中国科学院大学

University of Chinese Academy of Sciences

博士学位论文

开源处理器的架构迭代加速方法与指令调度架构研究

作者姓名：周耀阳

指导教师：孙凝晖 研究员

学位类别：工学博士

学科专业：计算机系统结构

培养单位：中国科学院计算技术研究所

2023 年 6 月

A Study of Architecture Iteration Acceleration and Instruction Scheduling Architecture for Open-source Processors

A dissertation submitted to

University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Doctor of Philosophy

in Computer Architecture

by

Zhou Yaoyang

Supervisor: Professor Sun Ninghui

Institute of Computing Technology, Chinese Academy of Sciences

June, 2023

中国科学院大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。承诺除文中已经注明引用的内容外，本论文不包含任何其他个人或集体享有著作权的研究成果，未在以往任何学位申请中全部或部分提交。对本论文所涉及的研究工作做出贡献的其他个人或集体，均已在文中以明确方式标明或致谢。本人完全意识到本声明的法律结果由本人承担。

作者签名：

日　　期：

中国科学院大学
学位论文授权使用声明

本人完全了解并同意遵守中国科学院大学有关收集、保存和使用学位论文的规定，即中国科学院大学有权按照学术研究公开原则和保护知识产权的原则，保留并向国家指定或中国科学院指定机构送交学位论文的电子版和印刷版文件，且电子版与印刷版内容应完全相同，允许该论文被检索、查阅和借阅，公布本学位论文的全部或部分内容，可以采用扫描、影印、缩印等复制手段以及其他法律许可的方式保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：　　导师签名：

日　　期：　　日　　期：

摘 要

随着 RISC-V 指令集的诞生，国内外兴起了开源芯片的热潮。各种新兴的硬件编程语言和开源处理器项目能大幅加速处理器芯片的 RTL 开发过程。但是对高性能处理器而言，当前的开源处理器（例如香山处理器和 Boom 处理器）与国际领先的商用处理器之间的性能差距较大。无论是商业公司还是开源社区，都希望基于当前的开源处理器项目迭代出商业级性能的处理器。如何加速处理器的架构迭代，如何设计出性能更高的微架构，是研究者与开源社区面对的重要挑战。

为了加速开源处理器架构迭代过程，本文围绕性能测算加速和性能精确建模开展研究。针对处理器微架构的性能不足的短板，本文围绕处理器的指令调度架构开展研究。本文的研究工作和主要贡献包括：

1）提出了一个针对 RISC-V 的敏捷性能评估框架，加速了香山处理器的性能测算速度。面对当前的主流 RTL 性能测算方法存在成本高、速度慢的问题，本文研究了如何用低成本的平台实现快速的处理器性能测算。本文设计了一种跨平台的 RISC-V 检查点 RVGCpt。该方法结合采样仿真方法，使得基于 RTL 软件仿真的性能测算时间从数年减少到数天。面对采样仿真中预热时间过长的问题，本文实现了面向 RTL 仿真的缓存功能预热技术和自适应混合预热方案 HyWarm，使得缓存预热时间缩短 85.7%。敏捷性能评估框架在一台 128 核服务器上仅需约 66 小时即可完成香山处理器的 SPECCPU 2006 分值测算，估算的分值相比于流片实测性能的误差约为 11%。

2）提出了一种性能分析框架以加速定位性能缺陷，并应用于构建香山处理器的精准微架构模拟器。在开源处理器的微架构设计迭代中，一个重要障碍是现有开源高性能处理器缺乏经过校准的微架构模拟器。为了校准微架构模拟器，性能计数器至关重要。现有的自顶向下分析框架和自底向上计数器相互脱节，无法快速定位产生性能差异的组件和指令序列。本文为香山处理器搭建了微架构模拟器 XS-GEM5，并提出了展开式自顶向下分析框架和并行访存缺失簇分析方法。当发现性能缺陷时，展开式自顶向下分析框架和并行访存缺失簇分析方法分别可以加速定位存在缺陷的处理器部件和触发缺陷的指令片段。

3）提出了针对动态数据流指令调度架构的分析模型和改进方法。扩大处理器核的规模是提升处理器性能的重要途径。由于发射窗口和物理寄存器的面积、时延关于指令窗口大小呈超线性增长，发射窗口和物理寄存器是限制处理器核规模的主要瓶颈之一。为了实现可扩展的乱序执行架构，本文利用了动态数据流指令调度架构来避免发射窗口和物理寄存器。本文提出了基于指令唤醒信息的分析模型来定位现存的动态数据流指令调度架构的瓶颈。在分析模型的指导下，本文以较小的功耗和面积开销达到了传统超标量架构约 94% 的性能。

关键词：芯片敏捷开发，开源处理器，负载采样，处理器性能建模，指令调度

Abstract

Since the birth of the RISC-V instruction set, there has been a surge of open-source chip projects internationally. Emerging hardware programming languages and open-source processors can significantly accelerate the RTL development. However, for general-purpose processors, existing open-source processors (such as Xiangshan processor and Boom processor) have a significant performance gap compared to internationally leading commercial processors. Both commercial companies and open-source community hope to develop processors with industry-level performance based on the current open-source processor projects. How to accelerate the performance iteration of processors and how to design microarchitectures with higher performance are important challenges faced by both researchers and the open-source community.

To accelerate the performance iteration of open-source processors, this thesis conducts research on performance measurement acceleration and performance accurate modeling. In view of the performance deficiencies of the processor microarchitecture, this thesis focuses on the research on the instruction scheduling architecture of the processor. The research work and main contributions of this thesis include:

1) Proposing an agile performance measurement framework for RISC-V, which accelerates the performance measurement of Xiangshan processor. Facing the problems of high cost and slow speed of current RTL performance measurement methods, this thesis studies how to achieve fast processor performance measurement with low-cost platforms. A cross-platform RISC-V checkpoint, RVGCpt, is designed in this thesis. Combined with the sampling method, RVGCpt reduces the performance measurement time with RTL software emulation from several years to several days. Facing the problem of long warm-up time in sampled simulation, this thesis implements a cache functional warm-up method and an adaptive hybrid warm-up scheme, HyWarm, for RTL emulation, which reduces the cache warm-up time by 85.7%. The agile performance measurement framework finishes the SPECCPU 2006 score estimation of Xiangshan processor on a 128-core server in only about 66 hours. The performance error between the estimated score by this method and the score evaluated with the real chip is about 11%.

2) Proposing a performance analysis framework to fastly locate performance differences and defects, which is applied in building an accurate microarchitecture simulator for Xiangshan processor. During the iteration of the microarchitecture design of open-source processors, an important obstacle is the lack of calibrated microarchitecture simulators for existing open-source high-performance processors. Performance counters are crucial for calibrating microarchitecture simulators. However, existing top-down

analysis frameworks and bottom-up counters are decoupled from each other and cannot quickly locate the components and instruction sequences that cause performance differences. This thesis builds a microarchitecture simulator XS-GEM5 for Xiangshan processor and proposes the Elaborated Top-down analysis framework and parallel memory miss cluster analysis method. When performance differences are found, the elaborated top-down analysis framework and parallel memory miss cluster analysis method fastly locate the source of performance differences.

3) Proposing an analytical model and performance improving method for the dynamic dataflow instruction scheduling architecture. Scaling up core size is an important way to improve processor performance. Due to the superlinear growth of the area and delay of the issue window and physical register file with respect to the instruction window size, the issue window and physical register file are one of the main bottlenecks that limit the size of the processor core. To achieve a scalable out-of-order execution architecture, this thesis uses a dynamic dataflow instruction scheduling architecture to avoid the issue window and physical register file. This thesis proposes an analytical model based on instruction wakeup signals to locate the bottlenecks of the existing dynamic dataflow instruction scheduling architecture. With the guidance of the analytical model, this thesis achieves about 94% of the performance of traditional superscalar architecture with lower power consumption and area overhead.

Key Words: Agile Chip Design, Open-Source Processor, Workload Sampling, Performance Modeling, Instruction Scheduling

目 录

图目录

表目录

符号列表

符号和缩略词说明

AGU	地址生成单元（Address Generation Unit）
ALU	算术逻辑单元（Arithmetic Logic Unit）
AMAT	平均访存延迟（Average Memory Access Time）
API	应用程序接口（Application Programming Interface）
ARF	体系结构寄存器堆（Architectural Register File）
ASIC	专用集成电路（Application-Specific Integrated Circuit）
BBV	基本块向量（Basicblock vector）
BTB	分支目标缓存（Branch Target Buffer）
CAM	内容可寻址存储器（Content Addressable Memory）
CISC	复杂指令集计算机（Complex Instruction Set Computer）
CPI	每条指令周期数（Cycle Per Instruction）
CPS	每秒周期数（Cycle Per Second）
CPU	中央处理器（Central Processing Unit）
C-AMAT	并行平均访存延迟（Concurrent Average Memory Access Time）
DQ	数据流队列（Dataflow Queue）
DRAM	动态随机存储器（Dynamic Random Access Memory）
DSP	数字信号处理器（Digital Signal Processor）
EDGE	显式数据流执行（Explicit Dataflow Graph Execution）
FMA	乘加指令（Fused Multiply-Add）
FPGA	现场可编程门阵列（Field-Programmable Gate Array）
FTB	取指目标缓存（Fetch Target Buffer）
HDL	硬件描述语言（Hardware Description Language）
ILP	指令级并行（Instruction-Level Parallelism）
IP	硬件知识产权模块（Hardware Intellectual Property Core）
IPC	每周期指令数（Instruction Per Cycle）
IPS	每秒指令数（Instruction Per Second）
IR	中间表示（Intermediate Representation）
KCPS	每秒千周期数（Kilo-Cycle Per Second）
KIPS	每秒千条指令数（Kilo-Instruction Per Second）
LJF	长作业优先（Longest Job First）

LRU	最近最少使用（Least Recently Used）
LSQ	加载/存储队列（Load/Store Queue）
LSDV	LRU 堆栈距离向量（LRU Stack Distance Vector）
LUT	查找表（Look-Up Table）
MCU	微控制器（Microcontroller Unit）
MLP	访存并行度（Memory-Level Parallelism）
MSHR	缓存缺失状态寄存器（Miss Status Holding Register）
OoO	乱序执行（Out-of-Order Execution）
PC	程序计数器（Program Counter）
PLRU	伪最近最少使用（Pseudo Least Recently Used）
PRF	物理寄存器堆（Physical Register File）
PTW	页表遍历（Page Table Walk）
RISC	精简指令集计算机（Reduced Instruction Set Computer）
ROB	重排序缓冲区（Reorder Buffer）
RTL	寄存器传输级（Register-Transfer Level）
SoC	系统级芯片（System on a Chip）
SRAM	静态随机存储器（Static Random Access Memory）
SSR	序列化后继表示（Serialized Successor Representation）
RAM	随机存储器（Random Access Memory）
TLB	页表缓存（Translated Lookaside Buffer）

第 1 章　绪论

1.1　研究背景及意义

现代的应用和数据对算力的需求越来越高，处理器芯片是信息时代提供算力的核心引擎。然而在先进制造工艺下，一款通用处理器经常需要耗费上亿的研发和流片成本。由于处理器芯片研发的投入高、风险高，业界只有少数公司能够从事处理器芯片的研发和持续迭代。有处理器芯片定制需求的公司希望通过各种办法来降低芯片开发的门槛，而开源芯片是降低门槛的一种途径。

随着 RISC-V 指令集的诞生，开源芯片项目和芯片敏捷开发方法得到了蓬勃发展。目前的开源芯片社区已经有了一些富有影响力的项目，例如加州大学伯克利分校开源的 Boom 和 Rocket 处理器 [1, 2]，中国科学院开源的香山处理器 [3]。这些开源项目既可以作为商业公司的基础代码降低公司的研发投入、加速产品研发，也可以作为学术界的基础研究，使得学术界有更接近业界的研究平台。

在众多开源处理器中，香山处理器作为性能最强的开源处理器，虽然已经能满足部分业务的需求，但是尚无法满足服务器芯片、移动设备芯片等重要场景。例如，业界常用于服务器的处理器 IP ARM N2 的 SPEC CPU2006 评估分值超过 15.0/GHz，而香山处理器第二代“南湖”架构的预估分值约为 10.42/GHz。因此，开源处理器追赶商业处理器仍然任重而道远。

在商业处理器的演进过程中，其微架构有两大发展趋势：1）处理器核规模变大；2）微架构设计变强。例如，从 Ivy Bridge 到 Sunny Cove，英特尔的处理器的指令窗口（重排序缓冲区）从 168 项扩大到了 352 项（表 1-1）。而苹果的 M1 处理器更是拥有约 630 项的指令窗口。在处理器的规模扩大的同时，为了获得相应的性能提升，还需要微架构设计与之匹配。以 ARM N2 处理器为例，它在指令供应方面从每周期供应最多 8 条指令提升到了 16 条，在数据供应方面增加了时间预取器（Temporal Prefetcher） [4]。

目前，无论从处理器核规模还是从微架构设计的角度来看，开源处理器（例如 Boom [2] 和香山 [3]）与商业处理器仍然存在较大的性能差距。以目前性能最强的开源处理器（香山处理器）为例，香山处理器的规模与 Skylake 相当，但是香山处理器的 SPECCPU 同频性能结果落后于 Skylake [3]，这说明香山处理器的微架构设计落后于 Skylake。在微架构设计落后的同时，开源处理器核的规模与工业界领先水平也存在差距（香山处理器 256 项 VS. 苹果 M1 630 项指令窗口）。

开源社区希望将开源处理器建设成像 Linux 这样在学术界和工业界都被广泛应用的创新开源平台。为了实现这样的愿景，开源处理器需要达到与商业处理器相抗衡的性能。而在追赶性能的过程中，能满足持续的微架构演进的平台和架构改进的方法论至关重要。

表 1-1 Intel 历代处理器微架构的规模扩展

Table 1-1 Core scaling of Intel microarchitectures.

	Ivy Bridge	Haswell	Skylake	Sunny Cove
重排序缓冲区容量	168	192	224	352
发射窗口容量	54	60	97	160
整数物理寄存器堆容量	160	168	180	280
浮点物理寄存器堆容量	144	168	168	224

1.2 论文研究面临的挑战

要提升微架构设计水平、提升处理器核的规模，现有的开源处理器面临着巨大的挑战。1）更先进的微架构和更大的处理器核规模意味着更大规模、更复杂的电路，而现有的处理器芯片性能评估方法面对大规模电路存在速度慢甚至失效的问题。2）处理器微架构的设计空间极大，寻找有效设计方案的过程需要不断试错，而现有的基于新型硬件描述语言的探索方法试错成本较高。3）虽然提升处理器核规模能带来性能收益，但是现有的微架构扩大规模后也会面临时钟频率下降的风险 [5, 6]。

针对开源处理器现状，处理器**性能评估方法**面临以下问题和挑战：

电路规模大，软件仿真时间过长。开源处理器映射到现场可编程门阵列（Field-Programmable Gate Array，FPGA）所需的资源量大。当映射到 FPGA 时，Boom 处理器 [2] 的最大规模配置只需要占用约 24 万查找表（Lookup Table，LUT），而香山处理器的单个核心需要占用约为 110 万 LUT。香山处理器的规模使得它只能被映射到顶级配置的商业 FPGA （例如 Xilinx VU19P [7]），无法被映射到常见的公有云 FPGA （Xilinx VU9P [7]）。这导致开源社区常用的基于公有云 FPGA 的仿真方法 [8] 无法用于香山处理器的性能评估。当 FPGA 仿真不可用时，RTL 软件仿真是一种替代方案。然而在大规模设计下，RTL 软件仿真器（例如 Verilator [9]）的仿真速度只有每秒 500-3000 周期。以这样的速度，需要数年时间才能完成 SPECCPU® 2006 的性能评测。

采样仿真预热时间长。当对程序进行片段采样来缩短仿真时间时，为了避免缓存冷启动导致性能估算误差，在每个采样片段开始前，都需要执行额外的、不参与性能统计的预热指令片段。预热过程在采样仿真的时间占比从 20% 到 80% 不等。

负载时长分布不均。因为仿真负载的时长差异较大，仿真完成时间往往由少数负载决定。超过 80% 的负载在仿真的前 1/4 的时间完成，这导致仿真的后 3/4 的时间只有少数负载在运行，计算资源的利用率较低。

针对开源处理器和开源模拟器现状，处理器微架构的**设计空间探索**面临以下几点挑战：

硬件描述语言试错成本高，微架构模拟器未对齐。当面对需要大规模架构

改动时，相比于架构模拟器，用硬件描述语言直接进行尝试的试错成本更高、面临的风险更大。在香山处理器的架构迭代中，即使使用了 Chisel [10] 进行开发，局部的架构重构仍需投入大量的人力和时间。例如重构香山处理器的乱序执行流水线删改了超过 8500 行代码；在编码工作完成后，还需要进行长达数月的功能和性能问题修复。此外，Chisel 相较于微架构模拟器，仍然面临调试反馈周期长、性能测评速度慢的问题。虽然微架构模拟器有更高的架构探索效率，但是主流的开源微架构模拟器 GEM5 的建模对象是 ALPHA 21264 [11, 12]，与香山处理器的差距较大。只有将架构模拟器与 RTL 相对齐，才能让模拟器产生可靠的结果 [13–15]，但是要将模拟器对齐到实际的处理器，需要投入大量的人力和时间 [13]。

性能计数器之间相互脱节。性能计数器可以帮助提高微架构模拟器对齐的效率 [13, 14]。传统的微架构性能计数器（例如缓存缺失率）并不总是与性能损失相关 [16, 17]。自顶向下性能计数器虽然能从宏观上定位到造成性能损失的性能瓶颈，但是它未能与微架构计数器建立联系。这导致从自顶向下性能计数器发现的高层性能瓶颈与微架构缺陷之间仍存在差距。

性能缺陷难以定位。性能计数器只能定位到存在性能缺陷的部件，而要找到具体的缺陷原因一般需要定位到发生异常的现场，例如导致出错的指令序列。现有的缺陷定位工作一般专注于功能缺陷 [3, 18]，很少考虑性能缺陷。并且性能缺陷一般不会导致程序崩溃，比功能缺陷更难发现。

针对开源处理器现状，要**扩展处理器规模**面临以下几点挑战：

传统超标量架构难以扩展，可扩展架构性能孱弱。扩大重排序缓冲区容量时，一般也要同时扩大发射窗口和物理寄存器堆的容量（表 1-1），因为发射窗口和物理寄存器堆也是挖掘指令级并行（Instruction-level parallelism，ILP）的重要组件。但是，扩大发射窗口和物理寄存器堆会对处理器的功耗和时序带来巨大挑战 [5, 6]。并且发射窗口和物理寄存器堆已经对当前的开源处理器的物理设计带来了压力 [2, 19]。为了克服上述缺陷，学术界尝试了许多设计，例如基于依赖的唤醒方式 [20]，基于数据流架构的调度方法 [21–24]，以及基于顺序核的局部乱序执行 [25, 26]。这些方案要么非常依赖于编译器的定制和优化，要么性能显著落后于传统超标量架构。

可扩展架构缺乏性能分析方法。基于数据流的指令调度架构和局部乱序执行架构是两类典型的可扩展架构，他们的依赖检测和指令调度方式都与传统超标量处理器存在较大差异。因此针对超标量处理器的传统性能分析模型对上述可扩展架构不适用。

1.3　论文的主要研究内容和贡献

1）在处理器芯片性能评估方法方面，本课题实现了敏捷性能评估框架来减少性能测算时间。针对香山处理器**电路规模大、软件仿真时间长**的问题，本文提出了 RISC-V 通用检查点 RVGCpt，结合 SimPoint 算法 [27]，通过采样方法大幅

缩短了香山处理器的性能测算时间。针对采样仿真**预热时间长、负载时长分布不均**等问题，本文提出了 HyWarm 混合预热方法，结合了 RTL 缓存功能预热和自动化预热需求分析工具，将预热过程的仿真指令数减少了 85.7%。经估算，需要数年时间才能完成香山处理器运行 SPECCPU 2006 的仿真，而在敏捷性能评估框架中，只需要约 54 小时就能完成 SPECCPU 2006 的性能测算。

2）为了加速设计空间探索，本文基于 GEM5 [12] 为香山开源处理器 [19] 设计了微架构模拟器 XS-GEM5。为了提升对齐微架构模拟器与香山处理器的速度，解决自顶向下性能计数器与微架构**性能计数器之间相互脱节**的问题，本文提出了展开式自顶向下分析框架。展开式自顶向下分析框架改进了自顶向下分析框架 [16, 28, 29]，为自顶向下性能计数器与微架构性能计数器建立了量化关系。为了**加速定位性能缺陷**，本文提出了并行访存缺失簇（Parallel miss cluster，PMC）分析器来自动化定位访存并行度异常的代码片段。

3）面对处理器核的规模难以扩大的问题，为了克服传统超标量架构的发射窗口和物理寄存器堆可扩展性较差的挑战，本文对可扩展性较好的动态数据流指令调度架构 Forwardflow 进行分析和改进。针对 Forwardflow 唤醒信息处理与传递机制，本文提出了一种**性能分析模型**帮助定位 Fowardflow 架构的性能瓶颈。在分析模型的指导下，本文设计了 Omegaflow 架构，解决了 Forwardflow **性能孱弱**的问题，达到了理想乱序执行 94% 的性能。Omegaflow 表明动态数据流指令调度架构是传统乱序执行的一种可行的替代方案。

1.4 论文的组织结构

图 1-1展示了本文的研究内容组织结构，其对应的内容安排如下：第一章（本章）从总体上介绍了本文的研究背景、各子课题的研究内容和意义。第二章回顾了相关领域的研究历程，包括高性能处理器性能测算与采样、高性能处理器的模拟器对齐高性能处理器的性能分析方法和超标量架构的局限性与改进。第三章

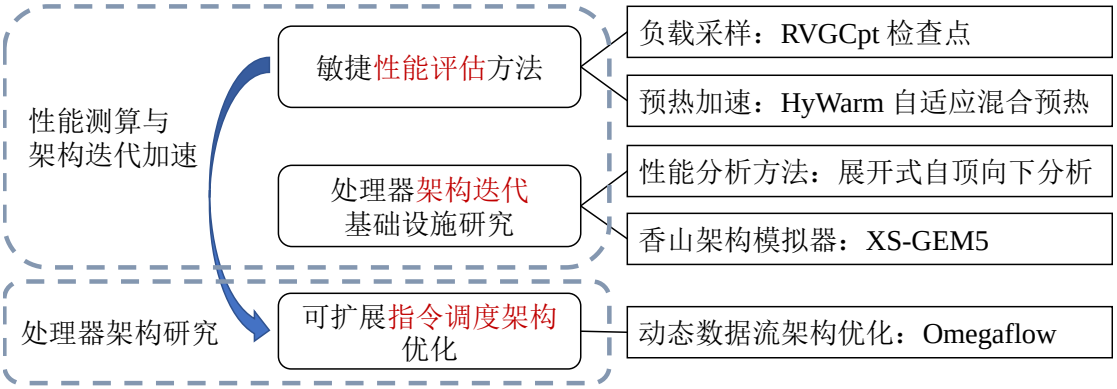


图 1-1 本文结构示意图

Figure 1-1 Structure of this thesis

介绍了在 RTL 上进行性能测算的挑战，以及如何通过采样和预热加速的方法来加速性能测算。第四章介绍了处理器架构迭代的基础设施搭建：改进自顶向下性能计数器架构用于加速架构模拟器与 RTL 的对齐。第五章介绍了如何基于动态数据流架构的思想改进调度执行引擎，以及在此基础上进行的性能分析和性能优化。第六章对本文的研究工作进行了总结。

第 2 章　处理器架构迭代与规模扩展相关研究综述

本章第 2.1节介绍开源处理器的发展现状。第 2.2节介绍处理器的性能评估方法和采样仿真相关研究。第 2.3节介绍处理器的微架构模拟器对齐相关研究。第 2.4节介绍处理器的性能计数器和性能分析方法相关研究。第 2.5节介绍超标量处理器的可扩展性问题和相关研究，包括学术界提出的替代架构研究和开源处理器的应对方案。

2.1　开源处理器

RISC-V 开源处理器生态。在 RISC-V 生态发展的早期，开源 IP 以微控制器（microcontroller unit, MCU）为主。自从加州大学伯克利分校开源了 Rocket Chip SoC 生成器和 Boom 处理器后，出现了对标工业级 IP 的一系列开源处理器。图 2-1展示了 RISC-V 处理器 IP 和 ARM 处理器 IP 的性能分布和对应关系。其中，中国科学院开源的香山处理器的第 1、2、3 代分别以 ARM Cortex A72、Cortex A76、Neoverse N2 为性能标杆进行设计 [30]。在此背景下，本文的三部分研究都致力于香山处理器的架构迭代。

Chisel [10] 是一种基于 Scala 语言的硬件描述语言。众多开源 RISC-V 项目都基于 Chisel 进行开发，例如 Rocket Chip SoC，Boom 处理器和香山处理器。Chisel 代码可以编译为 FIRRTL [31] 中间表示，进而生成 RTL 代码（Verilog）。它既可以生成可综合的 RTL 代码从而利用 FPGA 进行仿真，又可以利用 VCS [32] 或 Verilator [9] 等软件仿真工具进行仿真。

Chisel 相比于传统的 Verilog 语言有两方面的优势：1）Chisel 提供了基础的电路库使得底层电路的编写效率更高；2）编写 Chisel 代码时，可以利用 Scala 语言的特性实现高层次的抽象。这使得基于 Chisel 编写的代码比 Verilog 代码更加易于复用。例如 Boom 处理器复用了 Rocket Chip SoC 的完整核外环境，而香山处理器第一代复用了 Rocket Chip SoC 的总线和参数协商机制 [33]。

Rocket Chip [1] 是一款开源的 SoC 生成器，由加州大学伯克利分校研发。它能够从一个高级语言描述的硬件代码生成多种参数的设计。它能生成可综合的 RTL，并且这些 RTL 已经多次流片。强大参数化能力使它非常灵活，易于为特定应用进行定制。通过更改参数配置，用户可以生成从嵌入式处理器器到多核服务器芯片等各种不同规模的 SoC。

Boom 处理器 [2] 是首款开源的乱序执行处理器。在架构设计方面，Boom 的流水线前端使用了多分支预测器架构，流水线后端使用了统一的寄存器堆和统一的发射窗口。Boom 的流水线可配置性强，例如可以配置为每周期最多发射 2 到 4 条指令，取指、译码和提交宽度可以配置为 1、2、4。Boom 作为开源乱序执行处理器的先驱，在芯片敏捷开发实践、架构设计和物理设计等方面为社区都

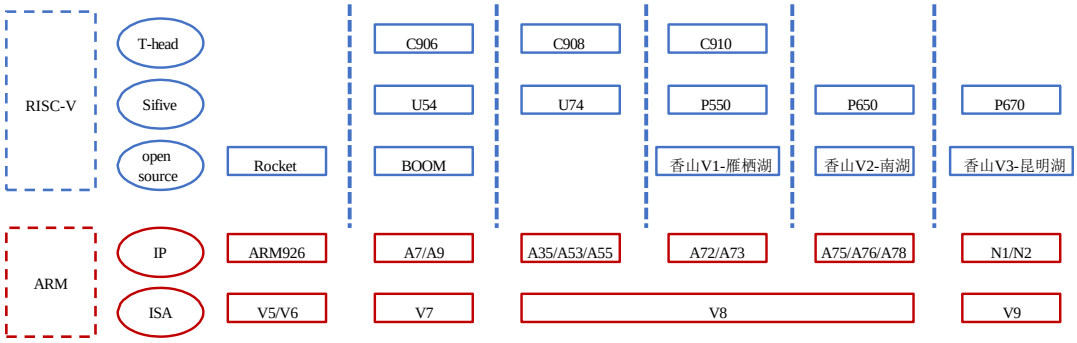


图 2-1 RISC-V 处理器 IP 与 ARM 的处理器 IP 性能分布（不包括微控制器），改编自文献 [30]

Figure 2-1 Performance distribution of RISC-V processor IP and ARM processor IP (excluding MCU)

提供了宝贵的经验。

香山处理器是首个达到工业级处理器性能的开源处理器。南湖架构是香山处理器的第二代，它在核心规模和微架构设计上都位于开源处理器的领先水平。南湖架构的译码和提交宽度为 6，后端的调度执行引擎拥有 2 条访存流水线，4 条整数流水线和 3 条浮点流水线。南湖架构是首个开源的解耦前端流水线设计 [34]，包含了 TAGE-SC 分支预测器 [35] 和 ITTAGE 间接跳转预测器 [36]。此外，南湖架构还拥有包含 Best-offset 预取器 [37] 和 Spatial Memory Streaming 预取器 [38] 的复合预取器。经本文第 3章提出的敏捷性能评估方法测算，南湖架构 SPECCPU® 2006 整数分值预估为 9.55 分/GHz，SPECCPU® 2006 浮点分值预估为 11.09 分/GHz。

2.2 高性能处理器性能测算加速

2.2.1 高性能处理器性能测算方法

表2-1列出了芯片设计过程中进行仿真的 4 种常用方法。其中 RTL 软件仿真器，传统 FPGA 和硬件仿真加速器是业界广泛采用的方法。在开源芯片潮流中，RTL 软件仿真器和云 FPGA 因为低成本、易于获取的特点被社区广泛采用。

FPGA 因为其易于重编程和丰富的数字信号处理器（Digital Signal Processor，DSP）资源，被广泛用于专用芯片的原型搭建和部署 [39–43]。而拥有大量逻辑资源和片上存储资源的 FPGA 芯片也被用于 CPU 的原型搭建 [8, 19]。FPGA 的仿真速度快、片上资源种类丰富、价格适中，适合用于中小型片上系统（System on a Chip，SoC）的搭建。为了强化 FPGA 的易用性，FireSim 框架使得 FPGA 评估可以在亚马逊云服务（Amazon Web Services，AWS）的 F1 实例上运行 [8]。通过云服务器，用户可以在短期内租用大量的 FPGA 来实现并行评估多个处理器

核，运行不同的配置或程序，极大地加速了性能评估流程。FASED 通过在 FPGA 上模拟动态随机存储器（Dynamic Random Access Memory，DRAM）的性能模型解决了 FPGA 上 CPU 与 DRAM 性能不匹配的问题 [44]。

但是 FPGA 存在两方面的缺点。第一是可调试性差：当电路系统出错时，只能通过集成逻辑分析器（Integrated Logic Analyzer，ILA [45]）来捕获少数信号。第二是可扩展性差：FPGA 的逻辑和存储资源有限，无法容纳大规模 SoC [3]。即使有一系列工作尝试改进 FPGA 的可扩展性、易用性和可调试性 [44, 46–50]，但是对大规模 SoC 而言，FPGA 仍需进一步改进 [51–53]。

硬件仿真加速器是专门设计的用于大型 SoC 的仿真系统，具有极高的可扩展性和极佳的可调试性。商业级的硬件仿真加速器（例如 Cadence Palladium [54] , Mentor Veloce [55] 和 Synopsys Zebu [56]）能够兼顾不错的仿真速度和极佳的可调试性：以 1-5MHz 的速度运行，以接近软件仿真的方式输出调试信息。但是其昂贵的价格决定了它们无法在社区被广泛推广。

RTL 软件仿真器 [9, 32, 57] 在仿真过程中可以输出丰富的调试信息，并且可以在普通的 CPU 服务器上运行，非常易于获取和使用、适合社区推广。但是当仿真大型 SoC 时，它们只能达到数 KHz 的仿真速度。并且当开启调试信息输出后，它们的仿真速度会进一步下降 [3]。实验表明，Verilator [9] 仿真大规模处理器核运行 SPECCPU 应用的速度为 0.5KIPS-3KIPS，以这样的速度运行完 SPECCPU® 2017 中最长的子项需要超过 7 年时间。这表明无法用 RTL 软件仿真进行完整的工业级基准测试。

开源 RTL 仿真器 Verilator [9] 和商业仿真器 VCS [32] 是被开源处理器社区广泛采用的两大仿真平台。Verilator 通过块重排序、网表级优化获得了比商业仿真器更快的仿真速度。Verilator 还能将仿真任务划分到多个 CPU 核上并行执行来提升仿真速度。ESSENT 仿真器 [57] 使用了重用不变信号等技术，获得了相较于 Verilator 1.5-11.5 倍的性能提升。然而即使获得了近一个数量级的提升仍然无法让 RTL 仿真速度胜任香山处理器的性能评估（预计耗时数月到一年）。

表 2-1 常用的 RTL 性能评估方法对比

Table 2-1 Comparison of common RTL performance evaluation methods

	仿真速度	典型价格	可调试性	典型可容纳设计
RTL 软件仿真器	~1kHz	~￥5-10 万	★★★	商业级 SoC
公有云 FPGA	~100MHz	~￥240-3600/天	★	Boom 处理器
私有 FPGA	~100MHz	~￥40 万	★	香山处理器
硬件仿真加速器	~1MHz	>￥1000 万	★★★	商业级 SoC

2.2.2 仿真采样方法

在 CPU 微架构的研究中，时钟精确的架构模拟器和 RTL 仿真器一样面临着速度慢的问题，为了解决该问题，学术界开展了深入的研究。最经典的两种采样方法是 SimPoint [27] 和 SMARTS [58]，它们开创了两种性能采样方法的流派：带权重采样和均匀采样。两种方法都可以大幅减少仿真的指令数（减少到 0.1% 以下）。

如图 2-2(a) 所示，**带权重采样**将程序划分为片段，通过功能模拟器 profiling 得到各个片段的程序特征，然后对这些特征应用机器学习算法进行分类或者聚类。对同一个类别的片段，从该类（class）或者簇（cluster）种选取最具代表性的片段来代表整个类或簇的性能。而该片段对应的权重则是该类/簇的片段的数量在整个程序中的占比。

如图 2-2(b) 所示，**均匀采样**在程序中选取大量的、均匀的采样点来估算整体性能（如图 2-2(b) 所示）。当采样点均匀且足够密集时，拥有特定特性的程序片段占比越大被采养的次数就更多，从而反映出程序的整体特性。均匀采样相比于带权重采样的优势是不需要提前 profiling，可以用于无法复现的在线应用的采样。但是均匀采样为了采样准确性需要的采样点较多，因此在仿真速度上比带权重采样慢。因为它经常与快进（fast-forwarding）技术一起使用，这导致它无法像带权重采样一样通过并行仿真多个检查点来缩短时间。因此该流派主要被诟病的点在于功能模拟的速度仍然不够快。

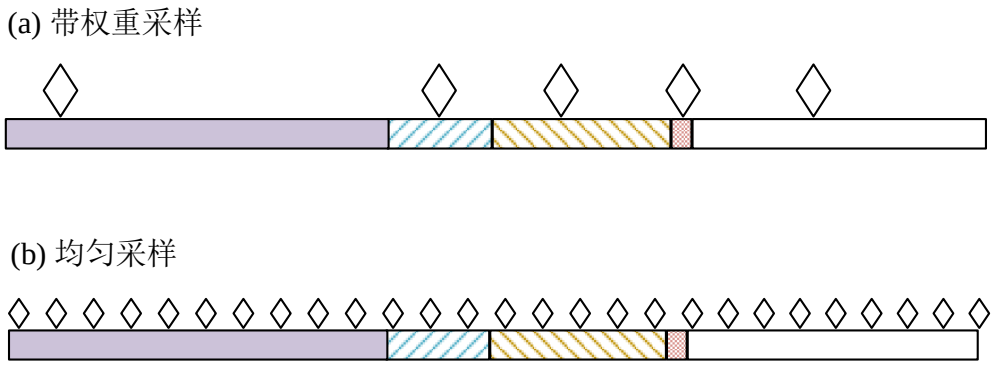


图 2-2 两种主流的程序采样方法：(a) 带权重采样方法通过离线分析获得程序不同片段的特征，根据片段长度对单个采样点赋予不同权重。(b) 均匀采样无需前提获取程序的阶段特征，通过选取密集而均匀的采样点来覆盖程序的不同阶段。

Figure 2-2 Two mainstream program sampling methods: (a) Weighted sampling method obtains the characteristics of different program periods through offline analysis and assigns different weights to individual sampling points based on the length of each period. (b) Uniform sampling method does not require offline profiling of the program's period characteristics. It covers different periods of the program by selecting dense and uniform sampling points.

2.2.2.1　带权重采样

特征是指程序片段的特征，它可能是基本块的执行次数（基本块向量）、分支预测难度（分支熵）、内存访问的局部性等。在带权重采样中，一般把指令数为 N 的程序切分为 m 个长度为 n 的片段 $p_i, i \in [0, m)$，为每一个程序片段采集特征 f_i。

基本块向量。SimPoint [27] 使用的特征是指令片段的基本块向量（Basic block vector, BBV），基本块向量的第 j 维对应于程序中的一个基本块 b_j。在片段 p_i 中，假设程序进入基本块 b_j 的次数为 c_{ij}，基本块 b_j 的指令数量为 a_j，则基本块向量的第 j 维等于进入该基本块的次数乘以基本块的指令数量：$f_i[j] = c_{ij} \times a_j$。向量的数值与指令数量相关是因为通常基本块的指令数量越多，该基本块执行时间越长，对整体性能的影响越大。

两个片段的基本块向量的某一维相似的物理含义可能是两个片段 1) 进入同一个函数的次数相似；2) 在分支中选择同一个方向的次数相似 3) 执行同一个循环体的次数相似。因此程序行为相似的两个片段在基本块向量的相似度也会较高。

基本块向量最大的优点是，它独立于微架构设计，因为基本块进出次数只与程序本身和编译器有关，与微架构设计无关。这使得基本块向量的收集可以在运行极快的功能模拟器上完成。但是它也存在局限性：同一个代码段的访存局部性特征、分支预测难度特征可能发生变化，而此时拥有不同特征的片段的基本块向量可能是相似的，但是性能可能有较大差异。例如快速傅里叶变化中的访存步长和重用距离会随着计算轮次而变化 [59]。为了解决上述问题，有工作提出利用 LRU stack distance 和分支熵来描述程序片段的访存特征和控制流特征 [60–63]。

LRU stack distance（LSD）定义为对同一地址的某次访问与上一个访问之间访问的唯一数据元素数量。LRU stack distance vector（LSDV）统计了片段 p_i 中每一个重用距离出现的次数 [59–61]。LSDV 反映了程序的局部性特征，局部性越好的程序的 LSDV 中的小数据越高频。如图 2-3所示，图 2-3(a) 的访存局部性比图 2-3(b) 更差，在 LSDV 中的表现是图 2-3(a) 中的小数据频次更高，直方图的分布更靠近 0。文献 [59] 表明将 LSDV 与 BBV 结合可以解决程序的局部性变化带来的采样误差。

部分工作把 LSD 与重用距离（Reuse distance）视为等同的概念 [61, 64]。但是也有部分工作将重用距离与 LSD 区别定义：在文献 [65] 中，重用距离被定义为两次访问之间的访问次数 [61]，而 LSD 被定义为两次访问之间的唯一数据元素数量。如无特别说明，本文将 LSD 与重用距离视为等同的概念。

分支熵定义为给定分支指令在给定上下文中分支方向的熵 [63]。其中上下文是分支预测器所使用的输入特征，一般为分支方向历史。假设在给定上下文中，某分支指令发生跳转的概率为 Pr，则分支熵为

$$E(Pr) = -Pr \times \log_2(Pr) - (1 - Pr) \times log_2(1 - Pr),$$

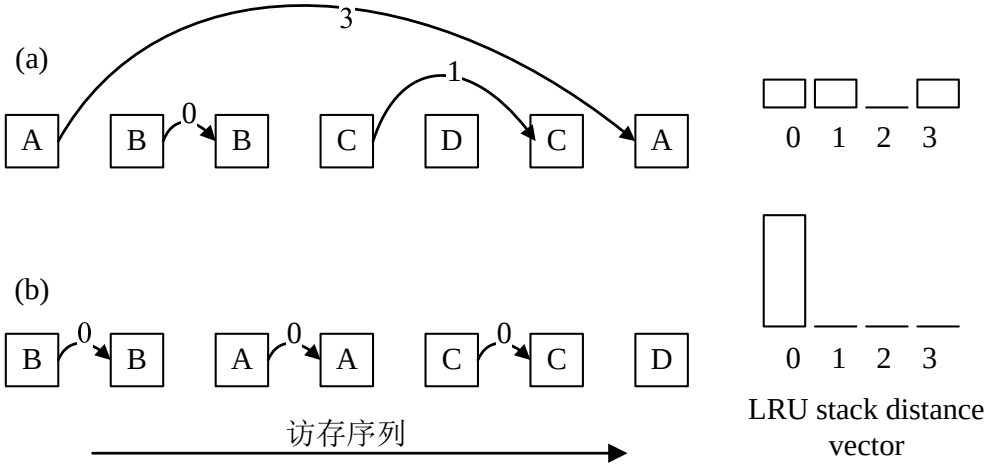


图 2-3 LRU stack distance vector（LSDV）示意图

Figure 2-3 Diagram of LRU Stack Distance Vector (LSDV)

分支熵描述了在给定上下文中分支跳转方向的多样性，越多样则分支越难以预测。

分类和聚类过程根据**特征**将程序片段划分为不同的类或簇。在机器学习领域，分类是根据带标注的数据集（training set）习得某种规律对新数据进行分类，而聚类是对无标注的数据集（unlabeled data set）根据特定标准将数据集划分为不同的簇 [66]。在程序的性能采样领域，无标注的聚类方法最为常用，例如 SimPoint [27] 使用了 k-means 聚类算法 [67]。采用有标注的分类算法较为困难，一方面，要获取带有性能标记的数据集较为耗时，另一方面，要获得足够的准确率，所需的样本数量较多。

2.2.2.2 功能预热与预热加速

无论是带权重还是不带权重的采样，都会面临**冷启动**问题：当程序正常从头开始运行时，近期访问的数据块会存留在缓存中。但是当从检查点恢复时，缓存处于全空的状态，导致采样段得到超额的缓存缺失，测得性能低于实际性能。为了让缓存的表现更接近从头开始运行的状态，一般需要用模拟器先运行一段不参与性能统计的片段进行预热。

在最初的工作中，预热过程与性能采样使用同一个 CPU 模型，这种方式被称为**全细节预热**（detailed warmup） [27]。但是后来的工作发现，预热过程可能占整个仿真过程的很大的比例 [58, 62, 68]。为了加速预热过程，文献 [58] 提出了**功能预热**：在功能 CPU 模型上模拟缓存等部件的行为，再将缓存状态迁移到时钟精确的 CPU 模型上。文献 [68] 通过对应用的预热需求进行预测，以缩短预热需求较小的应用的预热时间。文献 [69] 将虚拟化技术应用到功能预热，这使得功能模拟器能以接近本地执行的速度进行功能模拟跳过部分不需要携带功能缓存

的代码段，大大加速了功能预热。基于统计的缓存预热（statiscal cache warmup）利用访存重用信息预测访存命中情况，来完全消除对全细节预热或功能预热的需求，进一步缩短仿真时间 [65, 70, 71]。

2.2.2.3　多线程采样

除了上述针对单线程应用采样以外，也有工作针对多线程应用采样展开研究。多线程采样的最大挑战是在功能模拟器上程序执行的速度与架构模拟器以及真实处理器上的执行速度不一致，导致多个线程到达同步点的时机与真实处理器拥有不同的状态。这会导致共享缓存中的数据状态和等待线程的状态与架构模拟器的真实状态不同，进而无法采样到真实的多线程状态 [72]。为了解决多线程采样的问题，文献 [72] 提出了基于时间的采样，让不同线程可以以时间为单位进行同步，而不是以指令数为单位。为了在功能模拟器上估算时间（周期数），需要估算程序的执行速度（IPC），文献 [72] 对此提出了启发式的 IPC 推测方案。此后，文献 [59] 将 SimPoint 采样算法应用到了多线程采样中，得到了更高的加速。文献 [73] 提出了以任务为单位在功能模拟器上进行线程的进度估计，让不同的处理器核上的线程维持相同的进度，避免了基于时间的采样时速度较慢的问题。

2.2.2.4　模拟器的采样仿真

应用性能采样方法进行性能评估需要微架构模拟器或者 RTL 仿真器提供相应的功能。研究采样功能的工作相对较少 [12, 69, 74]，而对 RTL 仿真的采样功能的支持工作则更为缺乏。为了帮助理解采样仿真，本小节介绍 GEM5 对体系结构检查点和缓存预热的支持。

全系统模拟 VS. 系统调用模拟。在讨论性能采样的功能支持之前，必须先了解模拟器的仿真模型。一般地，模拟器有两种模拟方式：模拟系统调用和模拟全系统。如图 2-4所示，在模拟系统调用（System call emulation，SE）中，用户只需要提供应用的可执行文件，模拟器与应用交互的界面是指令集和系统调用。而在模拟全系统模式（full system emulation，FS）中，用户需要提供完整的系统镜像，模拟器（硬件系统）与软件交互的界面是指令集和输入输出（I/O）。在常见的模拟器中，GEM5 和 QEMU 都能支持这两种模式，其中 QEMU 将系统调用模拟称为用户模式（user mode）。

检查点是模拟器所模拟的系统某一时刻的状态。对系统调用模拟器而言，检查点包含的是应用程序的状态，包括进程控制块中的内容和进程的页表项。对全系统模拟器而言，检查点包含了硬件系统的所有状态，包括内存、体系结构寄存器、硬件设备的状态等。全系统模拟器的检查点不需要特地备份进程的状态，这是因为进程的状态一定在内存和体系结构寄存器中。

GEM5 的检查点支持。粗略地讲，可以认为 GEM5 的检查点文件是一个 key-value 格式的文件，键（key）对应类的信息，键值（value）是类的状态值。为了支持系统调用模式的检查点，GEM5 为每一个需要保存的状态的类都实现了

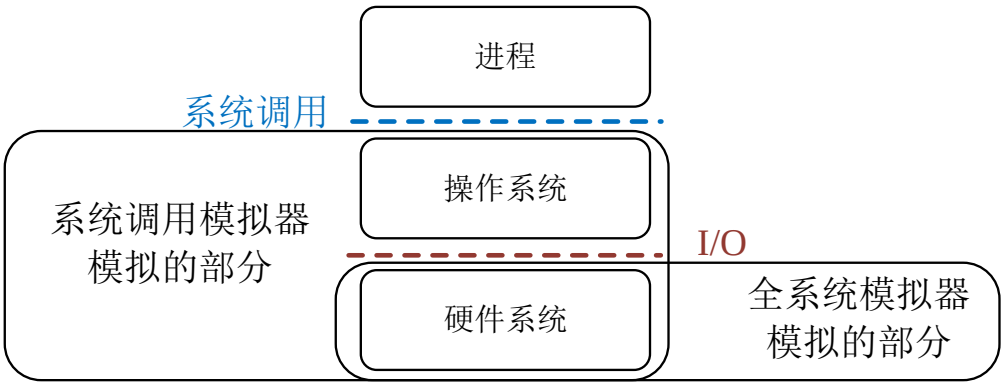


图 2-4 模拟系统调用和模拟全系统的差异

Figure 2-4 The difference between system call emulation and full system emulation

serialize 和 *unserialize* 方法。*serialize* 方法将类的主要状态保存到检查点文件中，为每一个域标注键和键值；*unserialize* 方法从检查点文件中读取该类的状态，将域的键值恢复到键所标注的类（图 2-5(a)）。

GEM5 的功能预热支持。为了支持功能预热，GEM5 支持从功能 CPU 核模型切换到时钟精确的 CPU 核模型，并且在切换时保持内存、缓存和硬件设备的状态不变。GEM5 为每个需要保留预热状态的类实现了 *takeOverFrom(old)* 函数，当调用 *x.takeOverFrom(y)* 时，新的模拟对象 *x* 会从 *y* 中获取状态。例如，执行 *newCpu.takeOverFrom(oldCpu)* 会让 *newCpu* 的指令缓存和数据缓存的指针都会指向 *oldCpu* 预热后的指令缓存和数据缓存（图 2-5(b)）。

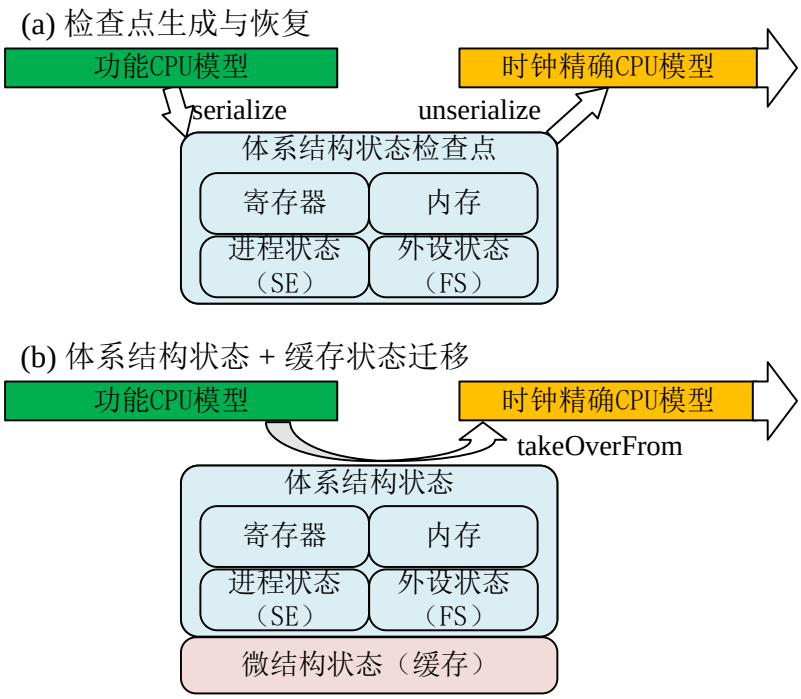


图 2-5 GEM5 对检查点和功能预热相关支持

Figure 2-5 Support of Checkpoints and Functional Warmup in GEM5

2.3　高性能处理器的模拟器对齐

曾经有一系列工作研究架构模拟器与真实处理器的对齐 [13, 14]。这些工作都采用**微基准测试**来从外部观测模拟器和真实处理器的性能差异，指导进行性能对齐。

文献 [14] 提出了一套帮助对齐的微基准测试方法论，将测试类型分为三部分控制转移测试集（C），执行引擎测试集（E）和访存测试集（M），并为三类测试集提供了一系列的微基准测试。作者们在 sim-alpha 模拟器上，将微基准测试观测到的指标对齐到 Compaq 公司的 Alpha 21264 处理器（DS-10L 工作站）。

文献 [14] 的数据表明 sim-alpha 与 DS-10L 工作站的性能误差在微基准测试上都在 10% 以内，平均误差仅为 2%。但是在 SPECCPU 2000 上，sim-alpha 的平均性能建模误差达到了 18%，最大误差高达 43%。这意味着，微基准测试的测试结果并不能很好地反映架构模拟器与真实处理器的性能差距。同样大的性能误差尺度在用 GEM5 建模 Cortex-15 的工作上也可以看到 [75]。

这说明，仅仅对齐指令延迟、访存延迟等基本参数对高性能处理器的模拟器对齐是不够的。这是因为工业级的基准测试会对处理器的各个部件施加压力。而这种压力除了与基本指令延迟相关外，还和各个子系统的带宽、推测准确率、推测错误恢复速度有关。

此外，文献 [13, 14, 75] 没有系统地分析剩余的性能误差是由于哪些具体的微

架构设计造成的，这受限于人力投入和商业处理器的架构设计的保密性 [76, 77]。虽然有工作分析过架构级别的不对齐对性能的影响 [15]，但是讨论的主要是单点的不一致，并没有提供系统性的帮助模拟器与真实处理器对齐的方法论。

2.4 性能计数器与性能分析方法

2.4.1 自顶向下分析方法

“每指令周期数堆栈”（CPI stack ）是一种表示性能的常用方式，它将性能分解为基准 CPI 和多个缺失事件的性能损失。CPI stack 可帮助深入了解应用程序在给定微处理器上的行为，因此被广泛使用。然而，在超标量乱序处理器的并行和乱序执行能力给计算 CPI 堆栈造成了独特的挑战：因为正常执行和缺失事件，例如缓存缺失、TLB 缺失和分支预测错误可能同时发生，这些缺失事件并不一定最终造成性能损失。

为了找出真正造成性能损失的缺失，文献 [16, 28] 提出了区间分析方法（interval analysis），一种根据缺失事件对流水线阻塞的贡献来计算 CPI stack 的方法。如图 2-6所示，区间分析方法将处理器的执行过程拆解为多个区间，区间的边界是缺失事件的发生点，他们让流水线彻底阻塞。基于此方法，可以准确地统计缺失事件对执行时间的贡献，得到更准确的自顶向下 CPI 堆栈 [28]。

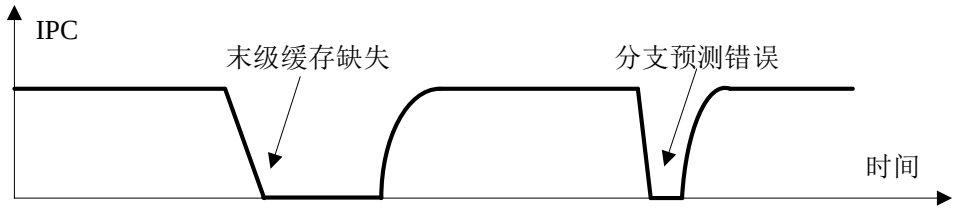


图 2-6 区间分析方法示例，改编自文献 [28]

Figure 2-6 Example of interval analysis

文献 [16, 28] 的统计方法虽然非常准确，但是需要引入一个用于记录全局分支指令状态的前端缺失事件表（Front-end Miss Event Table，FMT），在特定的情况下，需要对 FMT 的分支预测错误惩罚域的每一行进行更新，复杂度较高。为了避免全局结构，文献 [29] 提出了另一个折衷的统计分支预测错误惩罚的方法：用全部提交的指令数减去全部分发的指令数来估计分支预测错误惩罚。虽然这种统计方法会将分支预测错误路径上的后端阻塞时间归因到后端，但这部分时间占比较小，不会影响性能瓶颈分析。

2.4.2　性能瓶颈定位

基于经典性能计数器和自顶向下性能计数器，学术界提出了多种性能瓶颈定位方法 [17, 78]。文献 [78] 提出了一种基于自顶向下性能计数器的挖掘性能提升点的方法论。文献 [17] 在大量的经典性能计数器中寻找出对应用程序性能影响较大的计数器集合。影响性能的功能部件（例如浮点除法 [78]）一般与自顶向下性能计数器中的成分有明显的相关性，但文献 [17] 发现部分事件在自顶向下性能计数器中不存在，对性能仍有较大影响。在实践中，也不乏与自顶向下性能计数器缺乏联系的性能优化点。

本文认为，造成这种现象的原因是，现有的自顶向下性能分析框架过于强调成分之间的“可加性”，而缺乏对高性能处理器中各类事件的重叠、并行的描述。这与现代处理器处处强调并行、掩盖的设计准则不够匹配。例如，无论在数据中心应用中，还是在传统服务器应用中，推测错误和访存受限都是 CPI stack 中的重要组成部分 [79, 80]。访存受限时间除了受到访存层次结构的影响外，还受访存并行度影响 [81, 82]。推测错误的损失由多个成分共同决定，它们相互之间也有重叠和掩盖的关系。

2.4.3　访存并行度分析

自从访存并行度（Memory-Level Parallelism，MLP）被正式提出以后 [81]，它已经成为现代处理器最重要的性能指标之一。先后有大量的工作尝试优化 MLP，例如文献 [83] 通过 Runahead Execution 来提高 MLP，文献 [84] 通过增大指令窗口大小来提高 MLP，文献 [25] 在顺序执行的处理器上通过对访存指令进行有限的并行执行来提高 MLP。而在 MLP 性能计数器方面，现有方法可以通过监控缺失状态处理寄存器（Miss Status Holding Register，MSHR）来统计系统的整体 MLP [82]。但是即使发现了 MLP 异常，目前也缺乏自动化定位 MLP 瓶颈的手段。

综上所述，自顶向下性能分析是现存的性能分析框架中最擅长从宏观上定位性能瓶颈的方法，但是它未能涵盖高性能处理器的中广泛存在的并行、延迟掩盖，导致它无法直接地关联到很多自底向上性能计数器和微架构设计。这导致自顶向下框架给出宏观的性能瓶颈之后，经常无法给出进一步的性能分析方向，需要细致的人工筛选 [78]。其中，访存并行度和推测错误惩罚作为高性能处理器设计的重要性能指标，在现有的自顶向下体系中没有被充分考虑。

2.5　超标量架构的局限性及应对方案

2.5.1　传统超标量架构规模扩展的局限性

在传统超标量架构中，指令窗口是为了挖掘指令级并行和访存级并行不可或缺的结构，指令窗口越大，挖掘指令级并行和访存级并行的机会越多。但是在指令窗口扩大的过程中，需要相应地扩大其他部件才能获得收益，例如物理寄存

器堆和指令发射窗口（表 1-1）。然而，过大的物理寄存器堆和发射窗口会严重影响频率和功耗，使 CPU 的工作频率和并行度挖掘之间产生矛盾 [6]。

物理寄存器堆用于存放拥有目的寄存器的指令的计算结果，一般用一个多读口、多写口的 RAM 实现，他的时延由读写口的数量和总的项数决定。文献 [5] 指出，在总项数较少时（小于 256），物理寄存器堆的线延迟时延与读写口数和项数主要呈线性关系。当总项数较大时，物理寄存器堆的线延迟关于读写口数的函数将呈二次方增长。与此同时，物理寄存器的大小与指令窗口的大小成正比（典型的商业处理器中比值约为 0.75），因此要想获得较大的指令窗口（比如 400 项），传统的物理寄存器堆需要相应地扩展到 300 项左右（表 1-1），这会给物理设计带来极大的挑战，很可能会损害 CPU 的工作频率。

指令发射队列负责唤醒指令，选择就绪指令并发射到执行单元。选择就绪指令有很多高效的妥协方案可以兼顾性能和时延，而唤醒操作一般用内容可寻址存储器（Content Addressable Memory，CAM）实现，需要将物理寄存器编号广播到 CAM 的所有单元，部分设计方法甚至需要将物理寄存器值广播。广播式唤醒要求信号在一个周期内传播到所有的单元，而信号需要广播的距离取决于数据位宽和发射宽度。文献 [5] 指出，广播式发射队列的线延迟与发射宽度和发射队列项数呈平方关系，因此发射队列的可扩展性较差。

面对超标量架构规模难以扩大的局限性，学术界提出了多种架构改进方案，本文将这些方案分为 3 个技术路线：1）基于传统超标量结构，通过降低乱序执行部件的复杂度来提高可扩展性；2）基于顺序执行架构，通过分片乱序执行来挖掘访存级并行；3）基于数据流思想，不依赖于物理寄存器堆和发射队列实现乱序执行。此外，工业界和开源社区普遍采用了定制物理设计或架构层面的妥协来提高可扩展性。本章接下来先介绍工业级和开源社区目前的应对方案以及可扩展性差给开源处理器带来的挑战，然后介绍学术界的多种架构改进方案。

2.5.2 工业界和开源社区的方案

针对集中式寄存器堆和广播式发射队列，工业界主要从物理设计的层面解决。例如，使用定制的动态电路来实现发射队列可以减小发射队列的面积 [5]；再例如，复制两份物理寄存器堆可以降低单个物理寄存器堆的时延，但是代价是花费更大的面积和功耗 [11]。此外，定制物理寄存器堆几乎成为了业界通用做法 [2, 85]。例如，Boom 处理器通过定制物理寄存器将面积占比超过一半的物理寄存器缩小到了可接受范围 [2]。但是定制物理寄存器堆需要耗费的人力和时间是难以避免的，这导致即使 RTL 部分使用了开源代码快速构建，物理设计也无法快速完成 [2]。

此外，工业界应对集中式物理寄存器堆的另一种方法是在分派时读寄存器堆，这样寄存器堆的读口数与分派宽度一致，降低了寄存器堆的读口数量。但是这使得发射队列不仅需要广播物理寄存器 tag，还要广播寄存器值。这样的设计会导致发射队列的面积大、功耗高。如图2-7所示，发射队列（保留站，rs_xxx）

的面积几乎达到了缓存面积的一半（memBlock）。并且这种设计的可扩展性很差，将上图所示的发射队列容量增加 1/3 导致整个处理器核的面积增加了 10%，印证了文献 [5] 所的计算结果。

综上所述，目前开源社区的高性能处理器的解决方案主要是两种：1) Boom 选择通过物理设计（半定制）来解决物理寄存器堆的频率问题，这拖慢了敏捷开发的周期 [2]；2) 香山处理器选择发射前读寄存器堆来解决物理寄存器堆的频率问题，这导致发射队列占据了大量的面积（图2-7）[19]。因此，目前开源社区的高性能处理器都仅仅妥协地解决了部分问题，并且遗留的问题影响到了迭代速度和芯片面积。如果不解决物理寄存器堆和发射队列的问题，很难通过扩大处理器规模来获得性能提升。

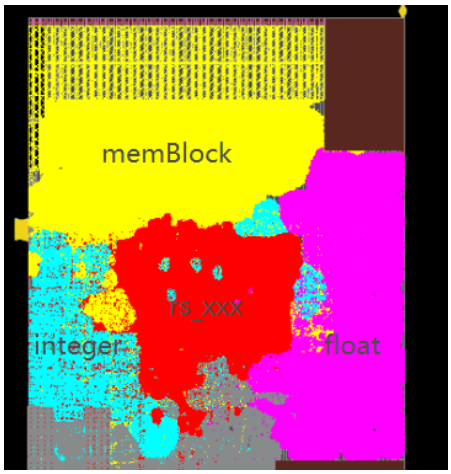


图 2-7 香山处理器的布局规划

Figure 2-7 The floorplan of Xiangshan processor

2.5.3　基于超标量架构提高可扩展性的改进

提高物理寄存器堆可扩展性。为了解决物理寄存器堆和发射队列带来的问题，研究者们尝试了很多改进。文献 [86–89] 通过缓存和多 bank 的方法来降低物理寄存器堆的物理设计难度。文献 [90–93] 通过提前释放占用的物理寄存器堆来降低乱序执行引擎对寄存器堆尺寸的需求。类似的，文献 [94] 通过虚拟寄存器编号的方式延迟物理寄存器的实际分配来提高对物理寄存器堆的利用率。

提高发射队列可扩展性。[95] 观察到很多动态指令在分派时只有 1 个未就绪的寄存器，利用这个优化这可以将寄存器的读口数量削减一半，将发射队列的广播面积减少一半。[20, 96] 观察到大部分动态指令的后继数量较少，可以让生产者追踪消费者的位置来避免发射队列带来的广播。[97] 提出可以让发射所占用的周期从一周期变为两周期，然后用推测的手段来减少性能损失。[98] 将多周期发射与基于依赖的唤醒方式结合，在性能和时延之间取得了更好的折衷。

虽然上述工作能在有限范围内缓解物理寄存器堆和发射队列带来的物理设计和功耗压力，但是这些设计都没有从根本上解决物理寄存器堆和发射队列的

问题，只是缓解了问题，无法任意地扩展指令窗口。为了获得更高的扩展性，学界有工作提出了基于顺序核的分片乱序执行和基于数据流架构的乱序执行。

2.5.4 基于顺序核的分片乱序执行

除了在原有的超标量架构基础上进行修改外，还有的研究工作尝试在顺序核的基础上改进，以获得超标量处理器的部分性能。例如，[25, 26] 提出了分片乱序执行架构（slice-OoO），将顺序执行的指令根据指令依赖分片，让相互之间没有依赖的指令片的头部访存指令可以并行执行。这使得顺序处理器可以在一定程度上获得乱序访存的能力，挖掘更多的访存并行度。虽然这些设计彻底避免了复杂的物理结构，但是这样的设计只能在部分程序上与传统超标量架构媲美，无法在高指令并行度的应用上达到超标量处理器的性能。

2.5.5 数据流架构

数据流架构在指令并行度和访存并行度两方面都有潜力与超标量架构相抗衡。最早的数据流架构诞生于超标量架构之前 [99, 100]，后来超标量架构被提出之后，研究者们发现超标量架构是在指令窗口内实现受限的数据流执行 [101]。当研究者们发现超标量架构的物理设计问题之后 [5, 6]，一部分研究者提出了用新的数据流架构来取代超标量架构 [22–24]。这些架构拥有分布式执行引擎（PE）的和分布式寄存器堆，使用了与 RISC 指令集不同的指令格式。编译器不再将 IR 的虚拟寄存器分配到体系结构寄存器，而是分配到一个 PE 的端口，即该虚拟寄存器在数据流架构中的“物理位置”。这样的设计使得每条指令只从具体的位置获取所需的值，无需从广播获取，也无需读取集中式的寄存器堆，避免了广播型发射队列和集中式物理寄存器堆。

与 RISC 指令集不同，显式数据流指令集 1）需要编译器了解具体的微架构信息；2）将指令和指令的操作数映射到特定的物理位置。具体来说，显式数据流指令集要求编译器负责从指令到 PE 的映射，从指令的操作数到 PE 的端口的映射。而在传统的 CISC 和 RISC 指令集中，编译器只需要了解体系结构寄存器等指令集层面的信息，无需进行微架构相关的、具体物理位置的映射。为了克服显式数据流集与现有的 RISC 生态不兼容的问题，同时从数据流的思想中获益，Forwardflow 架构 [21] 将 RISC 指令动态翻译为数据流依赖，从而在 RISC 指令集上也能避免集中式物理寄存器堆和发射队列。

第 3 章　敏捷性能评估方法

本章研究如何加速高性能处理器的性能评测。本章提出的敏捷性能评估方法采用了采样仿真机制和缓存预热加速机制，使得低成本、高效率的性能测算成为可能。

3.1　引言

在近几年的芯片敏捷开发浪潮中，得益于 Chisel[10]、BlueSpec[102]、PyMtl[103] 等新兴的硬件描述语言和 Rocketchip[1] 这样的 SoC 平台，直接基于 RTL 进行 CPU 架构设计变得越来越常见。随着社区的不断发展，基于这些语言的开源设计架构越来越先进，设计越来越复杂，例如加州大学伯克利分校开源的 Boom[2] 和中科院计算所开源的香山处理器 [19]。复杂的设计对现有的性能评估设施提出了更高的挑战，例如亚马逊云（AWS）的云化 FPGA 已经无法容纳目前最复杂的开源处理器核。

为了设计规模不受限制，部分工作采用了基于 RTL 软件仿真器的技术路线。但是对于复杂设计而言，RTL 软件仿真器在复杂设计下的速度仅有 500-3000 周期每秒 (CPS)[19]。为了加速仿真流程，利用采样和功能预热来减少仿真指令数是最常见的方法 [27]。采样方法起初是在 GEM5 等 CPU 架构模拟器上诞生的，要想直接在 RTL 软件仿真器上通过采样加速仿真，需要解决的问题是如何从高速模拟器序列化采样点和如何在 RTL 软件仿真器上反序列化检查点。为了解决该问题，本文提出了一种跨平台的采样格式 **RVGCpt**，利用 RISC-V 的特权指令来实现体系结构状态检查点的反序列化 [19]。

采样方法基于机器学习或者统计学方法来从冗长的程序中选择很少一部分片段作为代表来进行性能测量，进而估算整个程序的性能（如图3-1所示），它们的典型代表是 SimPoint [27] 和 SMARTS [58]。对于 SPECCPU® [104] 这样的大型程序而言，这些采样方法可以将总测量指令数缩减到千分之一以下。

采样方法会引入冷启动问题，即程序在特定采样点（Region of interest, ROI）开始执行时，处理器的微架构状态与从头开始运行是不同的。为此，研究者加入了预热机制，即在采样点开始之前增加一段不参与数据统计的预热执行阶段（如图 3-1(c) 所示），使微架构状态更贴近从头开始执行。在提高准确度的同时，预热也造成了额外的仿真指令数。为此，研究者通过功能预热、虚拟化预热等方法来降低预热带来的额外开销 [105]。

实验表明，即使应用了现有的检查点技术和 Verilator 的多线程加速 [9]，仍然存在需要仿真超 30 小时的检查点（40M 指令，图 3-2）。研究表明，为了降低单检查点的指令数，可以缩短单个检查点的采样长度 [27, 58]，但不能缩短预热长度，否则会损失性能 [62, 74]（详见 2.3 节）。因此预热时间占比会从现在的约

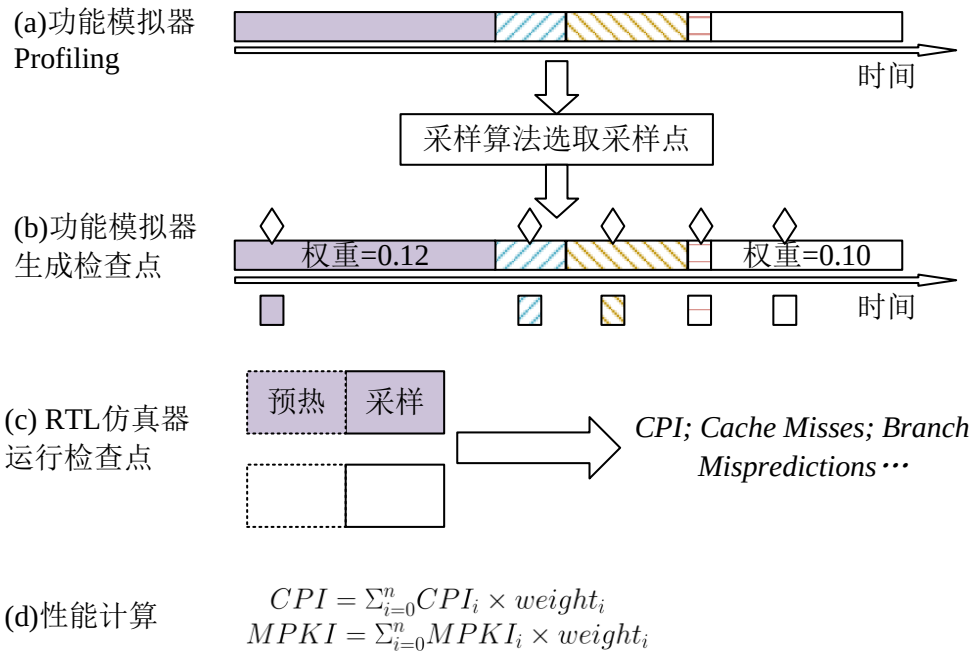


图 3-1 主流的采样方法工作流程

Figure 3-1 The workflow of the mainstream sampling methods.

50% 进一步上升，可见加速 RTL 预热非常重要。

为了加速 RTL 仿真预热，本文通过 profiling 根据程序预热需求将预热过程分为三段（图 3-3），然后对三段分而治之：完全跳过非必要预热，对可功能预热段采用功能预热加速，对必须采用 RTL 仿真的部分使用并行加速和调度优化。为了达成该方案，需要回答三个问题：1）在 RTL 仿真极其缓慢的背景下，面对快速迭代的设计，如何高效地 profiling；2）在快速迭代的设计上，如何实现可复用的功能预热；3）在任务时长分布不均且多任务间存在性能干扰的情况下，如何调度计算资源尽快完成全细节仿真。

问题 1：在 RTL 仿真极其缓慢的背景下，面对快速迭代的设计，如何高效地 profiling。有工作通过 profiling 获取应用的实际预热需求（指令数），这可以缩短不必要的预热长度 [62, 68, 71, 106]。但每当面对新设计时（比如缓存容量增加），都需要重新 profiling，反而增加了全细节仿真指令数 [68]。面对该挑战，本文 发现架构模拟器不仅运行速度比 RTL 快，而且可以得到近似的程序预热需求（详细实验数据见 3.5.5节）。因此，用模拟器替代 RTL 仿真进行 profiling 可以提高速度并兼顾准确度。

问题 2：在快速迭代的设计上，如何实现可复用的功能预热。尽管模拟器上的功能预热已有充分研究 [58, 62, 69]，但是没有现存的在 RTL 上实现功能预热的方法。其中最大的难题是，很难锚定 RTL 生成的 C++ 数据结构进行功能预热：因为从 RTL 生成的 C++ 数据结构可读性非常差，且变量名和存储结构会随着设

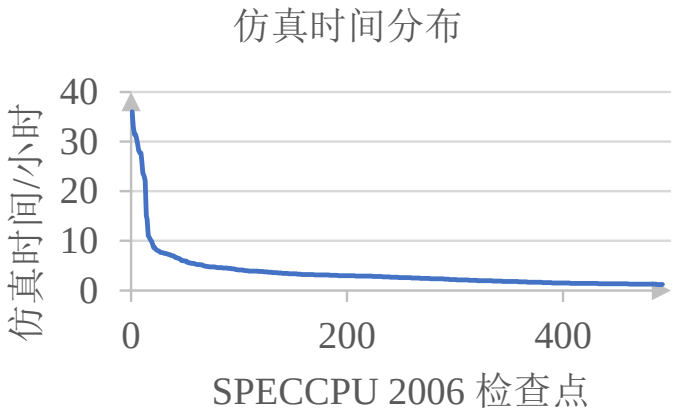


图 3-2 来自 492 个 SPECCPU® 2006 检查点的仿真时间分布。所有的检查点都用 16 线程加速运行。

Figure 3-2 The emulation time distribution of 492 SPECCPU® 2006 checkpoints. All checkpoints are emulated with 16 threads.

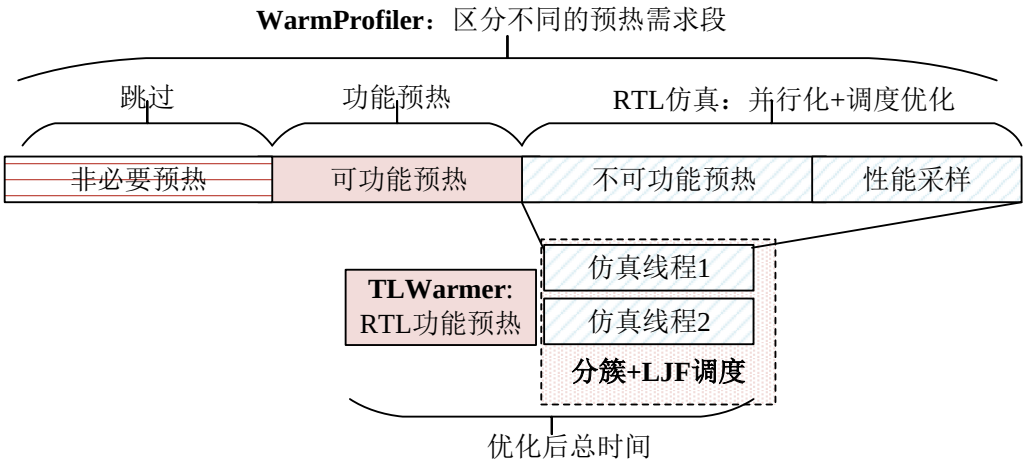


图 3-3 RTL 仿真预热的三个阶段

Figure 3-3 Three phases of warmup in RTL emulation .

计的变化而变化。面对该挑战，本文发现虽然生成的 C++ 数据结构会改变，但是总线协议是相对固定的，例如 AMBA [107]、TileLink [33]。如果锚定通用的缓存协议实现缓存功能预热，那么只要总线协议不改变就可以复用功能预热模块。

问题 3：在任务时长分布不均且多任务间存在性能干扰的情况下，如何调度计算资源尽快完成全细节仿真。图 4 表明少部分负载的仿真时间较长，该问题成为了影响仿真时间的支配因素，因此任务调度需要避免负载不均衡。此外，相较于服务器低负载运行单任务时的表现，本文发现在应用满载运行多个多线程任务时性能损失高达 45%（表 3-2）。为此，本文重新思考多线程和多任务对全系统吞吐的影响，提出了一种 CPU 核分簇方案（3.3.2节）。在此基础上，本文以 CPU 簇为调度粒度，对任务采用长作业优先（LJF）调度算法，该算法被证明最大完成时间在最差情况下只有最优调度的 4/3 [108, 109]。

综上所述，本文提出了 HyWarm 框架，它在 RVGCpt 的基础上进一步加速仿真，包括 1）WarmProfiler，基于架构模拟器进行预热需求分析，可以快速获得负载的预热需求；2）TLWarmer，通过锚定总线协议实现可复用的 RTL 功能预热；3）分簇 LJF 调度算法，实现最大化系统吞吐以及缩短最大完成时间。

本文在香山处理器 [19] 上应用了 RVGCpt，并利用 HyWarm 框架进行预热加速。RVGCpt 使得原本预计需要 7 年才能完成仿真的 SPECCPU2006 全细节仿真在 14 天内完成。在此基础上，TLWarmer 将 8 MByte 缓存预热时间从 10-100 小时缩短到少于 0.5 小时。综合 TLWarmer 和 WarmProfiler，HyWarm 能达到接近 25M 全细节预热的准确度（L1 缺失惩罚 95.1% VS. 91.3%，分支 MPKI 91.6% VS. 94.1%）。在准确度非劣于基线配置（25M）的情况下，相比于 25M 预热 + 随机调度的基线，在 RVGCpt 的基础上，HyWarm 将仿真完成时间缩短了 53%（表 3-5）。

3.2 背景与动机

3.2.1 硅前性能测算的主要方法

在芯片设计中，硅前性能测算对设计决策至关重要。现存的硅前性能测算手段主要包括三类：RTL 软件仿真器、基于 FPGA 的硬件加速仿真和基于仿真加速器的硬件加速仿真。

之前已经有很多研究致力于研究 FPGA 的硬件加速仿真相关问题。有的工作优化了从 CPU 模块到 FPGA 资源的映射，使得 FPGA 硬件加速仿真可以运行在更高的频率 [46]。有的工作利用 AWS 的云 FPGA 来提高 FPGA 硬件加速的易用性 [8]。还有工作利用 FPGA 进行电路的功耗预测和建模 [50]。然而上述工作都未能解决 FPGA 的资源容量有限导致无法容纳大规模的处理器设计的问题 [51]，例如 AWS F1 提供的 vu9p FPGA 芯片就因为 BRAM 和 LUT 开销无法容纳双核标配的香山处理器。

在业界，商业公司设计复杂 SoC 时往往会使用仿真加速器，例如 Cadence Palladium [54]、Mentor Veloce [55] 和 Synopsys Zebu [56]。仿真加速器拥有比 FPGA

更高的扩展性，但是仿真加速器的价格往往超过一千万人民币，这导致一个商业公司内部往往只有少量的仿真加速器。而且它的仿真速度一般在几 MHz [19]，比 FPGA 慢 10-100 倍，用一台仿真加速器来运行 SPECCPU® 2006 的性能测评至少需要几个月时间，这对实际项目也是不可接受的。

相比于 FPGA 和仿真加速器，RTL 软件仿真同时拥有最好的可扩展性和易调试性，它可以同时支持波形和文字打印，并且电路的规模只受限于服务器的内存容量。但 RTL 软件仿真器速度非常慢，根据仿真性能估算，如果用最快的开源 RTL 仿真平台 Verilator 来仿真香山处理器并评估 SPECCPU® 2006/2017 的性能，预估需要超过 7 年时间 [9]。

3.2.2　仿真采样方法

针对 RTL 软件仿真速度慢的问题，不难想到使用体系结构研究中常用的采样方法来减少仿真指令数。最广泛使用的两种采样方法是 SimPoint [27] 和 SMARTS [58]。SimPoint 通过功能模拟器 profiling 得到基本块向量，然后对基本块向量进行聚类，得到具有代表性的程序片段和相应的权重，通过运行这些片段来估算整体性能（如图 2-2(a) 所示）。SMARTS 则在程序中选取大量且均匀的采样点来估算整体性能（如图 2-2(b) 所示）。二者都可以大幅减少仿真的指令数（减少到 0.1% 以下）。在 SimPoint 和 SMARTS 的基础上，后续工作改进了它们的预热方法 [58, 62, 69–71, 110, 111]，增强了对多线程应用的支持 [59, 72, 73], 加强了对微架构相关特征的捕捉 [63]。

虽然上述采样算法可以减少总仿真指令数从而加速 RTL 软件仿真，但是这些采样算法无法直接应用于 RTL 软件仿真。将体系结构状态从功能模拟器迁移到 RTL 仿真器，需要两个过程：序列化和反序列化。其中序列化是指将体系结构状态检查点（寄存器值和内存）写入到磁盘或者宿主机内存中，以便其他仿真平台加载。而反序列化是指仿真器从磁盘或者宿主机内存中读取体系结构状态，以恢复到生成检查点的状态。GEM5 这样的模拟器与 RTL 仿真器最大的不同在于 GEM5 为序列化和反序列化都做了很多支持，但是 RTL 仿真器缺乏类似的序列化和反序列化的支持。

序列化功能模拟器的体系结构状态相对容易。因为在功能模拟器中，体系结构寄存器的状态存放于固定的数据结构中，因此易于序列化，而内存的状态可以通过整体拷贝的方式保存。但是反序列化，尤其是 RTL 仿真的反序列化则困难很多。在 GEM5 中，每一种 CPU 模型都提供了统一的反序列化体系结构寄存器的接口。要为 RTL 生成的 C++ 代码写一套统一的反序列化接口是不现实的，因为 RTL 生成的 C++ 数据结构是会随着 RTL 的修改变化的，并且可读性很差。

为了解决在 RTL 上对体系结构状态反序列化的问题，本文采用软件自动反序列化的方法来实现跨硬件平台的体系结构状态检查点（RVGCpt）。与 Dromajo [18] 相似，本文也选择了一个在指令集定义中较稳定的接口，并通过软件的方式来恢复体系结构状态。Dromajo 使用了 RISC-V 调试模块来恢复体系结构状态，而

RVGCpt 采用了 RISC-V 特权指令来恢复体系结构状。这两种体系结构检查点的实现方法都可以支持前述的 SimPoint 等采样方法，从而让 RTL 软件仿真的时间大幅缩短。RVGCpt 结合 SimPoint 让香山处理器的 RTL 软件仿真估算 SPECCPU® 2006 [104] 的时间从理论上数年缩短到了 2-14 天。

图 3-2展示了采用 RVGCpt 和 SimPoint 采样算法后，香山处理器仿真检查点的时间（40M 指令）。这些负载的仿真时间分布非常不均匀，假设计算资源充分多，那么多数负载在 10 小时内完成，而剩下的负载需要超过 30 小时，少数的检查点运行时间过长已经成为整个流程的关键路径。那么为了进一步加速性能测算，能否减少仿真指令数？研究表明在总采样指令数一定的前提下，减少每个采样片段执行的指令并相应地增加采样片段个数不会损失准确度 [27, 58]。例如，从 N 个 20M 的采样片段变为 4N 个 5M 的片段。在预热长度不变的情况下，这样虽然能保证准确度，但会增加仿真的总指令数。例如从 (20M+20M)*N 变为 (20M+5M)*4N，总仿真指令数增加了 150%，预热指令数占比上升到 80%。因此，为了缩短仿真时间需要缩短预热时间。

3.2.3 微架构状态预热加速

为了缩短微架构状态的预热时间，能否缩短预热长度？已有研究表明部分应用的预热需求超过 200M [62, 74]，因此在 20M-25M 的预热长度的基础上进一步的缩减已经不可行。尽管总预热长度无法缩短，但是研究者提出了功能预热方法以减少全细节预热的长度 [58, 62]：具体而言，功能预热方法采用更快的速度预热微架构模块的一部分功能，然后再使用全细节仿真来预热整个 CPU。

功能预热与全细节预热的最大区别是它用功能 CPU 来驱动缓存，提高预热速度。以 GEM5 模拟器为例，研究人员使用一个运行速度较快的顺序 CPU 驱动多级缓存。在缓存充分填充缓存块之后，再将 CPU 更换为乱序执行架构。得益于良好的软件接口，在 GEM5 模拟器上达成上述目标只需要将 SoC 的 CPU 指针指向乱序 CPU 即可。

但在 RTL 中，难以实现一个针对 RTL 生成的 C++ 数据结构的功能预热模块。这是因为处理器设计的不断迭代会带来新的 RTL 实现，而新的 RTL 所生成的 C++ 代码的数据结构也会随之变化，因此，旧有的功能预热模块难以被复用到新的处理器设计上。此外，如果使用这种思路，功能预热模块还需要比特精确地复刻 RTL 里的微架构状态，这对业界领先公司而言也是非常具有挑战性的，因为对齐模拟器的工作量巨大 [13, 76]。

文献 [62, 68, 74] 观察到 SPECCPU® 2006 中部分应用的预热需求很长，而另一部分应用的预热需求较短。因此，可以采取缩短预热需求较短的应用程序的预热长度的策略来节省整体的预热时间。为了确认这部分应用的预热需求，需要 profiling 多种不同的预热长度，找到性能准确度损失较小的最短预热长度。当后续重复运行同样的负载时，可以采用较短的预热长度进行预热。但是这种 profiling 方法并不适合快速迭代的架构，也不适合架构参数搜索，因为随着缓存

架构、规格的变化，旧有的 profiling 分析结果就会失效，导致无法通过大量重复运行来均摊 profiling 造成的开销 [68]。

3.3　观察与设计决策

3.3.1　混合预热加速的挑战与机遇

在 3.2.2节和 3.2.3节中，本文总结了处理器的 RTL 软件仿真中存在的三个挑战。1）RTL 仿真缺乏功能预热的功能支持，而且难以实现一个针对 RTL 生成的 C++ 数据结构的功能预热模块；2）当面对架构和参数变化时，难以通过多次运行来均摊 profiling 搜索预热需求的开销；3）由于少数检查点的运行时间过长，已经成为仿真时间的关键路径。

关于 RTL 预热，本文有 3 个观察。1）CPU 的部分模块使用了稳定的总线协议，例如缓存模块经常基于 TileLink 协议 [1, 112]，或者 AMBA 总线协议 [107]。虽然缓存内部的微架构设计会不断迭代，但是这些总线协议是稳定的。因此，只需要设计一套针对接口协议而不是针对微架构的功能预热方法，就能让功能预热模块在多个版本的 RTL 中得以复用。

2）同一个应用对不同模块的预热需求不同，尤其是分支预测模块的预热需求与缓存模块的预热需求往往差异较大（如图 3-4所示，*sjeng* 的分支 MPKI 在预热长度为 25M 时饱和，而二级缓存 MPKI 在预热长度为 4M 时即饱和）。因此，如果实现了功能预热，并提前了解某个应用对各个微架构模块的预热需求，就可以针对性地调节功能预热和全细节预热的长度，从而尽可能缩短完整预热所需时间。而为了“预知”负载的“可功能预热段”，需要提前测试多种可能的预热长度，从中选取既高效又精确的预热长度。

3）不同应用的预热需求不同。实验表明，约 38% 检查点的预热需求大于 20M，约 62% 检查点的预热需求少于 20M，其中约 57% 需求少于 5M（详细数据见3.5.4节）。如果能预知负载的预热需求，就可以在预热需求小的应用上节省大量时间，同时为预热需求大的应用提供更长的预热以获得更高的准确率。

为了对负载的预热需求进行 profiling，需要进行多种不同预热长度的搜索，Profiling 过程中，首先选取不同的预热长度，测量不同预热长度下的缓存 MPKI 或者分支 MPKI，从而构建如图 3-5所示的曲线。然后找到让 MPKI 满足误差阈值的最小预热长度。在搜索过程中，必须测试多个预热长度：如果对 9 个可选的预热长度进行二分查找（假设曲线单调），那么至少需要仿真 4 个点（$O(\log(n))$），则仿真指令数量就会增加约 4 倍。因此，如果不能通过多次运行来均摊，那么即使在预热阶段通过 profiling 缩短了仿真时间，所节省的时间也无法弥补 profiling 的代价。

面对 profiling 的挑战，本文有两个观察。首先，架构模拟器的仿真速度是 RTL 仿真器的 100 倍左右（Verilator 仿真香山约 1KCPS VS. GEM5 约 100KIPS），如果用架构模拟器进行 profiling，那么整体的仿真时间就不会被 profiling 拖累。

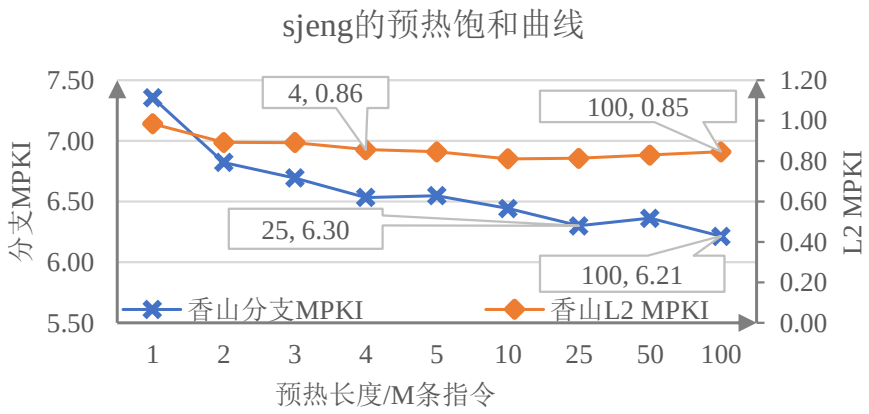


图 3-4 *sjeng* 的预热需求曲线。分支 MPKI 在预热长度为 25M 时饱和，而二级缓存 MPKI 在预热长度为 4M 时即饱和。

Figure 3-4 The curve of *sjeng*'s warmup demand. The branch MPKI saturates at 25M warmup length, while the L2 cache MPKI saturates at 4M warmup length.

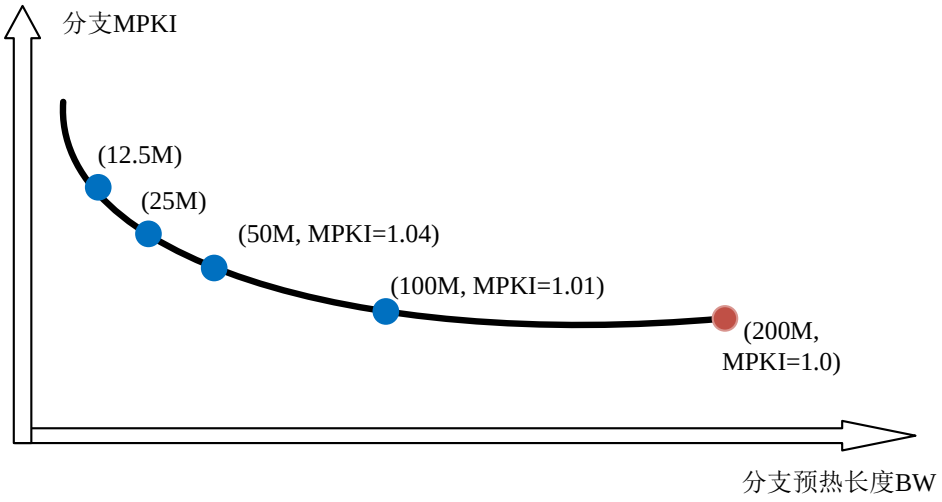


图 3-5 预热长度搜索过程

Figure 3-5 The process of searching for the warmup length

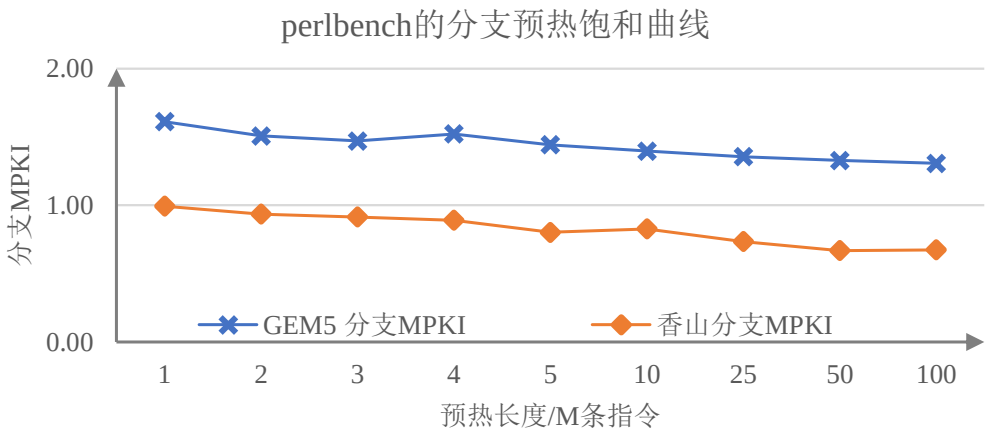


图 3-6 GEM5 模拟器与香山处理器的分支预测器预热需求展示出相似的趋势。

Figure 3-6 The warmup demand of the branch predictor in GEM5 and Xiangshan processor are similar.

相同的微架构算法的不同实现经过相同长度的预热之后留存的状态往往是相似的。这是受到了文献 [113] 启发：该文献设计了简化的 LRU 缓存用来恢复完整实现的 LRU 缓存的状态，并取得了不错的准确率。本文在预热需求上也观察到了类似的现象：如图 3-6所示，同为 32Kbyte 的 TAGE 预测器，香山处理器的 RTL 与 GEM5 的预热效果都在 50M 条指令附近饱和（更多数据见3.5.5节）。[1] 综上所述，为了高效地 profiling，可以用微架构算法和规格相似的架构模拟器来近似估计 RTL 的预热需求。

3.3.2　CPU 分簇与调度方案

得益于基于架构模拟器的 profiling 和基于总线协议的功能预热方案，本文大幅缩短了负载的预热时间，但是并没有改变整体的仿真时间分布特征：少量负载的仿真时长显著多于其它负载（如图3-9所示）。并且这些检查点在不恰当地调度下，有可能成倍地增加总仿真时间（如图3-7(a) 所示）。因此，调度任务使得负载均衡非常重要。

然而，在实际系统中对 Verilator 仿真的调度比传统的作业调度问题更复杂：1）很难预知任务时长；2）Verilator 提供了多线程加速功能，如何在多线程非线性扩展的情况下找到较好的调度策略；3）多个仿真任务之间存在性能干扰。尤其是 2）和 3），使得本就是 NP-hard 的任务调度问题变得更加复杂。

为了预知任务时长，本文利用 profiling 阶段在架构模拟器中得到的 IPC 来估算仿真任务的周期数，作为仿真时间的近似（Verilator 的总仿真时间与周期数正相关）。之前的工作表明，模拟器估算的 IPC 与 RTL 的真实 IPC 可能存在一

[1] MPKI 的差异来自与 GEM5 和香山处理器的分支预测器中对预热不敏感的部分，循环预测器 [114] 和 Statistical Corrector [35]。

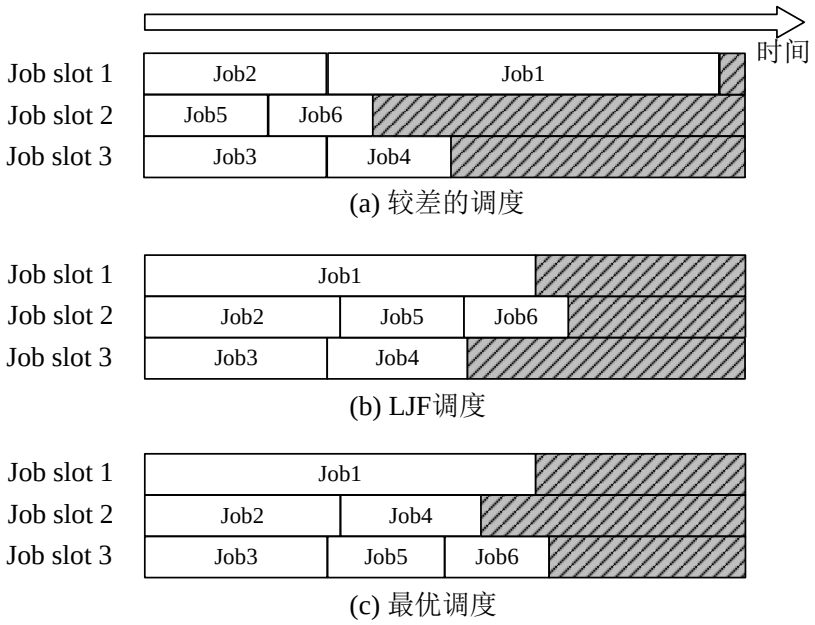


图 3-7 不同调度策略的最大完成时间对比

Figure 3-7 The comparison of the maximum completion time of different scheduling policies.

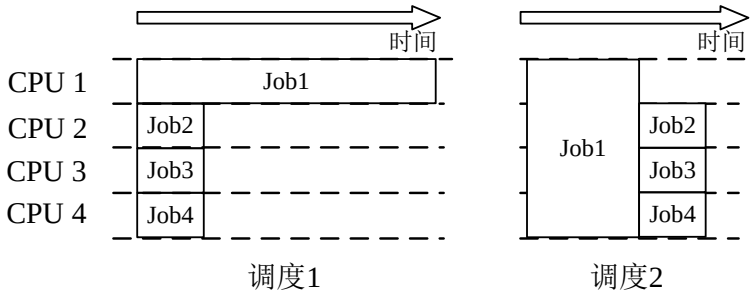


图 3-8 开启 Verilator 多线程仿真对调度策略的影响。调度 1 将 4 个任务都用单线程方式运行在 4 个 CPU 核上。调度 2 将 job1 用 4 线程运行在 4 个 CPU 核上，然后再运行 job2-4，缩短了总仿真时间。虽然调度 1 已经是单线程下的最优调度，但是仍然不如调度 2。

Figure 3-8 The impact of Verilator multi-threaded simulation on scheduling policies. Schedule 1 runs all jobs on 4 CPU cores with 1 thread per job. Schedule 2 runs job1 on 4 CPU cores with 4 threads per job, and then runs job2-4. Schedule 1 is already the optimal schedule in the scope of single-threaded emulation, but is still worse than schedule 2.

表 3-1 在服务器低负载时，Verilator 仿真的多线程扩展效率对比

Table 3-1 Comparison of multi-threading scaling efficiency when server load is low

线程数量	1 线程	4 线程	8 线程	16 线程
每核每秒指令数	190.82	538.28	450.94	321.27

表 3-2 在服务器满载时，Verilator 仿真的多线程扩展效率对比

Table 3-2 Comparison of multi-threading scaling efficiency when the server is fully loaded

	4 线程	8 线程	16 线程
每核每秒指令数	297.33	389.27	335.50

定差距，但误差一般在可接受范围内，并且能较好地反映程序之间 IPC 的相对差距 [14]。这种相对顺序的预测质量可以用归一化折现累积增益（NDCG [115]）描述，实验表明用 GEM5 模拟器预测的香山处理器的仿真时间的 NDCG 较高（NDCG=0.88，1.0 为完美预测）。

针对多线程调度问题，本文发现在任务时长分布不均时，为长任务开启多线程可以缩短仿真时间。如图 3-8所示，调度 2 比调度 1 的完成时间更短。由于 Verilator 的多线程扩展的性能收益是次线性的，如表 3-1所示，各种多线程选项中，吞吐最高的方案是配置为 4 线程。

然而，上述实验结果未能考虑任务间的性能干扰。当服务器的全部 64 个核都被仿真任务占用时，本文观察到了相反的性能数据：8 线程和 16 线程下的每核吞吐高于 4 线程（表 3-2）。这可能与 Verilator 巨大的访存足迹有关 [116]，而这类访存足迹巨大的应用会在末级缓存或内存带宽对邻居造成干扰 [117–122]。本文对比了运行单个 4 线程任务和满载运行 16 个 4 线程任务下的性能计数器，发现满载时的宿主机进程 IPC 比单任务时下降了约 66%。

基于表 3-1和表 3-2的实验，本文将 64 核服务器划分为 N 核的簇（N=8），总共有 64/N（8）个 CPU 簇，并以 CPU 簇为最小粒度进行调度。虽然将 CPU 核分簇调度不一定能得到全局最优调度，但是在考虑多核和系统干扰时最优调度难以获得，分簇调度至少是系统效率较高的一种方案。

在利用模拟器 IPC 估算仿真时间和将服务器划分为线程簇的基础上，任务调度问题可规约为传统的作业调度问题。虽然要保证图 3-7(c) 那样的最优调度非常困难（NP-hard）[123–127]，要尽可能避免出现图 3-7(a) 那样糟糕的调度。因此本文选择了长作业优先调度（LJF，图 3-7(b) ），已有工作证明 LJF 的最大完成时间在最差情况下是最优调度的 4/3 [108, 109, 127]。

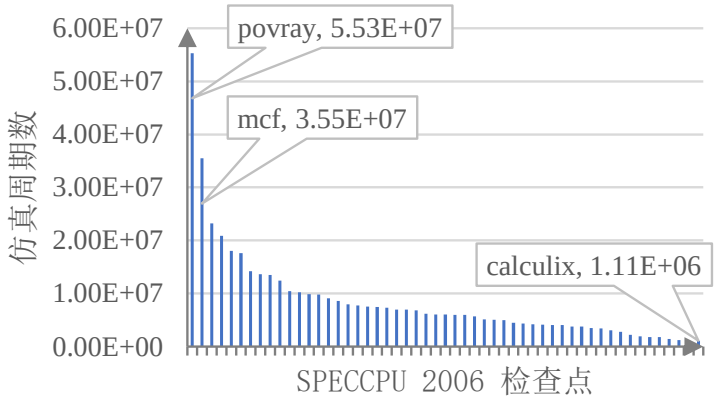


图 3-9 使用自适应预热时 53 个负载的全细节仿真周期数分布

Figure 3-9 The emulation cycle distribution of 53 workloads with adaptive warmup.

3.4 设计

3.4.1 RVGCpt

图3-10展示了 RVGCpt 的恢复原理。在 RVGCpt 的地址空间布局下，每次 RTL 仿真环境启动时，都会重置到初始化向量处（0x80000000），但是与普通的全系统环境不同，初始化向量处的指令不是 Bootloader，而是 RVGCpt 的恢复器。恢复器首先会检查 CPT Flag 字段，如果 CPT Flag 字段不匹配预设的 magic number，则说明不是从保存的检查点中恢复，会直接跳转到初始化向量处的指令（蓝色无条纹箭头所示）。如果 CPT Flag 字段匹配预设的 magic number，则说明是从检查点中恢复，恢复器会从 Register CPT 字段读取序列化到内存中的体系结构寄存器状态，逐个寄存器地恢复，完成恢复后跳转回生成检查点前一刻的 PC（红色条纹箭头所示）。

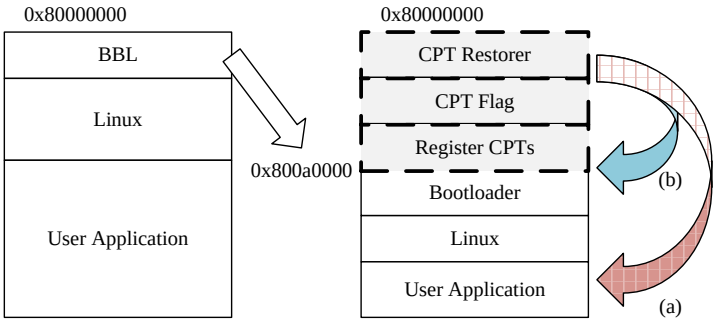


图 3-10 RVGCpt 的恢复过程

Figure 3-10 The recovery process of RVGCpt.

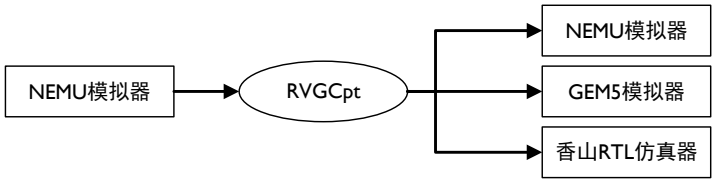


图 3-11 RVGCpt 在功能模拟器 NEMU 上生成，在 NEMU、GEM5 和香山处理器的 RTL 仿真器（Verilator）上恢复。

Figure 3-11 RVGCpt is generated on NEMU, and can be recovered on NEMU, on GEM5 and on the RTL emulator (Verilator) of the Xiangshan processor.

借用 RVGCpt 的恢复器，可以在任何支持 RISC-V 的模拟器或仿真器中恢复 RVGCpt 协议的检查点，如图3-11所示。让 GEM5 和 RTL 仿真器运行相同的负载，可以帮助进行设计空间的探索和决策。让功能模拟器 NEMU 与 GEM5 和 RTL 仿真器运行相同的负载，可以用于在线功能验证。

3.4.2　WarmProfiler

WarmProfiler 为了确定应用或检查点的预热长度需求，会在给定的处理器配置下尝试多种不同的预热长度，从而得到如图3-5所示的 MPKI 曲线。本文经验性地选择用 $9(1+N)$ 个样本点来绘制 MPKI 曲线，样本点的数量可以结合场景进行调整。

关于样本点的预热长度选取，在较长的区间选择几何级数递减的预热长度（100M，50M，25M，10M），在较短的区间采用等差数列递减的预热长度（5M，4M，…，1M）。本文没有选择样本的预热区间均匀长度分布（100M，90M，…，10M），这是因为预热需求较小的应用占多数，如果采用均匀分布会导致大量的应用落到 0 10M 的区间里并会选择 10M 的预热长度，即使它们的需求只有 1M。

首先，利用最长预热长度（例如 100M 指令）进行仿真，并计算出对应的 MPKI 值。然后，将此 MPKI 值作为基准，尝试找到能够达到特定 MPKI 阈值的最小预热长度。对余下的 $8(N)$ 个点，采用二分查找，最多需要运行 $3(log_2^N)$ 种配置就能找到满足要求的预热长度。

图 3-5展示了一个样本点数为 5 的搜索过程，假设基准点预热 200M，余下 4 个点为 100M、50M、25M、12.5M，假设 MPKI 阈值系数为 103%。首先测得 200M 预热下的 MPKI=1.0。然后，尝试 50M 预热，得到 MPKI=1.04，大于 MPKI 阈值 1.03，因此尝试更长的预热。然后，尝试 100M 预热，得到 MPKI=1.01。最终在满足要求（MPKI<1.03）的点中选取预热长度最短的点（100M）。该二分查找算法只在 MPKI 与预热长度的函数关系单调递减时保证选到满足要求的最短预热长度。当不是单调递减时会导致选不到最短预热长度，但是不会损失性能准确度。实践中，大部分应用的 MPKI 曲线都是单调的。这与之前的工作中预热越长 MPKI 越低的观察相符 [62, 71]。

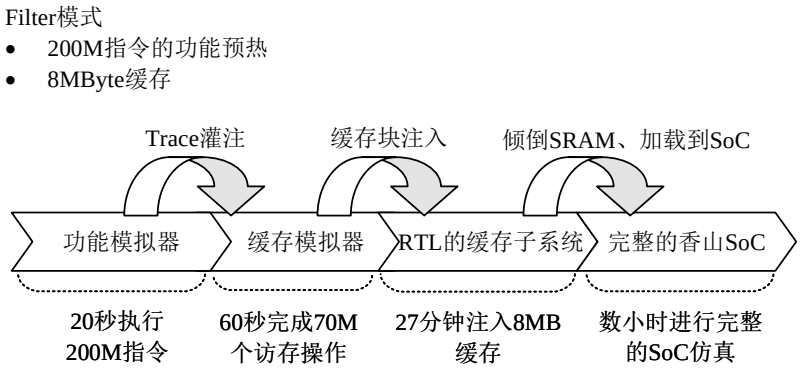


图 3-12 TLWarmer 在 Filter 模式下的工作流程

Figure 3-12 How TLWarmer works in Filter mode

一般地，缓存 MPKI 曲线与分支预测 MPKI 是相互独立的，因此需要两次进行上述搜索过程，分别找到缓存 MPKI 饱和点和分支预测 MPKI 饱和点。假设缓存 MPKI 的饱和预热长度为 CW，分支预测 MPKI 的饱和预热长度为 BW。其中缓存预热可以由功能预热完成，而分支预测器没有稳定接口，因此无法应用 TLWarmer 类似的思路，所以需要全细节预热。据此，可以计算出全细节预热的长度为 BW，功能预热的长度为 $FW = max(0, CW - BW)$。在实践中，因为功能预热的速度往往比全细节预热快 1-2 个数量级，因此可以始终选择运行固定的较长的 FW（100M、200M 等）。

3.4.3 TLWarmer

为了避免针对频繁变动的 RTL 生成的 C++ 进行功能预热，TLWarmer 的核心思想是利用像 TileLink 这样的总线协议给缓存注入预热后期望的缓存内容，注入一个缓存块的过程并不复杂，只需要按协议向缓存发送访存请求，就可以使相应的缓存块被取到缓存。值得讨论的问题有二：1) 完整的访存 trace 是庞大的，如何减少注入的访存请求。2) 为了在功能预热时加快仿真，本文选择了预热独立的缓存子系统，如何将独立的缓存子系统的缓存内容注入到完整的 SoC 的缓存子系统中。

本文把两种方案命名为 tracing 模式和 filter 模式。其中，tracing 模式按程序顺序依次发送所有的访存指令到缓存中，在该模式下，由 RTL 的缓存替换算法决定留下哪些缓存块。而 filter 模式类似于 [106] 中的理想 LRU 缓存，它用模拟器的替换算法作为过滤器，只将最终留存于模拟器的缓存中的块按照最近最少用的顺序注入 RTL 缓存中。Tracing 模式需要在 RTL 缓存中重放预热过程中所有的缓存请求，而 Filter 模式则只需要访问至多 N 个请求，其中 N 是缓存中缓存块的数量。因此 Filter 模式在 RTL 仿真器上产生的缓存请求更少且仿真周期数更少，是较优选择。

图 3-12展示了 Filter 模式下 TLWarmer 的典型工作流程。先用功能模拟器生

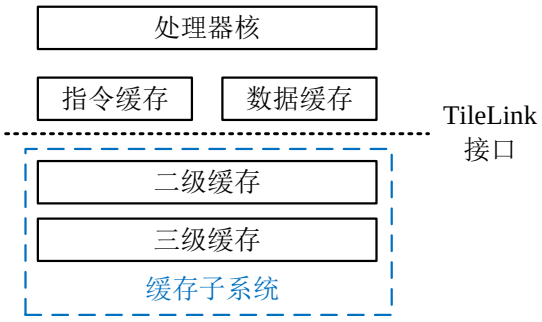


图 3-13 缓存子系统

Figure 3-13 Cache subsystem that receives TileLink requests

成访存 trace，然后在缓存模拟器中重放这些 trace，从而使缓存模拟器中按某种替换算法留下缓存块。实际上，如果功能模拟器带有缓存模拟功能，那么也可以让它们同时运行。

在缓存模拟器中留存的缓存块会通过 TileLink 协议注入到缓存子系统中（如图 3-13所示）。注入过程是由 RTL 缓存子系统接收并处理 128k 访存请求（8Mbyte/64Byte），所以该过程消耗的时间一般是几十分钟，显著多于前两阶段在模拟器上所花的时间（少于 10 秒）。因为在 TLWarmer 中模拟器部分占比很小，本文没有对模拟器上的过滤器进行进一步优化（例如应用 MRRL 技术 [106]）。

在获得了需要预热的块之后，下一个挑战是如何将缓存子系统的状态迁移到完整 SoC。这一过程中最大的挑战是缓存对象的碎片化问题：现代处理器的缓存并不是一块简单的 SRAM，而是需要分 bank 和分块（block）[112]，导致一块缓存经常包含几十个 SRAM 实例。为了避免人工地标注每个 SRAM 实例，本文利用 Chisel/Firrtl 实现了自动化的 SRAM 倾倒和恢复：在 Chisel 中选择特定的模块，为这些模块包含的 SRAM 对象打上标记，然后在 Firrtl 中对带标记的 SRAM 对象自动生成倾倒和恢复的代码。当缓存块注入完成后，缓存子系统的 SRAM 中的数据经由 Chisel 和 Verilog 提供的 API 被倾倒到很多个文件中，然后完整的 SoC 启动时会再次利用反向的 API 将这些文件加载到对应的 SRAM 中。

3.5　实验评估

本节先介绍 RVGCpt 的性能估算准确率，然后介绍 HyWarm 带来的性能评估速度的收益。在 HyWarm 部分，本文会先展示 WarmProfiler 对 SPECCPU® 的预热需求的分析，借此可以估算功能预热能减少的仿真指令数；接下来，本文会展示在功能预热的情况下，各个检查点的准确率和仿真时间对比，以证明功能预热对准确度和速度的增益；最后，本文会展示负载的运行时间分布不均的问题，以及如何通过分簇并行调度来缓解少数负载仿真时间过长的问题。

表 3-3 香山处理器的微架构配置

Table 3-3 The microarchitecture configuration of Xiangshan processor

部件	微架构与配置
分支预测器	16K TAGE-SC + ITTAGE + RAS + 4K BTB
一级数据缓存	128KB, 8 路数据缓存
一级指令缓存	128KB, 8 路指令缓存
二级缓存	1MB 8 路非包含
三级缓存	6MB 6 路非包含
一级指令 TLB	40 项
一级数据 TLB	136（128 x 4k 页 + 8 x 2M 页）
二级 TLB	2K 项
取指宽度	每周期 8*4Byte 指令
译码重命名宽度	每周期 6 条指令
ROB/LQ/SQ	256/80/64
物理寄存器堆	192 整数；192 浮点
执行单元	Int: 4xALU, 2xMDU, 1xMisc Float: 4xFMA, 2xMisc Mem: 2xLd AGU, 2xSt AGU

3.5.1 实验配置与测量指标

实验配置。本文的实验负载是 SPECCPU® 2006 中用 SimPoint 选取的检查点 [27]，采用了 RVGCpt 格式。由于 HyWarm 相关实验需要测试大量的预热长度配置，为了缩短实验时间，本文为每一个子项的每一个输入数据选取权重最大的 SimPoint 检查点来进行实验（共 53 个检查点）。在 HyWarm 的实验中，本文所采用的香山处理器架构配置如表 3-3所示。重放访存 trace 所使用的缓存模拟器的规格与香山处理器的缓存规格一致。

测量指标。本文研究 RVGCpt 的准确率时所用的主要指标为 SPECCPU® 的分值预测误差。本文研究预热长度所使用的指标主要包括 CPI、一级缓存缺失惩罚（L1 cache miss penalty，L1MP）和分支跳转 MPKI（Branch misprediction per kilo-instructions, 分支 MPKI）。其中，L1MP 用于讨论缓存预热效果，分支跳转 MPKI 用于讨论分支预测器预热效果，CPI 用于讨论整体预热效果。需要注意的是，本文没有纳入三级缓存 MPKI 作为测量指标，是因为在现有的实现中较难区分来自二级缓存的 demand 请求和预取请求。虽然没有包含三级缓存 MPKI 的数据，但是一级缓存缺失惩罚中包含了三级缓存中 demand miss 带来的性能损失。

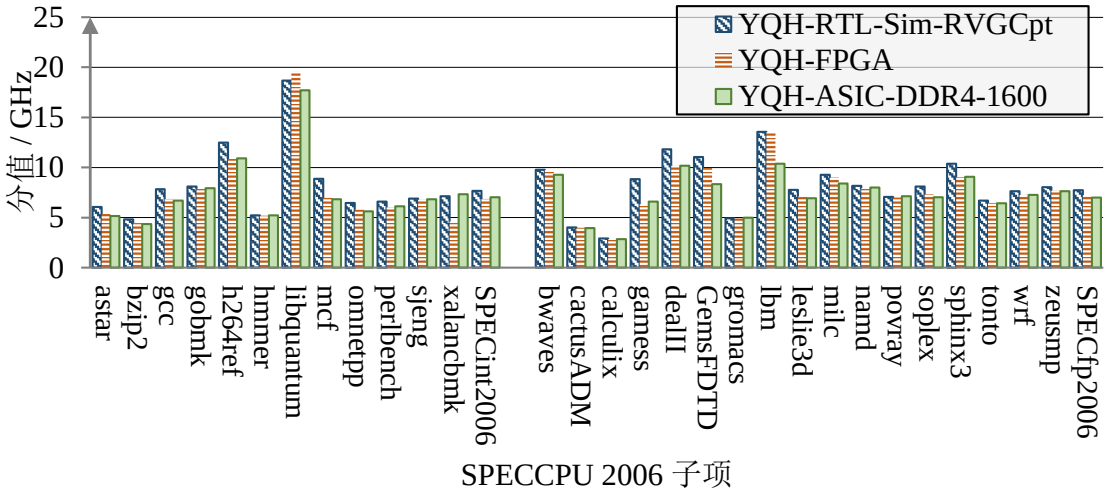


图 3-14 RVGCpt + SimPoint 估算的香山处理器 SPECCPU® 2006 分值

Figure 3-14 The SPECCPU® 2006 score of Xiangshan processor estimated by RVGCpt and SimPoint

3.5.2　RVGCpt 性能估算准确度

图 3-14展示了使用 RVGCpt 和 SimPoint 估算的香山处理器雁栖湖架构 [3] 的 SPECCPU® 2006 分值（YQH-RTL-Sim-RVGCpt），SimPoint 采样仿真时使用了 25M 预热长度。作为对比的是在 FPGA 上仿真所得的分值（YQH-FPGA）和流片实测的分值（YQH-ASIC-DDR4-1600）。相比于流片实测分值，使用 RVGCpt 和 SimPoint 估算的分值平均误差约为 11%。其中，误差较大的子项都是访存密集型应用，例如 *mcf* 和 *lbm*，这可能是因为仿真所使用的内存模型（DRAMSim3 [128]）与流片所使用的访存控制器 IP 不完全一致。虽然存在一定的误差，但是 RVGCpt 仍然能在项目中起到指导设计决策的作用。未来还可以通过改进内存模型和改进采样算法来进一步降低估算误差。

3.5.3　HyWarm 的整体性能评估

在性能估算准确率方面，HyWarm 相比于 3.5.2节中的 RVGCpt 的 25M 的普通预热方法，仅会额外造成约 0.1% 的 CPI 误差（表 3-6）。在性能测算速度方面，HyWarm 可以在一台单路 AMD EPYC 服务器（64 核）上用 6.95 小时完成 53 个检查点的性能测算，相比普通预热方法节省了超过 50% 的时间（表 3-9）。假设 RVGCpt 总检查点数为 1000，那么在一台双路 EPYC 服务器（128 核）上，预估只需约 66 小时就能完成基于 RVGCpt 和 HyWarm 的性能测算。本节接下来会介绍 HyWarm 识别出的应用预热需求，以及在此基础上进行功能预热和混合预热带来的性能收益。

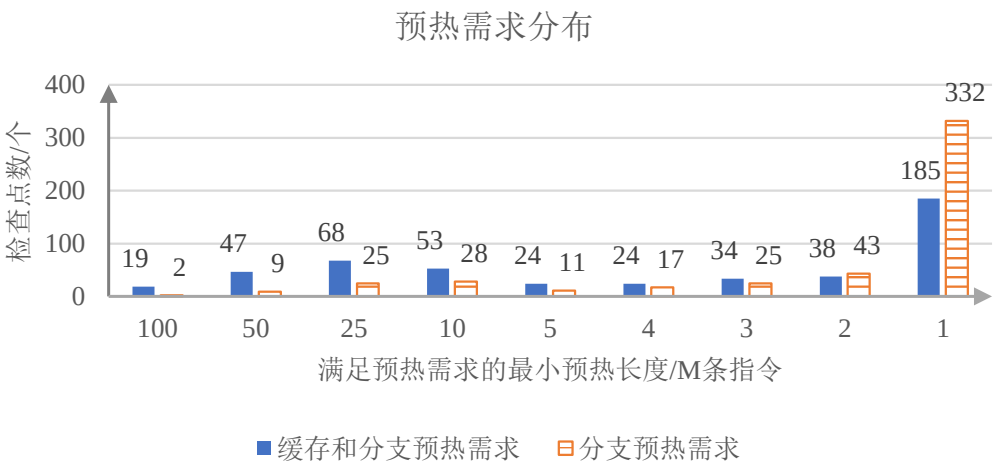


图 3-15 检查点的预热需求分布

Figure 3-15 The distribution of warmup demand of checkpoints

3.5.4 预热需求分析的准确度评估

图3-15展示了 WarmProfiler 在 492 个 SPECCPU® 检查点上 profiling 的结果。本文首先统计了每个检查点的总预热需求（图 3-15蓝色实心柱）：取分支预测器预热需求和缓存预热需求的较大值作为预热长度。缓存或分支预测器预热需求的计算方法在 3.4.2节中有详细描述。总预热需求展示出要达到逼近 100M 预热效果时，通过预先 profiling 可以减少的仿真指令数。如果没有 profiling，需要对所有的检查点都保守地运行 100M 指令的预热。相反的，有了 profiling 的指导，只有 19 个检查点需要 100M 预热，只有 47 个检查点需要 50M 预热。相对于 100M 预热的 baseline，将总的预热指令数减少了 85.7%。

需要注意的是，并非缓存和分支预测都必须使用全细节预热，因为 TLWarmer 预热缓存不需要经过全细节预热。在此基础上，只考虑分支预测器预热，那么全细节预热的需求将会进一步被压缩：只有 2 个检查点需要 100M 的全细节预热，只有 9 个检查点需要 50M 的全细节预热（图 3-15橙色条纹柱）。如果以 100M 全细节预热为基线，总的全细节预热指令数减少了 95.6%。

在 HyWarm 中，本文借助了 GEM5 这样的架构模拟器来 profiling 获得预热需求。为了验证 GEM5 所获得的预热需求是否与 RTL 的实际预热需求一致，本文对比了每个子项权重最大的检查点在 GEM5 和 RTL 上的预热需求曲线。实验结果表明，当 GEM5 的参数与 RTL 的处理器一致的情况下，他们的预热需求的趋势也较为接近。如图 3-16所示，多数子项的模拟器和 RTL 的分支预热需求一致。而像 *gobmk.trevoc* 这样的子项在 GEM5 上展示出了比 RTL 更长或更短的预热需求，这会增加一些不必要的预热长度或引入一定的误差，但是占整体的比例较小（7.5%）。

预热敏感性。本文还发现了部分应用对预热长度并不敏感。以缓存为例，本

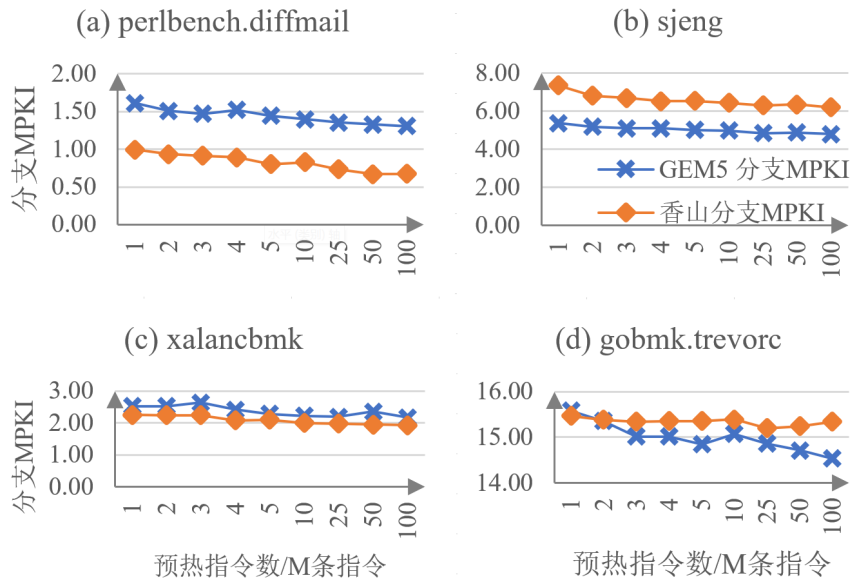


图 3-16 GEM5 模拟器与香山处理器的预热需求曲线

Figure 3-16 Warmup demand curves of GEM5 and Xiangshan processor

文计算了各个子项的一级缓存缺失惩罚与预热长度之间的相关性（Pearson’s correlation）。在 53 个负载中，34 个负载的 L1MP 与预热长度的相关性小于-0.5（预期 L1MP 与预热长度负相关），但是余下 19 个负载没有显著的负相关。例如 *GemsFDTD*、*lbm*、*libquantum* 等负载的 L1MP、每千条指令二级缓存缺失次数均与预热长度没有明显的相关性，本文将这类负载称为预热不敏感应用。在接下来讨论预热效果时，仅讨论缓存或分支预热敏感型负载。

3.5.5　功能预热和混合预热的性能评估

本小节先展示功能预热节省预热时间的作用；然后展示混合预热既能减少总仿真时间，又能兼顾的性能评估的准确率。本文对比了当采样长度为 5M 时的多种预热方案的仿真完成时间和性能准确率。表 3-4列出了这些预热配置，其中配置 *(X+Y)* 表示长度为 *X* 的功能预热之后进行长度为 Y 的全细节预热，*Ada* 表示根据 WarmProfiler 的结果自适应选取的功能预热和全细节预热长度，*FixedFW(95+5)* 则是在没有 WarmProfilier 的指导下只利用 TLWarmer 进行固定长度的功能预热和全细节预热。

缓存功能预热。图 3-17展示了不同预热方案相对于 100M 全细节预热的 L1MP 增幅。实验结果表明 FixedFW (95+5) 的混合预热方案显著优于 0+5 的全细节预热方案，这体现了功能预热的有效性。图 3-17中，Ada 和 FixedFW 部分负载的 L1MP 比 100M 预热的基线更低（出现了负值），这可能与替换算法、预取算法有关，因为 TLWarmer 只能恢复缓存的 SRAM 中的内容，不能恢复预取器和影响块替换的 metadata 的状态。对一部分负载，这会让有用的缓存块更容易留下，而对另一部分负载会让有用的缓存块。这也解释了为什么 *FixedFW* 的方案无法达到 100M 的全细节预热基线的 L1MP。

表 3-4 预热配置列表

Table 3-4 Warmup configurations

	功能预热	全细节预热	性能测量
0+100	-	100M	5M
0+50	-	50M	5M
0+25	-	20M	5M
0+10	-	10M	5M
0+5	-	5M	5M
Ada	100M-DW	自适应 (DW)	5M
FixedFW (95+5)	95M	5M	5M

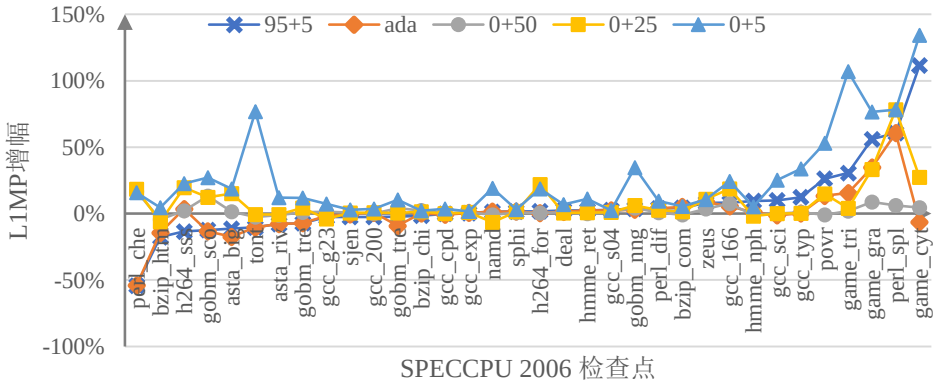


图 3-17 不同预热方案关于 L1MP 的评估误差。数据点按 FixedFW 方案的 L1MP 相对增幅排序。

Figure 3-17 Relative L1MP increase of different warmup configurations. Data points are sorted by the relative increase on L1MP of FixedFW.

从仿真总时间来看，功能预热的时间取决于需要恢复的缓存块的个数，一般在 5 分钟-30 分钟内，这显著少于 5M 的全细节预热的时间。也就是说，混合功能预热方案 FixedFW 只需要比 0+5M 方案略多的时间（38.5*h* VS. 35.8*h*，表 3-5），就能获得与 0+25M 相当的缓存预热效果。

自适应混合预热。根据 3.5.5节的性能评估结果，负载的预热需求往往是随着负载特征变化的，盲目地选取 *FixedFW(95+5M)* 的方案实际上无法适应这种变化。WarmProfiler 可以寻找到对性能评估准确度牺牲较小（比如缓存 MPKI、CPI 等指标）同时又节省时间的全细节预热长度。在图 3-17、图 3-18和图 3-19中由 WarmProfiler 指导的混合预热方案被标注为 *Ada*(adaptive)。表 3-6表明自适应混合预热达到了与 0+25 接近的准确率。表 3-5表明仿真 53 个负载所需的总串行仿真时间为 54.40h，仅略大于 0+10 方案。每个子项完整的仿真时间数据见表 A-1。

表 3-5 不同预热方案的总仿真时长对比

Table 3-5 The total emulation time comparison of different warmup schemes.

时间/h	0+5	0+10	0+25	*Fixed*	*Ada*
总计	35.8	**52.7**	**105.5**	38.5	**54.4**

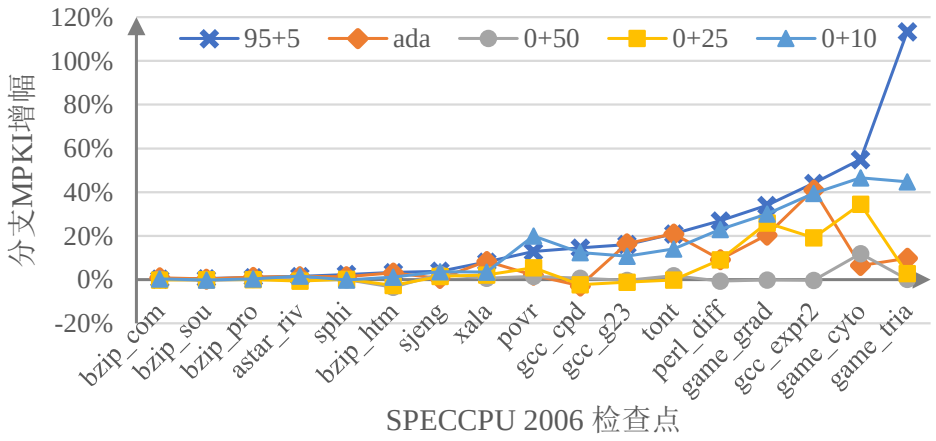


图 3-18 不同预热方案对分支 MPKI 误差的影响。数据点按 FixedFW 方案的分支 MPKI 相对增幅排序。

Figure 3-18 Relative branch MPKI increase of different warmup configurations. Data points are sorted by the relative increase on branch MPKI of FixedFW.

以 SPECCPU® 2006 的 *gamess_cytosine* 子项为例，在使用 *FixedFW* 混合预热方案时，*gamess_cytosine* 的 L1MP、分支 MPKI 和 CPI 增幅都较大。而 WarmProfiler 发现它的全细节预热需求很大，即使 0+50M 的全细节预热也无法使他的分支 MPKI 达到接近 0+100M 的情况。因此，为了保证准确性，WarmProfiler 直接为 *gamess_cytosine* 选择了 0+100M 的预热方案。在 *gamess_triazolium* 上也观察到类似的现象，WarmProfiler 为该负载选择了 75+25M 的混合预热方案。

WarmProfiler 的误差带来的影响。为了探究 WarmProfiler 的误差带来的影响，本文将基于模拟器的预热需求预测和完美预测作对比，其中完美预测直接利用 RTL 进行预热需求搜索得到分支预测器预热饱和点。表 3-7列出了所有 GEM5 预测预热需求偏低造成 MPKI 偏高的负载点（由于更低的 MPKI 更接近完整的 100M 预热，因此可以认为 MPKI 更低是更准确的表现），其中仅 4 个负载点遭遇了超过 0.1 的分支 MPKI 误差。而根据计算，0.1 的分支 MPKI 误差导致的 CPI 误差一般不超过 0.3%。在预热时间方面，实验结果表明模拟器预测的预热饱和点误差造成了 8.85% 的额外仿真时间。作为对比，直接用 RTL 进行预热需求搜索会增加数倍的时间，因此用模拟器预测预热需求更加高效。

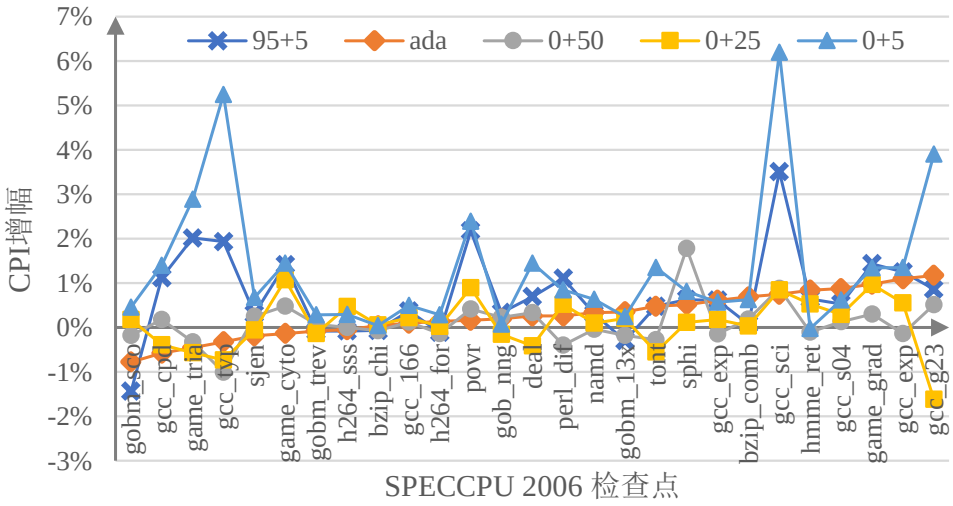


图 3-19 不同预热方案对 CPI 误差的影响。数据点按 Ada 方案的 CPI 相对增幅排序。

Figure 3-19 Relative CPI increase of different warmup configurations. Data points are sorted by the relative increase on CPI of Ada.

表 3-6 不同方案的平均准确率对比

Table 3-6 Accuracy comparison of different warmup configurations.

准确率	Ada	0+50	0+25	0+10
CPI	99.6%	99.8%	99.7%	99.1%
分支 MPKI	91.6%	98.9%	94.1%	85.2%
L1MP	95.1%	97.5%	91.3%	82.8%

3.5.6 并行调度的性能评估

虽然本文的方案大幅缩短了预热所需指令数，但是并不能改变总仿真时间在每个负载上的分布。如图 3-9所示，检查点的仿真周期数非常不均衡，少数负载由于预热指令较多（图 3-9的 *povray*）、IPC 较低（图 3-9的 *mcf*），它们仿真所需的时间显著长于其他负载。如果简单地让 53 个负载并行地开始仿真，那么仿真的后半段会只剩下几个最慢的负载。负载时长不均衡的问题不仅存在于自适应预热方案，在普通预热方案下也存在（图 3-2）。

分簇调度和 LJF 调度的重要性。第3.3.2节用简单的例子定性说明了 CPU 多核并行仿真给调度带来好处（图 3-8），而进一步的实验结果印证了分簇的收益。本文定义了**调度均衡度**用于描述负载均衡的程度：调度均衡度 = 平均完成时间/最长完成时间。表 3-8展示了 64 核下 3 种配置的调度均衡度，在同样的负载集下，CPU 簇数量越少，调度均衡度越高，无论是随机调度还是 LJF 调度都

表 3-7 WarmProfiler 预测预热饱和点造成的分支 MPKI 误差（增高），比较基准为根据 RTL 的数据完美预测的 MPKI 饱和点。

Table 3-7 Branch MPKI error (increase) caused by WarmProfiler. The baseline is the MPKI saturation point predicted by with perfect prediction provided by RTL.

负载点	完美预测 MPKI	MPKI 增高	MPKI 增高百分比 (%)
gcc_expr2	0.443	**0.177**	39.9
gcc_g23	0.973	**0.172**	17.7
tonto	0.506	**0.117**	23.1
gamess_gradient	0.430	**0.112**	26.1
gcc_scilab	7.687	0.090	1.2
xalancbmk	2.003	0.079	3.9
gcc_s04	0.163	0.070	42.8
perl_diffmail	0.669	0.066	9.8
h264ref_foreman	0.042	0.064	151.9
astar_rivers	3.422	0.053	1.6

表 3-8 簇的数量对调度均衡度的影响。3 种配置的含义：4 簇 =4 簇 ×16 核；8 簇 = 8 簇 ×8 核；16 簇 = 16 簇 ×4 核。

Table 3-8 The impact of cluster number on schedule balance. The meaning of 3 configurations: 4 clusters = 4 clusters × 16 cores; 8 clusters = 8 clusters × 8 cores; 16 clusters = 16 clusters × 4 cores.

调度均衡度	随机调度	LJF 调度
4 簇	0.93	0.99
8 簇	0.76	0.98
16 簇	0.54	0.63

是如此。这说明 CPU 簇越少，调度的“难度”越低，越容易达到均衡的调度。而从另一个角度看，LJF 总是能获得比随机调度更高的均衡度。在端到端性能对比中（表 3-9），同样能观察到 LJF 的优势：无论是在 64 核（8 簇 x 8 核）还是 128 核（16 簇 x 8 核），LJF 比随机调度节省了 11-20% 的时间。

表 3-9 LJF 调度与随机调度的仿真时间对比。其中 Ada+LJF 是 HyWarm 提出的方案，25+5+随机调度是基线方案。

Table 3-9 The emulation time comparison between LJF and random scheduling. Ada+LJF is the scheme proposed by HyWarm, and 25+5+random scheduling is the baseline.

仿真时间/h	随机调度	LJF 调度	提升
Ada，8 核 x8 簇	8.77	6.95	20.75%
Ada，8 核 x16 簇	6.26	5.38	14.06%
25+5，8 核 x8 簇	14.91	13.25	11.13%
25+5，8 核 x16 簇	11.67	9.35	19.88%

3.6 本章小结

本章提出一个敏捷性能评估框架，以缩短处理器的性能测算时间。由于香山处理器电路规模大、软件仿真时间长，本章提出了一种 RISC-V 通用检查点 RVGCpt，并结合 SimPoint 算法通过采样方法大幅减少了仿真指令数量，进而缩短了性能测算时间。此外，本文还提出了 HyWarm 混合预热方法，结合了 RTL 缓存功能预热和自动化预热需求分析工具，将预热过程的仿真指令数减少了 85.7%。敏捷性能评估框架使得基于 RTL 软件仿真的性能测算从不可用（数年）变为可用（数十小时）。

第 4 章　架构迭代基础设施研究

前一章介绍了用于处理器 RTL 仿真的敏捷性能评估方法，该方法采用了仿真检查点、功能预热和混合预热来加速性能评测。为了进一步缩短性能评测反馈时间和提升设计空间探索效率，本章研究如何为香山处理器构建性能计数器体系和搭建与香山处理器精确对齐的微架构模拟器。

4.1　引言

根据业界经验，在高性能处理器的架构迭代中，用微架构模拟器进行架构探索比直接基于 HDL 进行架构探索更加高效。若在 HDL 上直接进行架构探索，不仅 HDL 开发和性能测评均慢于微架构模拟器，而且 HDL 探索新功能失败所损失的时间和人力成本也高于微架构模拟器。现有的开源架构模拟器（GEM5[12]）尚未与现有的开源处理器（Boom[2] 和香山处理器 [3, 19]）精确对齐，直接基于未对齐的微架构模拟器进行架构探索得出的结论很可能是不可靠的 [13–15]。

GEM5 的乱序执行 CPU 的建模对象是 ALPHA 21264[11, 12]，要将 GEM5 完全对齐到香山处理器的工作量很大 [13]。面对巨大的工作量，为了提高定位性能差异和性能缺陷的效率，为 GEM5 与香山处理器构建一套协助分析性能差异的性能计数器体系至关重要。

性能计数器是指导架构设计的重要工具，也是定位性能差异的重要工具。传统的微架构性能计数器一般是 ad-hoc 、自底向上的，它们与 CPU 微架构设计强相关，本章接下来的部分用“自底向上性能计数器”来指代缓存 MPKI 等微架构性能计数器。从众多的自底向上性能计数器中寻找系统的性能瓶颈并不容易，因为微架构事件最终不一定贡献到性能损失 [17]。为了从最终对整体性能造成的影响的角度自顶向下地分析 CPU 的性能瓶颈，一系列工作提出并改进了自顶向下性能计数器 [16, 28, 29]。如图4-2所示，自顶向下性能计数器量化了 CPU 流水线和存储层次各部分对性能的影响，并且支持对粗粒度性能计数器的细化，从而实现自顶向下定位系统瓶颈。

然而，现有的自顶向下框架中的成分都是时间片，成分间的连接关系都是“相加”。这限制了自顶向下框架的表达能力，使之无法描述事件之间相互的覆盖、多个事件之间的并行等关系。因此高性能处理器中为了掩盖缺失事件、提高并行度而设计的组件对性能的影响无法在自顶向下框架中的成分中得到体现。这可能导致自顶向下框架的分析结果产生错误的性能损失归因。例如，当受限于三级缓存的成分（图 4-1(a) 中的 L3）较高时，性能损失的原因不一定直接来自三级缓存设计缺陷，而是来自影响三级缓存访存并行度的其它因素。

为此，本文提出了展开式自顶向下分析框架，该框架将访存并行度这样描述并行度的成分纳入考虑，使得访存并行度、缓存缺失率等与微架构设计紧密联系

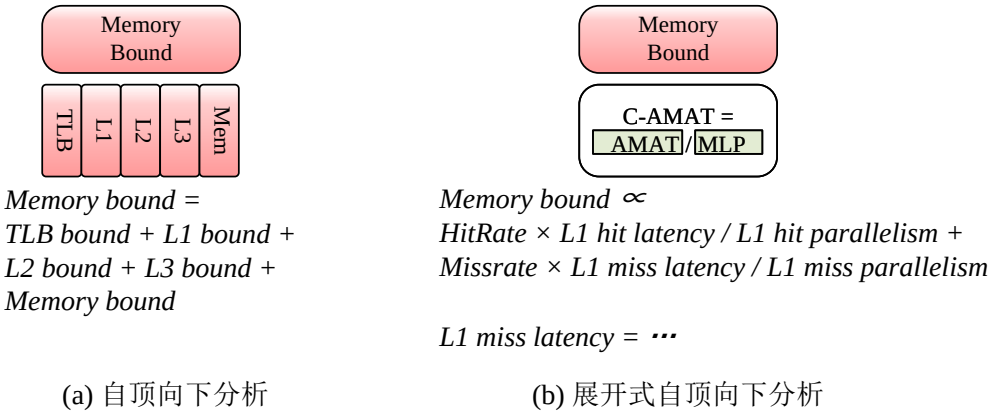


图 4-1 展开式自顶向下分析与自顶向下分析对受限于访存的成分有不同的分解方式

Figure 4-1 The difference between Elaborated Topdown analysis and Topdown analysis in decomposing the memory-bound component

的指标出现在展开式自顶向下分析框架中。为了描述不同量纲的成分之间的关系，展开式自顶向下分析框架允许使用乘法、除法、取最大等运算来描述父成分与子成分之间的关系。如图 4-1(b) 所示，展开式自顶向下分析框架中的受限于访存成分可以表示为一级缓存访问延迟、命中率、缺失率和访问并行度的函数。

展开式自顶向下分析充分考虑了访存并行度在访存受限时的影响，可以帮助架构师发现访存并行度存在问题。为了进一步加速访存并行度相关问题的定位，本文还提出了并行访存缺失簇（Parallel miss cluster，PMC）分析器来寻找性能缺陷发生的指令区间。

本文的主要贡献包括 1）改进了自顶向下分析方法，提出了展开式自顶向下分析框架，扩展了自顶向下分析的表达能力，增强了自顶向下分析与微架构设计的联系。2）提出了并行访存缺失簇分析器，用于加速定位访存并行度导致的性能缺陷、性能差异。3）初步对齐了 GEM5 微架构模拟器与香山处理器在 SPECCPU® 2006 上的性能表现，定位了余下的性能差异的原因。

本文的结构如下:4.2节介绍模拟器对齐和性能计数器的相关背景, 其中4.2.4节介绍了自顶向下性能计数器与自底向上之间割裂的问题，以及造成的影响。4.3节介绍展开式自顶向下分析框架对自顶向下性能计数器的补充以及如何将自顶向下性能计数器与自底向上计数器建立联系，其中4.3.4节介绍细粒度的访存并行度异常检测方法。4.4节介绍模拟器对齐过程中，性能测量的微基准测试与性能计数器的具体实现。4.5节介绍模拟器与 RTL 对齐后的性能对比，以及在对齐模拟器过程中应用展开式自顶向下分析框架的实例研究。

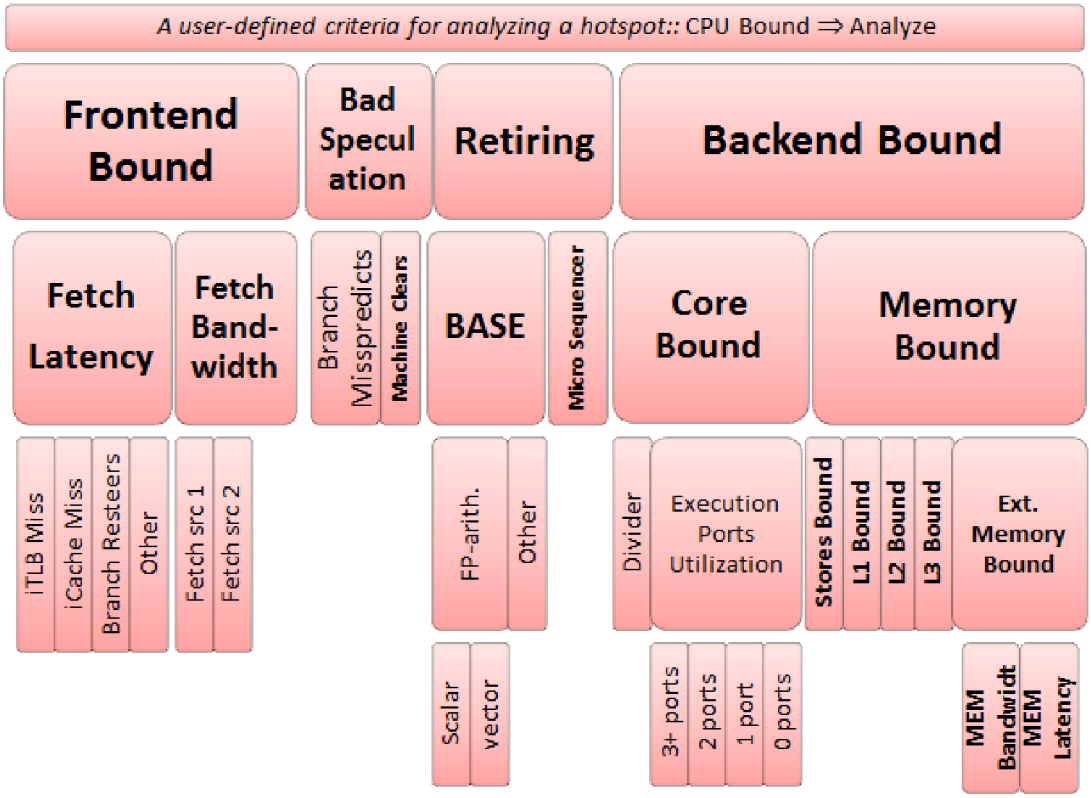


图 4-2 自顶向下分析方法，来自文献 [29]

Figure 4-2 The top-down analysis method proposed in [29]

4.2　背景和动机

4.2.1　架构模拟器的对齐与性能探索

在第 3章中，本文将微架构模拟器的采样方法迁移到了 RTL 仿真从而使得在 Chisel 生成的 RTL 上进行一些设计决策成为可能。但是，受限于多方面的原因，直接基于 Chisel 进行激进的架构探索仍然无法达到与架构模拟器（例如 GEM5）相媲美的效率。如表4-1所示，由于从 Chisel 生成 RTL 仿真器的速度较慢、RTL 仿真器的速度也较慢，因此在 Chisel 上进行架构探索时等待 debug 和性能数据的时间可能达到架构模拟器 10 倍以上。RTL 上进行 debug 和性能测评的速度慢的原因是 RTL 仿真的速度慢（见第 3章）。

表 4-1 基于微架构模拟器和基于 Chisel 的架构探索效率对比

Table 4-1 Comparison of the efficiency of architecture exploration based on microarchitectural simulators and based on Chisel

时间花费/h	基于微架构模拟器	基于 Chisel
单次 debug	0.5	5
单次性能测评	4	66
20 次 debug + 5 次性能测评	30	430

要使用模拟器进行设计空间探索，最大的问题是未经校准的架构模拟器可能会产生扭曲甚至错误的结果 [15]。过去，将模拟器与真实硬件对齐的主要障碍是商业公司内部保留了大量架构设计细节，使得学术界无法获取到真实架构设计与实现 [76, 77]。幸运的是，开源处理器的出现消除了这一障碍。然而，高性能处理器设计非常复杂，即使有开源代码作为参照，要精确对齐模拟器仍需巨大工作量。为了提高微架构模拟器对齐和架构探索的效率，本文应用了微基准测试和各类性能计数器。

4.2.2 基本性能测量与自底向上计数器

在业界共识中，进行架构模拟器对齐和处理器性能验证（Validation）的第一步是利用微基准测试进行基本的性能测量 [13, 14]。其中，指令延迟测量和指令组合的延迟测量是最常见的测量对象 [13]。指令延迟的测试用例可能会枚举所有组合情况，或者随机生成指令序列 [13]。在此基础上，为了测试更多微架构相关的性能特点，还需要手写很多测试用例来测试分支预测器、缓存等部件的性能 [14]。

在检查基本的性能对齐结果时，大家一般会采用每周期提交指令数（IPC）和自底向上性能计数器作为指标。例如，可以用缓存 MPKI 来检查缓存容量、预取算法和替换算法的对齐程度。类似的，可以用分支指令 MPKI 来检查分支预测部件的对齐程度。

4.2.3 自顶向下性能计数器

微基准测试和自底向上性能计数器能反映系统各部分的基本性能，但是由于高性能处理器的复杂性，缺失事件不一定造成性能损失，同时缺失事件也无法解释所有的性能损失 [17, 28]。例如，可能会观察到两个处理器的分支 MPKI、缓存 MPKI 没有明显差异，但 CPI 却存在很大差异。当这种情况发生时，为了定位性能差异，架构师不得不手动分析大量的性能计数器来定位造成性能差异或者掩盖性能差异的因素。为了加速这一过程并减少人力投入，研究人员提出了自顶向下性能计数器 [16, 29]。

自顶向下性能计数器的理论基础是区间分析（Interval analysis），它将 CPU 执行状态分为基本状态、阻塞状态和空闲状态。基本状态是指令正常分发的情况，阻塞状态是由于流水线后端出现长延迟指令导致的流水线阻塞，空闲状态是由于前端指令供应不足或流水线冲刷导致无法分发指令 [16]。然后在此基础上进一步细分，空闲或阻塞的原因可以归因于等待指令 TLB（I-TLB）、等待指令缓存（I-Cache）、分支预测错误冲刷以及指令碎片化等问题（如图 4-2所示）。

针对后端阻塞的归因一般比较明确，文献 [16] 和文献 [29] 中都有详细的分类和描述。然而，不同自顶向下方案在分支预测错误造成的性能损失统计方法上存在差异。文献 [16] 使用前端缺失事件表（Front-end miss event table, FMT）来追踪每条分支指令状态。当正常提交时，将 FMT 中记录的 I-TLB 和 I-Cache 时间加到全局计数器中；当发现预测错误时，则需要将 FMT 中记录的该分支预测

错误损失的时间加入全局计数器。为准确统计分支预测错误损失，FMT 中每项惩罚计数器都需逐周期累加，这也是文献 [16] 最复杂的部分之一。

为了避免在 FMT 中针对每条分支指令都统计推测错误惩罚，文献 [29] 提出了另一种统计推测错误的方法。该方法通过全部发射指令数减去全部提交指令数来估算在分支预测错误的路径上执行指令的性能损失。需要注意，该方法导致两种统计方法之间存在微小的差异：推测错误路径上阻塞事件，在文献 [16] 中被记为推测错误惩罚，而在文献 [29] 中则被视作后端阻塞造成的性能损失。在整体上，本文遵循了文献 [16] 的统计思路，但是在统计分支预测错误惩罚的方法上，本文借鉴了文献 [29] 的方法来降低复杂度。

4.2.4　展开式自顶向下分析框架的动机

本节首先介绍因为自顶向下分析方法没有显式地描述事件之间的重叠和并行而片面地归纳性能瓶颈或者错误定位性能瓶颈的问题。然后介绍，自顶向下性能计数器与自底向上性能计数器相互割裂的问题。

在 CPU 流水线中，分支预测错误导致的指令气泡与后端推测错误恢复的阻塞会发生重叠，现有的自顶向下性能计数器对待重叠的事件时，同一个时间片发生的阻塞或者空闲只能被归因到一种原因。这会导致在 CPI stack 上只会出现二者之一，从而让架构设计者误判性能瓶颈。

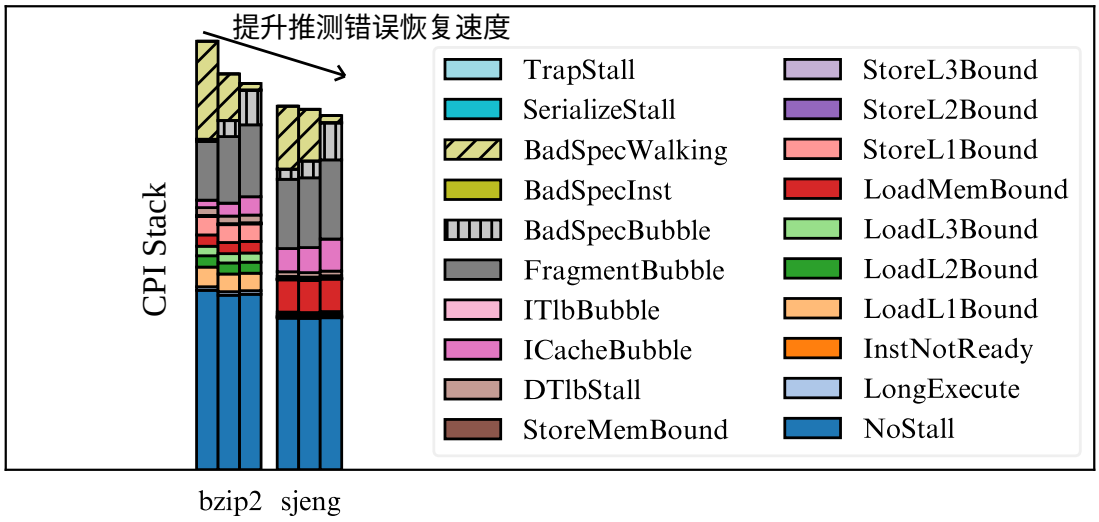


图 4-3 优化推测错误恢复机制对 *bzip2* 和 *sjeng* 的 CPI stack 的影响

Figure 4-3 The impact of optimizing the recovery mechanism for mis-speculation on the CPI stack of *bzip2* and *sjeng*.

例如，自顶向下的 CPI stack 总是优先统计推测错误后的后端阻塞（BadSpecWalking），而分支预测错误导致的指令气泡（BadSpecBubble）容易在 CPI stack 中被忽略。图 4-3展示了用两种方法改进推测错误的恢复速度后的 CPI stack，从左到右的三根柱子分别表示了改进前、第一种改进方法和第二种改进方法的 CPI stack。图 4-3说明在一定范围内改进推测错误的恢复速度能获得显著的收益，

但是超过一定范围后，CPI stack 上分支预测错误导致的指令气泡就会显著增加，无法进一步降低 CPI。事实上，优化后端的推测错误恢复一般不影响分支预测错误的指令气泡，CPI stack 中的分支预测错误气泡成分增加是因为在推测错误所导致的后端阻塞减少之后，被掩盖的分支预测错误气泡在 CPI stack 中暴露出来。

这种掩盖在 CPI stack 中放大了推测错误后的后端阻塞的影响，低估了分支预测错误气泡的影响。如果架构设计者尝试优化分支预测错误气泡，整体性能和 CPI stack 可能都不会发现显著变化，从而错误地得出“优化分支错误气泡作用较小”的结论。

造成该问题的根本原因是自顶向下过度强调成分之间的可加性，而被迫选取了可加的指标。这样设计造成了两个缺陷：

- 为了让事件计数器之和等于程序的总执行周期数，不得不丢弃相互重叠的事件中部分的计数器值。
- 为了使事件计数器之间可加，只能纳入以周期数为量纲的性能计数器。

其中缺陷 1 导致了上述的 CPI stack 的成分发生重叠而掩盖性能瓶颈的问题。而缺陷 2 则限制了自顶向下的表达能力，与自底向上计数器相互割裂。

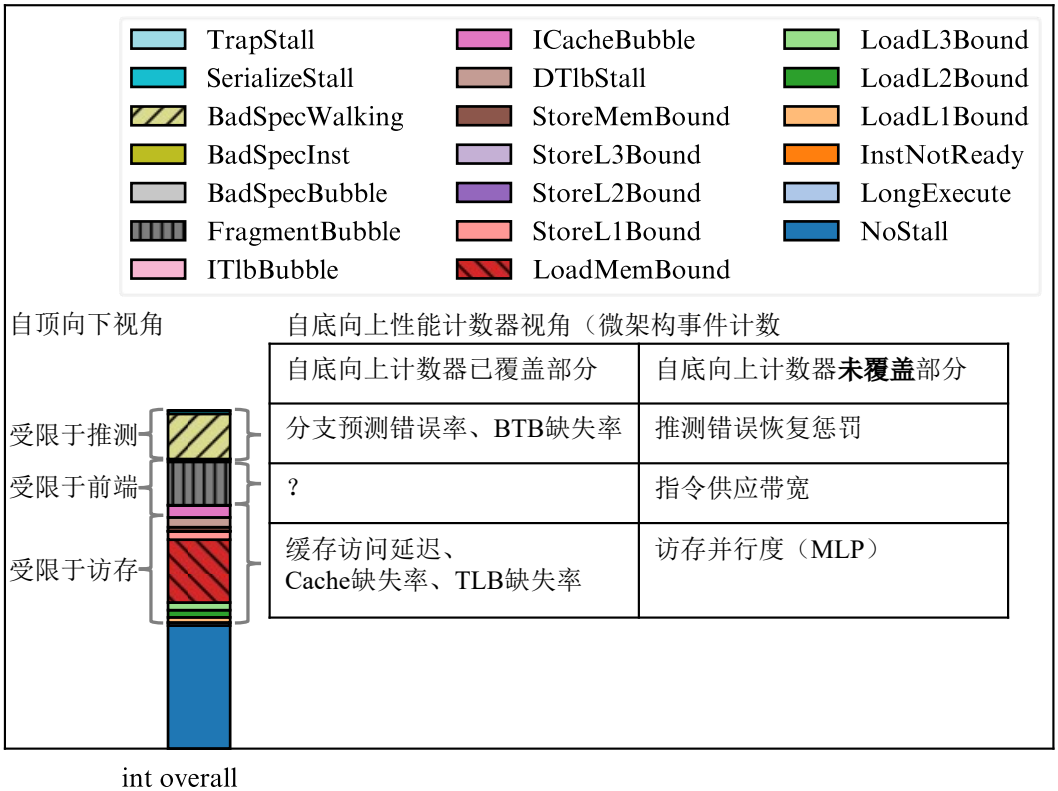


图 4-4 自顶向下性能计数器与自底向上性能计数器的关联

Figure 4-4 The relationship between top-down performance counters and bottom-up performance counters.

图 4-4展示了 SPECCPU® 2006 的整数测试集的 CPI stack 构成，其中受限于

推测错误、受限于前端和受限于访存的占比较大。但是要解释这部分的性能瓶颈所需的性能计数器几乎都没有被囊括在自顶向下性能计数器中，而经典的性能计数器只能部分解释上述性能瓶颈。例如受限于推测错误的周期数，不仅与分支预测器 MPKI 有关，也和推测错误的恢复惩罚有关；受限于访存的周期数，不仅与各级访存延迟和缓存 MPKI 有关，也和访存并行度有关。

这些具体的微架构事件相关的事件计数器并不能简单相加。如果简单地把推测错误恢复、访存缺失等事件的周期数等相加，会得到一个大于程序总执行周期数的结果。因为这些事件之间存在大量的重叠和并行。实际上，文献 [28, 82] 都表明简单地叠加微架构事件的周期数难以预测程序的性能。

为了解决上述问题，本文提出了展开式自顶向下分析框架，它在自顶向下分析方法的基础上，利用除法来描述事件的并行度对事件造成的性能损失的影响，利用 max 函数来描述事件之间的重叠。它使得不可加的计数器可以纳入自顶向下分析框架中，并且使得事件的并行与重叠能够被准确地表达。

4.3　展开式自顶向下分析框架

4.3.1　展开式自顶向下分析概览

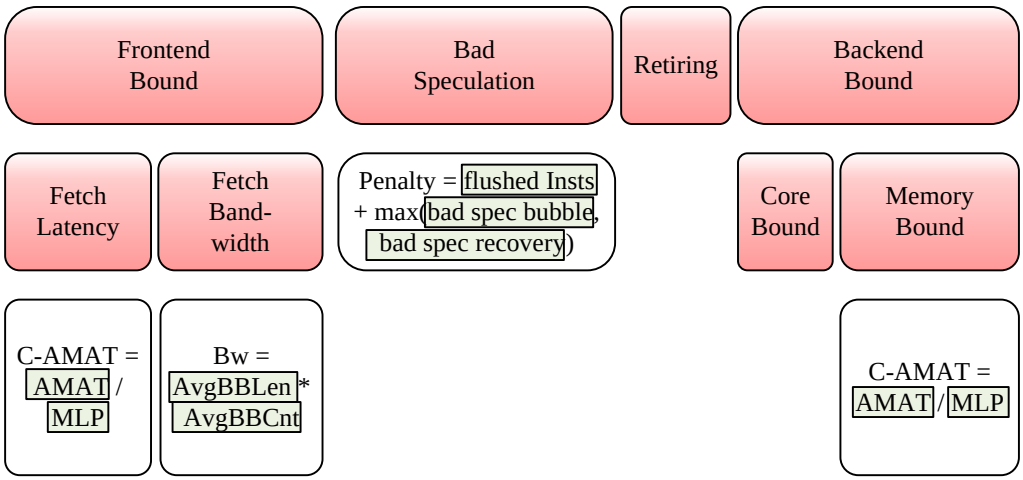


图 4-5 展开式自顶向下分析方法

Figure 4-5 The elaborated top-down analysis method.

图 4-2和图 4-5分别展示了传统的自顶向下分析方法和展开式自顶向下分析方法。图 4-5用带有底色的矩形标记出了两者的相同之处：保留了顶层的四种成分以及第二层对受限于前端和后端的子成分的划分。这部分未改变的成分在图 4-5中省略，其余节点在展开式自顶向下分析中被重新组织。

展开式自顶向下分析与文献 [29] 最大区别在于，它的叶子成分不再是单纯地将时间片相加，而是时间片和其它类型的计数器或指标通过数学关系进行组合。这允许我们描述高性能处理器的复杂设计带来的事件重叠与并行。

具体地，推测错误的子成分被明确地划分为三部分：推测错误的指令气泡（redirect bubble），冲刷指令损失的时间片（flushed slots）和推测错误恢复时后端的阻塞时间（recovery stall）。取指带宽的子成分被划分为平均基本块长度和平均基本块供应数量。取指延迟与访存受限的子成分被划分为平均访存延迟（AMAT）和访存并行度（MLP）。

在本节的后面部分，本文首先会详细介绍展开式自顶向下分析框架的顶层节点与子成分之间的复杂定量关系，通过放弃子成分之间可加性实现了对事件重叠与并行的描述，从而更准确地描述各叶子指标的对性能的最终贡献。此外，根据本研究的观察和相关工作 [25, 82, 129]，都发现访存并行度是对系统性能影响最大的因素之一。为此，本文针对访存并行度设计了性能计数器和微观的访存并行度异常检查工具。

4.3.2 展开式自顶向下分析模型

4.3.2.1 展开推测错误惩罚

在推测错误惩罚的子成分统计时，展开式自顶向下分析与文献 [29] 的做法略有不同：在文献 [29] 中，推测错误的指令气泡被统计到前端子成分中，而冲刷指令和推测错误恢复的损失被合并统计。而在展开式自顶向下分析中，本文将三种性能损失严格区分开，按照如下方式定义它们：

- 冲刷指令损失的时间片：因推测错误而被冲刷的指令在分发时所占用的时间片。
- 推测错误恢复的阻塞时间：推测错误后遍历重排序缓冲区（ROB）以恢复重命名表所耗费的时间。
- 重定向指令气泡：推测错误被检出后，指令流被重定向，导致取指缓冲区和流水线中的指令被冲刷，从而形成气泡。

文献 [29] 中，三者是简单的加和关系，但是这与高性能处理器的实际情况不符。高性能处理器中，推测错误恢复的阻塞和重定向指令气泡是相互重叠的，最终的性能损失一般由二者中较大的决定。因此在 CPI stack 的柱状图中，一定存在重叠的部分。为了避免忽视被掩盖的部分的重要性，本文采用公式而非柱状图来描述推测错误惩罚：

$$\text{bad speculation penalty} = T_{\text{flushed insts}} + \max\{T_{\text{recovery}}, T_{\text{redirect bubble}}\},$$

$$T_{\text{flushed insts}} = \#\text{mispred} \times \text{branch resolve latency},$$

$$T_{\text{recovery}} = \#\text{mispred} \times \#\text{recovery cycle per mispred},$$

$$\#\text{recovery cycle per mispred} = \frac{\#\text{instructions to walk}}{\text{walking width}},$$

$$T_{\text{redirect bubble}} = \#\text{mispred} \times \#\text{bubble per mispred},$$

$$\#\text{mispred} \propto \text{branch MPKI}。$$

整理以上关系可得

$$\text{bad speculation penalty} \propto \text{branch MPKI} \times (\text{branch resolve latency} + \max\{\#\text{recovery cycle per mispred}, \#\text{bubble per mispred}\})。 \tag{4-1}$$

图 4-6将这种相互重叠关系可视化。用矩形的面积来表示推测错误惩罚，矩形的高正比于每千条分支错误率（branch MPKI），矩形的长取决于分支解决速度、错误恢复速度和重定向气泡长度之和。错误恢复速度和重定向气泡长度之间可以相互掩盖，总面积取决于更大的一方。

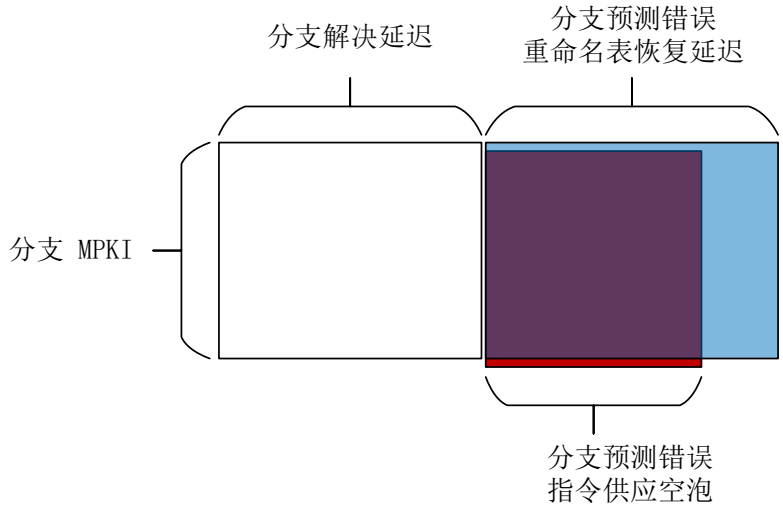


图 4-6 推测错误惩罚的可视化

Figure 4-6 Visualization of bad speculation penalty

4.3.2.2　展开取指带宽

目前的高性能处理器大多采用了解耦前端 [34, 130–132]，如图 4-7所示，在解耦前端中，取指与分支预测可以独立运行。本文将取指带宽定义为分支目标缓冲区（BTB）命中、指令缓存命中、后端无阻塞时的指令供应能力。而不满足该条件时的性能损失，被归类到其它成分中。

分支预测、取指和派送指令到译码这三个过程分别可能因为（BTB）缺失、指令缓存缺失和后端阻塞而被打断。其中 BTB 缺失时，会影响分支预测的准确率，造成的性能损失被归类到推测错误大类中。指令缓存缺失时，会造成指令供应空白，造成的性能损失被归类到取指延迟大类中。后端阻塞导致指令无法译码时，造成的性能损失被归类到受限于后端大类中。

取指带宽在文献 [29] 中被划分为 *fetch1* 和 *fetch2* 两部分，而事实上，指令供应带宽不仅受限于从指令缓存取指的速度，还受限于分支预测器的速度。展开式自顶向下分析对此进行了更详细的划分。

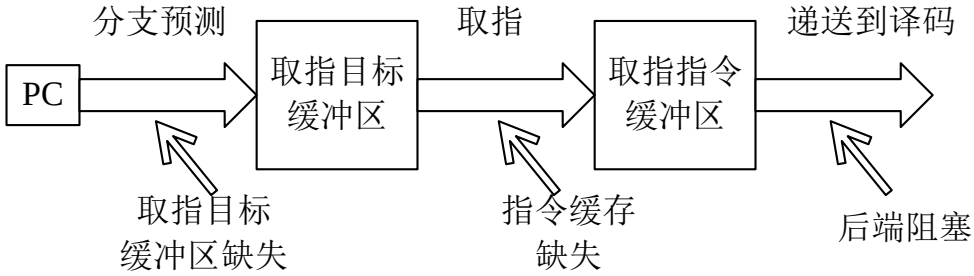


图 4-7 解耦前端的指令供应流水线。1) 分支预测器根据 PC 或者基本块地址预测下一次取指目标，送到分支目标缓冲区（Branch target buffer，BTB）中 [34]。如果分支预测过程中取指目标缓冲区缺失，则会产生一个平凡的预测，即预测指令流不会发生分支跳转。2) 取指级（Fetch）根据取指目标从指令缓存中获取指令，将指令存放到指令缓冲区（Fetch inst buffer）中。如果指令 TLB 或者指令缓存缺失，则需要等待缺失回填。当缺失发生时，若指令缓冲区中没有内容，则会形成流水线空泡。3) 指令缓冲区中的指令被送到译码阶段（Decode stage）进行译码。当流水线后端发生阻塞时，该过程可能被阻塞。

Figure 4-7 Instruction supply pipeline in decoupled front-end. 1) Branch predictor predicts the next fetch target based on PC or basic block address, and sends it to fetch target buffer [34]. If BTB miss occurs during branch prediction, a trivial prediction will be made, i.e., predicting this branch will not be taken. 2) Fetch stage fetches instructions from instruction cache according to fetch target, and stores them in fetch instrcution buffer. If I-TLB miss or I-cache miss occurs, the process will be stalled. If the fetch instruction buffer is empty when miss occurs, pipeline will be stalled. 3) Instructions in fetch instruction buffer are sent to decode stage for decoding. When pipeline backend is stalled, this process will be stalled.

本文把受限于指令供应带宽的性能损失称为碎片化气泡（Fragment bubble）。无阻塞的指令供应能力取决于每周期能供应的基本块个数和被供应的基本块的平均长度。

$$\text{fragment bubble per cycle} = \text{dispatch width} - \text{Avg basic block length} \times \text{Avg basic block count}, \quad (4\text{-}2)$$

其中，**被供应的基本块的平均长度（Avg basic block length）**有可能短于程序本身的基本块长度，因为一个基本块可能横跨多个指令缓存块，导致基本块被截断。而**每周期能供应的基本块的个数（Avg basic block count，ABBC）**取决于分支预测器的预测带宽和指令缓存的访存带宽。

$$\text{ABBC} = \min\{\text{branch prediction bandwidth}, \text{inst cache bandwidth}\}。 \quad (4\text{-}3)$$

等式 4-3表明 *ABBC* 受限于分支预测器的预测带宽。而分支预测器的预测带宽与流水线前端的微架构设计紧密相关。现代的分支预测器可在一个周期预测

多个基本块，例如部分微架构可以预测两个或者多个连续的无跳转的基本块 [34, 130–132]，甚至还有微架构可以在一个周期内预测多个发生跳转的分支 [4]。考虑到分支预测器的预测带宽与具体的微架构设计密切相关，本文不再对 *ABBC* 公式进行进一步展开。

4.3.2.3　展开访存延迟相关成分

对受限于访存和前端取指延迟的性能损失，展开式自顶向下分析框架采用基于并行平均访存并行度的方式进行建模。本文会建立该大类性能损失与访存并行度（MLP）、各级缓存访存延迟和缓存缺失率之间的数学关系，从宏观和微观两个层面分析 MLP 对访存性能的影响。此外，本文总结了 MLP 与常见的微架构部件的关系，作为 MLP 异常时的检查单（check list）。

从宏观层面来看，并行平均访存延迟是由整体的平均访存延迟和访存并行度共同决定的：

$$CAMAT_{Macro} = \frac{AMAT}{MLP},$$

因为访存请求之间的并行在各级缓存和主存中普遍存在，并且各级缓存的并行度不同，所以全局的平均访存并行度缺乏对微架构设计的指导意义。为了更好地指导微架构设计，本文需要将并行平均访存延迟分解为更细致的性能指标。

文献 [82] 证明了给定访存层次的 $CAMAT$ 可以按如下方法计算：

$$CAMAT = HitRate \times \frac{Latency}{Parallelism_{hit}} + MissRate \times \frac{AvgMissPenalty}{Parallelism_{miss}}, \quad (4\text{-}4)$$

其中 $MissRate$、$HitRate$ 和 $Latency$ 分别对应缺失率、命中率和命中后访问延迟，$Parallelism_{hit}$ 和 $Parallelism_{miss}$ 分别是命中和缺失的访存并行度。作为对比，原始的平均访存延迟（AMAT）如下：

$$AMAT = HitRate \times Latency + MissRate \times AvgMissPenalty, \quad (4\text{-}5)$$

对等式 4-4进行展开，可以从 CPU 核的视角计算各级缓存的并行平均访问延迟（$CAAT_{level}$）：

$$CAMAT = CAAT_{L1} = HitRate_{L1} \times \frac{Latency_{L1}}{Parallelism_{L1}} + MissRate_{L1} \times CAAT_{L2}, \quad (4\text{-}6)$$

$$CAAT_{L2} = HitRate_{L2} \times \frac{Latency_{L2}}{Parallelism_{L2}} + MissRate_{L2} \times CAAT_{L3}, \quad (4\text{-}7)$$

$$CAAT_{L3} = HitRate_{L3} \times \frac{Latency_{L3}}{Parallelism_{L3}} + MissRate_{L3} \times CAAT_{Mem}, \quad (4\text{-}8)$$

$$CAAT_{Mem} = \frac{Latency_{Mem}}{Parallelism_{Mem}}。 \quad (4\text{-}9)$$

根据上述公式，可得如图 4-8所示的计算树，树的根节点是并行平均访存延迟，而叶子节点是各访存层级的缺失率、访问延迟和访存并行度。文献 [82] 表明并行平均访存延迟（C-AMAT）与访存受限程序的性能的相关性比平均访存延迟（AMAT）更高。因此，上述分析方法相比于传统的自顶向下分析框架能更好地定位访存受限的原因。

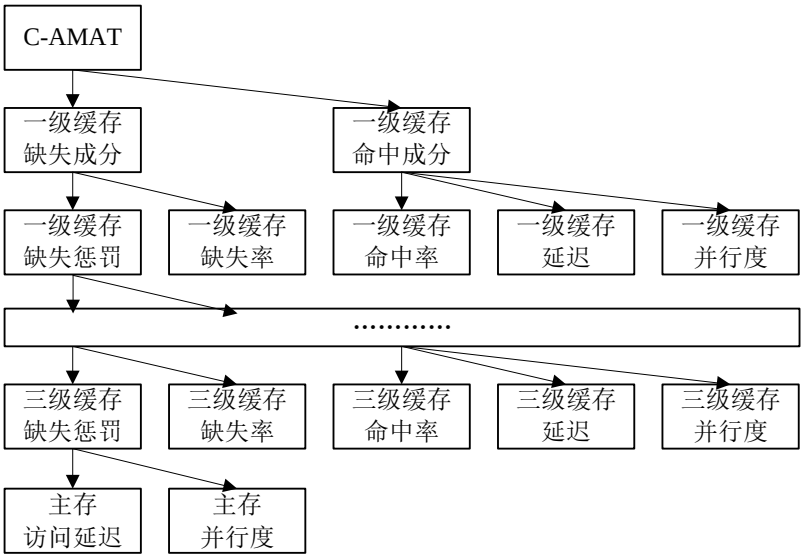


图 4-8 并行平均访存延迟取决于各访存层级的缺失率、访问延迟和访存并行度。

Figure 4-8 The concurrent average memory access time (C-AMAT) is determined by the miss rate, cache access latency, and memory-level parallelism of each memory hierarchy.

4.3.3 访存并行度检查表

在 4.3.2中，本文详细介绍了展开式自顶向下分析框架，其中访存并行度相关的部分是它的重要组成部分。当访存并行度发生异常下降时，现有的工作无法将访存并行度下降直接与具体的微架构部件联系起来。文献 [82] 中归纳了一些影响访存并行度的因素：发射宽度和 MSHR 数量，但是这些因素并不能完全解释所有的访存并行度异常下降的场景。文献 [29] 将访存并行度的异常下降导致的访存时间增加归类到各级缓存，可能导致性能瓶颈被错误归因。

在实际工作中，访存并行度相关的性能缺陷往往对性能造成较大的影响，但是因为影响访存并行度的因素很多，很难快速定位到具体的原因。为了快速定位访存并行度异常下降的原因，本文总结了访存并行度与微架构部件的相关矩阵（表 4-2）。在检查访存并行度时，由于外层缓存、主存的并行度主要受限于内层缓存的并行度，故应当遵循“从内至外”的原则。

一级缓存访存并行度可能受多方面影响：程序本身的并行度、指令窗口大小、访存推测算法和一级缓存吞吐。程序本身的并行度和指令窗口大小对访存并行度的影响在此前的工作中已有研究 [129]。一级缓存吞吐与地址生成器的数

量、缓存的读口数量、缓存的分 bank 机制有关。访存推测算法有可能产生假依赖，导致错失并行访存的机会。

当发现二级缓存访存并行度异常时，要先确认上述的一级缓存并行度相关因素没有问题。然后检查一级缓存 MSHR 和二级缓存吞吐。而三级缓存和主存的访存并行度异常也要遵循类似的顺序。需要注意的是，内层缓存的并行度异常下降时，会导致所有更外层的缓存的并行度异常下降。在很多访存密集型应用中，三级缓存和主存访问的成分的占比较大，当访存并行度异常时，这部分成分的增幅也比内层缓存更为明显。这可能误导架构设计者，认为性能损失归因于三级缓存和主存，从而遗漏对内层缓存并行度的检查。而把表格 4-2作为检查表，可以避免这种情况。

表 4-2 访存并行度与微架构部件的相关性矩阵

Table 4-2 The matrix of the relationship between memory-level parallelism and micro-architecture components.

	一级缓存并行度	二级缓存并行度	三级缓存并行度	主存并行度
程序依赖	✓	✓	✓	✓
指令窗口大小	✓	✓	✓	✓
访存推测算法	✓	✓	✓	✓
发射宽度和一级缓存吞吐	✓	✓	✓	✓
一级缓存 MSHR 和二级缓存吞吐		✓	✓	✓
二级缓存 MSHR 和三级缓存吞吐			✓	✓
三级缓存 MSHR 和内存并行度				✓

4.3.4　访存并行度异常定位

虽然本文通过分析模型描述了受限于访存的性能损失与各级缓存的并行度之间的关系，并且给出了访存并行度与微架构部件的相关性矩阵，但是由于影响访存并发的因素过多，即使通过计数器定位到了访存并行度异常的访存层级，距离找到具体的性能瓶颈或性能缺陷仍有一段距离。

正如软件工程中的 bug 定位一样，重现性能缺陷发生的现场是定位性能缺陷的关键。为了精确定位到性能缺陷或性能瓶颈的现场，本文提出了基于并行访存缺失簇（Parallel miss cluster，PMC）的 MLP 异常现场检测方法。首先，为了观测微观的访存并行度，本文定义了 PMC，它是同一个指令窗口内可以并行的访存缺失的集合。为了从海量的 PMC 中找到能够暴露出性能瓶颈或性能缺陷的 PMC，本文给出了不同场景下对 PMC 进行筛选的算法。

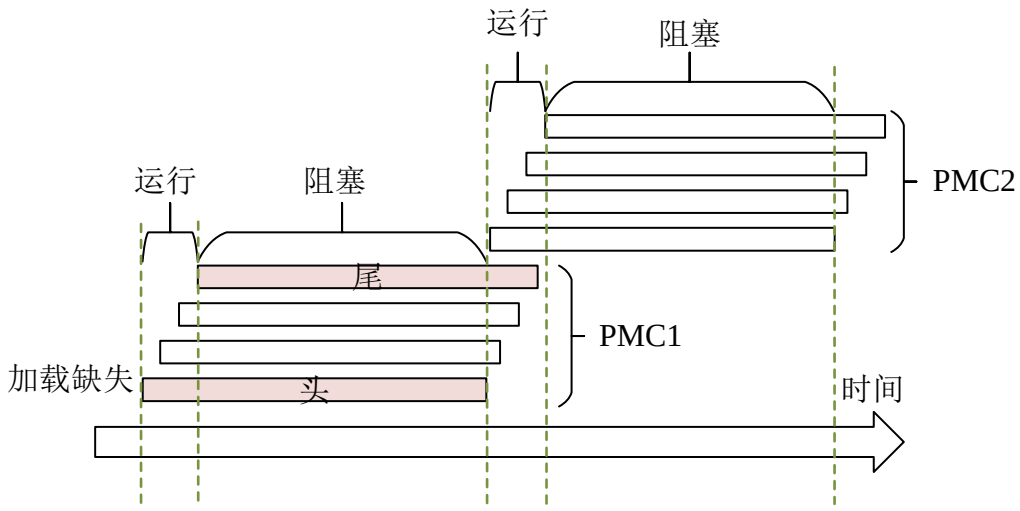


图 4-9 并行访存缺失簇示意图

Figure 4-9 The illustration of parallel miss clusters.

4.3.4.1 并行访存缺失簇的定义

图 4-9展示了两个 PMC 形成的四个区间：运行区间-阻塞区间-运行区间-阻塞区间。这里的区间的定义标准与文献 [28] 中的区间分析定义很相似：运行区间是有指令分发的区间，而阻塞区间是没有指令分发的区间。经观察发现，长延迟访存与区间分析有着密切的关系。因为片外访存延迟很大，长延迟访存必然造成流水线的阻塞，所以一个长延迟访存必然覆盖一个区间分析中的阻塞区间。

事实上 PMC 并非是唯一访存并行度的微观划分标准。在文献 [82] 中，作者在介绍访存并行度的统计方法时，将程序划分为了命中段（hit phase）和完全缺失段（pure miss phase），但是完全缺失段是为了准确统计访存并行度设计，而不是为了定位性能异常而设计。在访存密集型应用中，长延迟访存的并行度比片内的并行度更重要，因此以定位访存并行度异常为目的时，不需要考虑片内是否存在其它访存请求命中。

4.3.4.2 并行访存缺失簇划分方法

多数情况下，若同一个指令窗口内存在若干个可并行的长延迟访存，这些长延迟访存会在第一个长延迟访存响应之前被发射（如图 4-10(a) 所示）。这是因为从长延迟访存到流水线阻塞的时间一般只有数十个周期（香山处理器在典型配置下是 43 周期阻塞），只有当某个长延迟访存的前置指令依赖链条超过 150 个周期才能导致它的发射时间晚于 PMC 头部访存的响应（如图 4-10(b) 所示）。基于这样的观察，本文通过第一条长延迟访存的发射时间和响应时间来划分 PMC：所有在第一条长延迟访存发射之后、响应之前发射的长延迟访存都被划分到同一个 PMC。具体划分方法在算法 1中详细描述。

算法 1 将长延迟访存划分为并行访存缺失簇

算法 1 Dividing long-latency memory accesses into parallel miss clusters.

1: **procedure** PMCDIVIDER($loads$)

Require: $loads$: List of loads with latencies: [(start, end, SN)]

Ensure: List of parallel miss clusters ($PMCs$)

2: $PMCs \leftarrow \phi$ ▷ 输出的 PMC 列表

3: $workingPMC \leftarrow \phi$ ▷ 当前正在收集的 PMC

4: $restLoads \leftarrow loads$ ▷ 剩余的没有完成 PMC 划分的访存

5: $skidLoads \leftarrow \phi$ ▷ 属于当前指令窗口但是因为发射时间点太晚而无法纳入 PMC 中的访存

6: $workingPMCHead \leftarrow null$ ▷ 当前正在收集的 PMC 的最老的访存

7: **while** $restLoads \neq \phi$ **do**

8: **if** $workingPMC = \phi$ **then**

9: $workingPMCHead \leftarrow restLoads.popHead()$ ▷ 将最老的访存设为当前 PMC 的头

10: $workingPMC \leftarrow \{workingPMCHead\}$

11: **end if**

12: **for all** $load \in restLoads$ **do**

13: **if** $load.NS \leq workingPMCHead.SN + WindowSize$ **then** ▷ 完成了对当前指令窗口的遍历 **break**

14: **else if** $load.start \leq workingPMCHead.end$ **then** ▷ 该访存的发射时间太晚，不能纳入当前 PMC

15: $skidLoads \leftarrow skidLoads \cup \{load\}$

16: **else** ▷ 发射时间早于头指令响应，并且属于当前指令窗口

17: $workingPMC \leftarrow workingPMC \cup \{load\}$

18: **end if**

19: **end for**

20: $PMCs \leftarrow PMCs \cup \{workingPMC\}$ ▷ 完成当前 PMC 的构建

21: $workingPMC \leftarrow \phi$

22: $restLoads \leftarrow restLoads \cup skidLoads$ ▷ 将上一轮的 PMC 排除的访存重新加入 restLoads，以便下一轮的遍历

23: $skidLoads \leftarrow \phi$

24: $workingPMCHead \leftarrow null$

25: **end while**

26: **end procedure**

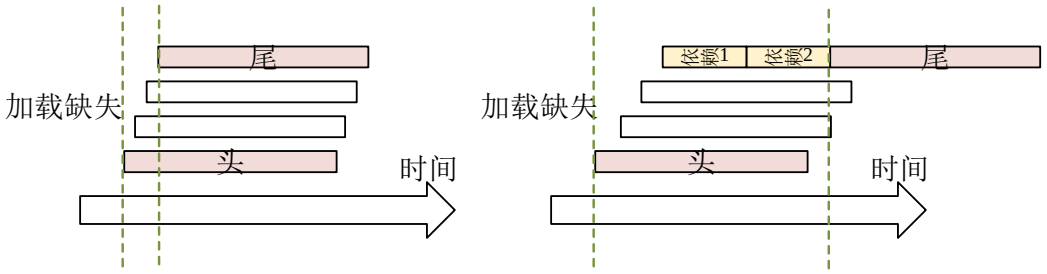


图 4-10 同一个 PMC 内的长延迟访存的发射时间相距可能较短 (a) 或较长 (b)

Figure 4-10 The illustration of two cases of parallel miss clusters: the interval between the two long-latency memory accesses might be short (a) or long (b).

4.3.4.3 筛选性能异常的并行访存缺失簇

完成 PMC 划分后，需要从大量的 PMC 中寻找性能异常的 PMC。然而由于每个 PMC 可能对应不同的程序片段（异质性），无法通过一个简单的 MLP 阈值来判断 PMC 是否有性能异常。面对 PMC 的异质性，本文的思路是为 PMC 选取参照对象，通过与参照对象对比并行度来判断 PMC 是否存在异常。参照对象可以是性能参考模型的 PMC，也可以是同一个指令片段的多次执行产生的 PMC。

当有性能参考模型时，PMC 对比相对简单：通过对比目标 CPU 和性能参考模型在同一个 PMC 上的 MLP 来判断是否存在异常。由于目标 CPU 和性能参考模型在微架构等方面可能存在差异，所以二者的 PMC 可能不完全一样。在此背景下，为了观察同一个 PMC，本文对二者的 PMC 列表取交集，从而找到对应了相同指令片段的 PMC 进行对比。相同指令片段定义为：执行同一个程序片段时，来自两个平台的两个 PMC 的起始指令的指令提交序号相同。区间分析的原理决定了两个微架构相似平台的 PMC 重合度一般较高。

当没有性能参考模型时，本文选取同一个指令片段的多次执行产生的多个 PMC 作为相互参照的对象。通常情况下，程序中存在大量的循环或者递归，而且相同的程序片段的访存依赖在大部分情况下是维持不变的 [133, 134]，因此同一个程序片段多次执行的访存并行度较为相似。进而，可以推论出同一个指令片段多次执行所得的 PMC 也会较为相似。

接下来介绍寻找同一个指令片段的多次执行产生的 PMC 的方法。首先，统计所有的 Load PC 发生长延迟缺失的次数，得到按照从大到小排序的列表 $loadsMissCounts$。从 $loadsMissCounts$ 中依次取出每个 Load PC，收集 PC 所在的 PMC，得到 PMC 集合 $topPMCs$。计算整个集合的 MLP 分布和平均 MLP，如果 MLP 分布出现了多个峰或集合中出现了 MLP 显著低于平均 MLP 的 PMC，就怀疑该 PMC 对应的指令片段存在性能缺陷或性能瓶颈，需要对该指令片段的

log 或者波形进行分析。

4.4　将展开式自顶向下分析框架应用到模拟器对齐

展开式自顶向下分析可以用于对齐微架构模拟器和处理器的 RTL 实现。虽然通过展开式自顶向下分析的 CPI stack，可以快速定位引起性能差异的源头，但它不是一枚“银弹”，还需要与其它方法相结合。这是因为当模拟器与 RTL 的架构差异较大时，CPI stack 所反映的性能差异可能缺乏参考价值。因此，首先需要对模拟器和 RTL 进行基本的架构参数对齐，然后才能使用展开式自顶向下分析和访存并行度分析工具来加速定位余下的性能差异。本章先介绍模拟器对齐的整体流程，然后介绍基本性能对齐部分的性能测量方法的实现。

4.4.1　模拟器对齐流程

对微架构模拟器和 RTL 实现进行基本架构参数对齐是对齐的第一步。和以前的模拟器对齐的工作相似 [13, 14]，本文首先通过微基准测试来对齐基本的指令延迟、访存延迟和分支预测错误惩罚。因为香山处理器的设计仍在不断演进，为了使上述延迟对齐更加自动化、避免流水线后端的改动导致对齐失效和重复劳动，本文设计一套独立于香山的 RTL 代码的测试集，MicroScope。以及基于 MicroScope 测量结果的自动化对齐工具 – LatencySolver。LatencySolver 根据测试集测得的指令延迟自动得出 GEM5 的延迟配置。

完成延迟对齐后，本文对分支 MPKI、缓存 MPKI 等常见的微架构性能指标进行了采集和分析。通过分析分支 MPKI 和访存 MPKI 存在显著差异的测试集子项，本文定位到了 GEM5 的预取器的性能缺陷、GEM5 的指令译码 bug，也为香山处理器发现了 DCache 替换算法的性能 bug。此外，本文还对齐了两个平台的解耦前端架构、分支预测算法、缓存预取算法和 TLB 架构。[1]

对齐上述成分并不能消除所有的性能差异，例如访存并行度差异。如图 4-11所示，在 C-AMAT 的计算树中，对齐基本延迟只能对齐图 4-11(a) 中的访存延迟；对齐缓存 MPKI 只能对齐图 4-11(b) 中的访存缺失率。根据 C-AMAT 的计算公式，还需要对齐访存并行度才能完成 C-AMAT 的对齐（图 4-11(c)）。为此，需要结合展开式自顶向下分析和访存并行度微观分析工具来对齐访存并行度才能完成所有访存相关的微架构的对齐。

表 4-3列出了本文在性能模拟器搭建和对齐过程中用到的五种性能分析工具各自的原理和应用场景。其中，MicroScope 和基本性能计数器能扫除最基本的性能差异，展开式自顶向下分析框架（自顶向下性能计数器 + 访存并行度性能计数器）和基于 PMC 的访存并行度异常定位工具负责填充 C-AMAT 计算树的剩余部分，完成访存子系统的对齐。

[1]因为这部分工作为团队协作完成，在本文中未予以展开。

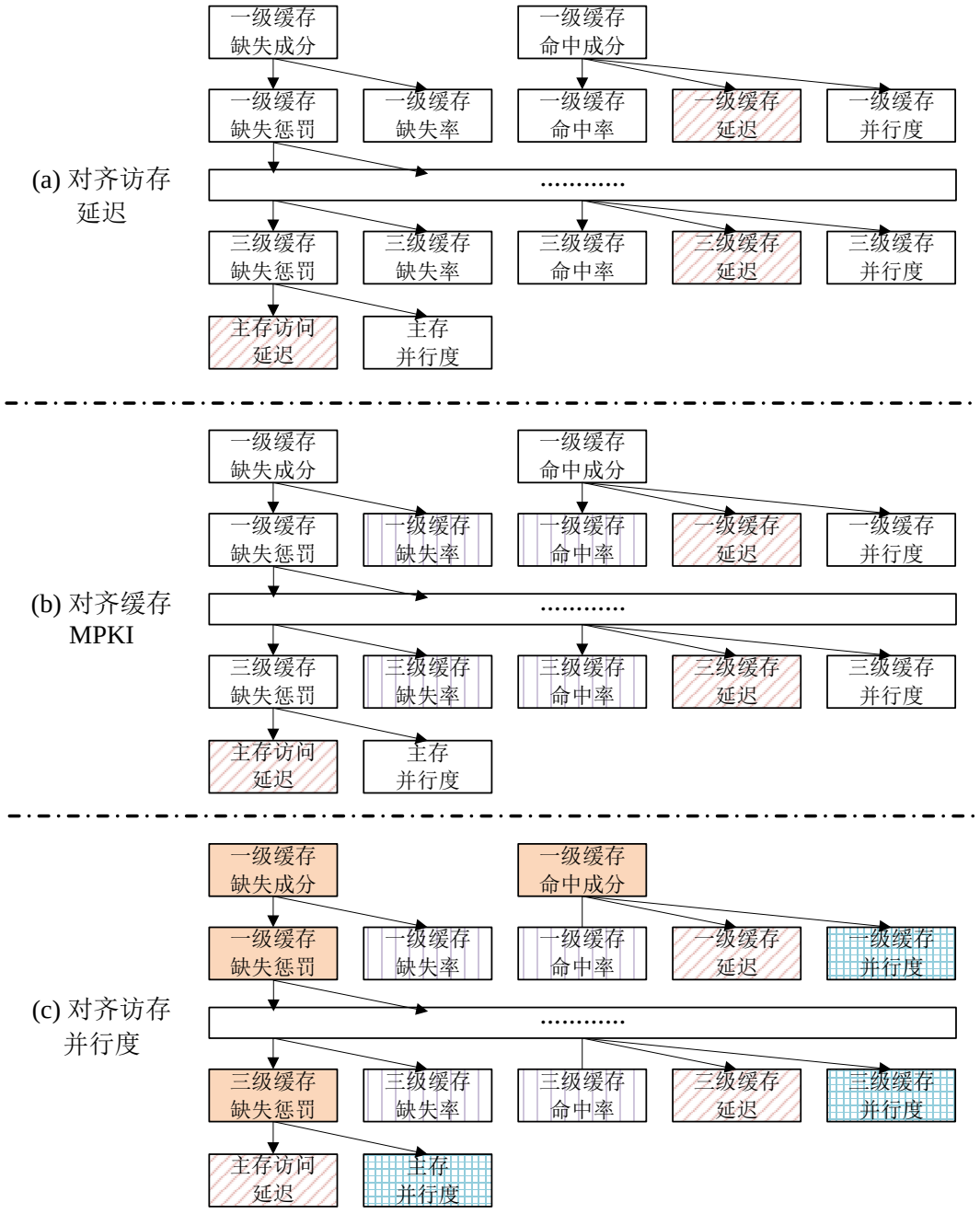


图 4-11 三部分对齐工作分别填充了 C-AMAT 的计算树的不同部分的节点。完成了访存并行度对齐后，完成了组成 C-AMAT 的所有成分。

Figure 4-11 Three steps of calibration fill different nodes of the C-AMAT tree. After calibrating the memory-level parallelism, all components of C-AMAT are calibrated.

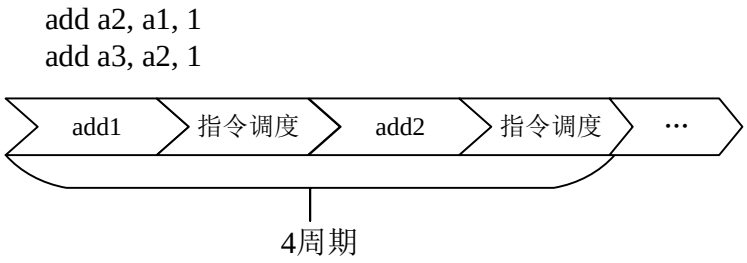


图 4-12 调度延迟给观察到的指令延迟带来的影响

Figure 4-12 How the scheduling delay affects the observed instruction delay

4.4.2　基本性能测量与对齐

MicroScope 是一套独立于 RTL 或者模拟器的测试集，直接用软件测量指令或者指令片段的延迟。LatencySolver 基于 MicroScope 测量结果计算出指令的执行延迟、不同指令组合的调度延迟、缓存的访存延迟等。

4.4.2.1　测量指令延迟

指令延迟通常是最容易测量的，但是由于高性能处理器的指令调度器经常会引入线延迟问题 [5]。如果坚持整个调度器的指令实现“背靠背”唤醒，将降低处理器的工作频率 [5, 6]。因此，实际的高性能处理器不得不放弃完整的“背靠背”唤醒 [19]，针对特定的指令组合引入额外的时钟周期。这会导致实际观测到的指令延迟与功能单元实现的延迟并不完全一致。如图4-12所示，当加法指令调度延迟为 1 时，如果从 ALU 到 ALU 有一周期的额外延迟，那么两个串行的加法指令需要四个时钟周期才能完成。此外，不同的指令组合可能会引入不同的调度延迟 [19]，这让指令延迟的测量变得更加困难。

由于指令的执行延迟和调度延迟总是相互交织，因此无法单独测量指令延迟或调度延迟。为了解决这一问题，LatencySolver 通过解整数线性方程组的方法来计算出指令的执行延迟和调度延迟。具体来说，LatencySolver 将指令的执行延迟和调度延迟视为方程组的变量，并将指令序列的执行时间视为方程组的常数项。在实践中，可能会遇到方程数量少于变量数量的情况，此时可以通过加入少量的先验知识作为约束，例如指令的执行延迟一定是正整数，指令组合间的调度延迟一定是非负整数，常见的 ALU 指令通常的执行延迟为 1。通过这种方法，MicroScope 和 LatencySolver 可以准确测量指令的执行延迟和调度延迟。

对齐过程中，需要先在香山处理器上运行 MicroScope 的指令序列，然后将 LatencySolver 得到的指令延迟和调度延迟写入架构模拟器的配置文件中，以便在仿真时使用这些延迟值。MicroScope 和 LatencySolver 对指令延迟和调度延迟的测量可以实现高度自动化。若 RTL 发生变化，只需要重新运行即可获得新的

指令延迟和调度延迟的配置文件。仅当作为先验约束的补充项发生变化时，才需要进行手动修改 LatencySolver，而 MicroScope 的测试序列本身则无需修改。

4.4.2.2 测量访存延迟

测量访存延迟需要综合考虑多个因素，包括缓存的多层级结构，TLB 的命中缺失情况和访存的模式能否被预取器识别。为了排除 TLB 的影响，MicroScope 在测量缓存延迟时只测量 TLB 命中的延迟，或直接测量物理地址的访问延迟。而为了避免被预取器识别访存模式，MicroScope 在测量缓存延迟时关闭了 RTL 和模拟器的预取器。解决上述两个问题后，访存延迟测量面临的主要挑战是如何让访存请求在期望的缓存层次上命中，并且为了多次测量消除误差，要让多次测量的请求落到同一缓存层次上。

首先，一级缓存的延迟测量相对简单，因为只要一个地址命中一级缓存一次之后，它就会一直存在于一级缓存中，不会被换出。只需要级联多个 load 指令，让他们串行地依次访问同一个地址即可。本文构造了一个指针，让该指针指向自己所在的地址，重复对指针进行解引用（如 Listing 1所示）。将这样的指令序列重复多次取平均值或者作差，就可以测量一级缓存的延迟。

```
1   lw a1, 0(a1)    ; load from *a1 into a1, assuming *a1 == a1
    lw a1, 0(a1)    ; load from *a1 into a1 again
```

Listing 1 串行访问同一地址以测量一级缓存延迟

但是，对二级和三级缓存的延迟测量有更多挑战，接下来以二级缓存为例进行说明。如果一个地址在二级缓存命中一次之后，它会被填充到一级缓存中，如果连续访存同一个地址，那么除了第一次访问以外，其他访问都会命中一级缓存。为了避免该问题，每次访问之后都必须将下次访问的地址增加至少一个缓存行的大小（因为关闭了预取器，无须担心被步长预取器预取）。

```
    ; cacheline_size = 0x40
2   ; assuming the content in each memory [x] is always x + cacheline_size
    ; assuming a1 = 0x10000
4   lw a1, 0(a1)    ; load from *a1 into a1, now a1 = 0x1040
    lw a1, 0(a1)    ; load from *a1 into a1 again, now a1 = 0x1080
```

Listing 2 串行访问增量的地址以测量二级和三级缓存延迟

在应用此测试方法之前，需要先填充要访问的内存区域，使每个缓存行的内容都是指向下一个缓存行的指针。然而，填充过程会导致填充内容在一级缓存中残留。为避免由此产生的干扰，需要使用缓存操作指令（CMO）清空一级缓存。如果 CPU 不支持 CMO 指令，则可以手动编写缓存刷新程序（CacheFlushser）来清除一级缓存的残留内容。请注意，当测量二级缓存时，不能过度冲刷以将缓存行从二级缓存中清除；同样，当测量三级缓存时，需要从一级和二级缓存刷新，但不能过度刷新以将缓存行从三级缓存中转移至主存中。

4.4.2.3　测量分支预测错误惩罚

分支预测错误惩罚包含两部分，第一部分是冲刷流水线后产生的流水线指令空泡，第二部分是恢复重命名表状态的过程。流水线空泡与恢复重命名表造成的性能损失大部分可以重叠，而最终的推测错误惩罚由二者中较大的决定。当发生推测错误时，如果从重排序缓冲区中被冲刷的指令数目较少，则一般性能损失由流水线空泡决定；否则性能损失由重命名表恢复的周期数决定。本文首先设计了预测错误后流水线空泡数量的测量方法，然后在此基础上增加重排序缓冲区（ROB）中被冲刷指令的数量，从而同时测量包括恢复重命名表所需的周期数和预测错误后流水线空泡数的预测错误惩罚。

为了测量流水线空泡造成的性能损失，本文构造了一个分支预测错误率为 50% 的微基准测试。该微基准测试中的条件分支总是依赖于一个数组 R 中的元素，用来判断某个元素是否大于 0.5。本文特意将数组 R 中的每个元素都初始化为 0-1 之间的随机浮点数，使分支方向无法被有效地预测。在一个 1000 次的循环中，每次循环都会执行一次条件分支，因此分支预测错误的期望次数为 500 次。由于循环体非常小，该程序的大部分执行时间都被流水线空泡所占据，因此可以近似地将总运行时间除以 500，来估计单次流水线空泡造成的性能损失。

上述微基准测试无法反映分支预测错误后恢复重命名表状态而阻塞流水线的性能损失，因为数组中的大部分元素都在缓存中命中，导致分支方向计算完成的时机比较早，发现错误后无需从 ROB 中冲刷很多指令。为了同时测量恢复重命名表所需的周期数和流水线空泡，本文对上述微基准测试进行了修改。在用随机的浮点数初始化数组 R 之后，利用 CacheFlusher 将数组 R 中的元素从缓存中替换掉。这样做会大幅推迟分支方向计算的完成时机，从而使得当分支预测错误时需要从 ROB 中冲刷大量指令，进而测试包含 ROB Walk 的访存推测错误惩罚。Listing 3描述了本文用来测量分支预测错误惩罚的微基准测试。

```
    R[i] = random_float(0.0, 1.0);
    flush_to_memory(R);
    for (int i = 0; i < 200; i++) {
        if (R[i] > 0.5) {
            // do something
        }
    }
```

Listing 3 测量分支预测错误惩罚的微基准测试

4.4.3　访存并行度的硬件性能计数器

为了快速确认访存并行度是否存在显著差异，本文引入了访存并行度测量技术，它包含两种测量机制：用在线的性能计数器统计发生流水线阻塞时的在途访存缺失数，得到整体的访存并行度；用离线访存记录基于并行访存缺失簇（PMC）来定位访存并行度发生异常的现场。其中后者已经在 4.3.4节中介绍，本节介绍在线性能计数器的实现方法。

如 Listing 4所示，MLP 性能计数器在每一级缓存统计该级缓存的在途缺失数量，增加对应的计数器。统计时需要注意两点：一是只在流水线阻塞时增加计

数器；二是只统计来自 CPU 的缺失，而不统计来自预取器的缺失。此方法统计出的 MLP 以分布的方式呈现，数组 MLPDist 的第 N 个元素的含义是在流水线阻塞时，有 N 个在途的缺失请求。MLP 分布越靠近 MaxMSHRCount，说明在流水线阻塞时有越多的在途缺失请求，说明系统的访存并行度越高。

```
MLPDist = int[MaxMSHRCount];
for_each cycle:
    if (core.pipeline_stall)
        for cache in multiLevelCaches:
            int miss_from_cpu = cache.miss.count - cache.miss.prefetch_count;
            if (miss_from_cpu > 0)
                MLPDist[miss_from_cpu]++;
```

Listing 4 MLP 性能计数器统计方法

表 4-3 在香山的架构模拟器对齐过程中用到性能分析工具

Table 4-3 Performance analysis tools used for calirating the simulator of Xiangshan processor

	MicroScope	基本性能计数器	自顶向下性能计数器	MLP 性能计数器	基于 PMC 的 MLP 异常定位
应用场景	对齐基本参数，例如计算指令延迟、访存延迟	对比分支预测错误率、缓存缺失率等基本指标。寻找替换算法、分支预测算法的差异	划分 CPI stack，快速定位性能瓶颈部件	检查访存子系统是否存在 MLP 异常	定位发生 MLP 异常的指令序列
基本原理	编写特定的指令序列，循环执行多次来估算循环体延迟	传统性能事件计数器，统计分支预测错误、缓存缺失的次数	统计 CPU 阻塞或空闲时发生的事件，计算这些事件在整体 CPI 中的贡献	在各级缓存中监控在途的、来自 CPU 核的访存缺失，统计 CPU 阻塞时在途的缓存缺失数量，得到 MLP 分布	将访存序列划分为多个 PMC，寻找模拟器与 RTL 在同一个 PMC 的并行度差异

4.5 实验评估

4.5.1 整体性能评估

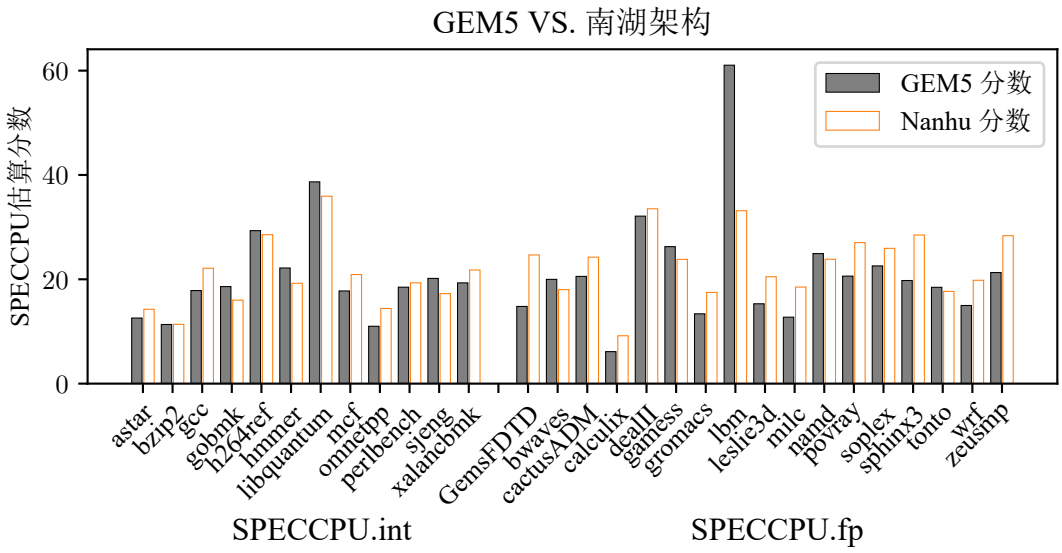


图 4-13 指令延迟对齐后的性能对比

Figure 4-13 The performance comparison after calibrating instruction delay

指令延迟对齐效果。图 4-13展示了指令延迟对齐后香山处理器与 GEM5 的性能对比。与文献 [14] 中观察到的现象相似，即使指令延迟完全对齐，RTL 与模拟器的性能差异仍然存在，这是因为指令延迟的差异只是模拟器与 RTL 的性能差异的一小部分，决定现代处理器性能的更多的是推测能力（分支预测和访存推测）和访存子系统（平均访存延迟和 MLP）。

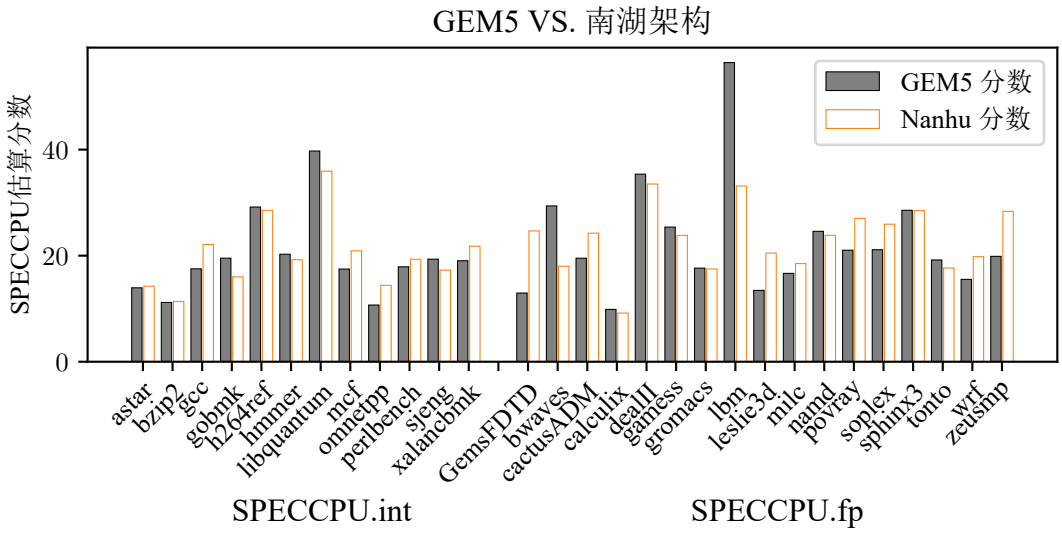


图 4-14 访存延迟对齐后的性能对比

Figure 4-14 The performance comparison after calibrating memory latency

访存延迟对齐效果。图 4-14展示了访存延迟对齐后的性能对比。对齐缓存和内存的访存延迟不但没有缩小模拟器与 RTL 的性能差距，反而增大了性能差距。例如在 *bwaves* 上，GEM5 领先 RTL 的幅度变大了；而在 *GemsFDTD*，*soplex* 和 *zeusmp* 上，GEM5 落后 RTL 的差距变大了。这并不是因为两个系统的差异变大，而是对齐的访存延迟让被掩盖的性能差异暴露了出来。

访存子系统性能缺陷的定位与修复。针对上述的明显的性能差距，本文在处理器对齐过程中依次应用了四种分析方法，大部分应用的性能差距原因都能被定位。例如，通过基本性能计数器发现香山处理器的一级数据缓存缺失率过高，通过自顶向下性能计数器发现 *GemsFDTD* 遭遇的 PTW 延迟过高问题，通过展开式自顶向下分析和 PMC 分析器定位到了 *soplex* 遭遇的访存依赖假阳性问题。

也存在少数四种分析方法都无法定位的情况，*GemsFDTD* 在 PTW 的延迟问题解决之后仍然存在性能差距。对于 *GemsFDTD*，本文采用了分段性能对比的方式来定位性能差异最大的部分，然后对这些部分进行人工分析。最终发现 *GemsFDTD* 的性能差异是性能计数器统计出错导致的，不属于微架构差异。

修复上述问题后的性能对比如图 4-15所示，*GemsFDTD* 和 *soplex* 等子项的性能差距缩小，但是 *bwaves* 和 *milc* 等子项的性能差距增大。在这里，香山处理器的 RTL 的 *bwaves* 一级数据缓存缺失率问题仍未解决。

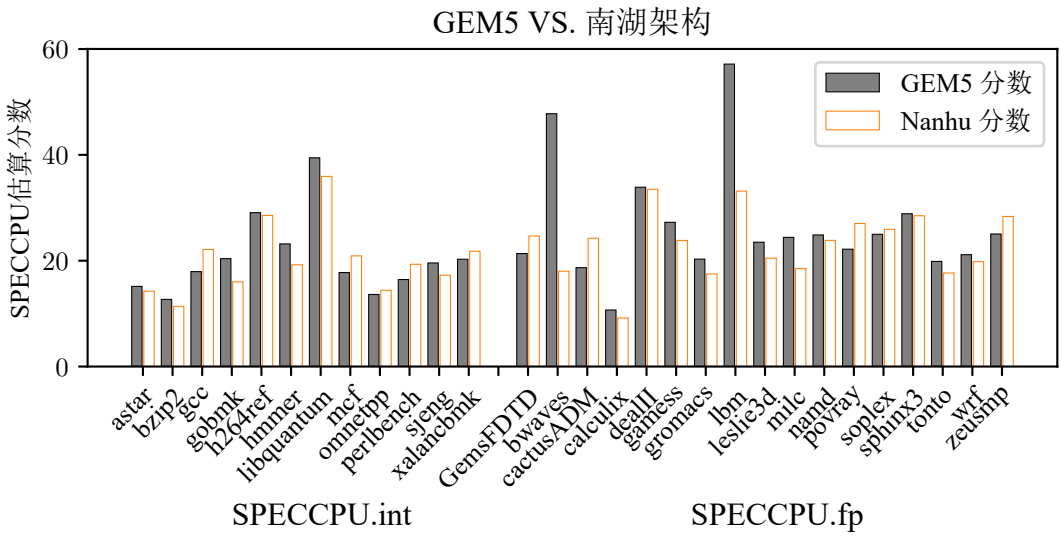


图 4-15 修复访存依赖假阳性等问题后的性能对比

Figure 4-15 The performance comparison after fixing the false positive prediction on memory dependency

在之前所有的测试中，*cactusADM* 在 GEM5 上的性能表现都不如香山的 RTL。这是因为 GEM5 的多预取框架和 Best Offset Prefetcher (BOP) [37] 存在缺陷，导致 BOP 预取器的覆盖率低，并且开启多预取框架后性能倒退。修复该问题之后，*cactusADM* 的性能表现得到了改善（如图 4-16所示）。

剩余性能差距分析。在修复了上述问题之后，仍然存在一些性能差距（图 4-

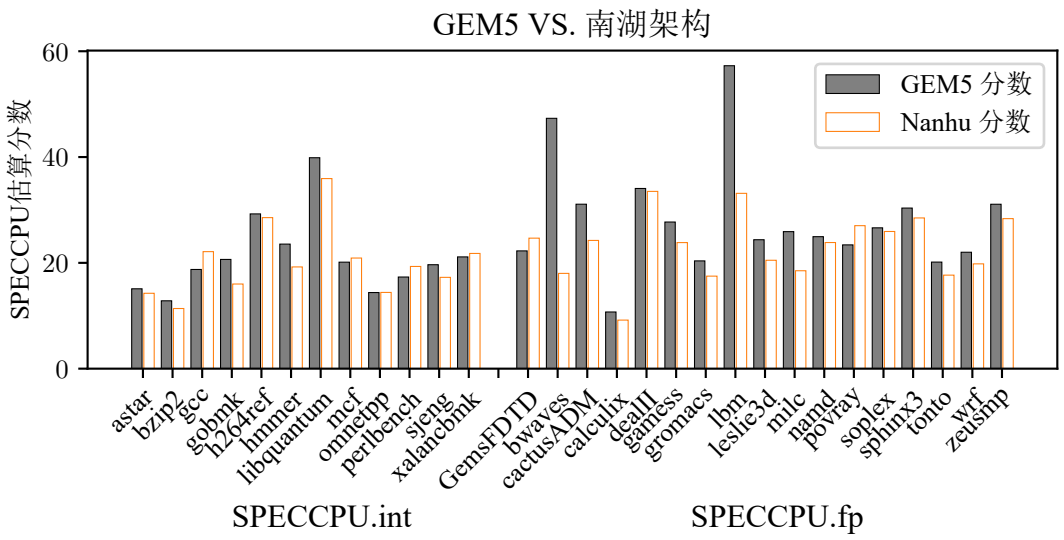


图 4-16 修复多预取框架后的性能对比

Figure 4-16 The performance comparison after fixing the multi-prefetcher framework

16）。本文对现存的性能差距的已知原因进行了整理。

- *bwaves* 的性能差距的一方面是因为香山处理器 RTL 一级数据缓存的缺失率过高，而另一方面是香山处理器的访存并行度异常。
- *lbm* 的性能差距是因为 GEM5 与 RTL 对 store 指令发出的请求的预取处理方式不同：GEM5 上可以获得每一个访存请求的发起 PC，因此 SMS 和 BOP 预取器都可以进行 store 指令的预取；而香山处理器将 store 请求先进行了合并再写入到一级数据缓存，因此无法准确获得 store 指令的发起 PC，进而无法让基于 PC 的 SMS 预取算法对 store 进行预取。恰巧准确的 store 预取对 *lbm* 的性能影响较大。
- *cactusADM* 的性能在多预取框架和 BOP 预取器修复之后由 GEM5 反超了 RTL。初步推测是由于 GEM5 上的预取算法实现得更准确或者更理想化。
- *gobmk* 的性能差距初步定位是前端的取指目标缓存 (FTB) 的缺失率不同，在 GEM5 上几乎没有 FTB 缺失，而在 RTL 上 FTB MPKI 约为 1。
- *gcc* 和 *povray* 是因为访存推测算法未对齐。

4.5.2 性能对齐的实例研究

4.5.2.1 香山处理器的数据缓存替换算法缺陷

在对齐架构模拟器与 RTL 的过程中，本文通过基本性能计数器发现了香山处理器一级数据缓存的缺失率比 GEM5 模拟器明显偏高。香山处理器专门添加了一个扩展 tag 来记录在最近被替换掉的缓存块中的命中情况。本研究发现 SPECCPU® 2006 的 *bwaves* 在扩展 tag 命中很多，表明替换算法经常替换掉有用的缓存块, 这说明要么 *bwaves* 的访存属于“颠簸”模式，要么是替换算法有性能缺陷。后经过检查发现香山处理器的数据缓存用于维护伪最近最少用（PLRU）的元数据的更新时机有问题：当多个访存请求在同一个缓存块缺失，在该缓存块

回填之前，不仅第一个访存请求会更新元数据，后续的请求也会更新元数据，这会导致 PLRU 的元数据维护出错。修复该问题后，*bwaves* 的一级数据缓存的缺失率降低到了正常水平。

4.5.2.2　GEM5 的地址翻译性能缺陷

在图 4-14中，在对齐了 GEM5 与香山处理器的访存延迟后，*GemsFDTD* 的性能大幅落后于香山处理器。通过 Topdown 分析发现，GEM5 上运行收集的 *GemsFDTD* 的 CPI stack 中，等待 TLB 的时间占比较高，部分片段超过了 10%（图 4-17）。从经验来看，因为 TLB 缺失率一般较低，一般较少出现 TLB 在 CPI stack 中占比较高的情况。

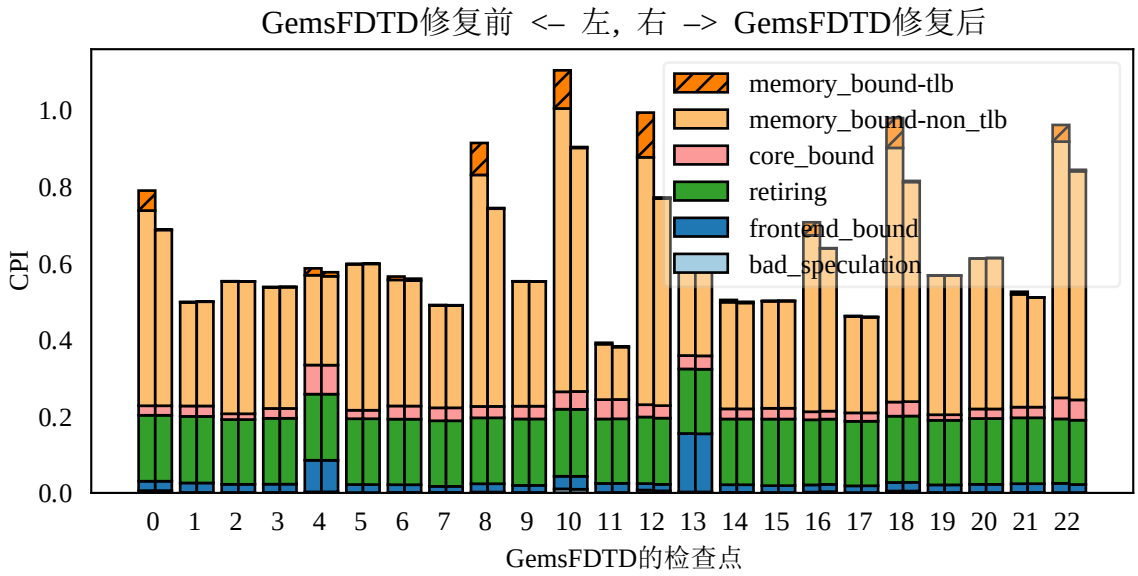


图 4-17 地址翻译性能缺陷对 *GemsFDTD* 的 top-down stack 造成的影响

Figure 4-17 The impact of the performance bug of address translation on the top-down stack of *GemsFDTD*

上述数据表明 GEM5 的 TLB 可能存在缺陷，因此本研究对比了香山处理器的 TLB 和 GEM5 的 TLB 的性能，发现两边 TLB 缺失率的次数相差无几。指令等待 TLB 的时间（亦称地址翻译时间），可以用下面的公式来估算：

$$\text{Translation time} = \text{TLB miss latency} \times \text{TLB miss rate} + \text{TLB hit latency} \times \text{TLB hit rate}$$

当模拟器与 RTL 的 TLB 命中率和命中延迟相近时，造成性能差异的变量仅剩 TLB 的缺失延迟。

当 TLB 缺失时，会进行页表遍历（Page Table Walk，PTW），将缺失的页表项从内存中加载到 TLB 中（图 4-18(a)）。因此 TLB 的缺失惩罚主要由 PTW 的延迟决定。通过分析调试信息，本项研究发现 *GemsFDTD* 的 PTW 延迟经常需要 200 周期左右，这说明 PTW 的访问页表的请求经常无法在缓存中被满足，需要从内存中获取。

通过对 PTW 访问的缓存块进行跟踪，本项研究发现 PTW 的足迹所在的缓存块在被访问后的较短时间后就被替换出 PTW 的局部缓存，在商业处理器和香山处理器中，PTW 的下游一般连接到二级缓存。但是后续的 PTW 访问相同缓存块的请求在二级缓存和三级缓存都发生了缺失。

这是因为为了对齐到香山处理器，本项研究将 GEM5 的二级和三级缓存的包含关系设为了 “mostly-exclusive”：上层缓存（例如一级数据缓存和 PTW 缓存）中的缓存行在下级缓存（例如二级缓存）中没有备份。在此配置下，出于性能考虑，无论缓存行是干净的还是脏的，都需要让上一级缓存在发生替换时主动将被替换块写回到下一级缓存（图 4-18(b)），而不是直接丢弃。而 PTW 缓存恰好没有开启该机制，导致页表项所在的缓存块被直接丢弃（图 4-18(c)）。定位到问题之后，修复方法较为简单：修改 PTW 缓存，使之总是将干净的块写回到下级缓存。

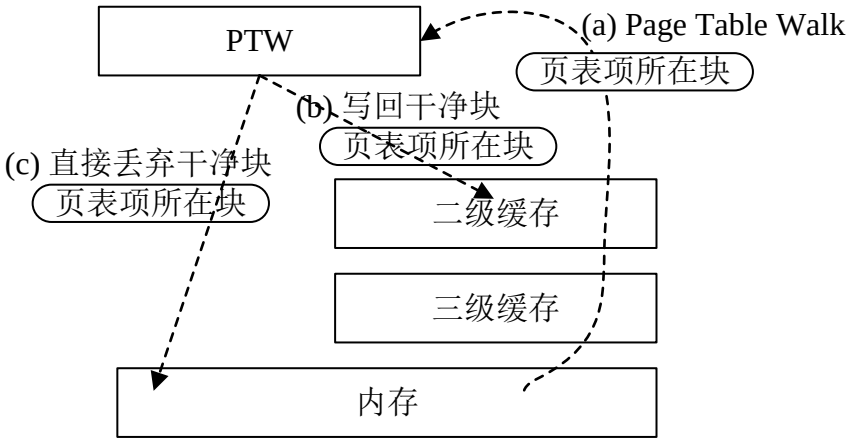


图 4-18 PTW 缓存处理干净缓存块的两种方式

Figure 4-18 Two ways to handle clean cache lines in PTW cache

4.5.2.3 GEM5 的访存推测算法假阳性性能缺陷

图 4-14反映出 *soplex* 在 GEM5 上的性能比香山处理器差，本项研究对比了两个平台的自顶向下性能计数器，发现差异较大的检查点的性能差异主要在于访存，尤其是三级缓存和主存的访问时间增加。此时，如果根据传统自顶向下，应该去定位三级缓存相关的问题。但是二者的基本访存相关性能计数器（缓存缺失率、访存延迟等）没有明显差异。

相反地，根据展开式自顶向下分析中的 C-AMAT 计算树，当缓存缺失率和访存延迟相近时，性能差异很可能来自于访存并行度的差异。而根据访存并行度检查表 4-2，三级缓存和主存的访问时间增加相关的访存并行度异常需要检查从核内到三级缓存等多个层级的访存并行度。

具体来讲，本文首先通过 MLP 计数器得到发生流水线阻塞时的平均 MLP，结合所有的长延迟访存的数量，利用文献 [16] 中的基于 interval 的分析方法，据

此可以估算出在该 MLP 下完成所有长延迟访存的最小时间，然后可以计算出最小 CPI。

$$\text{Min CPI} = \frac{\text{Min time}}{\text{Total instructions}},$$
$$\text{Min time} = \text{Interval cycles} \times \#\text{Intervals},$$
$$\#\text{Intervals} = \frac{\#\text{Long-latency loads}}{\text{Average MLP}},$$

其中 *Interval cycles* 是每个 Interval 的平均周期数，它的乐观估计是：

$$\text{Interval cycles} = \text{L3 miss latency} + \frac{\#\text{Interval instruction}}{\text{Dispatch width}},$$

根据计算，受限于 MLP，*soplex* 在 GEM5 上所能达到的最小 CPI 显著高于香山处理器达到的 CPI。

为了定位造成 MLP 损失的原因，本文应用了 4.3.4节中介绍的 PMC 分析器定位到性能缺陷的指令序列。根据指令的记录，分析确定产生 MLP 异常下降的长延迟访存指令与前面的长延迟指令之间没有数据依赖关系，即可并行的访存没有被并行执行。

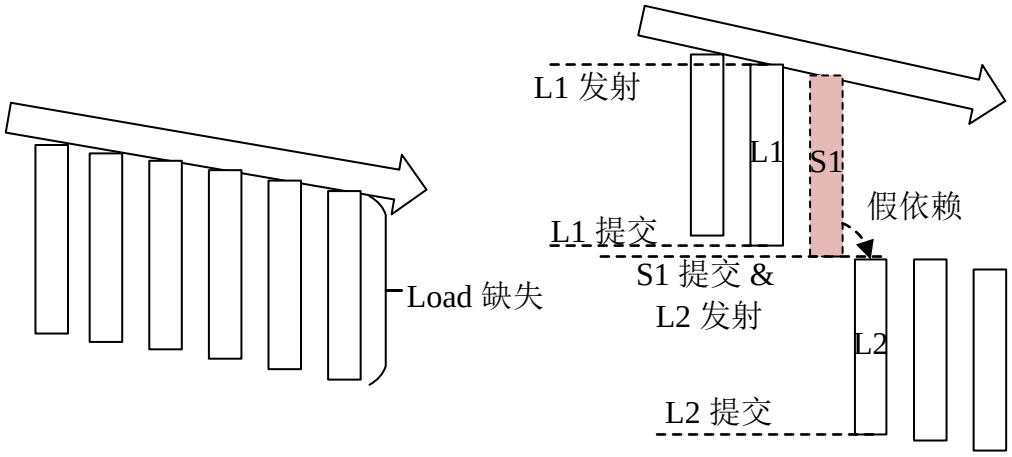
具体的访存并行度发生的原因如图 4-19所示。访存并行度异常的长延迟访存指令（L2）在上一个长延迟访存指令（L1）提交之后才就绪。而阻碍该指令就绪的原因并不是它的寄存器数据依赖关系，而是它被推测为依赖于一条 store 指令（S1）。当 L2 被推测为依赖于 S1 之后，它必须在 S1 指令的内容写回到数据缓存中之后才能执行。因为 S1 指令内容的写回必须等 S1 指令提交后才能发生，而因为指令是顺序提交的，S1 指令必须等更老的长延迟 L1 指令提交后才能提交。上述现象形成了如下的依赖链，导致了 L1 与 L2 无法并行执行：

$$T_{\text{L1 commit}} \leftarrow T_{\text{S1 commit}} \leftarrow T_{\text{S1 writeback}} \leftarrow T_{\text{L2 issue}}$$

这导致同样的访存序列，在没有错误的访存推测时，可以在 200 多周期完成；而当访存推测假阳性时，需要 400 多周期才能完成，流水线阻塞时间几乎翻倍。

经过进一步分析，本文发现 L2 与 S1 实际上不存在访存依赖关系，而是访存推测模块发生了假阳性推测。在处理器中，发生少量的错误推测是常见的，并不至于造成严重的性能损失。但是在 *soplex* 中存在大量的假阳性预测，导致了严重的 MLP 下降，且在香山处理器中并没有观察到该问题。这些假阳性判定是因为 GEM5 对 store 指令与 load 指令的重合判定过于保守，会将 *store *(addr + 8)* 与 *load *addr* 判定为重合。最后，通过修改 GEM5 的源码，将访存区间的重合判定修改为合理的参数，修复了这个问题。改进后 *soplex* 的性能在图 4-15中给出。

定位该性能缺陷的过程充分体现了展开式自顶向下分析相较于传统自顶向下分析的优势：展开式自顶向下分析通过 C-AMAT 的计算树和访存并行度检查表说明了三级缓存和主存的造成的阻塞时间过长不一定是三级缓存本身的设计问题，避免了定位性能异常过程中的误判。此外，PMC 分析器则加速了 MLP 相关异常的定位。



(a) s*oplex* 的理想 MLP　　(b) *soplex* 存在性能缺陷的MLP

图 4-19 访存推测假阳性在 *soplex* 上造成的访存并行度下降

Figure 4-19 Memory-level parallelism degradation caused by false positive memory dependency prediction on *soplex*.

4.5.3 处理器规模扩展的实例研究

在处理器的规格设计中，一般会根据系统的最坏情况访问延迟来确定指令窗口（ROB）的大小，然后根据 ROB 的大小以一定的比例来确定物理寄存器堆、发射队列和访存队列的大小。如果严格按照这样的思路设计，那么末级缓存缺失较少的应用因为的后端流水线阻塞也会较少。但是部分应用虽然在末级缓存缺失较少，但是在自顶向下 CPI stack 中仍可观察到大量的访存和后端流水线阻塞（如图 4-20中的 GEM5-normal 所示）。

本项研究对现有系统的访存延迟进行了分析，发现现有的指令窗口配置（256 项）无法完全覆盖片内最坏情况访存。如图 4-21(a) 和 (b) 所示，如果要满足 TLB 命中时的片内最坏访存延迟，需要 264 项的指令窗口；如果要满足 TLB 缺失、PTW 在二级缓存命中时的片内最坏访存延迟，需要 384 项的指令窗口。考虑到访存指令从分发到开始进行地址翻译还需要一定的周期数，还需要几个周期的冗余。为此，本项研究将指令窗口扩展到 400 项，以容忍 TLB 缺失时的片内最坏访存延迟。

指令窗口 400 的性能数据如图 4-20的 GEM5-ROB400 所示，他整体上将 SPECCPU® 2006 的分数提升了 7.6%。它让 *gromacs*，*tonto* 和 *dealII* 的 topdown CPI stack 中的访存和后端流水线阻塞的比例大幅下降，尤其是 *tonto* 和 *dealII* 的性能瓶颈几乎全部转化到了前端。

该数据说明在缓存延迟较大的架构中，提升指令窗口来容忍片内最坏访存延迟是有明显的性能收益的。存在多种途径可以获得这种性能收益，例如增加指

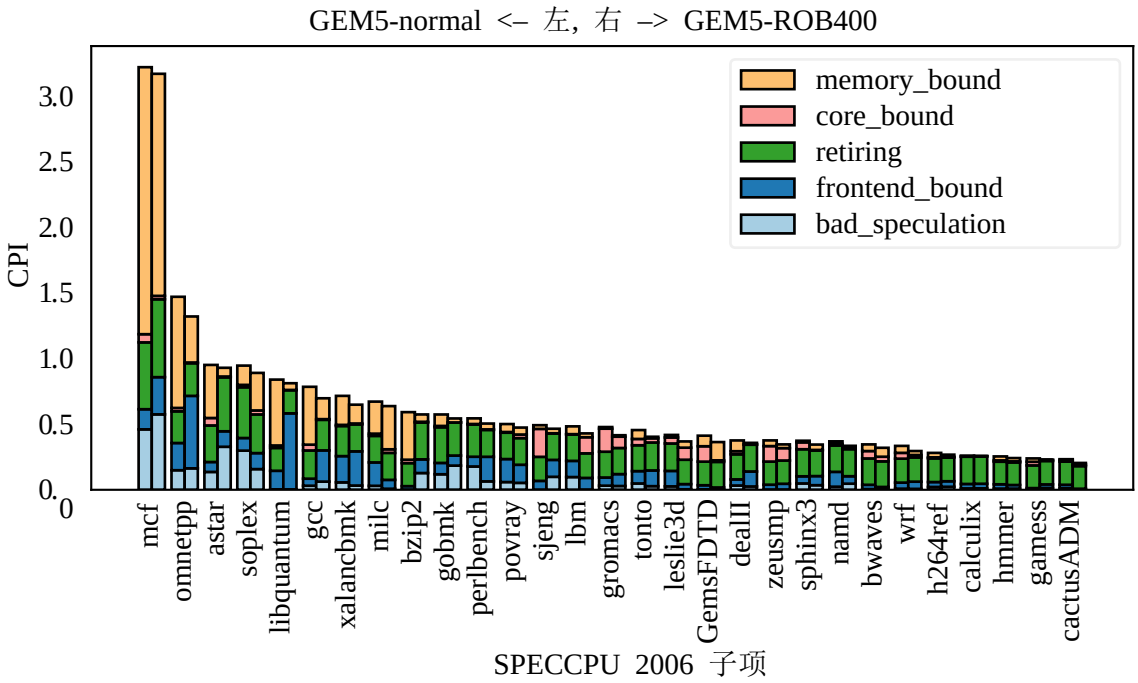


图 4-20 扩大指令窗口对自顶向下 CPI stack 的影响

Figure 4-20 The impact on top-down CPI stack when scaling-up the instruction window.

令窗口的大小、缩短访存延迟等。其中增大指令窗口是最简单的方法，但是可能给物理设计带来较大的挑战。这些挑战的潜在解决方案可能是寻找可扩展的物理寄存器堆替代方案，例如数据流架构 [21, 22] 或者寄存器懒惰分配或者提前回收 [90, 91, 93, 94, 135]。除增大指令窗口外，还可以通过其他方式来获得类似的收益，比如通过数据预取、TLB 预取、改进多级缓存等方式降低访存延迟来降低对指令窗口容量的需求。

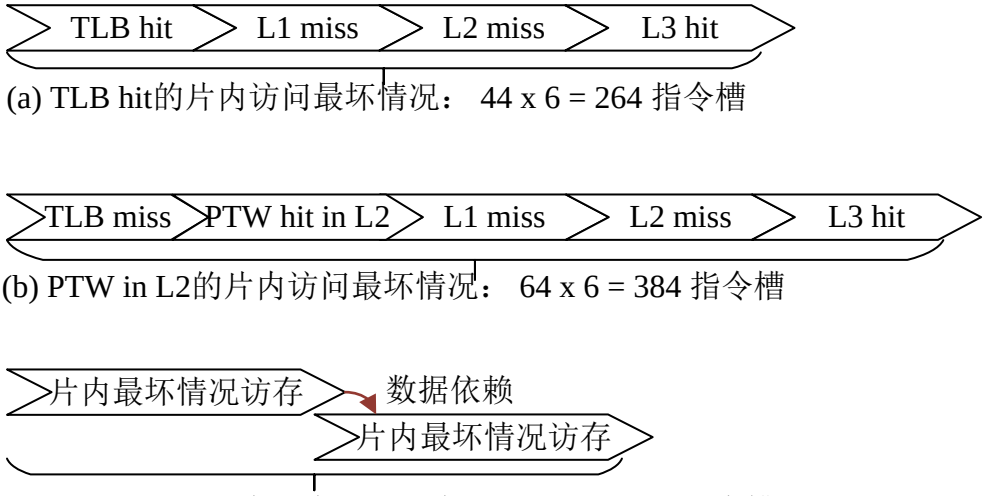


图 4-21 根据访存延迟指导指令窗口扩大规模

Figure 4-21 Scaling-up the instruction window according to the memory latency.

4.6 本章小结

为了快速进行设计空间探索，本文基于 GEM5 开发了香山处理器的微架构模拟器 XS-GEM5。为了提高微架构模拟器对齐效率，并解决自顶向下性能计数器与微架构性能计数器之间的脱节问题，本文提出了展开式自顶向下分析框架。该框架改进了自顶向下分析框架，建立了自顶向下性能计数器与微架构性能计数器之间的量化关系。精确对齐的模拟器和展开式自顶向下分析框架将会帮助香山处理器实现高效的架构迭代。

第 5 章　可扩展指令调度架构优化

无论是商业处理器的发展路径（表 1-1），还是 4.5.3节中关于扩大规模的收益的预测，都说明扩展处理器规模有助于性能提升。本章对处理器规模扩展的主要瓶颈 – 指令调度架构展开研究，提出了 Omegaflow 指令调度架构。Omegaflow 在可扩展性良好的同时，性能也逼近传统超标量架构。

5.1　引言

传统的高性能架构采用基于广播的发射队列和集中式物理寄存器堆，以充分挖掘应用程序的指令级并行度（Instruction-level parallelism，ILP）。4.5.3节的实验结果表明，增大指令窗口、物理寄存器堆和发射队列可以获得性能收益。但是，增大这些结构会带来较大的能效问题 [5, 6]，特别是在后摩尔时代。此外，这些结构还会影响时钟速率，并且需要定制设计，拖慢高性能处理器的布线流程 [2, 5]。

自 2000 年代以来，学术界和工业界尝试了许多设计来克服这些限制，例如：简化的乱序调度结构 [96, 98, 136, 137]，基于依赖的乱序调度 [20, 21]，显式数据流结构 [22, 23]，和基于顺序执行的部分乱序执行 [25, 26]。在这些提出的设计中，Forwardflow 架构被证明是一种潜在的替代方案，因为它简单且具有良好的能效。Forwardflow 可以使用简单的标准硬件设计，具有良好的能效并可以扩展指令窗口以提高性能。然而，Forwardflow 为了较好的能效付出了显著的性能代价，这使得它不能取代传统的 OoO 架构。[1]

本章希望探究 Forwardflow 的性能是否已经被充分挖掘、还能进一步提高 Forwardflow 的多少性能？为了回答这些问题，必须明确：*1*）什么限制了 *Forwardflow* 的整体性能，以及 *2*）如何消除这些障碍？为了帮助 Forwardflow 架构的性能分析，本文推导出了一个分析模型，以量化整体性能与各种因素之间的关系。通过这个模型，本文得出了以下关键发现。

令牌吞吐率和总令牌数（也称为唤醒消息）是在 Forwardflow 中限制应用程序性能的两个关键因素。令牌是在 Forwardflow 中被设计用来跟踪指令依赖关系，并在数据流队列（Dataflow Queue，DQ）的分布式组件之间传递。数据流队列是 Forwardflow 的发射引擎，它的职能类似传统乱序执行中的发射队列和物理寄存器堆。由于令牌需要占用 DQ 的硬件资源不能总是保证有空余，因此令牌在生成后并不能保证立即传递和接收。由于令牌的传输可能被延迟，因此对于给定数量的令牌，整体性能严重依赖于处理令牌的速度。以 SPECCPU 2017 中的 *namd* 为例，当在 Forwardflow 上以四个 DQ bank 运行时，平均每条指令生成 2.17 个令牌，这是一个不小的需求。然而，在四个 DQ bank 的 Forwardflow 中的整体令牌

[1] 文献 [21] 将 Forwardflow 与基于 32-entry SLIQ [98] 的基线 OoO 核心进行比较。而本文的基线是一个 OoO 核心，具有理想的统一发射窗口，总是保证指令的年龄序和背靠背唤醒。

吞吐率并不高，例如，在最乐观的估计下，每周期令牌吞吐量仅为 3.31 个。因此，该分析模型表明 *namd* 的性能（IPC）上限为 1.52，这与实验结果一致。

为了打破性能限制，必须提高令牌吞吐量或减少令牌总数。但是，不能通过粗暴地增加 RAM 端口或使用广播来实现这些目标，因为这会带来显著的开销（第 5.4.5 节）。幸运的是，本文有两个重要的观察：1）许多令牌并不发射指令；2）令牌总数与 Forwardflow 中串行依赖链的长度有关，存在缩短串行依赖链长度的机会。根据这两个观察，本文提出了一种新架构 **Omegaflow** 来解决性能问题，它包括三个新的子技术。

首先，根据观察 1，本文设计了**并行数据流队列 bank**，以大幅提高令牌处理速率。并行数据流队列 bank 的关键思想是差别化处理令牌，并在多个令牌访问 DQ bank 的不同位置时同时进行处理。其次，为了提高令牌通信带宽，并适应并行数据流队列 bank 的路由需求，本文采用了新的互连网络（16-16 Omega 网络和 16x16 crossbar）并设计了相应的路由机制。最后，基于观察 2，为了减少令牌的总数，本文引入了**完成即写回**机制。完成即写回机制缩短了不必要的依赖链（见 5.3.3节），从而减少了令牌的总量。

本文基于 GEM5 [12] 实现了 Omegaflow，并使用 SPECCPU 2017 进行评估。实验结果显示，与 Forwardflow 相比，Omegaflow 平均 IPC 提高了 24.6%，并且可以达到具有理想化调度器的 OoO 架构的 94.4% 的性能。功耗评估显示，与 Forwardflow 相比，Omegaflow 的能耗仅增加了 8.7%。RTL 综合报告显示，Omegaflow 对 Forwardflow 的关键路径的影响非常小。

总的来说，本章的贡献包括：

- 本文为 Forwardflow 提出了一个分析性能模型，揭示了应用程序需求（对于令牌的整体吞吐率的需求）与架构能力（令牌处理能力）之间的不匹配。
- 本文有两个关键的观察，有助于调和上述的不匹配问题。基于这两个观察，本文提出了 Omegaflow，它由三种新技术组成，解决了这种不匹配。
- 本文在 GEM5 模拟器中实现了 Omegaflow，并用 Chisel 实现了它的核心组件。实验结果表明，Omegaflow 的性能比基线设计高 24.6%，让基于动态数据流的指令调度朝着理想调度器迈进了一步。

5.2 背景和动机

本节首先回顾 Forwardflow，以说明 Forwardflow 如何以非传统方式实现乱序执行，即串行后继表示 (SSR)，然后展示 Forwardflow 如何使用数据流队列实现 SSR。接下来，本文列出了 Forwardflow 的优缺点，特别是性能局限性的来源。最后，阐述这些限制如何影响了性能，并讨论如何改进该体系结构。

5.2.1 Forwardflow 架构

在传统超标量架构中，指令依赖以体系结构寄存器名的方式来表示，由消费者记录生产者的“名字”，这被称为后向数据流。而 Forwardflow 架构中，由生产

者记录消费者的“位置”，这被称为前向数据流（Forward dataflow）。使用前向数据流的好处是，因为大部分生产者只有少量的消费者，可以有的放矢，避免像传统超标量架构中的广播行为。而针对拥有大量消费者的指令，Forwardflow 使用了串行后继表示（SSR）（图5-1），让依赖于同一个寄存器的消费者被串行唤醒。串行后继唤醒的设计理念是只为最常见情况预留数据通路，而传统超标量架构相当于为最差情况预留数据通路（广播），这使得 Forwardflow 有更好的性能功耗比。文献 [21] 的理想实验证明了基于 SSR 的理想的动态数据流架构的性能上限非常逼近理想的超标量架构。

串行后继表示（Serialized Successor Representation）

	指令元信息	目的寄存器		源寄存器1		源寄存器2	
		值	指针	值	指针	值	指针
I1	add	t2		a1		1	
I2	mul	s2		t2		2	
I3	sub	s3		t2		t1	
I4	add	s4		t1		4	

图 5-1 串行数据表示示意图

Figure 5-1 Illustration of Serialized Successor Representation

Forwardflow 在类似于传统超标量架构的重命名阶段构建串行后继表示，然后将串行后继表示存入数据流队列（Dataflow Queue，DQ）。逻辑上，数据流队列取代了超标量架构中的重排序缓冲区（ROB），物理寄存器堆和发射队列（图 5-2）。物理上，数据流队列采取了分布式的组织方式（图5-3）。数据流队列由一个或者多个 DQ 组（DQ group）组成（图5-3(a) 为 4 个 DQ 组），指令按顺序分派到每个 DQ 组，填满其中一个组再填下一个，构成一个环形队列。在一个 DQ 组内部，指令交替插入到多个 DQ bank（图5-3(b)(c)）。DQ bank 内部存放了指令元信息、寄存器值和标志后继位置的指针（图5-4）。通过这种布局，Forwardflow 可以向下收缩只使用一个 DQ 组来实现较低的功耗，也可以向上扩展将分布式的 DQ 组串联起来实现超大指令窗口（超过 1000 条指令）和超大发射宽度（64 发射）。

综上所述，串行唤醒和分布式布局帮助 Forwardflow 从源头上避免了广播式发射队列和集中式物理寄存器堆，但是同时也引入了性能问题：单 DQ 组的 Forwardflow 和传统超标量架构存在显著的性能差距，多 DQ 组的 Forwardflow 不能获得大幅性能提升。

5.2.2 串行后继表示的构建、存储与使用

为了介绍 Forwardflow 基于 SSR 的发射机制，本文先定义一些术语：

- *Committed ARF* 是用于存储已提交指令的结果的体系结构寄存器堆。
- 令牌是包含了消费者的位置（指针）和计算结果的数据的唤醒信息，从链表里的前驱指令传播到后继指令。

构建 SSR 链。重命名阶段负责生成链表的指针。对一条抵达重命名阶段的指令，如果它的寄存器操作数已经就绪，则从 *committed ARF* 直接获取寄存器值。如果操作数仍然未就绪，则先获得现有链表的尾部，然后生成一个从旧的尾指针出发、指向以该操作数即将分发的位置的新指针。在图5-1中，在重命名阶段，生成了指令 I2 和 I3 的指针：*I1.dest → I2.src1* 和 *I2.scr1 → I3.src1*。

存储 SSR 链。分发阶段将重命名阶段生成的指针存储到前驱指令的指针域，将指令的元数据和就绪的寄存器填充到数据流队列中的分发位置。指令元数据和寄存器的存储在数据流队列中是顺序填充的，即分发位置是确定的。但是指针域的更新不是顺序的，因为需要找到前驱指令并将后继的分发位置写入到指针域。例如，指针 I1.dest → I2.src1 将被发送到 I1 在 DQ 中的指针域，这一过程需要将该指针路由到 I1 所在的位置。I1 所在的位置可以从重命名表中获得。

基于 SSR 链进行指令发射。在发射阶段，SSR 机制沿着指针链表逐个唤醒消费者。当指令计算完成时，SSR 机制将首先获取链表头，指向第一个依赖于该指令的消费者。然后，将计算值的令牌发送到第一个消费者。此过程会不断地发送新的令牌到下一个后继（消费者），直到链表被遍历完。在图 5-1所示的链表中，当 I1 计算完成后，I2 将首先从 I1 接收包含 t2 寄存器值的令牌，然后将 t2 的寄存器值转发到 I3.src1。

SSR 链表的长度会从两方面影响性能。第一是 SSR 可能直接延迟指令发射，本文称为 **SSR 延迟**（例如图 5-1中，I3 会比 I2 晚一周期获得 t2 的值）。第二是由于每个 SSR 链的节点在发射时都会对应产生一个令牌，因此随着 SSR 链表长度的增加，传递的令牌的数量也会增加，这将间接影响性能。令牌数量如何影响性能将在第5.2.3节中讨论。

5.2.2.1 数据流队列

指令和 SSR 链表储存在数据流队列（DQ）中。DQ 接收从重命名阶段生成的指针，以构建链表，并在不同的 DQ 条目之间传递令牌以发射指令。接下来，将详细介绍 DQ 的结构以及 DQ 如何根据链表发射指令。

在数据流队列中，数据流队列 bank 是最小的可以独立处理令牌、发射指令的单元。前面提到过，一个令牌会携带唤醒信息和寄存器的值。当数据流队列 bank 从接收队列（Rx queue）接收到一个令牌后，数据流队列 bank 会将令牌中的寄存器值写入到寄存器域中，然后检查指令的元数据以确定指令是否可以发射。此外，只要链表节点的指针字段不是链表的末尾，数据流队列 bank 将不断

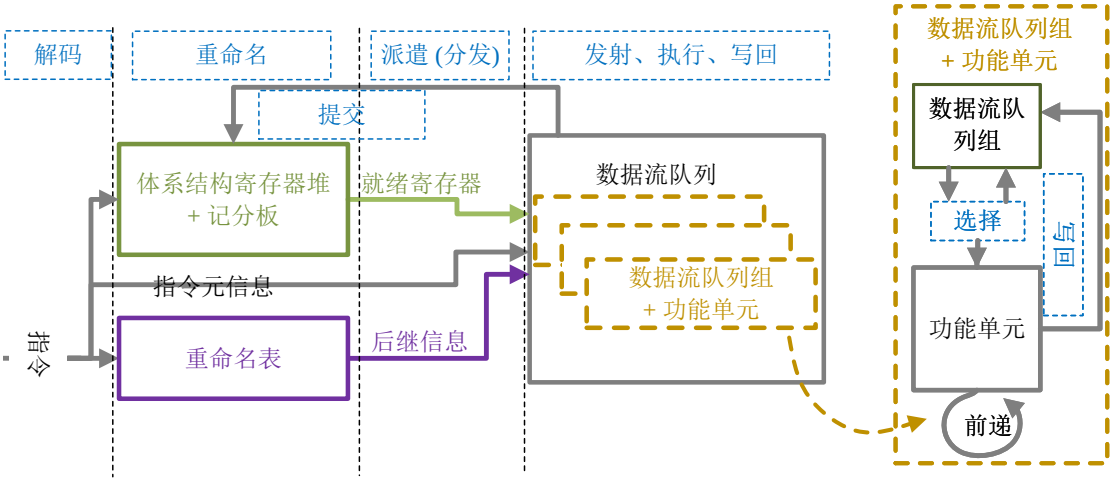


图 5-2 Forwardflow 的流水线划分

Figure 5-2 Pipeline stages of Forwardflow

生成以下一个后继为目标的新令牌，并通过发送队列（Tx queue）发送到后继所在的数据流队列 bank。图 5-4 是上述过程的一个例子。

多个数据流队列 bank 通过一个互联网络相连，形成一个数据流队列组（图 5-3(a)）。这些指令在多个 bank 之间交替摆放（图5-3(b)(c)）。每个周期，互联网络从各个数据流队列 bank 的发送队列获取令牌，并根据目的地将令牌路由到相应的接收队列。如果多个令牌的目标是同一个数据流队列 bank，则只有其中一个能被授权通过，其他的将在发送队列中等待。本文把这样的延迟称为**排队延迟**。图 5-5是一个令牌之间发生冲突的示例。

5.2.3　Forwardflow 的性能限制

在上一小节，本文介绍了 Forwardflow 如何利用串行化后继表示（Serialized Successor Representation，SSR）和数据流队列实现乱序执行。本小节，本文将讨论 Forwardflow 为了可扩展性和性能功耗比所做的设计决策的优点和局限性。然后，本文会建立 Forwardflow 所遭受的性能损失与其设计之间的定量关系。

Forwardflow 比传统的乱序执行核心更具能效优势有两方面的原因：1）*SSR* 通过跟踪指令依赖关系来避免广播唤醒消息；2）数据流队列通过有限的端口发射指令，这样可以避免端口密集型的物理寄存器堆。但是正如前文所指出的，SSR 机制和数据流队列不可避免地比理想调度造成了更多的指令唤醒延迟。研究表明，推迟指令发射可能会对某些工作负载产生严重的性能损失 [5, 138]。然而，对于 Forwardflow，仍然不清楚：1）什么样的工作负载更容易受到影响；2）推迟指令发射造成的性能损失在 Forwardflow 和理想调度之间的性能差距中占多大比例。本小节将通过逐步建立 Forwardflow 的微架构特征与程序负载特征之间的关系来回答这个问题。

因为每条指令都必须等到该指令所需的令牌全部被接收和处理后才能发射，

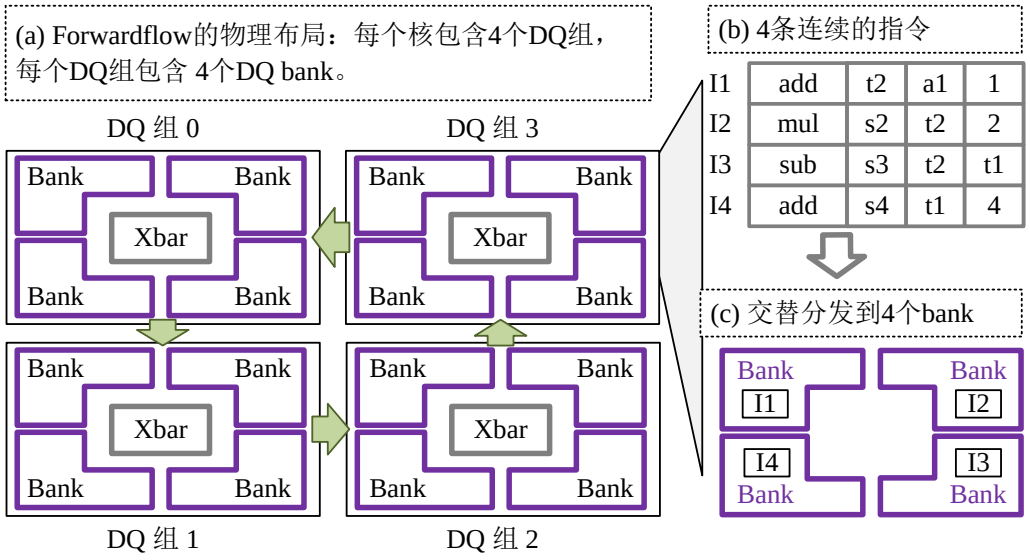


图 5-3 数据列队列的物理布局和层次结构

Figure 5-3 The floorplan and hierarchy of the Dataflow Queue

所以单个数据流队列组的发射速率由接收和处理令牌的速率决定。因此发射所有指令所需的时间取决于接收和处理令牌的速率以及令牌的总数：

$$N_{Cycles\ to\ issue} = \frac{N_{Tokens}}{TR},$$

其中 TR 表示接收和处理令牌的速率。根据 IPC 的定义，每周期提交的指令数小于等于每周期发射的指令数，因此：

$$IPC = \frac{N_{Insts}}{N_{Cycles\ to\ commit}} \leq \frac{N_{Insts}}{N_{Cycles\ to\ issue}},$$

将它们整合得到：

$$IPC \leq \frac{N_{Insts}}{N_{Tokens}/TR} = \frac{TR}{N_{Tokens}/N_{Insts}}, \tag{5-1}$$

根据上述公式的分母，可以定义一个新指标：**每条指令的令牌数**（token per instruction, TPI）：

$$TPI = N_{Tokens}/N_{Insts},$$

最后，将 TPI 代入公式 5-1，可得

$$IPC \leq \frac{TR}{TPI}, \tag{5-2}$$

公式 5-2回答了关于程序性能与令牌、程序特征的疑问:

1) 就程序特征而言, 越高的 TPI 越容易产生更低的 IPC。
2) 就 Forwardflow 架构而言, 限制 TR 就会限制它的整体性能。

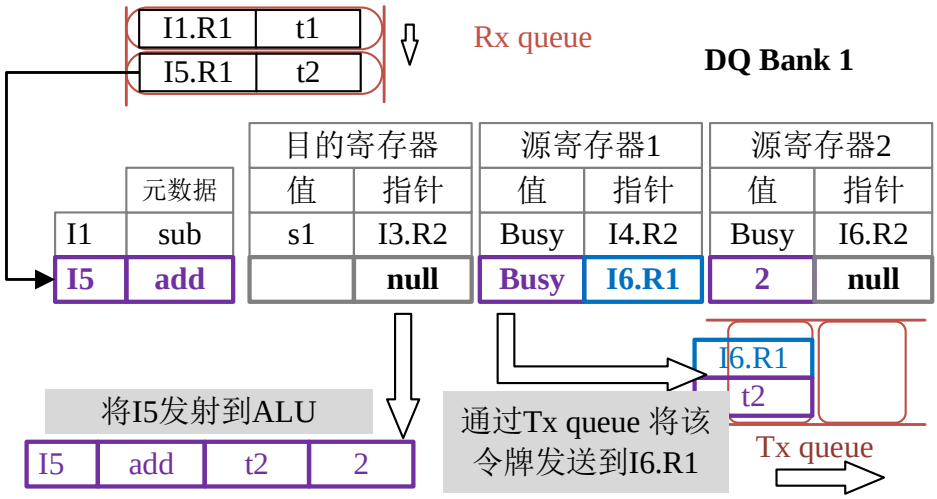


图 5-4 DQ bank 的结构和指令发射示意图

Figure 5-4 The structure of DQ bank and an instruction issue example in DQ bank

那么 Forwardflow 的 *TR* 是多少呢？对于一个 4 个数据流队列 bank 的数据流队列组（方便起见，本文将 4 bank 的 Forwardflow 称为 F-1 配置）而言，可以计算出在最理想的情况下它的 TR 的上限是 4.0：每个数据流队列 bank 每个周期至多处理一个令牌，一个含有 4 个数据流队列 bank 的数据流队列组每个周期最多处理 4 个令牌。如果再考虑 crossbar 发生冲突的话，TR 会更低：如果假设所有令牌的目的地是均匀且相互独立的，则 4x4 crossbar 能提供的带宽期望约为 3.31。因此，TR 的上限约为 $\min\{4.0, 3.31\} = 3.31$。

那么，上述限制对程序负载的性能影响有多大呢？本文从 SPECCPU 2017 基准测试集 [139] 中收集了如下四元组：*(benchmark, TPI, IPC, Ideal IPC)*。其中，*TPI* 和 IPC 是从 F-1 配置的 Forwardflow 中收集的。Ideal IPC 是从一个具有理想调度器的乱序执行核心中收集的。关于 F-1 和乱序执行核心的更详细配置可以在表 5-1 中找到。

图 5-6 中的结果表明，对于大多数工作负载来说，它们的性能上限远高于它们在 Forwardflow 中达到的实际性能。例如，图 5-6 中的模型可以解释 83.1% 的 *bwaves_0* 的性能损失。这表明

负载的 TPI 与 Forwardflow 的 TR 不匹配是性能瓶颈的关键因素。

上述推导和实验揭示了数据流队列组的 TR 较低，无法匹配程序负载的高 TPI ，造成了严重的性能损失。在下一节，本文将从两个方面来调和这种不匹配：增加数据流队列组的 TR 和降低工作负载的 TPI 。

5.2.4　观察和机会

正如在 5.2.1节中所讨论的，SSR 和数据流队列是 Forwardflow 的关键特性，它们使 Forwardflow 的能效较高。因此，完全废弃 SSR 或者对数据流队列组的处理

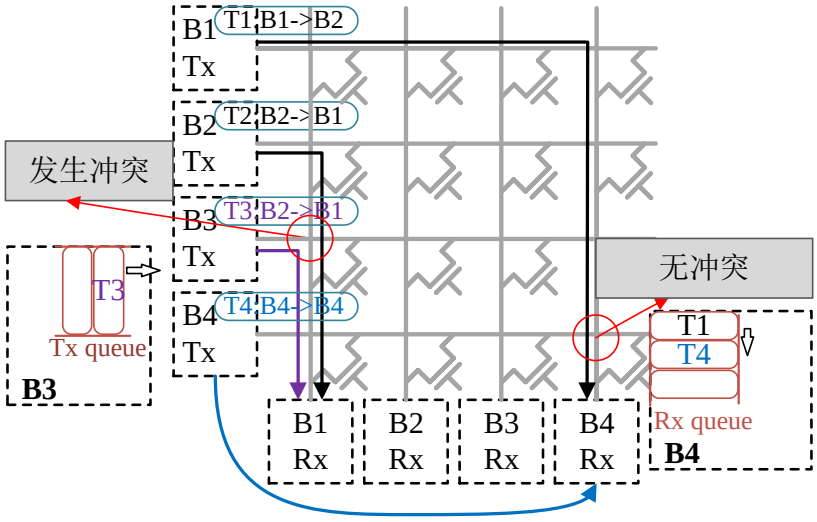


图 5-5 数据流队列发生 bank 冲突导致令牌排队的示例。T2 和 T3 发生了冲突因为它们的目的地相同。但是，T1 和 T4 不冲突，因为 T4 来自本地 bank 4，无需通过 Crossbar 发送。

Figure 5-5 An example of token conflict in DQ banks. Token T2 and T3 conflict because they have the same destination. However, T1 and T4 do not conflict because T4 is from local bank (bank 4) and does not need to be sent through the crossbar.

能力进行不当的提升都可能会损害 Forwardflow 的能效。为了指导对 Forwardflow 微架构的改进，本文对 Forwardflow 微架构中限制 TR 或影响 TPI 的因素进行了全面的分析。

5.2.4.1 什么限制了 Forwardflow 的 TR，如何提升？

Forwardflow 的 TR 受到两方面因素的限制，即每个数据流队列 bank 的令牌处理速率和互联网络的令牌传输带宽。虽然提升互联网络的带宽相对简单，但是提升每个数据流队列 bank 的令牌处理速率时则需要更多的设计考量。

考虑到每个抵达数据流队列 bank 的令牌都可能把一条指令的状态从未就绪改变为准备发射，Forwardflow 保守地假设每个令牌都会发射一条指令，并为每个抵达的令牌预留发射指令的机会。这样的设计决策使得每个数据流队列 bank 的令牌处理速率与指令发射速率严格绑定。而指令发射速率又受限于数据流队列 bank 的 RAM 端口数，这使得提升令牌处理速率和保持高效能之间产生了矛盾：提升指令发射速率必然会增加 RAM 端口数，从而损害能效。

为了进一步研究这种保守的绑定是否必要，本文收集了实际需要占用发射机会的令牌的数量。为了方便起见，本文定义了关键令牌 (*critical token*)：关键令牌抵达后，使得指令从未就绪变为就绪。否则，将这个令牌定义为非关键令牌 (*non-critical token*)。例如，图 5-4中的令牌 I5.R1 抵达后会导致指令 I5 发射。因此，令牌 I5.R1 就是一个关键令牌。相反，令牌 I1.R1 就是一个非关键令牌。

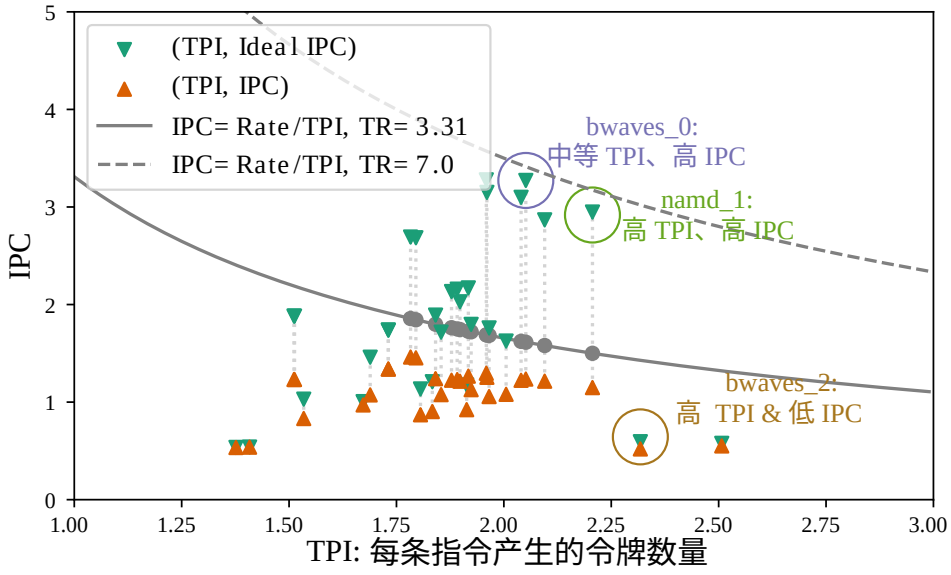


图 5-6 Forwardflow 性能的分析模型。图中的实线和虚线分别是当 TR 为 3.31 和 7.0 时，根据公式 5-2绘制的曲线。下三角和上三角分别表示在理想乱序执行核心和 Forwardflow 中程序负载所达到的 IPC。

Figure 5-6 The analytical model for Forwardflow. The solid and dashed lines are the curves drawn by formula 5-2 when TR is 3.31 and 7.0, respectively. The downward and upward triangles represent the IPCs achieved by the program workload in an ideal out-of-order core and Forwardflow, respectively.

本文从 SPECCPU 2017 基准测试中统计了 Forwardflow 架构执行过程中产生的令牌。实验表明，非关键令牌所占的比例很大，范围从 **39.4% 到 68.7%**（图 5-7）。虽然提高发射速率的代价较大，但是提高非关键令牌的消费速率的开销相对较低。通过简单的估算，如果将非关键令牌的消费速率提高到每周期 3.0 个，整体的令牌消耗速率会提高 **36%–84%**。

观察：非关键令牌所占的比例很大。
机会：可以通过分治的方式处理令牌，提高非关键令牌的消费速率，从而获得更高的消耗速率而不会引入过高开销。

5.2.4.2　什么因素影响了 TPI 以及如何降低

不难想到，TPI 主要由程序特征和编译技术决定。现在已有通过编译器降低 TPI 的方法 [138]，但是本文主要关注不改动 ISA 的硬件技术。本文发现，除了编译器外，TPI 还受到微架构设计的影响。

一方面，TPI 会随着指令窗口的增大而增大。根据本文的实验结果，当一个数据流队列组的指令总数从 128 增大到 192 时，TPI 平均增加了 5.21%，而当它增大到 384 时，TPI 又增加了 24.6%。然而，通过减小指令窗口的大小来降低 TPI 并不明智，因为指令窗口是挖掘指令级并行度的重要工具。

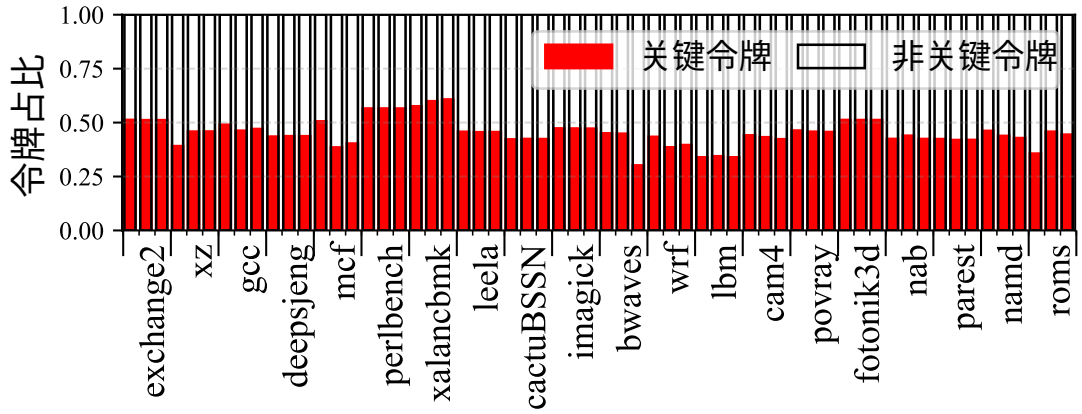


图 5-7 关键令牌所占比例

Figure 5-7 The proportion of critical tokens

另一方面，如第 5.2.1节所述，令牌的数量与 SSR 链表的长度正相关，而 SSR 链表的长度又受重命名机制的影响。Forwardflow 通过查询 *commited ARF* 来重命名指令。本文发现，尽管 *commited ARF* 包含了架构的精确状态，但是它不能及时地把最近计算出的指令结果传递给重命名阶段。不难想象，总是有一部分指令已经计算完成但是尚未提交。在 Forwardflow 的设计中，这些指令的值不能被正在重命名的指令获取。图 5-14(a)(b) 描述了这种情况：当指令 I1 计算完但是还没有提交时，依赖于 I1 目的寄存器 R1 的指令 I4 无法看到 R1 已经就绪。

观察: *Committed ARF* 无法将最新的寄存器值传递给新指令。 机会：可以使用一个额外的 ARF，使得指令可以在计算完成时立即写回到 ARF 中，以尽快传递寄存器值。

5.3 设计

在 5.2.4中的观察指导下，本文提出了 Omegaflow，Omegaflow 重新设计了数据流队列以提高单个数据流队列组的吞吐量（图 5-8），并添加了额外的体系结构寄存器堆（ARF）来缩短 SSR 链表的长度（图 5-9）。总体来说，Omegaflow 在 Forwardflow 的基础上做了三个主要的改进：

1. **并行数据流队列 bank** 。通过区分关键令牌和非关键令牌，Omegaflow 允许一个数据流队列 bank 每周期最多消耗 4 个令牌。
2. **互联网络增强**。为了适应并行数据流队列 bank 的路由需求并提高跨数据流队列 bank 的通信带宽，Omegaflow 将 4x4 的交叉网络替换为更宽的网络。
3. **完成即写回**。为了减少 SSR 链表的长度，Omegaflow 让指令在计算完成后立即写回到新增的 *completed ARF* 中，使得重命名的指令可以及时获得新计算的值。

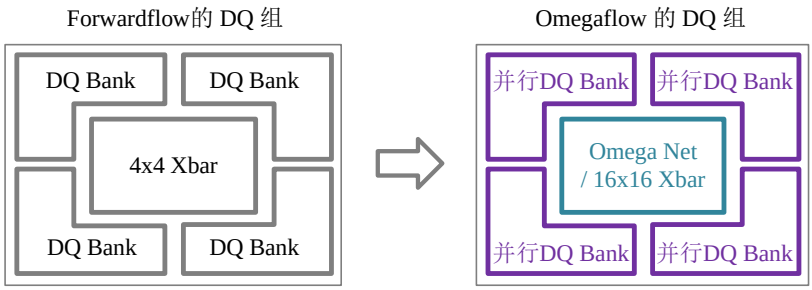


图 5-8 Omegaflow 通过把原始的数据流队列 bank 替换为并行数据流队列 bank，并把 crossbar 替换为 Omega 网络或 16x16 crossbar，提高了数据流队列的令牌吞吐率。

Figure 5-8 Omeagflow improves the token throughput of the Dataflow Queue by replacing the original DQ bank with the parallel DQ bank and replacing the crossbar with Omega network or a 16x16 crossbar.

5.3.1　并行数据流队列 bank

并行数据流队列 bank 的目标是提高令牌的消耗速率，面临的最大挑战是如何保持 Forwardflow 的高能效。因为在 Forwardflow 原本的设计中，每消耗一个令牌就需要一组 SRAM 的端口，暴力地提高消耗速率会导致需要多端口的 SRAM。幸运的是，本文观察到**很大一部分令牌都是非关键令牌**，而且**非关键令牌的资源消耗要远远小于关键令牌**。基于此观察，并行数据流队列 bank 先区分不同的令牌类型，然后允许让它们并行访问数据流队列 bank 的指针阵列来提升消耗速率。

并行数据流队列 bank 通过维护一个 *N-Busy* 数组来区分令牌类型，该数组表示对应的指令所依赖的令牌的数量。如果一个令牌到达时，在 *N-Busy* 数组中对应项的值为 1，则判定该令牌是关键令牌。否则，判定该令牌是非关键令牌。

并行数据流队列 bank 采用贪心的策略，假设所有到达的令牌都是非关键令牌，让它们并行访问指针阵列（前提是它们指向不同的指针阵列，没有发生冲突）。同时，并行数据流队列 bank 会检查 *N-Busy* 数组，如果并行访问的令牌中存在至少一个关键令牌，则在下一个周期发射对应的指令。如果并行访问的令牌中存在多个关键令牌，则只发射其中一个，其他的令牌则暂时放入队列中等待。

为了区分目的地不同的令牌，并行数据流队列 bank 把原来的统一的**接收队列**分成了四个**接收队列**，并让每一个**接收队列**连接到一个指针阵列。而保证令牌被路由到正确的**接收队列**的任务则被交给了交换网络来处理（第 5.3.2节）。

接下来，图 5-10详细地对比了原始的数据流队列 bank 和并行数据流队列 bank 。

1. 在灰底虚边矩形中，发送队列和接收队列的数量从一个增加到了四个，其中四个表示有三个源操作数和一个目的操作数。本文假设每条指令最多有三个源操作数。（本设计目标指令集是 RV64-G，其中包含三个源操作数的 FMA 指令 [140]。）

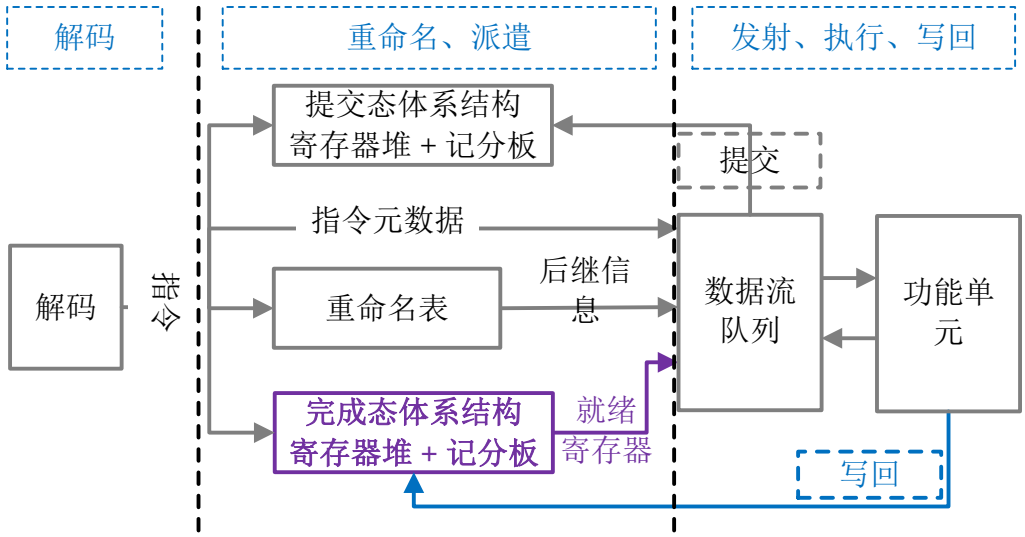


图 5-9 完成即写回对数据通路的修改

Figure 5-9 How writeback on completion (WoC) modifies the data path.

2. 在紫色斜纹实边矩形中，每个发送队列和接收队列都直接绑定到了对应的指针阵列。这样，不同队列中的非关键令牌就可以并行地被消耗。这也要求所有到达的令牌都已经被正确地路由到了对应的接收队列。该要求通过交换网络来实现，会在第 5.3.2节中展开讨论。
3. 在红色横纹实边矩形中，*N-Busy* 数组记录了每条指令的未就绪的操作数数量。并行数据流队列 bank 通过查询 *N-Busy* 数组来判断到达的令牌是关键令牌还是非关键令牌。如果发现多个关键令牌，则只有其中一个可以被授予访问寄存器值阵列和元数据阵列的权利。
4. 在蓝色网格纹实边矩形中，被授权的关键令牌可以访问寄存器值阵列和元数据阵列，获得发射所需信息。其他的关键令牌必须在接收队列中等待。与此相比，Forwardflow 把所有到达的令牌都当作关键令牌，并总是授予它们访问寄存器值阵列和元数据阵列的权利。

并行数据流队列 bank 的详细例子。图 5-11展示了一个并行数据流队列 bank 同时处理两个关键令牌（令牌 1: I5.R1 和 令牌 3: I1.R3）和一个非关键令牌（令牌 2: I9.R2）的例子。这些令牌可以并行地访问指针阵列并生成送往后继指令的令牌（①、②、③）。三个令牌并行访问 *N-Busy* 数组，其中令牌 1 和 令牌 3 被发现是关键令牌（④）。并行数据流队列 bank 每个周期最多只允许一个关键令牌访问寄存器值阵列和元数据阵列。当令牌 1 被授权访问后，它从目的指针阵列中读取了指令 I5 的目的寄存器的后继 I7.R3（⑥），I7.R3 需要在下一个周期中推测唤醒指令 I7 的第 3 个源操作数。与此同时，令牌 1 从源指针阵列中读取了指令 I5 的源寄存器的值和元数据（⑤）。从⑤和⑥中读取的信息会被打包并发送到功能单元（⑦）。与令牌 1 不同，令牌 3 被拒绝授权，并必须在接收队列中等待。注意，虽然令牌 3 对应的指令无法发射，但是因为令牌 3 可以读取指针阵列，所以

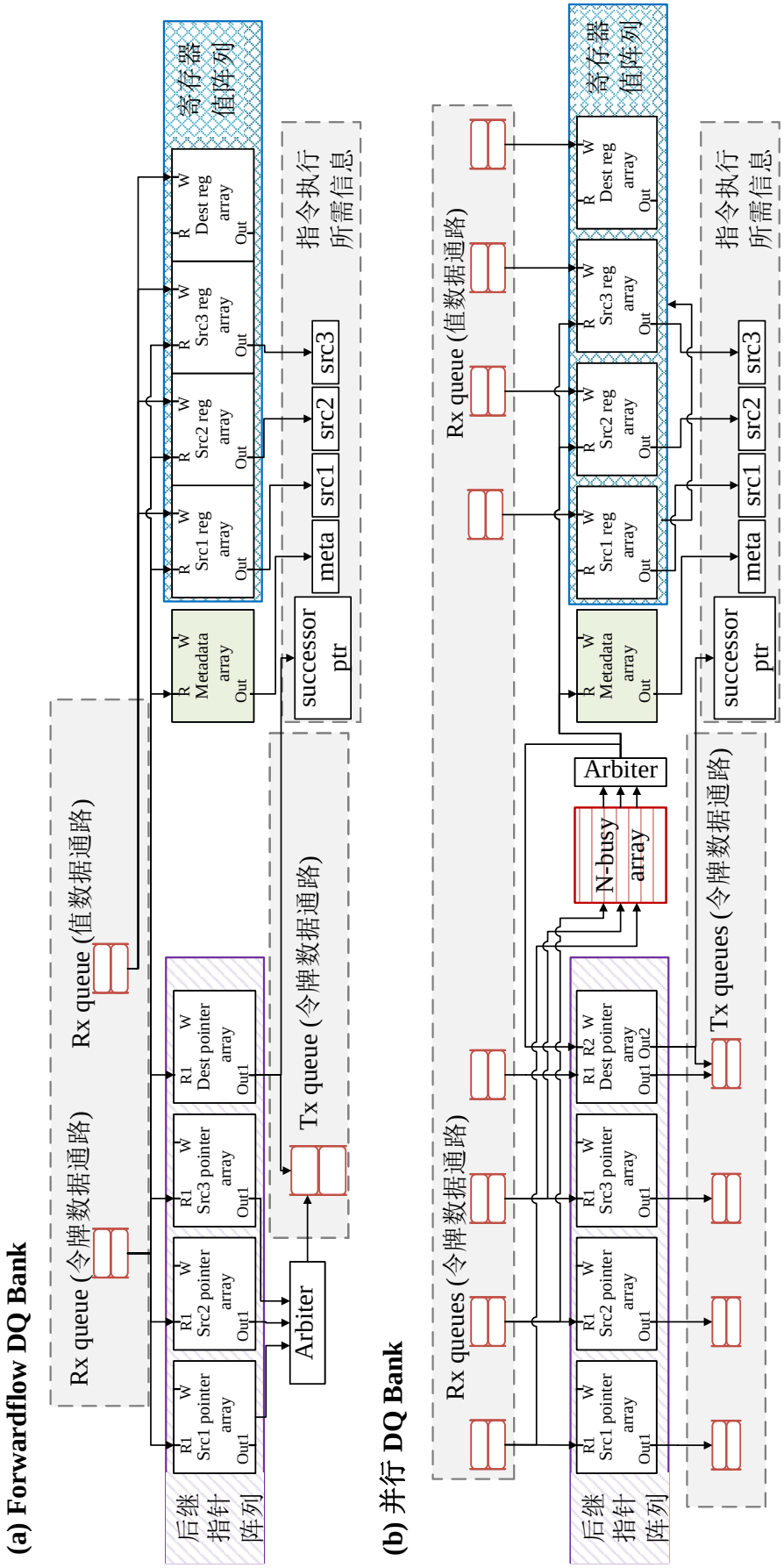


图 5-10 并行数据流队列 bank 与原始数据流队列 bank 的对比

Figure 5-10 Comparison between the original DQ bank and the parallel DQ bank

它的后继令牌可以在这个周期中发送，当下一个周期处理令牌 3 时，令牌 3 就会被标记、不生成后继令牌。

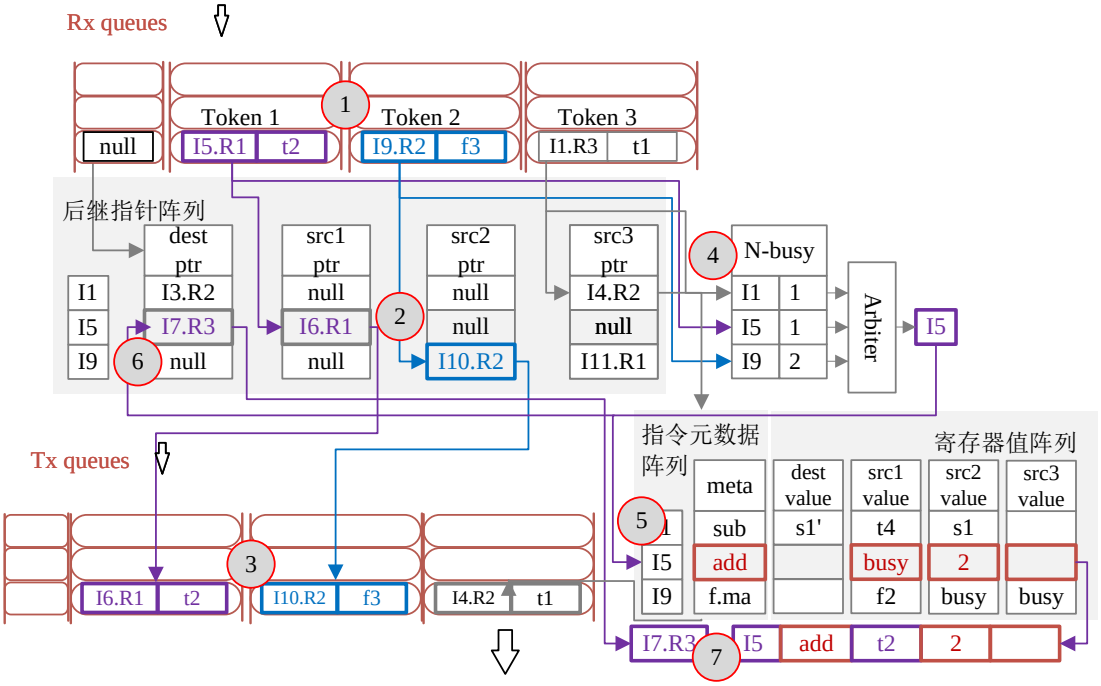


图 5-11 并行数据流队列 bank 处理多个令牌的流程

Figure 5-11 The workflow of parallel DQ bank processing multiple tokens.

不难看出，检查 *N-Busy* 数组的逻辑可能会增长关键路径、降低时钟频率。为了避免这个问题，本文给目的寄存器指针阵列增加了更多的端口，并且让收到的令牌并行访问目的寄存器指针阵列。根据 5.4.5节中的综合结果，这样设计可以缩短关键路径。注意，因为访问寄存器值阵列和元数据阵列可以被流水化、在下一个周期中处理，无需进行优化。流水化的发射阶段和寄存器读取阶段也不会破坏指令的连续发射。

5.3.2 互联网络

本节介绍 Omegaflow 中组内互联网络的设计考虑。在功能方面，Omegaflow 中组内互联网络的设计目标是将令牌路由到正确的并行数据流队列 bank 的接收队列中。在 Forwardflow 中的 crossbar 只需要将令牌路由到相应的数据流队列 bank，不存在多个接收队列。在性能方面，组内互联网络需要提高令牌在各个数据流队列 bank 间的交换速率，使之与并行数据流队列 bank 消耗令牌的速率相匹配。

互联网络的可选方案。本文保持和 Forwardflow 一样的规模：假设一个数据流队列 组有四个并行数据流队列 bank 。因为每个并行数据流队列 bank 有四个发送队列和接收队列，所以组内互联网络的输入输出端口各有 16 个。为了满足上述目标，一种选择是采用一个更大的 crossbar，比如一个 16x16 的 crossbar，或

者一个 8x8 的 crossbar 加上额外的路由逻辑。另一个选择是利用**多阶段互联网络**（MIN）[141]，比如 Omega 网络。本文将会评估这两种方案对性能和时序造成的影响。

路由和仲裁。在图 5-5中已经介绍了 crossbar 的路由和仲裁过程，在此本文补充介绍 Omega 网络的路由和仲裁过程。图 5-12展示了一个 16x16 的 Omega 网络将令牌路由到目的地的过程。**路由**：Omega 网络中的每个单元节点根据目的地的二进制表示来转发令牌。例如，令牌 1 的目的地是 *B1.I3.R2*，对应 bank 1 的第二个源寄存器阵列。该目的地可以用 *0110* 表示，即 01 表示 bank 1，10 表示第二个源寄存器阵列。根据上述表示，路由过程中经过的四个单元节点会把令牌转发到相应的端口：0 选上端口，1 选下端口。**仲裁**：图 5-12还展示了令牌 B1.I3.R2 和 令牌 B1.I1.R3 在节点 *Row2.Column3* 中的冲突。当发生冲突时，仲裁单元从上下的端口轮流选择一个令牌。在这个例子中，令牌 B1.I1.R3 必须等待到下一个周期，而令牌 B1.I3.R2 可以被转发到目的地。

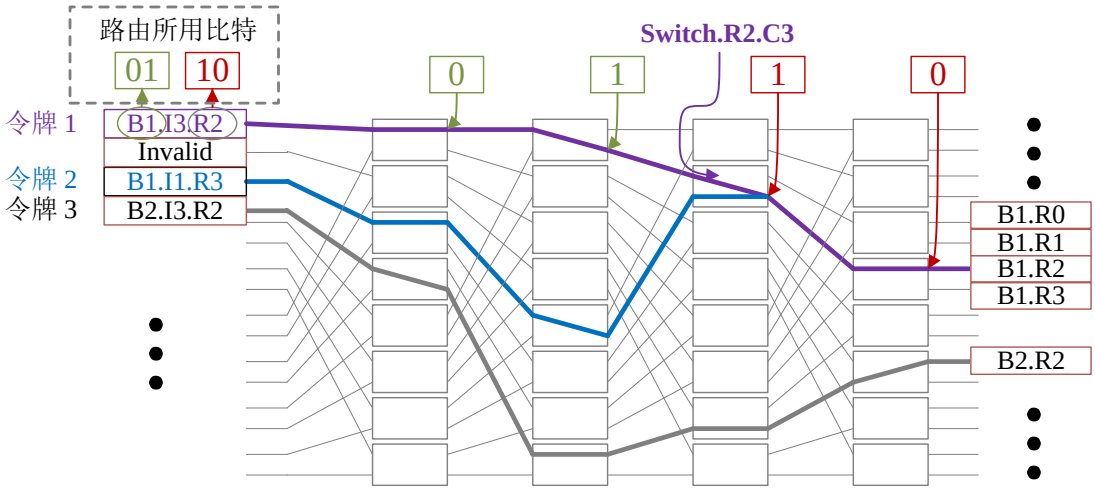


图 5-12 在 Omega 网络中进行路由和仲裁

Figure 5-12 Routing and arbitration in Omega network

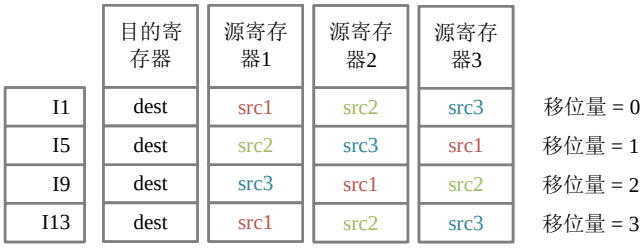


图 5-13 通过操作数混洗使得接收队列负载更均衡

Figure 5-13 Achieving better load balance with operand shuffle

接收队列的负载均衡。在实际情况下，令牌目的地分布并不均匀。从汇编指令的语义上看，大多数指令都有一个目的寄存器和至少一个源寄存器，但是只有

少数指令有三个源寄存器。因此，目的寄存器阵列和第一个源寄存器阵列对应的接收队列的负载要比其它接收队列大。为了解决这个问题，本文对指令的源操作数进行随机化的映射，这一操作在重命名和分发阶段进行。源操作数根据在数据流队列 bank 中的索引进行循环移位，如图 5-13所示。注意，循环移位后，指针链表和令牌仍然可以跟踪到正确的依赖关系，因为循环移位后的位置被记录在重命名表中。当指令被发射时，需要进行反向的循环移位，以恢复源操作数的正确语义。通过这样的方法，接收队列的负载均衡可以得到改善 [142]。另外，本文没有让目的寄存器操作数参与循环移位，因为这样做会增加设计实现的复杂度。

5.3.3 完成即写回

完成即写回的目的是通过增加一个**完成状态 ARF**（*Completed ARF*）使得指令可以在计算完成时立即写回到 ARF 中，从而缩短 SSR 链表长度。完成即写回的动机已经在第 5.2.4节中解释过。在此，本文先通过一个例子来说明完成即写回如何防止 SSR 链表过长。然后，讨论如何维护完成状态 ARF 和 Forwardflow 中现有的提交状态 ARF。

如果没有完成状态 ARF，如图 5-14(b) 所示，指令 I4 依赖于 I1 的目的寄存器（R1），但是 I4 无法在重命名、分发时获得 R1 已经被计算出来的结果。这导致了不必要的指针（紫色加粗实线箭头），该指针从 I3.src1 指向 I4.src。相反地，通过完成状态 ARF，可以将寄存器的最新状态传递给重命名阶段。在图 5-14 (c) 中，当 I4 被重命名、分发时，它可以直接从完成状态 ARF 中获取 R1 的值。在这种情况下，I4 有机会被更早地发射，并且完成状态 ARF 可以将 SSR 链表的长度从 3 缩短到 2。

为了维护完成状态 ARF，每当一个指令计算完成时，对应的功能单元会向完成状态 ARF 发送一个写回请求（图 5-9）。写回请求包括寄存器标签、结果和指令的数据流队列 ID（相当于 OoO 中的 ROB ID）。只有当数据流队列 ID 与最后一个将该寄存器标记为忙碌的指令的 ID 匹配时，完成状态 ARF 才会被更新。确保 ID 匹配是为了避免一个较老的指令覆盖掉由年轻指令设置的忙碌状态。

虽然有了完成状态 ARF，但是为了维护精确的体系结构状态，不能删除提交状态 ARF。每当发生异常或分支预测错误时，为了恢复完成状态 ARF，只需要将提交状态 ARF 中的内容复制到完成状态 ARF 中即可（同时也会复制寄存器计分板）。而重命名表则通过从数据流队列中遍历得到，与传统 OoO 处理器的重命名表恢复过程类似。

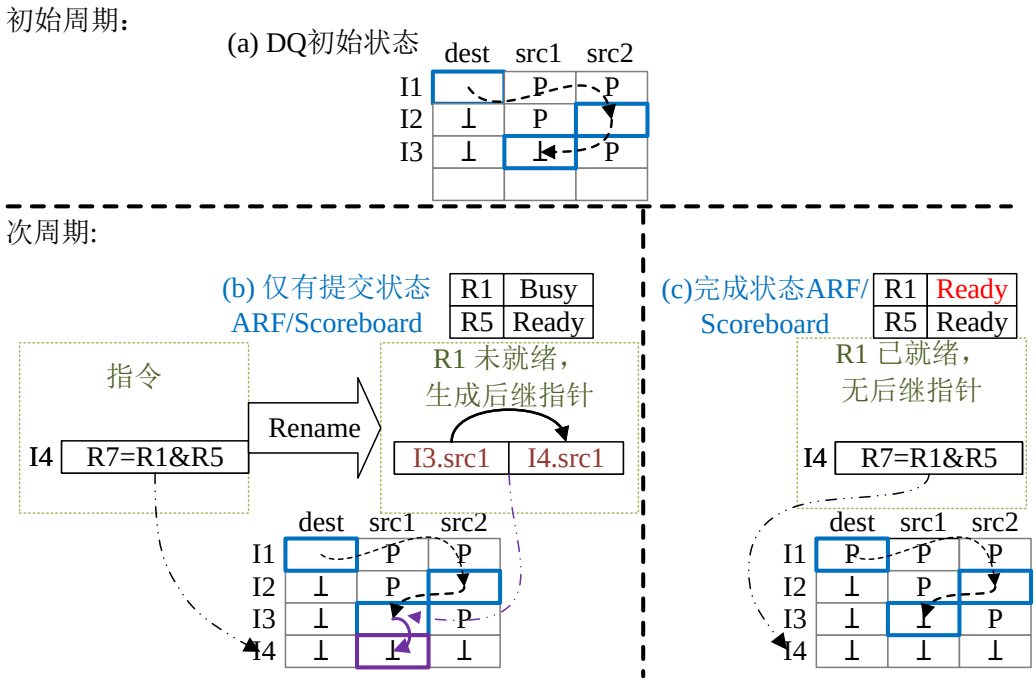


图 5-14 完成即写回的作用

Figure 5-14 How writeback on completion works

5.4　实验评估

在本节中，本文首先比较 Forwardflow 与 Omegaflow 的整体性能提升，对性能收益进行分解（5.4.2节）。然后分别展示缩短 SSR 链带来的收益（5.4.3节）和提升数据流队列组的令牌消耗速率带来的收益（5.4.4节）。然后给出了 Omegaflow 的开销估计（5.4.5节），这部分的数据表明 Omegaflow 的功耗开销非常小，不会影响时序。最后本文给出了关于扩展性、分支预测和内存消歧的补充实验结果。

5.4.1　性能评估方法论

为了评估 Omegaflow 相较于 Forwardflow 架构的性能提升，本文在 GEM5 [12] 中实现了 Omegaflow 和 Forwardflow，并与传统乱序执行架构进行对比。三种四发宽度的核的配置如表 5-1所示，建模频率为 2GHz。本文使用 SPECCPU 2017 基准测试 [139]（*wrf* 被排除在外，因为生成检查点的过程中出错）。本文使用 Simpoint [27] 为每个基准测试选择模拟点，SimPoint 的 maxK 设置为 30，然后从高权重的模拟点开始选择，以确保每个基准测试的 BBV 覆盖率高于 80%。对于每个模拟点，本文运行 50M 条指令进行预热，然后再运行 50M 条指令来收集统计数据。

时序评估。本文用 Chisel [10] 实现了 Forwardflow 和 Omegaflow 的主要组件，然后用 Synopsys Design Compiler 和 freePDK 15 [143] 对四个 bank 的数据流队列组进行综合。

表 5-1 Omegaflow 实验配置

Table 5-1 Configuration of Omegaflow

组件	乱序核	Omegaflow	Forwardflow
指令窗口 调度器	192 项 ROB 192 项理想化调度器	192 项数据流队列，分 4 个 bank	
LSQ	72 项 Load Queue 48 项 Store Queue	NoSQ [134]	
功能单元	4 个整数算术逻辑单元（+ 地址生成单元） 2 个整数乘除单元 2 个浮点加法器 2 个浮点乘除法法器		
数据流队列 bank	-	并行数据流队列 bank	原始 bank
互联网络	-	16x16 Omega 网络	4x4 Crossbar
完成即写回	-	Yes	No
指令选择	理想化指令选择器 [5]	四个分组调度的功能单元	
分支预测	64KB TAGE-SC-L 分支预测器 [35]		
一级指令缓存和数据缓存	32KB、8 路组相联、2 周期延迟		
二级缓存	2MB、8 路组相联、24 周期延迟，BOP 预取器 [37]		
三级缓存	16MB、16 路组相联、40 周期延迟		
主存	DDR3 1600 8x8		

功耗和面积。因为 Forwardflow 和 Omegaflow 共享缓存、分支预测器和 LSU 等组件，本文只评估了修改后的部分的功耗和面积。具体来说，本文估算了乱序核中的 192 项 ROB、物理寄存器文件（PRF）和一个 80 项调度器的功耗和面积，这是一个典型的实际乱序核可能采用的配置。本文假设调度器是 Circ 调度器，因为 Circ 调度器的行为与 CACTI 提供的标准 CAM 匹配 [144]。本文估算了 Forwardflow 和 Omegaflow 中的数据流队列和 ARF 的功耗和面积（在 Omegaflow 中评估了两组 ARF）。本文用 CACTI [145] 计算这些组件的面积和单次访问功耗，并在 GEM5 [12] 中统计了它们的访问数据来估算整体功耗。上述对比中排除了内存相关的单元（LSQ、StoreSet 和 NoSQ），因为这些不是 Omegaflow 的主要关注点。注意，NoSQ 也适用于乱序核，而 LSQ+Storeset 的配置也可用于 Forwardflow 和 Omegaflow。

5.4.2 整体性能评估

在图 5-15(a) 中，本文比较了不同架构的性能提升。整体上，配置了 16x16 crossbar 的 Omegaflow 比 Forwardflow 提升了 26.3% 的 IPC，而 Omegaflow 与使用 192 项调度器的乱序核的性能差距只有 4.4%。这些性能提升的直接原因是并行数据流队列 bank 减少了令牌的传输延迟和提升了令牌的消耗速率。因为并行数据流队列 bank 的性能不受跨组延迟影响，Omegaflow 的性能提升比 Forwardflow 扩展到两个数据流队列组的性能提升更大。[2]

[2]在本文的实验中，从 Forwardflow 扩展到 Forwardflow +2G 的平均性能提升为 8.4%。而在 Forwardflow 的论文中，这个数字为 11.5%。这些差异可能是因为不同的实现细节、运行负载和架构配置。

每个改进的性能贡献如图 5-15(b) 所示。因为使用 16x16 crossbar 的 Omegaflow 和使用 Omega 网络时，各部分的性能贡献相似，因此本文在性能分解中仅展示了使用 Omega 网络的情况。性能分解表明，提升令牌的消耗速率和提高互联网络带宽对性能提升的贡献最大（因为这两个改进是耦合的，无法对他们的贡献分开评价）。第二大的贡献是通过完成即写回减少了 SSR 链的长度。

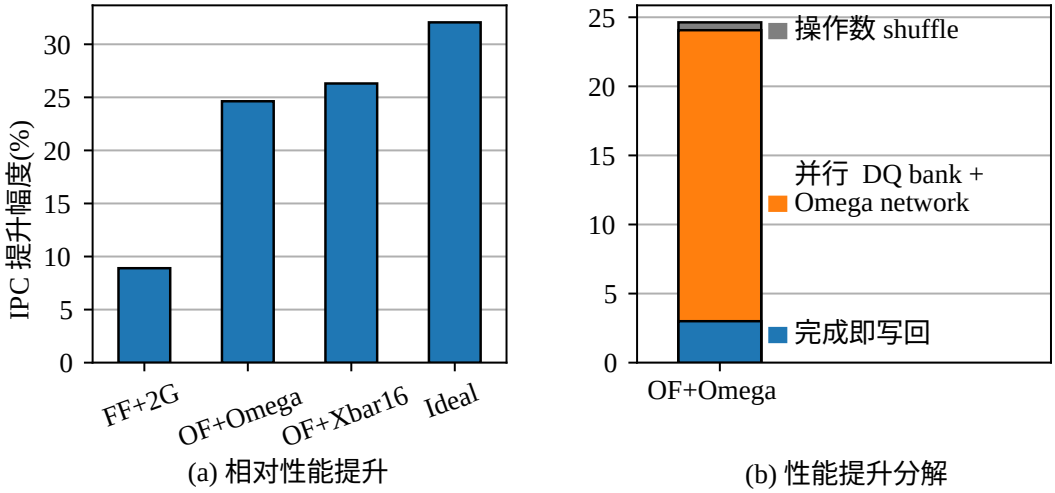


图 5-15 整体性能结果，性能基线为 Forwardflow。本文将 Forwardflow 记为 *FF*，将两个数据流队列组的 Forwardflow 记为 *FF+2G*，将拥有理想调度器的乱序核记为 *Ideal*，将 Omega 网络互联的 Omegaflow 记为 *OF+Omega*，将 16x16 crossbar 互联的 Omegaflow 记为 *OF+Xbar16*。

Figure 5-15 Overall performance results with Forwardflow as the baseline. We refer to Forwardflow as *FF*, to Forwardflow with two DQ groups as *FF+2G*, to an out-of-order core with an ideal scheduler as *Ideal*, to Omegaflow with Omega network as *OF+Omega*, and to Omegaflow with 16x16 crossbar as *OF+Xbar16*.

5.4.3　缩短串行化链带来的收益

本文在 Forwardflow 中使用了完成即写回机制来缩短 SSR 链，它在 Forwardflow 的基础上提升了 3.0% 的性能。接下来，本文从两个角度来分析这个性能提升。首先，完成即写回将 TPI 减少了 14.7%（图 5-16(a)），从而将令牌的总排队周期减少 9.6%（图 5-16(b)）。其次，完成即写回把平均 SSR 延迟周期减少了 53.8%（图 5-16(c)）。这表明完成即写回机制有效削减了 SSR 链的长度。

虽然完成即写回机制相对于其它组件技术贡献较少，但是他对一些 SSR 不友好的工作负载来说是重要的。图 5-17展示了完成即写回机制在 *Perlbench.checkspam* 上有 21.6% 性能提升。这些性能提升是因为令牌的排队周期和 SSR 延迟周期分别减少了 43.7% 和 80.4%。SSR 延迟周期的巨大减少表明，很大比例的 SSR 链都是因为计算结果没有及时传递给新指令导致的，可以被完成即写回机制削减。

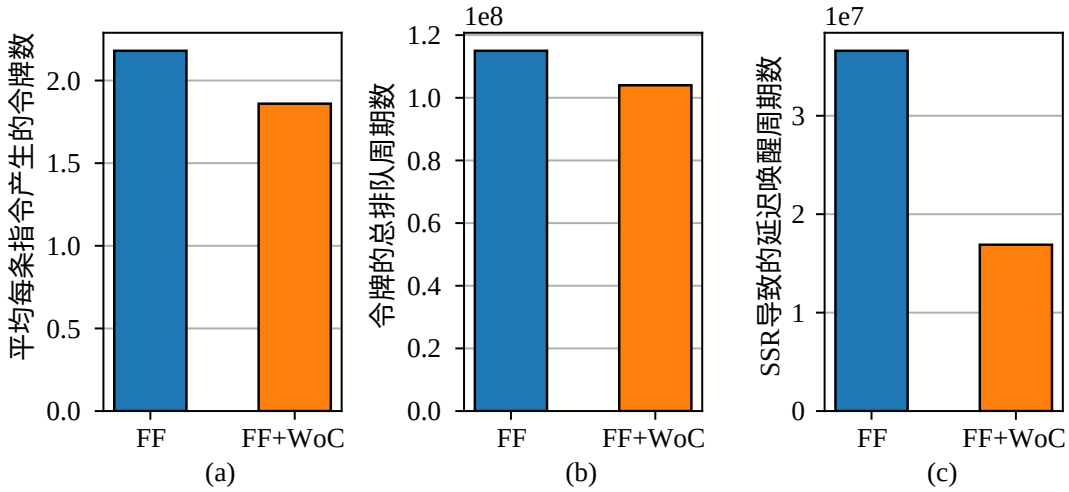


图 5-16 完成即写回带来的平均收益

Figure 5-16 The average performance improvement brought by writeback on compilation

5.4.4 增强令牌吞吐率带来的收益

为了展示提升令牌的消耗速率的作用，本文首先展示了两个仿真点 *imagick* 和 *lbm* 中令牌在互联网络中的流量分布（图 5-18）。流量分布数据表明，Forwardflow 不能满足工作负载的令牌消耗速率需求。尽管令牌的消耗速率需求很高，但 Forwardflow 中的令牌的消耗速率在大多数周期中仅达到 2 或 3。这与 5.2.4节中的计算结果一致，即 Forwardflow 中的 4x4 crossbar 会遇到端口冲突导致带宽下降。而并行数据流队列 bank 和更宽的互联网络可以通过提供更高的令牌消耗速率和传输带宽来满足工作负载的需求。

然后图 5-19表明并行数据流队列 bank 和更宽的互联网络可以降低令牌排队延迟、提升整体性能。这有力地论证了 Forwardflow 中的令牌消耗速率不足以满足工作负载的需求。在提升令牌消耗速率后，*imagick* 和 *lbm* 的令牌排队延迟分别减少了 83.5% 和 90.3%，端到端性能分别提升了 20.4% 和 22.6%。16x16 crossbar 的令牌排队延迟只有 Omega 网络的一半，但端到端性能提升约 2.2%。这表明并行数据流队列 bank 和 Omega 网络提供的令牌吞吐率已经满足了工作负载的需求，进一步提升带来的性能提升边际递减。此外，操作数混洗作为一个几乎没有开销的技术点带来了 0.5% 的性能提升。

namd 是受益最多的负载之一，第 5.2.4节中的分析模型也预测它的性能受到令牌吞吐率的严重影响。Omegaflow 让令牌排队延迟减少了 75.2%，端到端 IPC 提升了 68.4%（图 5-20(a)）。不过，令牌排队延迟的显著减少并非总是带来显著的 IPC 提升。例如，*cactuBSSN* 的令牌排队延迟减少了 92.8%，但 CPI 却只减少了 1.2%（图 5-20(b)）。类似的结果也在 *mcf* 和 *xalancbmk* 上观察到。这些负载要么是控制密集型，要么是内存密集型，它们的理想 IPC 很低。这些负载的性能提升也可以通过本文的分析模型来解释：低 IPC 的应用程序即使有高 TPI ，也不

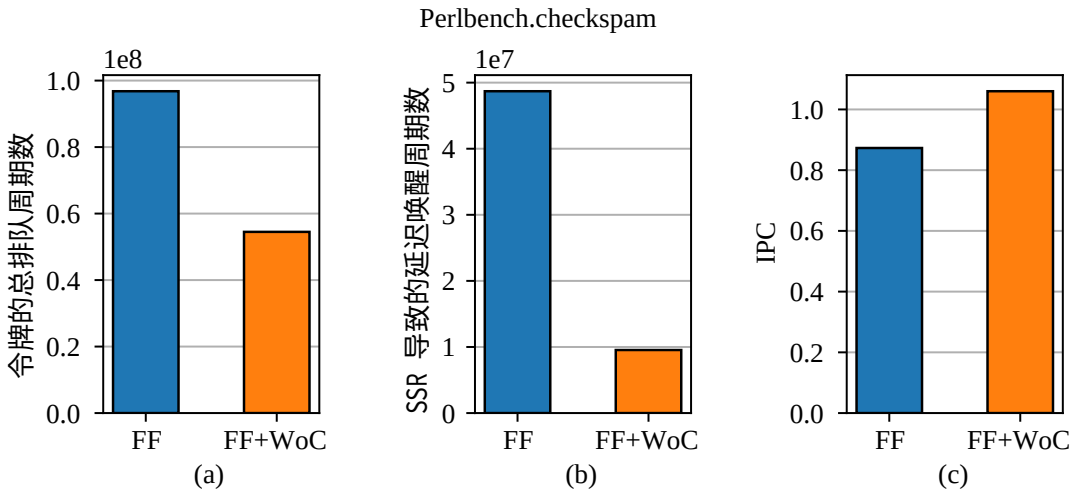


图 5-17 完成即写回给 *perlbench.checkspam* 带来的收益

Figure 5-17 The performance improvement brought by writeback on completion on *perlbench.checkspam*

会受到令牌吞吐率的限制，例如图 5-6中的 *bwaves_2*。

5.4.5　功耗、面积和时序评估

本节展示了 Omegaflow 和 Forwardflow 的功耗、面积和时序，表明 Omegaflow 的带来性能提升的同时不会带来显著的额外开销。在本节，本文提到 Omegaflow 时，指的是利用 Omega 网络互联的版本，而不是用 16x16 crossbar 互联。

表 5-4展示了 Omegaflow 和 Forwardflow 执行 200M 条指令的 SPECCPU 2017 负载的平均功耗和归一化的 IPC。相比于 Forwardflow，Omegaflow 的平均功耗增加了 8.7%，带来了 24.6% 的 IPC 提升。这表明为了 Omegaflow 的性能提升付出的功耗开销是值得的。

在与乱序核进行比较时，首先需要介绍用于比较的调度器。Shift 调度器是最理想的调度器，严格维护指令的年龄，需要复杂的结构，实现 80 项的 Shift 调度器来是不切实际的 [144]。Circ 调度器不严格维护指令的年龄，可以用 CAMs 来实现 [144]。与使用 80 项 Shift 调度器的乱序核相比，Omegaflow 达到了它 94.8% 的性能。与使用 80 项 Circ 调度器的乱序核相比，Omegaflow 达到了它 99.5% 的性能，节省了 8.3% 的能耗。

表 5-4展示了用 CACTI [145] 估算的 Omegaflow 和 Forwardflow 的面积开销。Omegaflow 的面积开销是 21.4%，带来了 24.8% 的性能提升。如果不使用并行数据流队列 bank 技术，而是简单粗暴地使用多端口 RAM 来实现 DQ bank，面积开销会增加 44.9%。这表明利用非关键令牌来提升令牌吞吐率是非常重要的，它可以节省很多面积开销。同时 Omegaflow 的面积开销显著优于使用 80 项 Circ 调度器的乱序执行（表 5-4，0.0987 mm^2 对比 0.1383 mm^2）。需要注意的是，在乱序执行处理器中，不同的指令调度器和物理寄存器堆的设计会导致不同的面积开销。

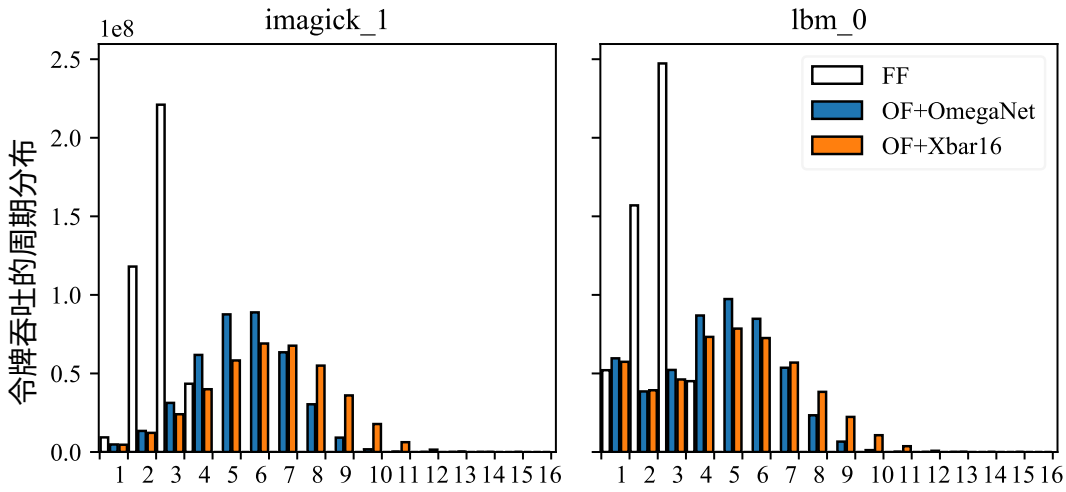


图 5-18 *imagick* 和 *lbm* 的令牌流量分布

Figure 5-18 The token flow distribution of *imagick* and *lbm*

表 5-2 4 种配置下的平均能耗和 IPC。其中，IPC 以 192 项的理想调度器为基准归一化。

Table 5-2 Average energy consumption and IPC of 4 configurations. The IPC is normalized to the ideal scheduler with 192 entries.

	乱序核 + 80 项 Shift 调度器	乱序核 + 80 项 Circ 调度器	FF	OF
能耗/mJ	-	2.632	2.220	2.416
归一化 IPC	0.997	0.949	0.757	0.944

这里只比较了 Circ 的面积，是因为面积建模工具 CACTI 只适合建模 Circ 调度器（基于 CAM），无法准确建模维护年龄序的 Shift 调度器的功耗面积。一般地，严格维护年龄序的 Shift 调度器的面积开销比 Circ 更大，因为 Shift 调度器在 Circ 的基础上额外增加了调度单元移位的功能。

表 5-3展示了将目标时钟周期设置为 160ps 时，DC 综合所得的 Omegaflow 和 Forwardflow 的时序结果。[3] 这些结果表明，与 Forwardflow 相比，Omegaflow 不会影响时序。然而，如果不采用第 5.3.1节中提到的优化技术，关键路径可能会受到影响。[4] 实验结果也说明 16x16 crossbar 可能会对时序产生轻微的影响，因此选择哪种互联网络需要根据 IPC 和频率综合考虑。

[3]FreePDK 15 只能用来比较不同设计的时序，但不能用来估计真实的时钟频率。 [143]

[4]无论是否优化，DQ 的两种实现在 IPC 方面表现都是一致的。

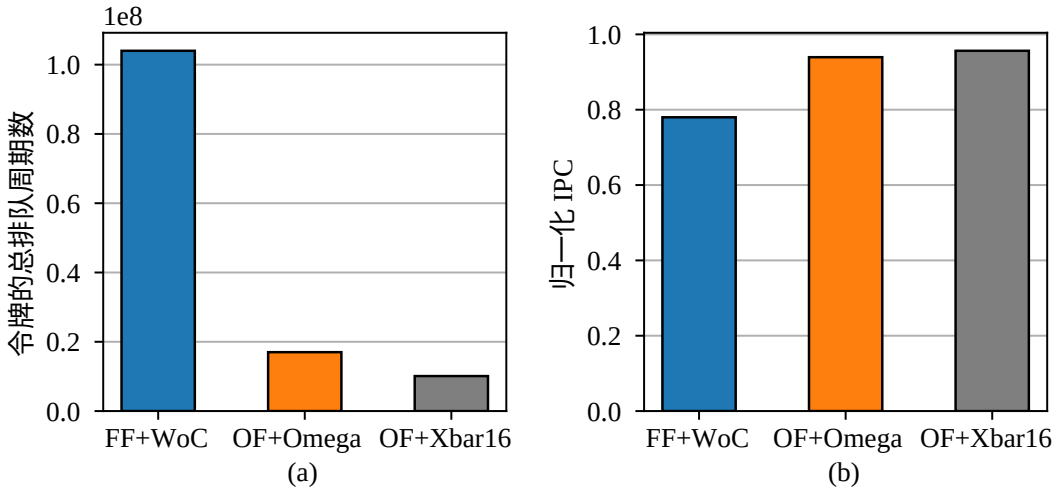


图 5-19 提高令牌吞吐率带来的性能收益

Figure 5-19 The performance improvement brought by increasing the token throughput

表 5-3 4 种不同的数据流队列设计给时序带来的影响

Table 5-3 The impact of 4 different Dataflow Queue designs on timing.

	Forwardflow DQ	Parallel DQ (without optimization)	Parallel DQ	Parallel DQ with Xbar16
时延裕度（slack）/ps	0.09	-18.59	0.0	-6.36

5.4.6　敏感性分析

5.4.6.1　规模扩展对比

本节补充了规模扩展的性能结果：从 4 个 bank 扩展到 8 个 bank，ForwardFlow 架构获得了约 8% 的性能收益，而 Omegaflow 架构获得了 3.5% 的性能收益。实验结果表明，*FF+2G* 相比于 *FF* 平均每条指令减少了 0.43 个周期的排队时间，而 *OF+2G* 相比于 *OF* 每条指令仅减少 0.067 个周期的排队时间，这说明 Forwardflow 的收益更大，是因为 Forwardflow 的单个数据流队列组的令牌吞吐率较低，而更

表 5-4 4 种架构的面积对比

Table 5-4 Area comparison of 4 architectures

	Forwardflow (ARF, DQ)	Forwardflow (ARF, multi-ported DQ)	Omegaflow (2*ARF, parallel DQ)	OoO (Circ Sched, PRF, ROB)
面积/mm^2	0.0813	0.1178	0.0987	0.1383

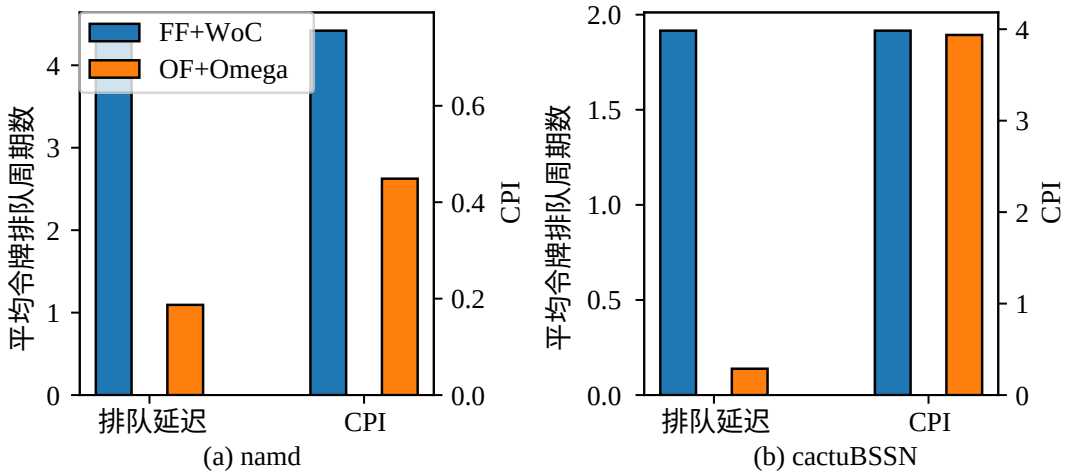


图 5-20 提高令牌吞吐率给 *namd* 和 *cactusBSSN* 带来的性能收益

Figure 5-20 The performance improvement brought by increasing the token throughput on *namd* and *cactusBSSN*

多的数据流队列 bank 可以为 Forwardflow 弥补单个 bank 的令牌吞吐率不足带来的性能损失。但是，从绝对性能来看，*OF+2G* 的性能（IPC）比 *FF+2G* 高 21.1%，说明扩大规模并不能完全弥补 Forwardflow 架构单个数据流 bank 性能孱弱的损失。

此外，Omegaflow 架构扩展到 2 个组后所获得的 3.5% 的性能收益仍然无法弥补其与传统乱序执行架构之间的 5.6% 的性能差距。这是因为 Omegaflow 架构仅解决了流水线后端的可扩展问题，而不能扩宽流水线前端，使得后端流水线提供的带宽无法被充分利用。为了充分发挥 Omegaflow 架构的性能优势，还需要解决流水线前端的可扩展问题，例如分支预测带宽、指令供应速率和重命名带宽等。

5.4.6.2 推测技术的影响

因为推测执行的水平可能会影响架构之间的性能差距，下面两小节对比了 Omegaflow 与乱序核 + 理想调度器在不同的推测执行方法下的性能差距。

关于分支预测器，本文将 TAGE-SC-L 替换为一个 oracle 分支预测器。使用 oracle 分支预测器时，Omegaflow 和乱序核 + 理想调度器的平均性能差距为 4.7%（作为对比，使用 TAGE-SC-L 时，性能差距为 5.6%）。这个结果表明，随着分支预测器的不断进步，更高的准确率不会给 Omegaflow 带来更大的性能差距。因为 Omegaflow 和乱序核 + 理想调度器都有相同大小的指令窗口，更强的分支预测器不会使某一方受到更多的益处。

关于访存依赖推测，本文对比了 Omegaflow 使用 NoSQ [134] 和传统的 LSQ + store set [133] 的性能。NoSQ 带来的平均性能提升只有 0.6%。在 NoSQ 的原始论文中，在乱序核上获得的性能收益也较小。

5.5　本章小结

本章针对 Forwardflow 架构的唤醒信息处理与传递机制，提出了一种分析模型，用于定位性能瓶颈。在该模型的引导下，Omegaflow 架构相比 Forwardflow 架构获得了大幅的性能提升，达到了理想乱序执行 94% 的性能。这表明动态数据流指令调度架构是传统乱序执行的一种可行的替代方案。

第 6 章　总结与未来工作

本文立足于开源处理器（香山处理器）和开源微架构模拟器（GEM5），以支持更大的处理器规模、更先进的微架构设计为目标，重点研究了处理器的性能测算加速，架构迭代加速和指令调度架构的优化。敏捷性能评估框架满足了规模越来越大的开源处理器的性能测评需求。架构迭代基础设施为香山处理器的持续架构迭代铺平了道路。指令调度架构优化工作为处理器规模的进一步扩展提供了可能的方向。

敏捷性能评估框架抓住了 CPU 指令集和缓存的总线协议较为稳定的特点，为快速迭代的架构设计了易于复用的检查点（**RVGCpt**）和混合预热方法（**HyWarm**），使得基于 RTL 软件仿真的性能测算从不可用（数年）变为可用（数十小时）。

在仿真易用性方面，RVGCpt 有两方面的特点：1）对硬件要求低，不依赖于专用硬件，可以在最普通的服务器上运行；2）对仿真环境的要求较低，一般只需要 100-200 行代码即可完成对 RVGCpt 的适配。这两个特点使得 RVGCpt 非常易于推广，成为了香山处理器开源社区的最常用的性能评估手段，被近十家公司和研究机构使用。而 HyWarm 的适配难度比 RVGCpt 更高，使用方法更复杂，它的普及程度更低。未来，为了使 HyWarm 得到更广泛的应用，提高其易用性非常重要。例如可以考虑强化预热框架的通用性。

在仿真时间方面，经过 HyWarm 减少不必要的 RTL 仿真之后，剩余的 RTL 仿真指令数难以进一步削减。为了进一步提升性能测算的速度，加速 RTL 仿真软件的速度至关重要。例如可以考虑实现仿真计算图的按需计算、结合微架构特征进行优化等。

在性能估算的准确率方面，SimPoint [27] 已经是非常经典的算法，它无法识别同一段程序的微架构特征发生变化造成的性能变化。可以考虑结合分支熵 [63]，访存足迹 [146] 和访存局部性 [59] 得到更丰富的程序片段特征，进而实现更准确的采样。

在负载适配方面，采样方法应该加强对新兴应用的支持，例如多线程任务 [147] 和数据中心应用 [79]。因为这些应用表现出与 SPEC CPU 不同的微架构特征 [80]。为了支持新兴应用，一方面需要在全系统仿真的软件层面增强运行时环境，另一方面需要支持基于时间的多核采样方法 [72, 148]。

架构迭代基础设施研究为香山处理器产出了初步对齐的架构模拟器。更重要的是，它为香山处理器和 GEM5 模拟器搭建了架构对齐和架构迭代所需的分析框架（**展开式自顶向下分析**和**并行访存缺失簇分析**），使得未来出现的性能差异、性能缺陷能够被快速地定位。这样的基础设施和方法论并不局限于香山处理器和 GEM5 模拟器，也可以被应用于其它 CPU 和模拟器之间的对齐。

在模拟器对齐方面，希望未来能利用这些基础设施进一步完善 GEM5 与香山处理器的对齐，使得对齐后的 GEM5 能成为学术界性能探索的架构基础，成为开源处理器架构迭代的指南针。

在性能分析方法方面，目前的**展开式自顶向下分析**仍然需要人工分析数据，而且在引入非时间片的量纲之后，可视化变得困难。希望未来的性能分析工具能自动识别出可能的性能瓶颈，并对多量纲的底层性能计数器进行可视化。

Omegaflow 在动态数据流架构的基础上迈进了一小步，为动态数据流架构提供了一种系统性的性能瓶颈分析思路。这种分析思路可以扩展到类似的基于令牌进行指令唤醒的其它架构 [22, 99, 138]。

目前，动态数据流架构虽然拥有强大的流水线后端扩展能力，但是受限于分支预测、取指、寄存器重命名的带宽，它的性能并没有完全得到释放。解耦分支预测器 [34]、多跳转分支预测器 [4] 和分区域取指 [149] 有望克服分支预测和取指的带宽限制。而并行化前端 [150] 可以解决寄存器重命名的带宽限制问题。动态数据流架构或许可以与上述技术相结合从而弥补它目前的流水线前端架构的不足。

附录

表 A-1 不同功能预热方案的总仿真时长对比（所有测试项）

Table A-1 The total emulation time comparison of different warmup schemes (all benchmarks).

时间/h	0+5	0+10	0+25	*Fixed*	*Ada*
总计	35.8	**52.7**	**105.5**	38.5	**54.4**
GemsFDTD	0.37	0.55	1.04	0.42	0.29
astar.bi	0.57	0.91	1.65	0.58	0.64
astar.ri	0.69	0.95	1.97	0.66	0.79
bwaves	0.57	0.92	1.68	0.60	0.43
bzip2.chi	0.30	0.43	0.81	0.30	0.22
bzip2.com	1.00	1.52	2.71	1.01	0.72
bzip2.htm	0.30	0.43	0.92	0.34	0.31
bzip2.lib	0.30	0.42	0.89	0.30	0.21
bzip2.pro	1.01	1.60	3.19	0.98	0.68
bzip2.sou	0.95	1.49	2.92	1.08	0.96
cactusADM	0.41	0.60	1.35	0.47	0.32
calculix	0.35	0.60	1.12	0.36	0.26
dealII	0.33	0.51	1.10	0.40	1.20
gamess.cy	0.33	0.49	1.00	0.36	3.46
gamess.gra	0.35	0.51	1.06	0.38	1.09
gamess.tri	0.33	0.50	0.92	0.34	1.10
gcc.166	0.42	0.61	1.33	0.48	1.34
gcc.200	0.90	1.17	2.72	0.89	0.71
gcc.cpde	0.54	0.86	1.63	0.62	1.75
gcc.expr2	0.58	0.86	1.76	0.63	1.03
gcc.expr	0.63	0.89	1.75	0.61	0.70
gcc.g23	0.55	0.76	1.54	0.66	0.43
gcc.s04	0.57	0.93	1.66	0.67	0.69
gcc.scil	0.90	1.10	2.34	0.94	2.48
gcc.type	0.92	1.44	2.62	0.91	1.57
gobmk.13x	0.94	1.51	3.08	0.99	1.66
gobmk.nn	0.85	1.28	2.61	0.92	0.61
gobmk.sco	0.97	1.34	2.70	0.98	0.66
gobmk.tr	0.95	1.30	2.63	0.87	0.98
gobmk.tr	0.71	1.07	2.26	0.73	1.17

gromacs	0.72	1.00	2.25	0.72	0.48
h264ref.f	0.44	0.58	1.21	0.47	0.45
h264ref.s	0.38	0.50	1.04	0.38	2.23
hmmer.nph	0.77	1.25	2.52	0.85	1.45
hmmer.re	0.80	1.21	2.43	0.92	0.79
lbm	0.67	1.02	2.08	0.74	0.57
leslie3d	0.51	0.78	1.43	0.51	0.35
libquantum	0.56	0.78	1.55	0.98	0.39
mcf	3.14	4.18	9.35	3.34	2.32
milc	0.42	0.59	1.26	0.46	0.34
namd	0.52	0.77	1.38	0.48	0.31
omnetpp	1.08	1.66	3.19	1.27	1.06
perl.che	0.46	0.68	1.29	0.47	0.83
perl.di	0.55	0.83	1.37	0.52	1.56
perl.spli	0.43	0.66	1.31	0.43	0.32
povray	0.55	0.88	1.65	0.54	5.39
sjeng	0.72	1.05	2.00	0.67	2.14
soplex.p	1.15	1.59	3.57	1.36	0.87
soplex.r	1.11	1.70	3.05	1.14	0.71
sphinx3	0.46	0.72	1.33	0.59	1.49
tonto	0.37	0.55	1.19	0.41	0.48
xalancbmk	0.89	1.42	2.56	1.17	1.03
zeusmp	0.51	0.75	1.53	0.58	0.39

参考文献

[1] Asanovic K, Avizienis R, Bachrach J, et al. The rocket chip generator [R]. Berkeley, CA, USA: EECS Department, University of California, Berkeley, 2016.

[2] Celio C, Chiu P F, Asanović K, et al. Broom: an open-source out-of-order processor with resilient low-voltage operation in 28-nm cmos [J]. IEEE Micro, 2019, 39(2): 52-60.

[3] Xu Y, Yu Z, Tang D, et al. Towards developing high performance risc-v processors using agile methodology [C]//Proceedings of the 55th Annual International Symposium on Microarchitecture. Chicago, IL, USA: IEEE Computer Society, 2022: 1178-1199.

[4] Pellegrini A. Arm neoverse n2: Arm's 2nd generation high performance infrastructure cpus and system ips [C]//Proceedings of the 2021 IEEE Hot Chips 33 Symposium. 2021: 1-27.

[5] Palacharla S, Jouppi N P, Smith J E. Complexity-effective superscalar processors [C]// Proceedings of the 24th International Symposium on Computer Architecture. Denver, CO, USA: ACM, 1997: 206-218.

[6] Agarwal V, Hrishikesh M S, Keckler S W, et al. Clock rate versus IPC: the end of the road for conventional microarchitectures [C]//Proceedings of the 27th International Symposium on Computer Architecture. Vancouver, BC, Canada: IEEE Computer Society, 2000: 248-259.

[7] Xilinx Inc. Virtex UltraScale+ [EB/OL]. [2023-04-20]. https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html.

[8] Karandikar S, Mao H, Kim D, et al. FireSim : FPGA-accelerated cycle-exact scale-out system simulation in the public cloud [C]//Proceedings of the 45th Annual International Symposium on Computer Architecture. Los Angeles, CA, USA: IEEE Computer Society, 2018: 29-42.

[9] Veripool.org. Veripool [CP/OL]. [2023-04-18]. https://www.veripool.org/verilator/.

[10] Bachrach J, Vo H, Richards B, et al. Chisel: constructing hardware in a scala embedded language [C]//Proceedings of the Annual Design Automation Conference. San Francisco, CA, USA: ACM, 2012: 1212-1221.

[11] Kessler R E. The ALPHA 21264 microprocessor [J]. IEEE micro, 1999, 19(2): 24-36.

[12] Binkert N, Beckmann B, Black G, et al. The gem5 simulator [J]. ACM SIGARCH Computer Architecture News, 2011, 39(2): 1-7.

[13] Black B, Shen J P. Calibration of microprocessor performance models [J]. Computer, 1998, 31(5): 59-65.

[14] Desikan R, Burger D, Keckler S W. Measuring experimental error in microprocessor simulation [C]//Proceedings of the 28th Annual International Symposium on Computer Architecture. Göteborg, Sweden: ACM, 2001: 266-277.

[15] Nowatzki T, Menon J, Ho C H, et al. Architectural simulators considered harmful [J]. IEEE Micro, 2015, 35(6): 4-12.

[16] Smith J E. A top-down approach to architecting CPI component performance counters [J]. IEEE Micro, 2007, 27(1): 84-93.

[17] Lv Y, Sun B, Luo Q, et al. Counterminer: mining big performance data from hardware counters [C]//Proceedings of the 51st Annual International Symposium on Microarchitecture: 2018-Octob. Fukuoka, Japan: IEEE Computer Society, 2018: 613-626.

[18] Kabylkas N, Thorn T, Srinath S, et al. Effective processor verification with logic fuzzer

enhanced co-simulation [C]//Proceedings of the 54th Annual International Symposium on Microarchitecture. Virtual Event, Greece: IEEE/ACM, 2021: 667-678.

[19] https://github.com/OpenXiangShan. Xiangshan out-of-order processor [CP/OL]. 2021[2023-04-18]. https://github.com/OpenXiangShan/XiangShan.

[20] Ramírez, Marco A and Cristal, Adrian and Veidenbaum, Alexander V and Villa, Luis and Valero M. Direct instruction wakeup for out-of-order processors [C]//Proceedings of the 2004 Innovative Architecture for Future Generation High-Performance Processors and Systems. Maui, HI, USA: IEEE Computer Society, 2004: 2-9.

[21] Gibson D, Wood D A. Forwardflow: a scalable core for power-constrained CMPs [C]// Proceedings of the 37th International Symposium on Computer Architecture. Saint-Malo, France: ACM, 2010: 14-25.

[22] Sankaralingam K, Nagarajan R, Liu H, et al. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture [C]//Proceedings of the 30th Annual International Symposium on Computer Architecture. San Diego, CA, USA: IEEE Computer Society, 2003: 422-433.

[23] Swanson S, Michelson K, Schwerin A, et al. WaveScalar [C]//Proceedings of the 36th Annual International Symposium on Microarchitecture. San Diego, CA, USA: IEEE Computer Society, 2003: 291-302.

[24] Burger D, Keckler S W, McKinley K S, et al. Scaling to the end of silicon with EDGE architectures [J]. Computer, 2004, 37(7): 44-55.

[25] Carlson T E, Heirman W, Allam O, et al. The load slice core microarchitecture [C]// Proceedings of the 42nd Annual International Symposium on Computer Architecture. Portland, OR, USA: ACM, 2015: 272-284.

[26] Kumar R, Alipour M, Black-Schaffer D. Freeway: maximizing MLP for slice-out-of-order execution [C]//Proceedings of the International Symposium on High-Performance Computer Architecture. Washington, DC, USA: IEEE Computer Society, 2019: 558-569.

[27] Sherwood T, Perelman E, Hamerly G, et al. Automatically characterizing large scale program behavior [C]//Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems. San Jose, CA, USA: ACM, 2002: 45-57.

[28] Eyerman S, Eeckhout L, Karkhanis T, et al. A performance counter architecture for computing accurate CPI components [C]//Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. San Jose, CA, USA: ACM, 2006: 175-184.

[29] Yasin A. A top-down method for performance analysis and counters architecture [C]// Proceedings of the International Symposium on Performance Analysis of Systems and Software. Monterey, CA, USA: IEEE Computer Society, 2014: 35-44.

[30] 余子濠. 指令集模拟器的软硬件协同加速技术研究 [D]. 中国科学院计算技术研究所, 2022.

[31] Li P S, Izraelevitz A M, Bachrach J. Specification for the FIRRTL Language [R]. Berkeley, CA, USA: EECS Department, University of California, Berkeley, 2019.

[32] Synopsys Inc. VCS - Functional Verification Solution - Synopsys [CP/OL]. [2023-04-07]. https://www.synopsys.com/verification/simulation/vcs.html.

[33] Cook H, Terpstra W, Lee Y. Diplomatic design patterns: a TileLink case study [C]// Proceedings of the 1st Workshop on Computer Architecture Research with RISC-V. Boston, MA, USA, 2017: 23.

[34] Reinman G, Austin T, Calder B. Scalable front-end architecture for fast instruction delivery [C]//Proceedings of the 26th Annual International Symposium on Computer Architecture: May. Atlanta, GA, USA: IEEE Computer Society, 1999: 234-245.

[35] Seznec A. A new case for the TAGE branch predictor [C]//Proceedings of the 44rd Annual International Symposium on Microarchitecture. Porto Alegre, Brazil: ACM, 2011: 117-127.

[36] Seznec A, Michaud P. A case for (partially) tagged geometric history length branch prediction [J]. J. Instr. Level Parallelism, 2006, 8.

[37] Michaud P. Best-offset hardware prefetching [C]//Proceedings of the International Symposium on High-Performance Computer Architecture. Barcelona, Spain: IEEE Computer Society, 2016: 469-480.

[38] Somogyi S, Wenisch T F, Ailamaki A, et al. Spatial memory streaming [C]//Proceedings of the 33rd International Symposium on Computer Architecture. Boston, MA, USA: IEEE Computer Society, 2006: 252-263.

[39] Putnam A, Caulfield A M, Chung E S, et al. A reconfigurable fabric for accelerating large-scale datacenter services [C]//Proceedings of the 41st Annual International Symposium on Computer Architecture: number 3. Minneapolis, MN, USA: IEEE Computer Society, 2014: 13-24.

[40] Morales-Sandoval M, Feregrino-Uribe C. On the design and implementation of an FPGA-based lossless data compressor [J]. Proceedings of the 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, 2015: 52-59.

[41] Ovtcharov K, Ruwase O, Kim J y, et al. Accelerating deep convolutional neural networks using specialized hardware [J]. Microsoft Research Whitepaper, 2015: 3-6.

[42] Fowers J, Ovtcharov K, Papamichael M, et al. A configurable cloud-Scale DNN processor for real-Time AI [J]. Proceedings of the 45th International Symposium on Computer Architecture, 2018: 1-14.

[43] Firestone D, Putnam A, Mundkur S, et al. Azure accelerated networking: Smartnics in the public cloud [C]//Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation. Renton, WA, USA: USENIX, 2018: 51-66.

[44] Biancolin D, Karandikar S, Kim D, et al. FASED: FPGA-accelerated simulation and evaluation of DRAM [C]//Proceedings of the International Symposium on Field-Programmable Gate Arrays. Seaside, CA, USA: ACM, 2019: 330-339.

[45] Xilinx Inc. Integrated Logic Analyzer v6.2 [CP/OL]. [2023-04-07]. https://docs.xilinx.com/v/u/en-US/pg172-ila.

[46] Tan Z, Waterman A, Cook H, et al. A case for FAME: FPGA architecture model execution [C]//Proceedings of the 37th International Symposium on Computer Architecture. Saint-Malo, France: ACM, 2010: 290-301.

[47] Magyar A, Biancolin D, Koenig J, et al. Golden gate: bridging the resource-efficiency gap between ASICs and FPGA prototypes [C]//Proceedings of the International Conference on Computer-Aided Design. Westminster, CO, USA: ACM, 2019.

[48] Ma J, Zuo G, Loughlin K, et al. Debugging in the brave new world of reconfigurable hardware [C]//Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems. Lausanne, Switzerland: ACM, 2022: 946-962.

[49] Iskander Y S, Patterson C D, Craven S D. Improved abstractions and turnaround time for FPGA design validation and debug [C]//Proceedings of the 21st International Conference

on Field Programmable Logic and Applications. Chania, Greece: IEEE Computer Society, 2011: 518-523.

[50] Kim D, Izraelevitz A M, Celio C, et al. Strober: Fast and Accurate Sample-Based Energy Simulation for Arbitrary RTL [C]//Proceedings of the 43rd Annual International Symposium on Computer Architecture. Seoul, South Korea: IEEE Computer Society, 2016: 128-139.

[51] Hung W N, Sun R. Challenges in large FPGA-based logic emulation systems [C]// Proceedings of the 2018 International Symposium on Physical Design. Monterey, CA, USA: ACM, 2018: 26-33.

[52] Agnesina A, Lim S K, Lepercq E, et al. Improving FPGA-based logic emulation systems through machine learning [J]. ACM Transactions on Design Automation of Electronic Systems, 2020, 25(5): 46:1-46:20.

[53] Biancolin D, Magyar A, Karandikar S, et al. Accessible, FPGA Resource-Optimized Simulation of Multiclock Systems in FireSim [J]. IEEE Micro, 2021, 41(4): 58-66.

[54] Cadence Inc. Palladium Emulation | Cadence [CP/OL]. [2022-12-22]. https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html.

[55] Siemens Inc. Veloce Hardware-assisted Verification System | Siemens Software [CP/OL]. [2023-01-08]. https://eda.sw.siemens.com/en-US/ic/veloce/.

[56] Synopsys Inc. Emulation | Synopsys [CP/OL]. [2023-01-08]. https://www.synopsys.com/verification/emulation.html.

[57] Beamer S, Donofrio D. Efficiently exploiting low activity factors to accelerate RTL simulation [C]//Proceedings of the Annual Design Automation Conference: 2020-July. San Francisco, CA, USA: IEEE, 2020.

[58] Wunderlich R E, Wenisch T F, Falsafi B, et al. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling [C]//Proceedings of the 30th International Symposium on Computer Architecture. San Diego, CA, USA: IEEE Computer Society, 2003: 84-97.

[59] Carlson T E, Heirman W, Van Craeynest K, et al. BarrierPoint: Sampled simulation of multi-threaded applications [C]//Proceedings of the International Symposium on Performance Analysis of Systems and Software. Monterey, CA, USA: IEEE Computer Society, 2014: 2-12.

[60] Ding C, Zhong Y. Predicting whole-program locality through reuse distance analysis [C]// Proceedings of the Conference on Programming Language Design and Implementation. San Diego, CA, USA: ACM, 2003: 245-257.

[61] Shen X, Zhong Y, Ding C. Predicting locality phases for dynamic memory optimization [J]. Journal of Parallel and Distributed Computing, 2007, 67(7): 783-796.

[62] Eeckhout L, Luo Y, De Bosschere K, et al. BLRL: Accurate and efficient warmup for sampled processor simulation [J]. Computer Journal, 2005, 48(4): 451-459.

[63] Pestel S D, Eyerman S, Eeckhout L. Micro-architecture independent branch behavior characterization [C]//Proceedings of the International Symposium on Performance Analysis of Systems and Software. Philadelphia, PA, USA: IEEE Computer Society, 2015: 135-144.

[64] Mattson R L, Gecsei J, Slutz D R, et al. Evaluation techniques for storage hierarchies [J]. IBM System Journal, 1970, 9(2): 78-117.

[65] Nikoleris N, Eklov D, Hagersten E. Extending statistical cache models to support detailed

pipeline simulators [C]//Proceedings of the International Symposium on Performance Analysis of Systems and Software. Monterey, CA, USA: IEEE Computer Society, 2014: 86-95.

[66] 周志华. 机器学习 [M]. 清华大学出版社, 2016.

[67] MacQueen J, Others. Some methods for classification and analysis of multivariate observations [C]//Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability. Oakland, CA, USA: Statistical Laboratory, University of California, Berkeley, 1967: 281-297.

[68] Luo Y, John L K, Eeckhout L. Self-monitored adaptive cache warm-up for microprocessor simulation [C]//Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing. Foz do Iguacu, Brazil: IEEE Computer Society, 2004: 10-17.

[69] Sandberg A, Nikoleris N, Carlson T E, et al. Full speed ahead: detailed architectural simulation at near-native speed [C]//Proceedings of the International Symposium on Workload Characterization. Atlanta, GA, USA: IEEE Computer Society, 2015: 183-192.

[70] Nikoleris N, Sandberg A, Hagersten E, et al. CoolSim: statistical techniques to replace cache warming with efficient, virtualized profiling [C]//Proceedings of the 16th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation. Konstantinos, Samos Island, Greece: IEEE, 2017: 106-115.

[71] Nikoleris N, Hagersten E, Carlson T E, et al. Directed statistical warming through time traveling [C]//Proceedings of the 52nd Annual International Symposium on Microarchitecture. Columbus, OH, USA: ACM, 2019: 1037-1049.

[72] Ardestani E K, Renau J. ESESC: a fast multicore simulator using time-based sampling [C]// Proceedings of the International Symposium on High-Performance Computer Architecture. Shenzhen, China: IEEE Computer Society, 2013: 448-459.

[73] Grass T, Carlson T E, Rico A, et al. Sampled simulation of task-based programs [J]. IEEE Transactions on Computers, 2019, 68(2): 255-269.

[74] Patil H, Isaev A, Hajiabadi A, et al. ELFies : executable region checkpoints for performance analysis and simulation [C]//Proceedings of the International Symposium on Code Generation and Optimization. Seoul, South Korea: IEEE Computer Society, 2021: 126-136.

[75] Gutierrez A, Pusdesris J, Dreslinski R G, et al. Sources of error in full-system simulation [C]//Proceedings of the International Symposium on Performance Analysis of Systems and Software. Monterey, CA, USA: IEEE Computer Society, 2014: 13-22.

[76] Hughes C J, Pai V S, Ranganathan P, et al. Rsim: Simulating shared-memory multiprocessors with ILP processors [J]. Computer, 2002, 35(2): 40-49.

[77] Akram A, Sawalha L. A survey of computer architecture simulation techniques and tools [J]. IEEE Access, 2019, 7: 78120-78145.

[78] Yasin A, Haj-Yahya J, Ben-Asher Y, et al. A metric-guided method for discovering impactful features and architectural insights for skylake-based processors [J]. ACM Transactions on Architecture and Code Optimization, 2019, 16(4): 46:1-46:25.

[79] Kanev S, Darago J P, Hazelwood K, et al. Profiling a warehouse-scale computer [C]// Proceedings of the 42nd Annual International Symposium on Computer Architecture: 13-17-June. Portland, OR, USA: ACM, 2015: 158-169.

[80] Ferdman M, Adileh A, Kocberber O, et al. Clearing the clouds: A study of emerging scale-out workloads on modern hardware [C]//Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems: Asplos. London, UK: ACM, 2012: 37-47.

[81] Glew A. MLP yes! ILP no [EB/OL]. 1998[2023-04-18]. https://people.eecs.berkeley.edu/~kubitron/asplos98/final.html.

[82] Sun X H, Wang D. Concurrent average memory access time [J]. Computer, 2014, 47(5): 74-80.

[83] Mutlu O, Stark J, Wilkerson C, et al. Runahead execution: An alternative to very large instruction windows for out-of-order processors [C]//Proceedings of the International Symposium on High-Performance Computer Architecture. Anaheim, CA, USA: IEEE Computer Society, 2003: 129-140.

[84] Ronen R, Mendelson A, Lai K, et al. Coming challenges in microarchitecuture and architecture architecture [J]. Proceedings of the IEEE, 2001, 89(3): 325-339.

[85] Hu W, Wang J, Gao X, et al. Godson-3: A Scalable Multicore RISC Processor with x86 Emulation [J]. IEEE Micro, 2009, 29(2): 17-29.

[86] Wallace S, Bagherzadeh N. A scalable register file architecture for dynamically scheduled processors [C]//Proceedings of the 5th International Conference on Parallel Architectures and Compilation Techniques. Boston, MA, USA: IEEE Computer Society, 1996: 179-184.

[87] Cruz J L, González A, Valero M, et al. Multiple-banked register file architectures [C]// Proceedings of the 27th Annual International Symposium on Computer architecture. Vancouver, BC, Canada: IEEE Computer Society, 2000: 316-325.

[88] Abella J, González A. On reducing register pressure and energy in multiple-banked register files [C]//Proceedings of the International Conference on Computer Design. San Jose, CA, USA: IEEE, 2003: 14-20.

[89] Tseng J H, Asanovic K. Banked multiported register files for high-frequency superscalar microprocessors [C]//Proceedings of the 30th Annual International Symposium on Computer Architecture. San Diego, CA, USA: ACM, 2003: 62-71.

[90] Martin M M, Roth A, Fischer C N. Exploiting dead value information [C]//Proceedings of the 30th Annual International Symposium on Microarchitecture. Research Triangle Park, NC, USA: ACM/IEEE Computer Society, 1997: 125-135.

[91] Martinez J F, Renau J, Huang M C, et al. Cherry: checkpointed early resource recycling in out-of-order microprocessors [C]//Proceedings of the 35th Annual International Symposium on Microarchitecture: November. Istanbul, Turkey: ACM/IEEE Computer Society, 2002: 3-14.

[92] Ergin O, Balkan D, Ponomarev D, et al. Increasing processor performance through early register release [C]//Proceedings of the International Conference on Computer Design. San Jose, CA, USA: IEEE Computer Society, 2004: 480-487.

[93] Quinones E, Parcerisa J M, Gonzalez A. Early register release for out-of-order processors with register windows [C]//Proceedings of the Parallel Architectures and Compilation Techniques: October. Brasov, Romania: IEEE Computer Society, 2007: 225-234.

[94] Monreal T, Gonzalez A, Valero M, et al. Delaying physical register allocation through virtual-physical registers [C]//Proceedings of the 32nd Annual International Symposium on Microarchitecture. Haifa, Israel: ACM/IEEE Computer Society, 1999: 186-192.

[95] Kim I, Lipasti M H. Half-price architecture [C]//Proceedings of the 30th International Symposium on Computer Architecture: number 2. San Diego, CA, USA: IEEE Computer Society, 2003: 28-38.

[96] Raasch S E, Binkert N L, Reinhardt S K. A scalable instruction queue design using depen-

dence chains [C]//Proceedings of the 29th Annual International Symposium on Computer Architecture. Anchorage, AK, USA: IEEE Computer Society, 2002: 318-329.

[97] Stark J, Brown M D, Patt Y N. On pipelining dynamic instruction scheduling logic [C]// Proceedings of the 33rd Annual International Symposium on Microarchitecture. Monterey, CA, USA: ACM/IEEE Computer Society, 2000: 57-66.

[98] Vivekanandham R, Amrutur B, Govindarajan R. A scalable low power issue queue for large instruction window processors [C]//Proceedings of the 20th International Conference on Supercomputing: July. Cairns, Queensland, Australia: ACM, 2006: 167-176.

[99] Nikhil R S, Others. Executing a program on the MIT tagged-token dataflow architecture [J]. IEEE Transactions on computers, 1990, 39(3): 300-318.

[100] Traub K R. A compiler for the MIT tagged-token dataflow architecture [D]. Massachusetts Institute of Technology, 1986.

[101] Patt Y N, Hwu W m, Shebanow M. HPS, a new microarchitecture: rationale and introduction [C]//Proceedings of the 18th Annual Workshop on Microprogramming. Pacific Grove, CA, USA: ACM/IEEE, 1985: 103-108.

[102] Nikhil R S. Bluespec System Verilog: efficient, correct RTL from high level specifications [C]//Proceedings of the 2nd International Conference on Formal Methods and Models for Co-Design. San Diego, CA, USA: IEEE Computer Society, 2004: 69-70.

[103] Lockhart D, Zibrat G, Batten C. PyMTL: a unified framework for vertically integrated computer architecture research [C]//Proceedings of the 47th Annual International Symposium on Microarchitecture. Cambridge, United Kingdom: IEEE Computer Society, 2014: 280-292.

[104] Standard Performance Evaluation Corporation. SPEC CPU® 2006 [CP/OL]. 2006[2023-01-26]. https://www.spec.org/cpu2006/.

[105] Wenisch T F, Wunderlich R E, Falsafi B, et al. TurboSMARTS: accurate microarchitecture simulation sampling in minutes [C]//Proceedings of the International Conference on Measurements and Modeling of Computer Systems. Banff, Alberta, Canada: ACM, 2005: 408-409.

[106] Haskins J W, Kevin Skadron J. Memory reference reuse latency: accelerated warmup for sampled microarchitecture simulation [C]//Proceedings of the International Symposium on Performance Analysis of Systems and Software. Austin, TX, USA: IEEE Computer Society, 2003: 195-203.

[107] ARM Inc. Learn the architecture - Introducing AMBA CHI [S/OL]. [2022-11-24]. https://developer.arm.com/documentation/102407/0100.

[108] Coffman E G, Sethi R. A generalized bound on LPT sequencing [C]//Proceedings of the Conference on Computer Performance Modeling Measurement and Evaluation. Cambridge, MA, USA: ACM, 1976: 306-310.

[109] Xiao X. A direct proof of the 4/3 bound of LPT scheduling rule [C]//Proceedings of the 5th International Conference on Frontiers of Manufacturing Science and Measuring Technology. Taiyuan, China: Atlantis Press, 2017: 486-489.

[110] Hassani S, Southern G, Renau J. LiveSim: going live with microarchitecture simulation [C]// Proceedings of the International Symposium on High-Performance Computer Architecture. Barcelona, Spain: IEEE Computer Society, 2016: 606-617.

[111] Vengalam U K R, Sharma A, Huang M C. LoopIn: a loop-based simulation sampling mechanism [C]//Proceedings of the International Symposium on Performance Analysis of Systems and Software. Singapore: IEEE Computer Society, 2022: 224-226.

[112] SiFive Inc. Sifive: Block-Inclusivecache-Sifive [CP/OL]. 2019[2023-01-25]. https://github.com/sifive/block-inclusivecache-sifive.

[113] Barr K C, Pan H, Zhang M, et al. Accelerating multiprocessor simulation with a memory timestamp record [C]//Proceedings of the International Symposium on Performance Analysis of Systems and Software. Austin, TX, USA: IEEE Computer Society, 2005: 66-77.

[114] Seznec A. The L-TAGE branch predictor [J]. Journal of Instruction-Level Parallelism, 2007, 9.

[115] Järvelin K, Kekäläinen J. Cumulated gain-based evaluation of IR techniques [J]. ACM Transactions on Information Systems, 2002, 20(4): 422-446.

[116] Khan T A, Brown N, Sriraman A, et al. Twig: profile-guided BTB prefetching for data center applications [C]//Proceedings of the 54th Annual International Symposium on Microarchitecture. Virtual Event, Greece: ACM, 2021: 816-829.

[117] Ma J, Sui X, Sun N, et al. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD) [C]//Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems. Istanbul, Turkey: ACM, 2015: 131-143.

[118] Kasture H, Sanchez D. Ubik: efficient cache sharing with strict QoS for latency-critical workloads [C]//Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. Salt Lake City, UT, USA: ACM, 2014: 729-742.

[119] Qureshi M K, Patt Y N. Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches [C]//Proceedings of the 39th Annual International Symposium on Microarchitecture. Orlando, Florida, US: IEEE Computer Society, 2006: 423-432.

[120] Leverich J, Kozyrakis C. Reconciling high server utilization and sub-millisecond quality-of-service [C]//Proceedings of the 9th European Conference on Computer Systems. Amsterdam, The Netherlands: ACM, 2014: 4:1-4:14.

[121] Muralidhara S P, Subramanian L, Mutlu O, et al. Reducing memory interference in multicore systems via application-aware memory channel partitioning [C]//Proceedings of the 44th Annual International Symposium on Microarchitecture. Porto Alegre, Brazil: ACM, 2011: 374-385.

[122] Delimitrou C, Kozyrakis C. IBench: Quantifying interference for datacenter applications [C]//Proceedings of the International Symposium on Workload Characterization. Portland, OR, USA: IEEE Computer Society, 2013: 23-33.

[123] Krause K L, Shen V Y, Schwetman H D. Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems [J]. Journal of the ACM, 1975, 22(4): 522-550.

[124] Horowitz E, Sahni S. Exact and approximate algorithms for scheduling nonidentical processors [J]. Journal of the ACM, 1976, 23(2): 317-327.

[125] Hochbaum D S, Shmoys D B. Using dual approximation algorithms for scheduling problems theoretical and practical results [J]. Journal of the ACM, 1987, 34(1): 144-162.

[126] Hochbaum D S, Shmoys D B. Polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach [J]. SIAM Journal on Computing, 1988, 17(3): 539-551.

[127] Graham R L. Bounds for certain multiprocessing anomalies [J]. Bell System Technical Journal, 1966, 45(9): 1563-1581.

[128] Li S, Yang Z, Reddy D, et al. DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator [J]. IEEE Computer Architecture Letters, 2020, 19(2): 110-113.

[129] Eyerman S, Eeckhout L. Per-thread cycle accounting in SMT processors [C]//Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. Washington, DC, USA: ACM, 2009: 133-144.

[130] Ramirez A, Santana O J, Larriba-Pey J L, et al. Fetching instruction streams [C]//Proceedings of the 35th Annual International Symposium on Microarchitecture. Istanbul, Turkey: ACM/IEEE Computer Society, 2002: 371-382.

[131] Santana O J, Ramirez A, Valero M. Enlarging instruction streams [J]. IEEE Transactions on Computers, 2007, 56(10): 1342-1357.

[132] Santana O J, Ramirez A, Valero M. Multiple stream prediction [C]//Proceedings of the 14th International Symposium on High-Performance Computing. Nara, Japan: Springer, 2008: 1-16.

[133] Chrysos G Z, Emer J S. Memory dependence prediction using store sets [C]//Proceedings of the 25th Annual International Symposium on Computer Architecture. Barcelona, Spain: IEEE Computer Society, 1998: 142-153.

[134] Sha T, Martin M M K, Roth A. NoSQ: store-load communication without a store queue [C]// Proceedings of the 39th Annual International Symposium on Microarchitecture. Orlando, FL, USA: IEEE Computer Society, 2006: 285-296.

[135] Monreal T, Viñals V, González A, et al. Hardware schemes for early register release [C]// Proceedings of the 31st International Conference on Parallel Processing. Vancouver, BC, Canada: IEEE Computer Society, 2002: 5-13.

[136] Canal R, González A. A low-complexity issue logic [C]//Proceedings of the 14th International Conference on Supercomputing. Santa Fe, NM, USA: ACM, 2000: 327-335.

[137] Huang M, Renau J, Torrellas J. Energy-efficient hybrid wakeup logic [C]//Proceedings of the International Symposium on Low Power Electronics and Design. Monterey, CA, USA: ACM, 2002: 196-201.

[138] Robatmili B, Li D, Esmaeilzadeh H, et al. How to implement effective prediction and forwarding for fusable dynamic multicore architectures [C]//Proceedings of the International Symposium on High-Performance Computer Architecture. Shenzhen, China: IEEE Computer Society, 2013: 460-471.

[139] Standard Performance Evaluation Corporation. SPEC CPU® 2017 [CP/OL]. 2017[2022-10-20]. https://www.spec.org/cpu2017/.

[140] Waterman A, Lee Y, Patterson D, et al. The RISC-V instruction set manual. Volume 1: user-level ISA, version 2.0 [R]. Berkeley, CA, USA: EECS Department, University of California, Berkeley, 2014.

[141] Wu C L, Feng T Y. On a class of multistage interconnection networks [J]. IEEE transactions on Computers, 1980, 100(8): 694-702.

[142] Mitra D, Cieslak R A. Randomized parallel communications on an extension of the omega network [J]. Journal of the ACM, 1987, 34(4): 802-824.

[143] NC State University. FreePDK15 | NC State EDA [CP/OL]. 2014[2023-04-18]. https://eda.ncsu.edu/freepdk/freepdk15/.

[144] Ando H. SWQUE: a mode switching issue queue with priority-correcting circular queue [C]// Proceedings of the 52nd Annual International Symposium on Microarchitecture. Columbus, OH, USA: ACM, 2019: 506-518.

[145] Muralimanohar N, Balasubramonian R, Jouppi N P. CACTI 6.0: a tool to model large caches [J]. HP laboratories, 2009, 27: 28.

[146] Xiang X, Ding C, Luo H, et al. HOTL: a higher order theory of locality [C]//Proceedings of the 18th Architectural Support for Programming Languages and Operating Systems. Houston, TX, USA: ACM, 2013: 343-356.

[147] Bienia C, Kumar S, Singh J P, et al. The PARSEC benchmark suite: characterization and architectural implications [C]//Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. Toronto, Canada: IEEE Computer Society, 2008: 72-81.

[148] Carlson T E, Heirman W, Eeckhout L. Sampled simulation of multi-threaded applications [C]//Proceedings of the International Symposium on Performance Analysis of Systems and Software. Austin, TX, USA: IEEE Computer Society, 2013: 2-12.

[149] Sharafeddine M, Jothi K, Akkary H. Disjoint out-of-order execution processor [J]. Transactions on Architecture and Code Optimization, 2012, 9(3): 19:1-19:32.

[150] Oberoi P S, Sohi G S. Parallelism in the front-end [C]//Gottlieb A, Li K. Proceedings of the 30th International Symposium on Computer Architecture. San Diego, CA, USA: IEEE Computer Society, 2003: 230-240.

致 谢

如果要问我从读博中获得了什么，那我首先想到的一定不是学位、论文、工作机会，而是认知的彻底改变。我不仅知道了计算机的微处理器是如何工作的，我还明白了我周围的世界是如何工作的。我明白了什么是“重要的事”，理解了“要做重要的事”。其次，我习得了如何高效地去完成重要的事。这种思维方式突破了学科的桎梏，渗透到了我生活的方方面面。

感谢导师孙凝晖老师和包云岗老师。在整个博士期间，二位老师一直给予我无私的指导和帮助。他们给我的研究提供了各方面的支持：从最基本的参考文献建议，到与业界、学界专家的交流机会，再到实验室的硬件资源支持。我觉得两位老师非常可贵的有两点：第一，两位老师极大地影响了我的学术品味，让我明白什么是“重要的问题”，什么是“有意义的工作”，什么是“有价值的研究”。如果说博士最重要的能力是发现问题，那么两位老师就是为我塑造了问题的“估值函数”。第二，老师们在引导我的同时也给了我极大的自由度，对我的兴趣方向给予了支持。老师们并不急功近利，鼓励我们挖掘科研和项目中的“支线任务”。

感谢我的妻子张帆进。感谢她从本科开始支持我做系统结构相关研究，支持我攻读博士学位。感谢她在我最困难的时候陪伴我，鼓励我，哪怕在我世俗意义上非常失败的时候，也持续肯定我的能力和价值。我永远不会忘记在我们拍婚纱照那天收到我第一篇论文[1]的接收函。感谢她在我一无所有的时候选择了我。感谢她让我更好地认识自己，感谢她让我认识更好的自己。

感谢实验室的唐丹老师、王卅老师、解壁伟老师、余子濠老师、甄好老师对我的关心和指导。感谢实验室同学对我的关心和帮助，包括但不限于展旭升、靳鑫、黄博文、徐渊、郭梦影、杨城、刘卉、薛晗、颜志远、常子豪、郭静、王诲喆、刘志刚、张传奇、勾凌睿、韩济民、刘国栋、徐易难、金越、蔺嘉炜、王华强、王凯帆、张林隽、张紫飞、赵雪岩……感谢与我共事的韩博阳、高子博、陈子航、徐岩、段震伟、刘沁雨、张潮等同学。感谢南京大学在本科阶段对我的计算机系统基础的培养，包括但不限于袁春风老师，徐峰老师，许畅老师，蒋炎岩老师，杨若瑜老师，张泽生老师和王帅老师。感谢我的论文评审人陈明宇老师、陈云霁老师、孙广宇老师、叶笑春老师和翟季冬老师对论文的指导。

感谢我的父母，在一个并不富裕的农村家庭支撑我学习 22 年。感谢他们在家庭压力最大的时候，独自扛下压力，让我能专注于学习和科研。我曾在本科毕业时写给他们一封信，说让他们再辛苦几年，换来我更高的上限。我现在距离履行自己的承诺更近了一步。

感谢开源操作系统 Linux 为我整个博士生涯的实验提供了全程支持。感谢

[1]Zhou, Yaoyang, Zihao Yu, Chuanqi Zhang, Yinan Xu, Huizhe Wang, Sa Wang, Ninghui Sun, and Yungang Bao. ”Omegaflow: a high-performance dependency-based architecture.” In Proceedings of the ACM International Conference on Supercomputing, pp. 152-163. 2021.

开源芯片项目香山处理器和开源微架构模拟器 GEM5 为本文所有子课题提供了实验平台。感谢开源操作系统 Android 和 Syberia OS 支撑起了我的日常通讯和娱乐。感谢开源输入法 RIME 帮助我完成了这篇博士论文。我是狂热的开源爱好者，我相信人类的一切信息类基础设施终究走向开源。正因为此，我认为我能够围绕开源处理器和开源微架构模拟器开展博士课题的研究工作是非常幸运的。希望有一天，我能使用基于开源处理器的计算机。

作者简历及攻读学位期间发表的学术论文与研究成果

作者简历：

周耀阳，四川省成都市人，中国科学院计算技术研究所博士研究生。

2013 年 9 月–2017 年 7 月，在南京大学计算机科学与技术系获得理学学士学位。

2017 年 9 月–2023 年 6 月，在中国科学院计算技术研究所攻读博士学位。

已发表（或正式接受）的学术论文：

1. **Zhou, Yaoyang**, Zihao Yu, Chuanqi Zhang, Yinan Xu, Huizhe Wang, Sa Wang, Ninghui Sun, and Yungang Bao. ”Omegaflow: a high-performance dependency-based architecture.” In Proceedings of the ACM International Conference on Supercomputing, pp. 152-163. 2021.
2. **周耀阳**, 韩博阳, 蔺嘉炜, 王凯帆, 张林隽, 余子濠, 唐丹, 王卅, 孙凝晖, 包云岗. HyWarm：针对处理器 RTL 仿真的自适应混合预热方法 [J]. 计算机研究与发展.（已录用）
3. *Jin, Xin, ***Yaoyang Zhou**, Bowen Huang, Zihao Yu, Xusheng Zhan, Huizhe Wang, Sa Wang, Ningmei Yu, Ninghui Sun, and Yungang Bao. ”QoSMT: supporting precise performance control for simultaneous multithreading architecture.” In Proceedings of the ACM International Conference on Supercomputing, pp. 206-216. 2019. (joint first author)
4. Xu, Yinan, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin et al. ”Towards Developing High Performance RISC-V Processors Using Agile Methodology.” In 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 1178-1199. IEEE, 2022.
5. Jin, Xin, Ningmei Yu, **Yaoyang Zhou**, Bowen Huang, Zihao Yu, Xusheng Zhan, Huizhe Wang, Sa Wang, and Yungang Bao. ”Supporting Predictable Performance Guarantees for SMT Processors.” IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences 103, no. 6 (2020): 806-820.

申请或已获得的专利：

1. 周耀阳. 处理器性能评估方法, 装置, 电子设备及可读存储介质: 中国, CN115658455A[P]. 2023.