the "ldr" instruction with post-indexed addressing comprises two uops, with one uop launched at either port 7 or 8, and the other uop launched at port 1, 2, or 3. The intricate combinations between these uops make determining the correct port mappings exceedingly difficult. To address this issue, PMEvo (Ritter and Hack 2020) proposes the use of evolutionary algorithms to automatically derive the port mappings for out-of-order (OOO) processors. It provides support for both X86 and AArch64 architectures, but after we implement this tool to Kunpeng 920 processor, we find the correctness of the result is relatively low. To this end, we adopt the formalism proposed in uops.info Abel and Reineke (2019) to manually get the number of uops and port usage of these complex instructions. In brief, we run the instructions with determined port usage and the instructions with undetermined port usage together one by one and see the throughput result to make sure whether there is interference between these two types of instructions. If not, we can assure that the port usage between these two types of instructions is not overlapped. By applying lots of manual efforts, we determine the number of uops and port usage information of Kunpeng 920's instructions.

4.3 Implementing TSV 110 on OSACA

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

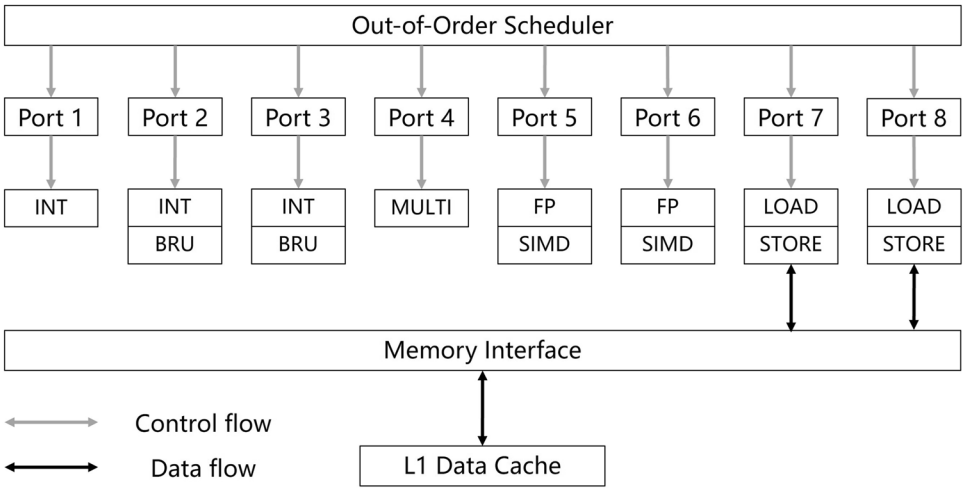

Fig. 2 Overview of the out-of-order execution back end and memory subsystem components of a Kunpeng 920 core

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, etc., 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.

In order to conduct a comparative analysis of the accuracy of OSACA and llvm-mca, we select five distinct real-world applications and extract basic blocks from these applications using the DynamoRIO platform (further details provided in Sect. 5.2). Subsequently, we compare

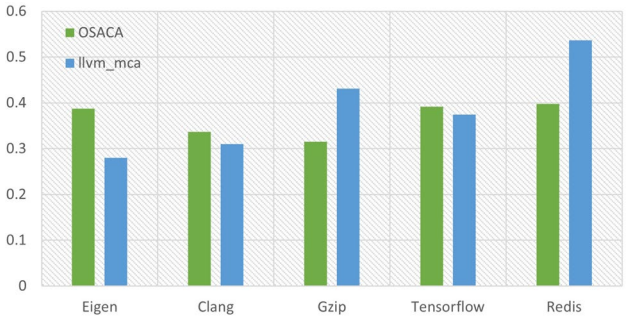

Fig. 3 Mean error rate of OSACA and baseline version of LLVM-MCA on 5 different workloads

the prediction cycles of OSACA and llvm-mca with the actual values obtained from BHive (refer to Sect. 5.1) to determine the mean error rate of each application, as illustrated in Fig. 3.

Considering the constraints of the OSACA framework that inhibit the incorporation of micro-architecture nuances and consequently restrict its capacity for methodical precision enhancement, and having taken into account the analogous error rates observed between OSACA and llvm-mca, we opt to employ llvm-mca as the ongoingly evolving framework.

4.4 Implementing TSV 110 on LLVM-MCA

Although llvm-mca provides native support for Kunpeng 920, as we can see from Fig. 3, its accuracy is not quite desirable. To further investigate the potential of llvm-mca, we calibrate it with Kunpeng 920's micro-architecture that we discovered, thus improving the accuracy of this tool.

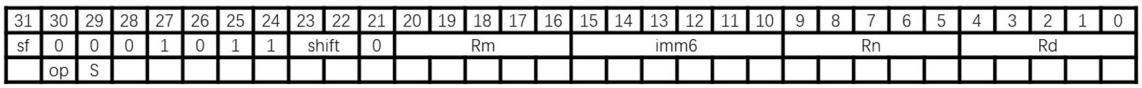

Fig. 4 The encoding of 'ADD (shifted register)' instruction in Armv8-A Instruction Set Architecture

Table 1 The assembler symbol of shift segment in the encoding of 'ADD (shifted register)' instruction

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

4.4.1 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 of 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 Fig. 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 which translates to 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 1. To this end, we assign this reserved item to the instruction without shift, and this simple solution works perfectly with llvm-mca.

4.4.2 Move elimination

Move elimination units are commonly existent in modern high-performance processors due to their ability to eliminate certain register movement operations by register renaming. After devising extensive experiments to reveal the existence of a move elimination unit on Kunpeng 920 processor, we find that Kunpeng 920 will perform move eliminations on the move instructions with register-register operands, such as 'mov x1, x2'. 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.

4.4.3 Zero/one idioms

zero/one-idioms are another important unit that is substantially existent in modern high-performance processors due to their ability to calculate certain instructions whose results are zero or one, e.g., the result of 'sub x1, x1' is always zero. We establish abundant experiments in Kunpeng 920 to observe the possible zero-idioms and one-idioms operations in the X86 architecture, such as eor, com, sub, and so on. We do not find that the microarchitecture has optimized them, so we assume that there is no relevant unit in Kunpeng 920. Further details regarding the specifics and efficacy of each step accuracy improvement can be found in Sect. 6.2

5 Measurements and benchmarks

5.1 Accurate binary code runtime environment

In order to build a runtime environment system to accurately measure the throughput of a basic block as the baseline, we need a set of suitable benchmarks. BHive (Chen et al 2019) is presented to provide a benchmark for systematic validation of performance models of x86 basic blocks, and it also provides a runtime environment for Arm binary assembly codes (bhi 2021). To be specific, BHive will redirect all memory access to a memory page that is mapped in advance, as a consequence, this feature naturally matches the assumption 'All memory access hit L1 cache', which results in providing a perfect measurement for the throughput result of the static code analyzers. However, the tool suffers from a significant drawback whereby it utilizes the performance monitoring unit (PMU) to determine the number of cycles, resulting in a substantial penalty of approximately 300 cycles in absolute error. Although this error may not be significant for long program executions with millions or billions of execution cycles, it poses a significant challenge for small basic block execution cycles of a few hundred or thousand. Despite the possibility of reducing the relative error by unrolling basic blocks extensively, this solution is not always feasible. To address this issue, we propose a simplified approach that involves directly obtaining cycle data using the mov instruction to read the Cycle Count Register value in assembly

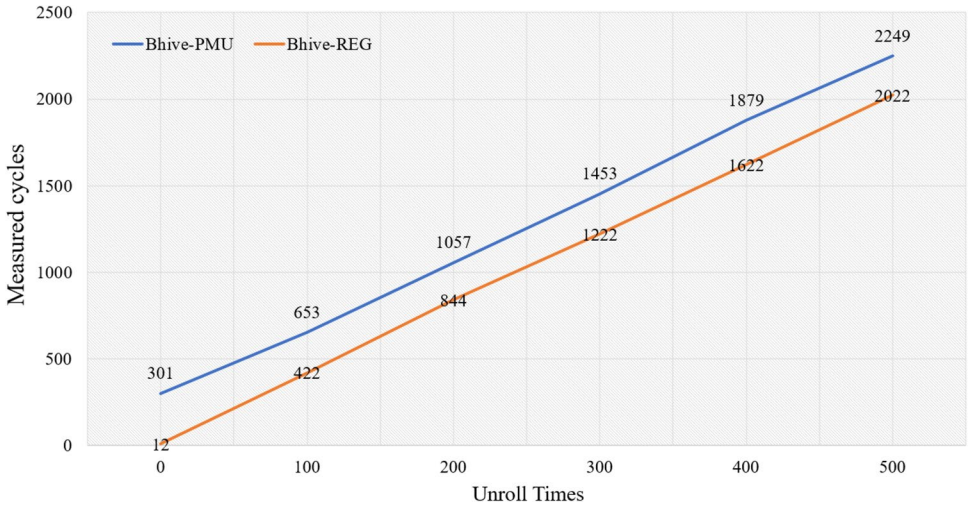

Fig. 5 Cyclic performance discrepancies between bhive utilizing pmu and optimized bhive leveraging register file data

code, resulting in a slight error of approximately 10 cycles. This technique circumvents the latency caused by the PMU API call and eliminates unnecessary PMU event information. As depicted in Fig. 5, We select basic block in List 4 as BHive input to showcase that our optimized Bhive implementation effectively reduces superfluous measurement overhead. We also opensourced our code in Github.[3]

5.2 Benchmarks: basic blocks from real-world applications

Table 2 shows the applications which we chose to collect basic blocks for benchmarking. The applications are chosen based on several objectives. Firstly, the set of applications should cover a diverse range of domains in order to accurately represent real-world workloads (Chen et al 2019). Secondly, the basic blocks should reflect the typical usage patterns of performance model users. Compiler developers, for instance, encounter basic blocks from general-purpose programs that exhibit different characteristics compared to those from high-performance kernels. Lastly, it is deemed important to include traditional high-performance computing applications as a significant component in evaluating the accuracy of performance models, e.g., Fastest Fourier Transform in the West (FFTW) and Linear Algebra PACKage (LAPACK).

In addition to traditional HPC applications, we select a set of representative applications written in general-purpose languages such as C and C++. These applications include Clang (Lattner and Adve 2004b) (compiler), Redis (in-memory database), and Gzip (compression). These commonly used applications employ sophisticated algorithms and data structures, resulting in a diverse range of basic blocks. Furthermore, we also selected a set of widely-used applications that also utilize hand-optimized high-performance kernels, including TensorFlow (Abadi et al 2016) (machine learning), Embree (Wald et al 2014) (ray tracing), and FFmpeg (multimedia).

Table 2 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)	High performance	6127
Clang	Compiler	35598
Redis	Database	5716
GZip	Compression	3749
TensorFlow	Machine Learning	161450
Embree	Ray tracing	7272
FFmpeg	Multimedia	33485
Total		281491

We use the highest Kunpeng 920 optimization settings (-O3, NEON SIMD, etc.) to compile all applications. It is particularly important for high-performance applications such as FFTW.

We implement the benchmarks building procedure in *three steps*, as depicted in Fig. 6:

First, we dynamically capture every assembly instruction executed by the process with the help of a runtime code manipulation system named DynamoRIO (Bruening et al 2003). We opted for dynamic analysis over static disassembly due to the identification of instances where static disassemblers failed to differentiate between padding bytes and instructions (Chen et al 2019).

[3] https://github.com/qcjiang/bhive-reg.

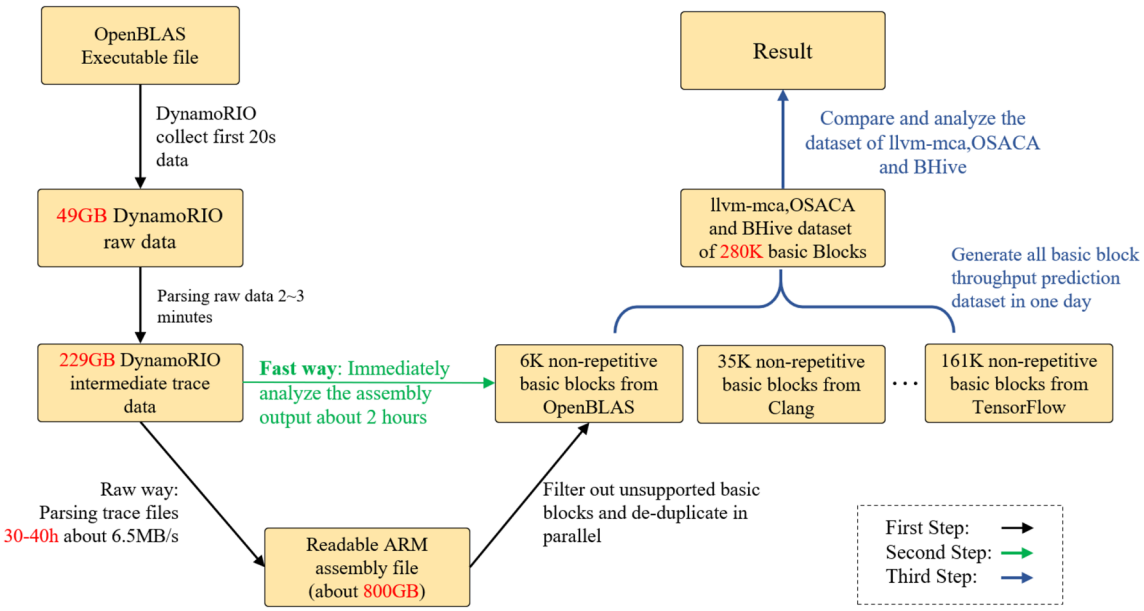

Fig. 6 Workflow of executing, extracting, collecting and analyzing basic blocks, using OpenBLAS as an example

We use the official benchmarking input of each applications to simulate realistic execution, except for FFmpeg and Gzip, which to our knowledge do not have official benchmarks. We test Eigen on two sparse linear algebra workloads: sparse matrix-matrix multiplication (Eigen_MM) and sparse matrix–vector multiplication (Eigen_MV), as shown in Fig. 7. We test LAPACK and OpenBLAS at the same level-3 matrix-matrix multiplication.

As exemplified by the black arrow in Fig. 6, representing the first step, capturing raw assembly code is extremely time consuming and storage consuming. For example, collecting 20 s OpenBLAS program data will need about 40 h to analyze and 1 TB of storage to store the result.

In the *second* step, as indicated by the green arrow in the Fig. 6, we use an in-time method to process the assembly code instead of saving all the assembly code to eliminate the 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.

As illustrated by the blue arrow in Fig. 6, in the *third* step, 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 for download on Github.[4]

6 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.

6.1 Evaluation metrics

6.1.1 MAPE

The mean absolute percentage error (MAPE), also known as mean absolute percentage deviation (MAPD), is a measure of the 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.

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

6.1.2 Kendall's tau

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

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

where Nc denotes the number of concordant pairs and Nd denotes the number of discordant pairs. As argued by Chen et al (2019), Kendall's tau coefficient can prove to be more useful than the MAPE for example for compiler optimizations. Compiler developers tend to prioritize the relative ordering of program execution time when selecting the optimal candidate programs, over their absolute execution time.

6.2 Result

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 3. Our optimized version provides more accurate predictions in all cases. 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.

It is noteworthy that the Eigen (MM) baseline error is significantly high, reaching up to 143.05%. This is due to the fact that the llvm-mca instruction-based simulation model erroneously assumes the presence of a data dependency of register x1 between the loop of instruction ldr q0,[x1,#0x10]!. Further details about this issue are presented in Sect. 8. To address this issue, we resolve this case by dividing the instruction into two instructions so that its micro-operations (uops) could be represented accurately.

In addition, we evaluated our two-step optimization strategy, as presented in Table 4. Due to the relatively low proportion of move instructions within the overall basic block set, we observed a significant improvement in accuracy through the first step of our strategy, which involves correcting the instruction modeling.

6.3 Heatmap analysis

We also draw heat maps to compare our optimized llvm-mca and BHive result on Kunpeng 920 shown in Fig. 7, and the baseline version of llvm-mca is shown in Fig. 8. The heatmap illustrates the relative frequencies of occurrence of cases, with darker colors indicating higher frequencies. Points above the diagonal line indicate predicted throughput values that are greater than the measured values. Conversely, points below the diagonal line indicate predicted throughput values that are less than the measured values.

Table 3 Comparison of different applications llvm-mca results on BHive

Application	llvm-mca-optimized		llvm-mca-baseline	
	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

6.3.1 Mitigating overestimated predicted throughput

Comparing the optimized and original versions of the heatmap, it is evident that our optimized version has effectively addressed the issue of overestimating predicted throughput in Clang, Redis, and HPC applications. This is demonstrated by the notable reduction in the number of points located above the diagonal line in these applications.

To demonstrate how we mitigate overestimated predicted throughput, we select a representative example that consists of basic block as shown in list 5. The basic block is executed for 500 iterations, and the baseline model gave an overestimated predicted throughput of 3500. However, the actual value and the optimized predicted throughput were both 900. This discrepancy can be attributed to an erroneous understanding of the data dependency involving the WZR register in the Bitwise OR (shifted register) instruction ORR $< W_d >$,WZR, $< W_m >$, which is equivalent to MOV $< W_d >$, $< W_m >$ on ARM. In the baseline version, this led to a false assumption of data dependency with the zero register, which does not in fact exist. We corrected this instruction modeling error in our optimized version.

Table 4 Evaluation of two-step optimization strategy

Metrics		
Opt-strategy	MAPE (%)	Kendall
Baseline	35.84	0.2404
Instruction modeling	16.26	0.6102
Instruction modeling + move elimination	13.32	0.6926

6.3.2 Rectifying low-predicted throughput

Upon comparing the heatmaps before and after the optimization of TensorFlow application, it is evident that the count of points with low-predicted throughput below the diagonal has significantly decreased.

To illustrate our approach to rectifying low-predicted throughput below the diagonal, we select a typical example, as shown in list 6. When the basic block is executed for 500 iterations, the baseline model predicts a low throughput of approximately 2000. In reality, both the actual value and the optimized predicted throughput are approximately 6500. In this particular example, LDAXR (Load-Acquire Exclusive Register) is a memory access instruction that possesses atomicity checked by exclusives monitors, leading to higher latency than ordinary ldr instructions. By obtaining the exact latency of LDAXR instruction in Sect. 4.2, we improved the accuracy of the predicted throughput after optimization.

Although we are able to rectify the issue of low predicted throughput below the diagonal line, there remained scattered points in the heatmap that could not be easily fixed due to the functional defect of llvm-mca. This issue is further discussed in Sect. 8.

In summary, our optimizations on llvm-mca result in a 22.52% improvement on average MAPE, as demonstrated by simulation results in 14 different applications presented in Table 3. Our tool achieves 86.7% accuracy in predicting the throughput of all basic blocks, providing system developers with a more credible means of statically predicting the throughput of basic blocks.

7 Application

In this section, we will show the applications that our optimized llvm-mca can be leveraged in the following topics:

1. Guiding the program's optimization by alleviating the bottleneck from register-register dependency and resource pressure in the pre-compile stage.
2. Providing the performance guidance for CPU microarchitecture parameters design in the pre-evaluation phase.

7.1 Accuracy deviation misleads the optimization

In this section, we illustrate how the improvement in accuracy is crucial for avoiding incorrect optimization directions using the two examples presented in Sect. 6.3.

As shown in list 5, the overestimated predicted throughput in the baseline version can mislead compiler developers into thinking that the data dependency is the performance bottleneck of the code fragment. Our optimized version correctly indicates that the data dependency bottleneck does not exist, and increasing the instruction dispatch width is the next optimization direction.

List 6 presents an instance where the low predicted throughput in the baseline version results from incorrect data on the memory access instruction latency. This can cause erroneous judgment on the number of computing instructions that can be covered by the memory access instruction latency, leading to incorrect estimation of whether the program is compute-intensive or memory-intensive. This in turn results in severe deviation of the optimization direction.

These two examples illustrate that only when the results of the static code analyzer achieve high accuracy can the application of the static code analyzer be reliable and widely accepted.

7.2 Program optimization

In this section, we explain how static code analyzer alleviates performance bottlenecks when introducing a new application.

7.2.1 Hotpots identification

Dense matrix–vector multiplication is chosen as the exemplar for optimization, with the multiplication of an N by N matrix with a vector of length N to facilitate further problem discussion. We compile and run the application with the highest optimization settings (-O3, NEON SIMD, etc.) and dynamically capture every assembly instruction executed by the process with the help of a runtime code manipulation system named DynamoRIO (Bruening et al 2003). By leveraging the method of dynamic basic block capture, we identify the core basic block whose execution frequency increases proportionally with the size of the problem as the optimization target. Listing 7 presents the most frequently used basic block during the execution of dense matrix–vector multiplication. We delve into how our optimized llvm-mca can be leveraged to exhibit and mitigate the port usage bottleneck of this basic block, thus guild the optimization of dense matrix–vector multiplication.

Listing 5 An example of Overestimated Predicted Throughput

```
mov x7, x9
mov w6, w11
mov w5, w27
mov x4, x20
mov x2, x26
mov x1, x28
mov x0, x25
```

Listing 5 An example of Overestimated Predicted Throughput

Listing 6 An example of Low-Predicted Throughput

```
add x0, x22, #0x608
add x0, x0, #8
ldaxr w1, [x0]
```

Listing 6 An example of Low-Predicted Throughput

7.2.2 Bottleneck 1: low resource consumption

Given an assembly code sequence, llvm-mca not only estimates the total execution cycles, but also gives the following analyses:

- Processor resource usage: The evaluation of resource pressure considers the aggregate pressure exerted on each instruction across all ports. When an instruction can use multiple ports of the same type, llvm-mca implements a CPU-like simulation strategy of randomized port selection by averaging the port pressures, as shown in Listing 8.
- Data dependencies: To identify data dependencies, the analysis involves keeping track of register and memory access in each instruction and simulating the pipeline. As demonstrated in Listing 9, there is a write-after-write (WAW) dependency of the multiply-accumulate (mla) instruction to register q0 between two loops in Matrix–vector multiplication demonstration.

llvm-mca predicts the total cycles of executing basic block in a loop of iterations depending on the critical path length. Therefore, the total cycles can be modeled as follows:

$$total\_cycles = first\_loop\_cycles + critical\_path\_length * (loops\_num - 1),$$

where *first_loop_cycles* is the number of basic block execution cycles in the initial iteration. *Critical_path_length* is defined as the maximum number of cycles required for one loop of instructions to complete after the initial iteration. And *loops_num* refers to the number of iterations to run. The calculation of the total number of cycles is illustrated in the pipeline simulation diagram presented in Listing 10. The first loop's independent execution time is 17 cycles, while each of the following loops requires 8 cycles due to the data dependency. With a loop count of 1000, the expected total cycle value provided by llvm-mca is

$$17 + 8 * (1000 - -1) = 8009.$$

7.2.3 Optimization 1: loop unrolling

Listing 11 reveals that there is untapped potential to enhance resource utilization, as indicated by the 0.00% Resource Pressure value, and data dependency must be addressed. Loop unrolling was selected as the initial approach to improve resource utilization and eliminate instruction data dependencies. As the reconstructed basic block is shown in Listing 12, the loop unrolling approach extracts the benefits of SIMD parallelism by maximizing the utilization of vector registers.

After implementing the optimization strategy of eight-fold loop unrolling, the pipeline simulation view reveals that the optimized code's total cycles are

$$31 + 16 * (125 - -1) = 2015,$$

where 31 is the number of cycles for the first loop, 16 denotes the critical path latency of eight mla instructions, and 125 signifies the number of loops after applying eight-fold unrolling. Thus, the loop unrolling optimization unlocks a hidden potential to 4-fold performance improvement, as validated by the BHive's output.

7.2.4 Bottleneck 2: strained port utilization

As illustrated in Listing 14, the utilization of the 'TSV110UnitFSU1' port by mla instruction has emerged as the new bottleneck in the program's performance. The timeline view in Listing 13 indicates that the limited throughput of the 'TSV110UnitFSU1' port hampers the performance of the

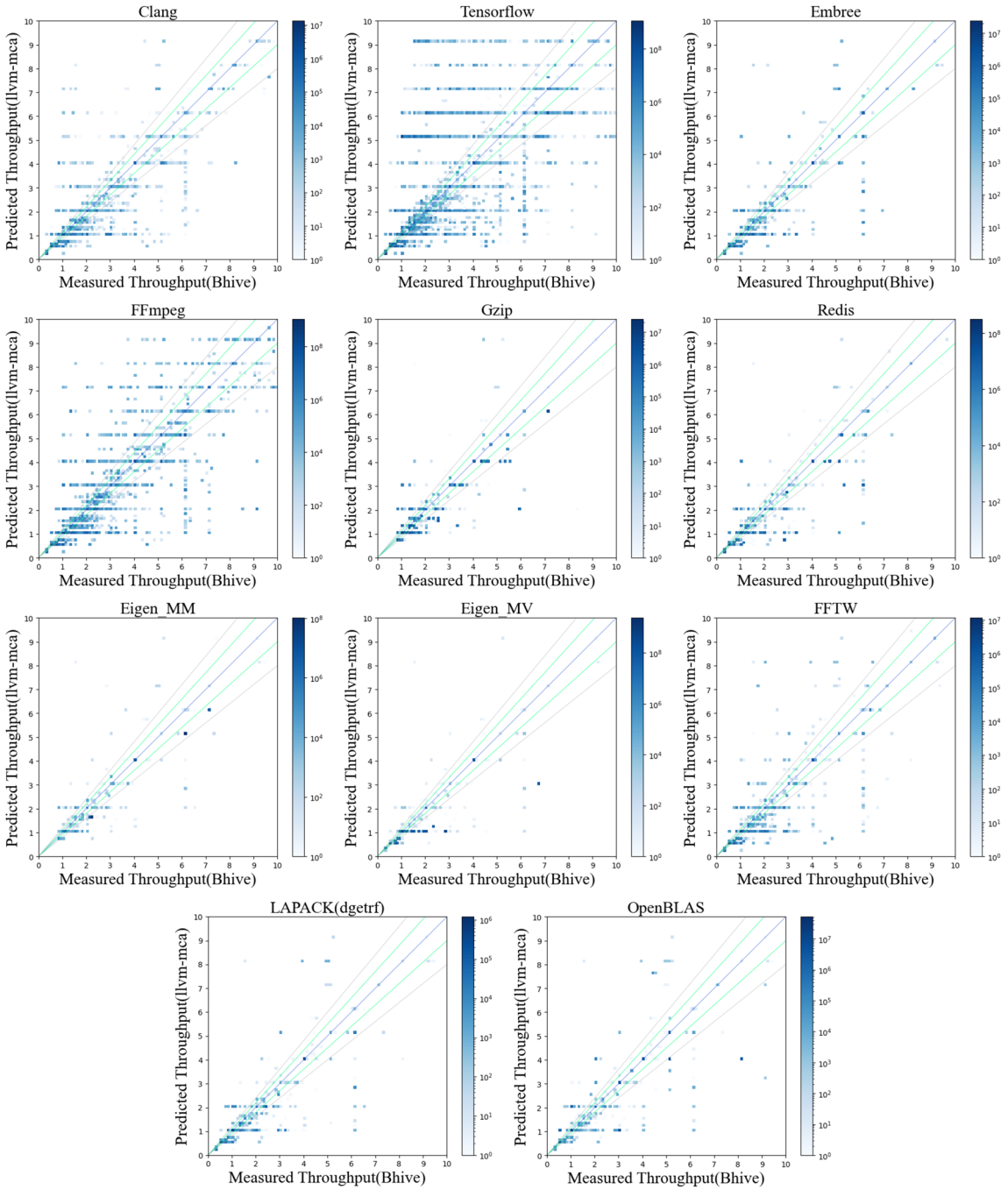

Fig. 7 Heatmaps of optimized version of llvm-mca for basic blocks with a measured throughput of less than 10 cycles/iteration on Kunpeng 920, the green dash line in each figure denotes 10% error rate and the purple dash line denotes 20% error rate

subsequent continuous 'mla' instructions in the latter half of the basic block. Moreover, the 'ldr' instruction in the first half of the basic block was hidden behind the high latency of the multiply-add instructions. Therefore, we make a key observation that the basic block's throughput is constrained by the bad port dispatching of 'mla' instruction.

7.2.5 Optimization 2: port re-usage

As stated in Sect. 4.2, under extensive experiments, we discover that the 'mla' instruction could only be executed on two FP/ASIMD units (TSV110UnitFSU1 and TSV110UnitFSU2). Here, we make a key observation that the 'mla' instructions in this basic block are dependency-free. In other words, by orchestrating the 'mla' instructions to two FP/

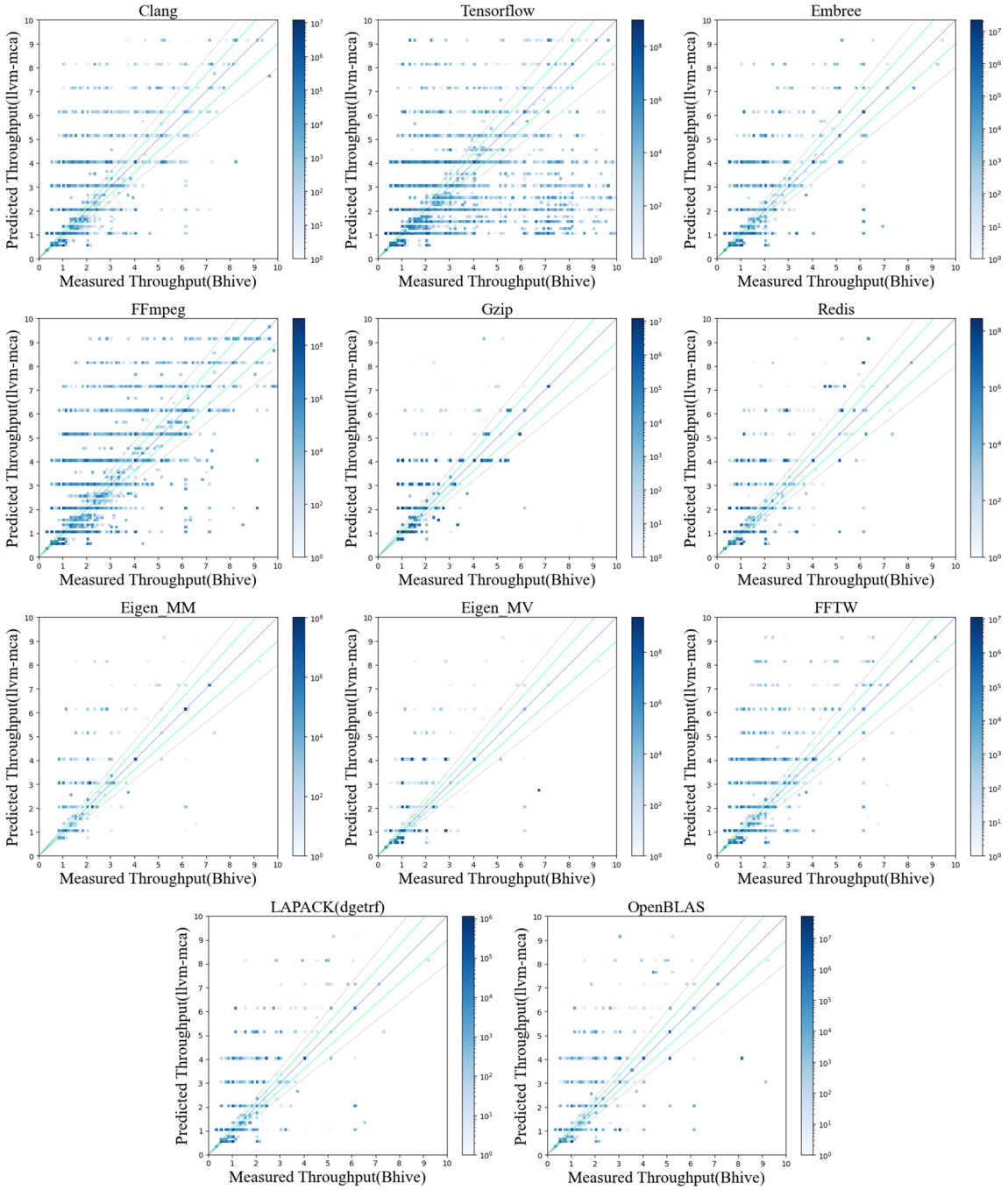

Fig. 8 Heatmaps of baseline version of llvm-mca for basic blocks with a measured throughput of less than 10 cycles/iteration on Kunpeng 920, the green dash line in each figure denotes 10% error rate and the purple dash line denotes 20% error rate

ASIMD units, we can further extract the benefits of Kunpeng 920 processor. In this way, the theoretical throughput of this basic block drops to

$$25 + 9 * (125 - -1) = 1141.$$

Port reusage results in a nearly double speed compared to the previous step. Following two rounds of optimization guided by llvm-mca, we further enhance the performance of 7x compared to the baseline version. Therefore, this subsection elaborates that llvm-mca can be leveraged as a promising way for programmers to exploit the underlying performance bottleneck systematically and alleviate it correspondingly.

Listing 7 Fundamental basic block for dense Matrix-vector multiplication

```
ldr q2, [x0, x2]
ldr q1, [x1, x0]
add x0, x0, 16
cmp x0, x3
mla v0.4s, v2.4s, v1.4s
```

Listing 7 Fundamental basic block for dense Matrix-vector multiplication

Listing 8 Resource pressure analysis of baseline basic block through llvm-mca

```
Resources:
[0.0] - TSV110UnitAB
[0.1] - TSV110UnitAB
[1]   - TSV110UnitALU
[2]   - TSV110UnitFSU1
[3]   - TSV110UnitFSU2
[4.0] - TSV110UnitLdSt
[4.1] - TSV110UnitLdSt
[5]   - TSV110UnitMDU

Resource pressure per iteration:
[0.0] [0.1] [1]   [2]   [3] [4.0] [4.1] [5]
1.50  1.50  1.00 2.00   -  1.00  1.00   -

Resource pressure by instruction:
[0.0] [0.1] [1]   [2]   [3] [4.0] [4.1] [5]  Instructions:
0.01  0.01  0.98  -     -   -    1.00   -   ldr     q2, [x0, x2]
0.49  0.49  0.02  -     -  1.00   -      -   ldr     q1, [x1, x0]
0.50  0.50   -    -     -   -     -      -   add     x0, x0, #16
0.50  0.50   -    -     -   -     -      -   cmp     x0, x3
 -     -     -   2.00   -   -     -      -   mla     v0.4s, v2.4s, v1.4s
```

Listing 8 Resource pressure analysis of baseline basic block through llvm-mca

7.3 Guidance for CPU design in pre-evaluation stage

Performance modeling and analysis have become an essential part of the co-design process in the design of contemporary computing systems, particularly for high-performance microprocessors (Mukherjee et al 2002). As Fig. 9 shows, performance modeling is intricately involved in every step of CPU design. Information changes in each step of CPU design, such as logic and circuit design, affect performance modeling parameter configurations and impact the final design choice, including throughput and latency estimates and feasibility of circuit layout.

Cycle-accurate full-system simulators, which are commonly used for performance modeling, often suffer from slow simulation speed. Static code analyzer(e.g., llvm-mca) is a fast and lightweight tool that uses static code analysis to identify performance-critical instructions or sequences. CPU designers can use this tool to guide their design choices, such as pipeline optimizations or custom hardware modules for specific instructions (e.g., hardware multiplier for multiplication-intensive code).

By incorporating llvm-mca components before the traditional performance modeling, designers can pre-evaluate and screen out unsuitable designs using llvm-mca, and reduce the design space that needs to be evaluated by the traditional cycle-accurate simulator. As shown in Fig. 9, the vacant space within the dashed box denotes the design space eliminated in pre-evaluation stage. This approach can yield a substantial reduction in the time required for conventional performance modeling, thereby abbreviating the entire performance modeling workflow.

Compared to the traditional performance model, the static code analyzer differs in several significant ways:

7.3.1 Precision

We endeavor to ensure that the hardware details simulated by gem5 are as closely aligned with those of the Kunpeng 920, as discussed in Sect. 4.1, which highlights the intricate microarchitectural nuances of the Kunpeng processor. In cases where specific microarchitecture details of the Kunpeng processor are unknown, we utilize the default configuration settings of the ARMv8 architecture. Our evaluation of gem5's SE mode on the Kunpeng architecture, as compared to the runtime cycle count obtained through perf, demonstrates a remarkable level of precision. This precision is observed across various applications, as illustrated in Table 5. Although the static code analyzer may not possess the same degree of precision as gem5, it is still capable of conducting rapid pre-evaluation and exclusion of unsuitable designs during CPU design.

Listing 9 Critical sequence analysis of baseline basic block through llvm-mca

```
Critical sequence based on the simulation:
        Instruction              Dependency Information
 +-< 4. mla v0.4s, v2.4s, v1.4s

      < loop carried >

     0. ldr q2, [x0, x2]
     1. ldr q1, [x1, x0]
     2. add x0, x0, #16
     3. cmp x0, x3
 +-> 4. mla v0.4s, v2.4s, v1.4s  ## REGISTER dependency:  q0

      < loop carried >

 +-> 4. mla v0.4s, v2.4s, v1.4s  ## REGISTER dependency:  q0
```

Listing 9 Critical sequence analysis of baseline basic block through llvm-mca

Listing 10 Pipeline timeline view of baseline basic block through llvm-mca

```
Timeline view:
                    0123456789          0123456789          0123456789
Index     0123456789          0123456789          0123456789

[0,0]     DeeeeeeER .    .    .    .    .    ldr         q2, [x0, x2]
[0,1]     DeeeeeeER .    .    .    .    .    ldr         q1, [x1, x0]
[0,2]     .DeE----R .    .    .    .    .    add         x0, x0, #16
[0,3]     .D=eE---R .    .    .    .    .    cmp         x0, x3
[0,4]     .D=====eeeeeeeeER   .    .    .    mla         v0.4s, v2.4s, v1.4s
[1,0]     . DeeeeeeE------R   .    .    .    ldr         q2, [x0, x2]
[1,1]     . DeeeeeeE------R   .    .    .    ldr         q1, [x1, x0]
[1,2]     .  DeE----------R   .    .    .    add         x0, x0, #16
[1,3]     .  D=eE---------R   .    .    .    cmp         x0, x3
[1,4]     .  D==========eeeeeeeeER.    .    mla         v0.4s, v2.4s, v1.4s
[2,0]     .   DeeeeeeE------------R.    .    ldr         q2, [x0, x2]
[2,1]     .   DeeeeeeE------------R.    .    ldr         q1, [x1, x0]
```

Listing 10 Pipeline timeline view of baseline basic block through llvm-mca

7.3.2 Speed

The traditional full-system cycle-accurate simulator, such as gem5 (Binkert et al 2011) is based on the cycle-by-cycle simulation. But the static code analyzer like llvm-mca is designed to be a lightweight and fast tool that can quickly analyze the performance of assembly code on a specific CPU. It is typically much faster than gem5.

We establish an experimental procedure to measure the speed gap between llvm-mca and gem5: First, we employed the basic block collection framework mentioned in Sect. 5.2 to gather the basic blocks from various applications. Next, we conduct a summation of the simulated time of the hot basic blocks from llvm-mca. Finally, we compare the obtained results from gem5 simulation to evaluate the speed discrepancy.

According to Table 6, llvm-mca demonstrates a marked improvement of three orders of magnitude in simulation time compared to gem5. This improvement is attributed to llvm-mca's consistent analysis time for each basic block, leading to a simulation time that is proportional only to the number of distinct hot basic blocks. In contrast, gem5's simulation time is often constrained by the magnitude of the program input.

7.3.3 Scalability

Static code analyzer like llvm-mca provides a user-friendly infrastructure with a library of modules that model different hardware components, such as TLB and loop stream detector (LSD). With this library, designers and researchers can easily reuse, extend, and modify architectural components to quickly build specific complex cpu models. Meanwhile, by extending llvm-mca to support more advanced features of modern CPUs, such as hardware security features or new memory technologies, it is possible to achieve a more accurate representation of the CPU and its behavior.

Static code analyzer is a lightweight tool to be easily used and can be integrated into existing development workflows more easily than a full-system simulator like Gem5. By combining static code analyzer with other tools such as memory simulators like DRAMSim2 (Rosenfeld et al 2011) or Ramulator (Kim et al 2016), it is possible to achieve a more comprehensive simulation of computer systems that includes CPU performance and memory behavior. However, it is important to note that adding more components can also increase the complexity of the model and potentially reduce its speed. Balancing accuracy and performance is an important consideration when developing and using CPU models for performance analysis.

Listing 11 Throughput bottleneck analysis of baseline basic block through llvm-mca

```
Cycles with backend pressure increase [ 12.49% ]
Throughput Bottlenecks:
  Resource Pressure       [ 0.00% ]
  Data Dependencies:      [ 12.49% ]
  - Register Dependencies [ 12.49% ]
  - Memory Dependencies   [ 0.00% ]
```

Listing 11 Throughput bottleneck analysis of baseline basic block through llvm-mca

Listing 12 Basic block using loop unrolling optimization for matrix-vector multiplication

```
ldr     q0, [x0, x10]
ldr     q24, [x8, x0]
ldr     q23, [x7, x0]
ldr     q22, [x6, x0]
ldr     q21, [x5, x0]
ldr     q20, [x4, x0]
ldr     q19, [x3, x0]
ldr     q18, [x2, x0]
ldr     q17, [x1, x0]
add     x0, x0, 16
mla     v2.4s, v0.4s, v24.4s
cmp     x0, x9
mla     v3.4s, v0.4s, v23.4s
mla     v4.4s, v0.4s, v22.4s
mla     v5.4s, v0.4s, v21.4s
mla     v6.4s, v0.4s, v20.4s
mla     v7.4s, v0.4s, v19.4s
mla     v16.4s, v0.4s, v18.4s
mla     v1.4s, v0.4s, v17.4s
```

Listing 12 Basic block using loop unrolling optimization for matrix-vector multiplication

Listing 13 Pipeline timeline view after using loop unrolling optimization

```
Timeline view:
                    0123456789          0123456789          0123456789
Index     0123456789          0123456789          0123456789

[0,0]     DeeeeeeER .    .    .    .    .    .  ldr  q0, [x0, x10]
[0,1]     DeeeeeeER .    .    .    .    .    .  ldr  q24, [x8, x0]
[0,2]     .DeeeeeeER.    .    .    .    .    .  ldr  q23, [x7, x0]
[0,3]     .DeeeeeeER.    .    .    .    .    .  ldr  q22, [x6, x0]
[0,4]     . DeeeeeeER    .    .    .    .    .  ldr  q21, [x5, x0]
[0,5]     . DeeeeeeER    .    .    .    .    .  ldr  q20, [x4, x0]
[0,6]     .  DeeeeeeER   .    .    .    .    .  ldr  q19, [x3, x0]
[0,7]     .  DeeeeeeER   .    .    .    .    .  ldr  q18, [x2, x0]
[0,8]     .   DeeeeeeER  .    .    .    .    .  ldr  q17, [x1, x0]
[0,9]     .   DeE-----R  .    .    .    .    .  add  x0, x0, #16
[0,10]    .    D=eeeeeeeER   .    .    .    .  mla  v2.4s, v0.4s, v24.4s
[0,11]    .    DeE--------R  .    .    .    .  cmp  x0, x9
[0,12]    .    .D==eeeeeeeER .    .    .    .  mla  v3.4s, v0.4s, v23.4s
[0,13]    .    .D====eeeeeeeER    .    .    .  mla  v4.4s, v0.4s, v22.4s
[0,14]    .    . D=====eeeeeeeER  .    .    .  mla  v5.4s, v0.4s, v21.4s
[0,15]    .    . D=======eeeeeeeER.    .    .  mla  v6.4s, v0.4s, v20.4s
[0,16]    .    .  D========eeeeeeeER   .    .  mla  v7.4s, v0.4s, v19.4s
[0,17]    .    .  D==========eeeeeeeER .    .  mla  v16.4s, v0.4s, v18.4s
[0,18]    .    .   D==========eeeeeeeER    .  mla  v1.4s, v0.4s, v17.4s
[1,0]     .    .   DeeeeeeE------------R   .  ldr  q0, [x0, x10]
[1,1]     .    .    DeeeeeeE------------R   .  ldr  q24, [x8, x0]
[1,2]     .    .    DeeeeeeE------------R   .  ldr  q23, [x7, x0]
[1,3]     .    .    .DeeeeeeE-----------R   .  ldr  q22, [x6, x0]
[1,4]     .    .    .DeeeeeeE-----------R   .  ldr  q21, [x5, x0]
[1,5]     .    .    . DeeeeeeE----------R   .  ldr  q20, [x4, x0]
[1,6]     .    .    . DeeeeeeE----------R   .  ldr  q19, [x3, x0]
[1,7]     .    .    .  DeeeeeeE---------R   .  ldr  q18, [x2, x0]
[1,8]     .    .    .  DeeeeeeE---------R   .  ldr  q17, [x1, x0]
[1,9]     .    .    .   DeE-------------R   .  add  x0, x0, #16
[1,10]    .    .    .   D========eeeeeeeER  .  mla  v2.4s, v0.4s, v24.4s
[1,11]    .    .    .   D=eE-------------R  .  cmp  x0, x9
[1,12]    .    .    .    D=========eeeeeeeER.  mla  v3.4s, v0.4s, v23.4s
```

Listing 13 Pipeline timeline view after using loop unrolling optimization

Listing 14 Critical sequence analysis after using loop unrolling optimization. RINT is anabbreviation for resource interference

```
 Critical sequence based on the simulation:

      Instruction                         Dependency Information
+< 18. mla v1.4s, v0.4s, v17.4s

   < loop carried >

   0.  ldr q0, [x0, x10]
   1.  ldr q24, [x8, x0]
   2.  ldr q23, [x7, x0]
   3.  ldr q22, [x6, x0]
   4.  ldr q21, [x5, x0]
   5.  ldr q20, [x4, x0]
   6.  ldr q19, [x3, x0]
   7.  ldr q18, [x2, x0]
   8.  ldr q17, [x1, x0]
   9.  add x0, x0, #16
+> 10. mla v2.4s, v0.4s, v24.4s  #RINT: TSV110UnitFSU1 [probability:99% ]
   11. cmp x0, x9
+> 12. mla v3.4s, v0.4s, v23.4s  #RINT: TSV110UnitFSU1 [probability:100%]
+> 13. mla v4.4s, v0.4s, v22.4s  #RINT: TSV110UnitFSU1 [probability:100%]
+> 14. mla v5.4s, v0.4s, v21.4s  #RINT: TSV110UnitFSU1 [probability:100%]
+> 15. mla v6.4s, v0.4s, v20.4s  #RINT: TSV110UnitFSU1 [probability:100%]
+> 16. mla v7.4s, v0.4s, v19.4s  #RINT: TSV110UnitFSU1 [probability:100%]
+> 17. mla v16.4s, v0.4s, v18.4s #RINT: TSV110UnitFSU1 [probability:100%]
+> 18. mla v1.4s, v0.4s, v17.4s  #RINT: TSV110UnitFSU1 [probability:100%]

   < loop carried >

+> 10. mla v2.4s, v0.4s, v24.4s  #RINT: TSV110UnitFSU1 [probability:99% ]
```

Listing 14 Critical sequence analysis after using loop unrolling optimization. RINT is an abbreviation for resource interference.

8 Discussion

Although our optimized llvm-mca can provide good accuracy results on different workloads, there still exists a 13.3% error rate, we discover this remaining stumbling block comes from the two following factors.

8.1 Micro-architecture exploration

Due to the lack of corresponding events in PMU units, it is hard for us to exploit the specifics in micro-architecture in depth, e.g., macro/micro fusion, scheduler, etc. As a result, we cannot further add specifics to llvm-mca to build a parametric simulation model to faithfully represent the execution process, thus improving the accuracy of our predictions systematically.

8.2 Uops-based simulation model

Although llvm-mca needs uops information as input, it is in fact instructions-based simulation tool, which means the minimum unit of the simulation is instruction instead of uop. However, modern processors no matter whether reduced instruction set computer (RISC) or complex instruction set computer (CISC) architecture will further divide the instructions to uops after the instructions are decoded. In the entail execution back end in these processors, the units like scheduler, ALU, etc., will execute the uops instead of instructions. To this end, a uops-based simulation model can further explore the potential of static code analyzers.

To illustrate the discrepancy between instruction-based simulation and uops-based simulation, we refer to the case mentioned in Sect. 6.2. Specifically, the instruction ldr q0,[x1,#10]! can be split into two micro-operations: add x1,x1,#10 and ldr q0,[x1]. While the instruction-based simulation predicted the cycle count to be 2503, the actual cycle count obtained through uops-based simulation is 500, as indicated in Listing 15.

9 Conclusion

In this work, we implement the 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. After guiding the optimization of

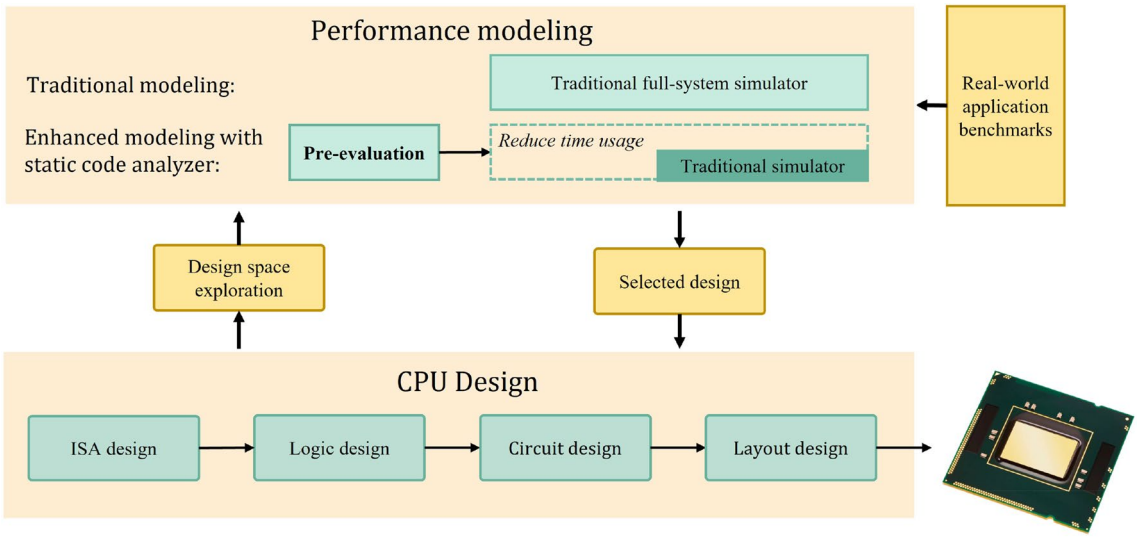

Fig. 9 Workflow of CPU design guided by performance modelings. Enhanced modeling is much faster than traditional modeling by using llvm-mca in pre-evaluation stage

Table 5 Cycles comparison between perf and gem5 for various demonstrations

Application	Perf cycles	Gem5 cycles	Relative error (%)
Matrix-vector multiplication	272058382	270340386	0.63
Merge sort	140051195	142381554	1.66
Binary research	120217497	116937068	2.72
Average			1.67

Table 6 Simulation time comparison between llvm-mca and gem5 for various demonstrations

Application	llvm-mca	Gem5	Speedup
Matrix-vector multiplication	1.67 s	1411.50 s	845.2
Merge sort	0.95 s	1782.07 s	1875.8
Binary research	0.52 s	951.78 s	1830.3
Average			1517.1

Matrix–vector multiplication and comparing it with traditional performance models in CPU design, static code analyzer shows great potential for ongoing evolution and refinement as a means of CPU performance analysis. Its high speed, adaptability, extensibility, and capacity for cycle-accurate simulation make it an essential instrument for comprehending and optimizing the performance of code on current-generation CPUs.

Acknowledgements This work is partially supported by the National Natural Science Foundation of China (Grant No. 62102389).

Author contributions ST and QJ formulated the original idea and wrote the main manuscript. ZC and XH designed part of the experiments. JC and HA supervised the project and provided fundings. All authors reviewed the manuscript.

Declarations

Conflict of interest The authors declare no competing interests.

Listing 15 Pipeline timeline view of instruction-based simulation and uops-basedsimulation for an example basic block

```
Instructions-based simulation:
    [0,0]     DeeeeeER  .    .    .    . .   ldr   q0, [x1, #10]!
    [1,0]     D=====eeeeeER  .    .    . .   ldr   q0, [x1, #10]!
    [2,0]     .D=========eeeeeER  .    . .   ldr   q0, [x1, #10]!

Uops-based simulation.
    [0,0]     DeER .    .              . .   add   x1, x1, #10
    [0,1]     D=eeeeeER .              . .   ldr   q0, [x1]
    [1,0]     D=eE----R .              . .   add   x1, x1, #10
    [1,1]     D==eeeeeER.              . .   ldr   q0, [x1]
    [2,0]     .D=eE----R.              . .   add   x1, x1, #10
    [2,1]     .D==eeeeeER              . .   ldr   q0, [x1]
```

Listing 15 Pipeline timeline view of instruction-based simulation and uops-based simulation for an example basic block.

References

Abadi, M., Barham, P., Chen, J., et al.: Tensorflow: A system for large-scale machine learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. USENIX Association, USA, OSDI'16, p 265-283 (2016)

Abel, A., Reineke, J.: uops. info: characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp 673–686 (2019)

Abel, A., Reineke, J.: uica: accurate throughput prediction of basic blocks on recent intel microarchitectures. In: Proceedings of the 36th ACM International Conference on Supercomputing, pp 1–14 (2022)

Alappat, C., Meyer, N., Laukemann, J., et al.: 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)

Binkert, N., Beckmann, B., Black, G., et al.: The gem5 simulator. ACM SIGARCH Comput. Archit. News **39**(2), 1–7 (2011)

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

Charif-Rubial, A.S., Oseret, E., Noudohouenou, J., et al.: Cqa: a code quality analyzer tool at binary level. In: 2014 21st International Conference on High Performance Computing (HiPC), IEEE, pp 1–10 (2014)

Chen, Y., Brahmakshatriya, A., Mendis, C., et al.: 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, pp 167–177 (2019)

Dayhoff, J.E.: Neural network architectures: an introduction. Van Nostrand Reinhold Co (1990)

Eeckhout, L.: Computer architecture performance evaluation methods. Synth. Lect. Comput. Archit. **5**(1), 1–145 (2010)

Fog, A.: The microarchitecture of intel, amd, and via cpus. https://www.agner.org/optimize/microarchitecture.pdf (2022). Accessed 11 Sept 2023

Hammer, J., Hager, G., Wellein, G.: Ooo instruction benchmarking framework on the back of dragons. SC18 SRC Poster (in review) (2018)

HiSilicon.: Kunpeng 920 chipset. https://www.hisilicon.com/en/products/Kunpeng/Huawei-Kunpeng/Huawei-Kunpeng-920 (2021). Accessed 11 Sept 2023

Hofmann, J.: ibench-instruction benchmarks (2017). URL https://githubcom/RRZE-HPC/ibench

Huawei.: Taishan v110 - microarchitectures - hisilicon. https://en.wikichip.org/wiki/hisilicon/microarchitectures/taishan_v110 (2022). Accessed 11 Sept 2023

Huh, J., Tuck, J.: Improving the effectiveness of searching for isomorphic chains in superword level parallelism. In: 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, pp 718–729 (2017)

Intel architecture code analyzer user's guide. https://software.intel.com/content/dam/develop/external/us/en/documents /intel-architecture-code-analyzer-3-0-users-guide-157552.pdf (2017). Accessed 11 Sept 2023

llvm-mca - llvm machine code analyzer. https://llvm.org/docs/CommandGuide/llvm-mca.html (2022). Accessed 11 Sept 2023

Jiang, Q., Tan, S., Cao, Z., et al.: Quantifying throughput of basic blocks on arm microarchitectures by static code analyzers: A case study on kunpeng 920. In: 2022 IEEE 24nd International Conference on High Performance Computing and Communications; IEEE 20th International Conference on Smart City; IEEE 8th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), IEEE (2022)

Kim, Y., Yang, W., Mutlu, O.: Ramulator: a fast and extensible dram simulator. IEEE Comput. Archit. Lett. **15**(1), 45–49 (2016). https://doi.org/10.1109/LCA.2015.2414456

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

Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004., IEEE, pp 75–86 (2004a)

Lattner, C., Adve, V.: Llvm: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004., pp 75–86, https://doi.org/10.1109/CGO.2004.1281665 (2004b)

Laukemann, J., Hammer, J., Hager, G., et al.: Automatic throughput and critical path analysis of x86 and arm assembly kernels. In: 2019 IEEE/ACM Performance Modeling, pp. 1–6. Benchmarking and Simulation of High Performance Computer Systems (PMBS), IEEE (2019)

Laukemann, J., Hammer, J., Hofmann, J., et al.: 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, pp 121–131 (2018)

Lozano, R.C., Carlsson, M., Drejhammar, F., et al.: Constraint-based register allocation and instruction scheduling. In: International Conference on Principles and Practice of Constraint Programming, Springer, pp 750–766 (2012)

McGovern, A., Moss, J.: Scheduling straight-line code using reinforcement learning and rollouts. Advances in neural information processing Systems 11 (1998)

Mendis, C., Amarasinghe, S.: goslp: globally optimized superword level parallelism framework. Proceedings of the ACM on Programming Languages 2(OOPSLA):1–28 (2018)

Mendis, C., Renda, A., Amarasinghe, S., et al.: Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In: International Conference on machine learning, PMLR, pp 4505–4515 (2019)

Mukherjee, S., Adve, S., Austin, T., et al.: Performance simulation tools. Computer **35**, 38–39 (2002). https://doi.org/10.1109/2.982914

Pohl, A., Cosenza, B., Juurlink, B.: Portable cost modeling for auto-vectorizers. In: 2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), IEEE, pp 359–369 (2019)

Pohl, A., Cosenza, B., Juurlink, B.: Vectorization cost modeling for neon, avx and sve. Perform. Eval. **140**(102), 106 (2020)

Reimplementation of the bhive profiler (2021). https://github.com/gilbertmike/bhive. Accessed 11 Sept 2023

Renda, A., Chen, Y., Mendis, C., et al.: Difftune: Optimizing cpu simulator parameters with learned differentiable surrogates. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, pp 442–455 (2020)

Ritter, F., Hack, S.: Pmevo: portable inference of port mappings for out-of-order processors by evolutionary optimization. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp 608–622 (2020)

Rosenfeld, P., Cooper-Balis, E., Jacob, B.: Dramsim2: a cycle accurate memory system simulator. IEEE Comput. Archit. Lett. **10**(1), 16–19 (2011). https://doi.org/10.1109/L-CA.2011.4

Sanchez, D., Kozyrakis, C.: Zsim: fast and accurate microarchitectural simulation of thousand-core systems. ACM SIGARCH Comput. Architect. News **41**(3), 475–486 (2013)

Seiferth, J., Alappat, C., Korch, M., et al.: Applicability of the ecm performance model to explicit ode methods on current multi-core processors. In: International Conference on High Performance Computing, Springer, pp 163–183 (2018)

Stengel, H., Treibig, J., Hager, G., et al.: Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In: Proceedings of the 29th ACM on International Conference on Supercomputing, pp 207–216 (2015)

Stephenson, M., Amarasinghe, S., Martin, M., et al.: Meta optimization: improving compiler heuristics with machine learning. ACM SIGPLAN Not. **38**(5), 77–90 (2003)

Treibig, J., Hager, G.: Introducing a performance model for bandwidth-limited loop kernels. In: International Conference on Parallel Processing and Applied Mathematics, Springer, pp 615–624 (2009)

Wald, I., Woop, S., Benthin, C., et al.: Embree: a kernel framework for efficient CPU ray tracing. ACM Trans. Graph. (2014). https://doi.org/10.1145/2601097.2601199

Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009)