|  |  |
| --- | --- |
| Computer Architecture Performance Evaluation Methods | 计算机体系结构性能评估方法。 |
| Lieven Eeckhout | Lieven Eeckhout |
| Ghent University | 根特大学 |

|  |  |
| --- | --- |
| ABSTRACT | 摘要 |
| Performance evaluation is at the foundation of computer architecture research and development. Contemporary microprocessors are so complex that architects cannot design systems based on intuition and simple models only. Adequate performance evaluation methods are absolutely crucial to steer the research and development process in the right direction. However, rigorous performance evaluation is non-trivial as there are multiple aspects to performance evaluation, such as picking workloads, selecting an appropriate modeling or simulation approach, running the model and interpreting the results using meaningful metrics. Each of these aspects is equally important and a performance evaluation method that lacks rigor in any of these crucial aspects may lead to inaccurate performance data and may drive research and development in a wrong direction. | 性能评估是计算机体系结构研究和开发的基础。现代微处理器非常复杂，架构师不能仅凭直觉和简单的模型来设计处理器系统。对于引导研发过程走向正确的方向，恰当的性能评估方法至关重要。然而，进行严格的性能评估并非微不足道的，因为性能评估涉及很多方面，比如挑选工作负载、选择合适的建模或仿真方法、运行模型以及使用有价值的度量指标解释结果。每一个方面都同等重要。在这些关键方面中任何一方面缺少严格规划和实施，性能评估都可能得出不正确的性能数据，并可能将研究和开发推向错误的方向。 |
| The goal of this book is to present an overview of the current state-of-the-art in computer architecture performance evaluation, with a special emphasis on methods for exploring processor architectures. The book focuses on fundamental concepts and ideas for obtaining accurate performance data. The book covers various topics in performance evaluation, ranging from performance metrics, to workload selection, to various modeling approaches including mechanistic and empirical modeling. And because simulation is by far the most prevalent modeling technique, more than half the book’s content is devoted to simulation. The book provides an overview of the simulation techniques in the computer designer’s toolbox, followed by various simulation acceleration techniques including sampled simulation, statistical simulation, parallel simulation and hardware-accelerated simulation. | 本书的目标是对当前计算机体系结构性能评估的先进技术进行概述，特别强调探索处理器架构的方法。本书聚焦在获得准确的性能数据的基本概念和想法。这本书涵盖了性能评估的各种话题，从性能度量，到工作负载选择，到各种建模方法（包括机制和经验建模）。由于仿真是目前为止最流行的建模技术，本书一半以上的内容都是关于仿真的。这本书提供了计算机设计师工具箱中的仿真技术的概述。随后介绍各种仿真加速技术，包括采样仿真、统计仿真、并行仿真和硬件加速仿真。 |
| KEYWORDS | 关键词 |
| computer architecture, performance evaluation, performance metrics, workload characterization, analytical modeling, architectural simulation, sampled simulation, statistical simulation, parallel simulation, FPGA-accelerated simulation | 计算机体系结构、性能评估、性能度量指标、工作负载特性、分析建模、架构仿真、采样仿真、统计仿真、并行仿真、FPGA加速仿真 |

|  |  |
| --- | --- |
| Preface | 前言 |
| GOAL OF THE BOOK | 本书的目的 |
| The goal of this book to present an overview of the current state-of-the-art in computer architecture performance evaluation, with a special emphasis on methods for exploring processor architectures. The book focuses on fundamental concepts and ideas for obtaining accurate performance data. The book covers various aspects that relate to performance evaluation, ranging from performance metrics, to workload selection, and then to various modeling approaches such as analytical modeling and simulation. And because simulation is, by far, the most prevalent modeling technique in computer architecture evaluation,more than half the book’s content is devoted to simulation.The book provides an overview of the various simulation techniques in the computer designer’s toolbox, followed by various simulation acceleration techniques such as sampled simulation, statistical simulation, parallel simulation, and hardware-accelerated simulation. | 本书的目标是对当前计算机体系结构性能评估的先进技术进行概述，特别强调探索处理器架构的方法。本书聚焦在获得准确的性能数据的基本概念和想法。这本书涵盖了性能评估的各种话题，从性能度量，到工作负载选择，到各种建模方法（包括机制和经验建模）。由于仿真是目前为止最流行的建模技术，本书一半以上的内容都是关于仿真的。这本书提供了计算机设计师工具箱中的仿真技术的概述。随后介绍各种仿真加速技术，包括采样仿真、统计仿真、并行仿真和硬件加速仿真。 |
| The evaluation methods described in this book have a primary focus on performance. Although performance remains to be a key design target, it no longer is the sole design target. Power consumption, energy-efficiency, and reliability have quickly become primary design concerns, and today they probably are as important as performance; other design constraints relate to cost, thermal issues, yield, etc. This book focuses on performance evaluation methods only, and while many techniques presented here also apply to power, energy and reliability modeling, it is outside the scope of this book to address them. This does not compromise on the importance and general applicability of the techniques described in this book because power, energy and reliability models are typically integrated into existing performance models. | 本书中描述的评估方法主要聚焦在性能上。尽管性能仍然是一个关键的设计目标，但它不再是唯一的设计目标。功耗、能效和可靠性已经迅速成为主要的设计关注点。当前，他们可能和性能一样重要。其他设计约束涉及到成本、热问题、良率等。本书只关注性能评估方法，而这里提出很多方法也适用于功耗、能耗和可靠性建模。这不是在重要性和普适性上，对于本书描述的技术的妥协，而且因为功耗、能耗和可靠性模型通常集成在已有的性能模型中。 |
| The format of the book is such that it cannot present all the details of all the techniques described in the book. As mentioned before, the goal is to present a broad overview of the stateof-the-art in computer architecture performance evaluation methods, with a special emphasis on general concepts. We refer the interested reader to the bibliography for in-depth treatments of the specific topics covered in the book, including three books on performance evaluation [92; 94; 126]. | 这本书的体例使得它不能呈现书中所描述的所有技术的所有细节。如前所述，我们的目标是对计算机体系结构评估方法的最新进展进行广泛的概述，特别强调一般性概念。我们推荐有兴趣的读者阅读参考文献，以深入理解书中涉及的具体话题，包括关于性能评估的三本书[92;94;126]。 |
| BOOK ORGANIZATION | 本书的组织结构 |
| This book is organized as follows, see also Figure 1. | 本书组织结构如下，参见图1。 |
| Chapter 2 describes ways to quantify performance and revisits performance metrics for single-threaded workloads, multi-threaded workloads and multi-program workloads. Whereas quantifying performance for single-threaded workloads is straightforward and well understood, some may still be confused about how to quantify multi-threaded workload performance. This is especially true for quantifying multiprogram performance. This book sheds light on how to do a meaningful multiprogram performance characterization by focusing on both system throughput and job turnaround time. We also discuss ways for computing the average performance number across a set of benchmarks and clarify the opposite views on computing averages, which fueled the debate over the past two decades. | 第2章描述了量化性能的方法，并重新审视了单线程、多线程和多程序工作负载的性能度量。虽然量化单线程工作负载的性能很简单，而且很容易理解，但如何量化多线程工作负载性能仍然令人感到困惑。在量化多程序性能时尤其如此。这本书阐明了如何通过关注系统吞吐量（system throughput）和作业周转时间（job turnaround time）来进行有意义的多程序性能表征。我们还讨论了对一组基准测试程序（benchmark）计算平均性能数字的方法，并澄清了关于计算平均值的相反观点。这些观点在过去20年里引发了辩论。 |
|  | |
| Figure 1: Book’s organization. | 图1：本书组织结构。 |
| Chapter 2 describes ways to quantify performance and revisits performance metrics for single-threaded workloads, multi-threaded workloads and multi-program workloads. Whereas quantifying performance for single-threaded workloads is straightforward and well understood, some may still be confused about how to quantify multi-threaded workload performance. This is especially true for quantifying multiprogram performance. This book sheds light on how to do a meaningful multiprogram performance characterization by focusing on both system throughput and job turnaround time. We also discuss ways for computing the average performance number across a set of benchmarks and clarify the opposite views on computing averages, which fueled the debate over the past two decades. | 第2章描述了量化性能的方法，并重新审视了单线程、多线程和多程序工作负载的性能度量。虽然量化单线程工作负载的性能很简单，而且很容易理解，但如何量化多线程工作负载性能仍然令人感到困惑。在量化多程序性能时尤其如此。这本书阐明了如何通过关注系统吞吐量（system throughput）和作业周转时间（job turnaround time）来进行有意义的多程序性能表征。我们还讨论了对一组基准测试程序（benchmark）计算平均性能数字的方法，并澄清了关于计算平均值的相反观点。这些观点在过去20年里引发了辩论。 |
| Chapter 3 talks about how to select a representative set of benchmarks from a larger set of specified benchmarks. The chapter covers two methodologies for doing so, namely Principal Component Analysis and the Plackett and Burman design of experiment. The idea behind both methodologies is that benchmarks that exhibit similar behavior in their inherent behavior and/or their interaction with the microarchitecture should not both be part of the benchmark suite, only dissimilar benchmarks should. By retaining only the dissimilar benchmarks, one can reduce the number of retained benchmarks and thus reduce overall experimentation time while not sacrificing accuracy too much. | 第三章讨论如何从一组较多的基准测试程序中选择一组具有代表性的基准测试程序。本章涵盖了两种方法，即主成分分析以及Plackett和Burman的实验设计。这两种方法背后的想法是，在固有行为和/或与微架构的交互中表现出相似行为的基准测试程序不应该都包含在基准测试套件中，只有不同行为的基准测试程序应该出现在基准测试套件中。通过只保留不相似的基准测试程序，可以减少基准测试程序的数量，从而减少整体实验时间，同时又不会过多地牺牲精度。 |
| Analytical performance modeling is the topic of Chapter 4. Although simulation is the prevalent computer architecture performance evaluation method, analytical modeling clearly has its place in the architect’s toolbox. Analytical models are typically simple and, therefore, very fast.This allows for using analytical models to quickly explore large design spaces and narrow down on a region of interest, which can later be explored in more detail through simulation. Moreover, analytical models can provide valuable insight, which is harder and more time-consuming to obtain through simulation. We discuss three major flavors of analytical models. Mechanistic modeling or whitebox modeling builds a model based on first principles, along with a good understanding of the system under study. Empirical modeling or black-box modeling builds a model through training based simulation results; a model, typically, is a regression model or a neural network. Finally, hybrid mechanistic-empirical modeling aims at combining the best of worlds: it provides insight (which it inherits from mechanistic modeling) while easing model construction (which it inherits from empirical modeling). | 第四章的主题是分析性能建模。虽然仿真是目前流行的计算机体系结构性能评估方法，但分析建模显然在架构师的工具箱中占有一席之地。分析模型通常是简单的，因此运行非常快。这允许使用分析模型来快速探索大型设计空间，并缩小感兴趣的区域，稍后可以通过仿真进行更详细的探索。此外，分析模型可以提供有价值的观点，而通过仿真获得这样的观点更难和更耗时的。我们讨论了分析模型的三个主要方面。机制建模（Mechanistic modeling）或白盒建模（White-box modeling）建立一个基于基本原理的模型，同时很好地理解正在研究的系统。经验建模（Empirical modeling）或黑箱建模（black-box modeling）通过基于仿真结果的训练建立模型。典型的模型是回归模型或神经网络。最后，机制-经验混合模型旨在结合两方面的优势：它提供洞察力（继承自机制建模），同时简化模型构建（继承自经验建模）。 |
| Chapter 5 gives an overview of the computer designer’s toolbox while focusing on simulation methods. We revisit different flavors of simulation, ranging from functional simulation, (specialized) trace-driven simulation, execution-driven simulation, full-system simulation, to modular simulation infrastructures. We describe a taxonomy of execution-driven simulation, and we detail on ways for how to deal with non-determinism during simulation. | 第五章概述了计算机设计师的工具箱，同时重点介绍了仿真方法。我们回顾了不同种类的仿真，从功能仿真、（专门的）记录（trace）驱动仿真、执行驱动仿真、全系统仿真，到模块化仿真基础组件。我们描述了执行驱动仿真的分类，并详细介绍了如何在仿真期间处理不确定性的方法。 |
| The next three Chapters 6, 7, and 8, cover three approaches to accelerate simulation, namely sampled simulation, statistical simulation and through exploiting parallelism. Sampled simulation (Chapter 6) simulates only a small fraction of a program’s execution. This is done by selecting a number of so called sampling units and only simulating those sampling units. There are three challenges for sampled simulation: (i) what sampling units to select; (ii) how to initialize a sampling unit’s architecture starting image (register and memory state); (iii) how to estimate a sampling unit’s microarchitecture starting image, i.e., the state of the caches, branch predictor, and processor core at the beginning of the sampling unit. Sampled simulation has been an active area of research over the past few decades, and this chapter covers the most significant problems and solutions. | 接下来的第6、7和8章将介绍三种加速仿真的方法，即采样仿真、统计仿真和利用并行加速仿真。采样仿真（第6章）只仿真程序执行的一小部分。这是通过选择一些所谓的采样单元，并且只仿真这些采样单位来实现的。采样仿真有三个挑战：(i)选择什么采样单元；(ii)如何初始化采样单元的架构起始镜像（寄存器和内存状态）；(iii)如何估计采样单元的微架构起始镜像，即在采样单元开始时缓存、分支预测器和处理器核心的状态。采样仿真在过去的几十年里一直是一个活跃的研究领域，这一章涵盖了最重要的问题和解决方案。 |
| Statistical simulation (Chapter 7) takes a different approach. It first profiles a program execution and collects some program metrics that characterize the program’s execution behavior in a statistical way. A synthetic workload is generated from this profile; by construction, the synthetic workload exhibits the same characteristics as the original program. Simulating the synthetic workload on a simple statistical simulator then yields performance estimates for the original workload. The key benefit is that the synthetic workload is much shorter than the real workload, and as a result, simulation is done quickly. Statistical simulation is not meant to be a replacement for detailed cycle-accurate simulation but rather as a useful complement to quickly explore a large design space. | 统计仿真（第7章）采用不同的方法。它首先分析一个程序的执行，并收集一些程序特性度量，以统计的方式描述程序的执行行为。从这个程序特性文件生成一个合成工作负载；通过控制构建过程，合成工作负载表现出与原始程序相同的特征。然后在一个简单的统计仿真器上仿真合成工作负载，生成原始工作负载的性能估计。关键的优势是，合成工作负载比实际工作负载短得多，因此可以快速完成仿真。统计仿真并不是要取代周期精确的详细仿真，而是作为快速探索大型设计空间的有效补充。 |
| Chapter 8 covers three ways to accelerate simulation by exploiting parallelism. The first approach leverages multiple machines in a simulation cluster to simulate multiple fragments of the entire program execution in a distributed way. The simulator itself may still be a single-threaded program. The second approach is to parallelize the simulator itself in order to leverage the available parallelism in existing computer systems, e.g., multicore processors. A parallelized simulator typically exploits coarse-grain parallelism in the target machine to efficiently distribute the simulation work across multiple threads that run in parallel on the host machine. The third approach aims at exploiting fine-grain parallelism by mapping (parts of) the simulator on reconfigurable hardware, e.g., Field Programmable Gate Arrays (FPGAs). | 第8章介绍了利用并行性来加速仿真的三种方法。第一种方法利用仿真集群中的多台机器，以分布式的方式仿真整个程序执行的多个片段。仿真器本身可能仍然是单线程程序。第二种方法是将仿真器本身并行化，以利用现有计算机系统中的并行性，例如多核处理器。并行化的仿真器通常利用本地计算机中的粗粒度并行性，有效地将仿真工作分发给在主机上并行运行的多个线程。第三种方法旨在通过将仿真器（部分）映射到可重构硬件上，如现场可编程门阵列（FPGA），以利用细粒度的并行性。 |
| Finally, in Chapter 9, we briefly discuss topics that were not (yet) covered in the book, namely measurement bias, design space exploration and simulator validation, and we look forward towards the challenges ahead of us in computer performance evaluation. | 最后，在第9章中，我们简要地讨论了书中未涉及的主题，即测量偏差、设计空间探索和仿真器验证，并展望了我们在计算机性能评估方面面临的挑战。 |
| Lieven Eeckhout  June 2010 | Lieven Eeckhout  2010年6月 |

|  |  |
| --- | --- |
| CHAPTER 1 Introduction | 第1章 绪论 |
| Performance evaluation is at the foundation of computer architecture research and development. Contemporary microprocessors are so complex that architects cannot design systems based on intuition and simple models only. Adequate performance evaluation methodologies are absolutely crucial to steer the development and research process in the right direction. In order to illustrate the importance of performance evaluation in computer architecture research and development, let’s take a closer look at how the field of computer architecture makes progress. | 性能评估是计算机体系结构研究和开发的基础。现代微处理器非常复杂，架构师不能仅凭直觉和简单的模型来设计处理器系统。对于引导研发过程走向正确的方向，恰当的性能评估方法至关重要。为了展示性能评估在计算机体系结构研究和开发中的重要性，让我们近距离观察计算机体系结构领域是如何发展的。 |
| 1.1 STRUCTURE OF COMPUTER ARCHITECTURE (R)EVOLUTION | 1.1 计算机体系结构演进的结构 |
| Joel Emer in his Eckhert-Mauchly award speech [56] made an enlightening analogy between scientific research and engineering research (in this case, computer systems research) [55]. The scientific research side of this analogy is derived from Thomas Kuhn’s theory on the evolution of science [110] and describes scientific research as is typically done in five steps; see Figure 1.1(a). The scientist first takes a hypothesis about the environment and, subsequently, designs an experiment to validate the hypothesis — designing an experiment often means taking a random sample from a population.The experiment is then run, and measurements are obtained. The measurements are then interpreted, and if the results agree with the hypothesis, additional refined experiments may be done to increase confidence in the hypothesis (inner loop in Figure 1.1); eventually, the hypothesis is accepted. If the outcome of the experiment disagrees with the hypothesis, then a new hypothesis is needed (outer loop), and this may potentially lead to what Kuhn calls a scientific revolution. | Joel Emer在他获得Eckhert-Mauchly奖的演讲[56]中，对科学研究和工程研究（在这里是计算机系统研究）做了一个启发性的类比[55]。类比中科学研究方面源自Thomas Kuhn关于科学进化的理论[110]，并将典型的科学研究描述为五个步骤，见图1.1 (a)。科学家首先提出一个关于环境的假设，然后设计一个实验来验证这个假设——设计一个实验通常意味着从总体中随机抽取样本。随后进行实验，得到测量结果。接下来对测量结果进行解释。如果结果与假设一致，可以进行额外的细化实验来增加假设的置信度（图1.1的内环）；最终，这个假设被接受了。如果实验结果与假设不一致，那么就需要一个新的假设（外环），这可能会导致Kuhn所说的科学革命。 |
|  | |
| Figure 1.1: Structure of (a) scientific research versus (b) systems research. | 图1.1：(a)科学研究和(b)系统研究的结构。 |
| The procedure in systems research, in general, and computer architecture research and development, in particular, shows some similarities, see Figure 1.1(b). The architect takes a hypothesis about the environment — this is typically done based on intuition, insight and/or experience. One example hypothesis may be that integrating many small in-order processor cores may yield better overall system throughput than integrating just a few aggressive out-of-order processor cores on a single chip. Subsequently, the architect designs an experiment to validate this hypothesis.This means that the architect picks a baseline design, e.g., the architect determines the processor architecture configurations for the in-order and out-of-order architectures; in addition, the architects picks a number of workloads, e.g., the architect collects a number of compute-intensive, memory-intensive and/or I/O-intensive benchmarks. The experiment is then run, and a number of measurements are obtained. This can be done by running a model (analytical model or simulation model) and/or by doing a real hardware experiment. Architects typically run an extensive number of measurements while sweeping through the design space. The key question then is to navigate through this wealth of data and make meaningful conclusions. Interpretation of the results is crucial to make correct design decisions.The insights obtained from the experiment may or may not support the hypothesis made by the architect. If the experiment supports the hypothesis, the experimental design is improved and additional experimentation is done, i.e., the design is incrementally refined (inner loop). If the experiment does not support the hypothesis, i.e., the results are completely surprising, then the architect needs to re-examine the hypothesis (outer loop), which may lead the architect to change the design or propose a new design. | 一般来说，系统研究的过程与计算机体系结构研究和开发的过程有一些相似之处，见图1.1(b)。架构师对环境进行假设——这通常是基于直觉、洞察力以及经验。一个关于假设的例子是，集成许多小的顺序处理器核心，可能比在单个芯片上集成几个乱序处理器核心得到更高的系统吞吐量。随后，架构师设计了一个实验来验证这个假设。这意味着架构师选择一个基准设计，例如，架构师确定顺序和乱序架构的处理器架构配置；此外，架构师还会挑选一些工作负载，例如，架构师会收集一些计算密集型、内存密集型以及I/O密集型的基准测试程序。然后进行实验，得到了一些测量结果。实验可以通过运行一个模型（分析模型或仿真模型）和/或做一个真实的硬件实验来实现。架构师在扫描设计空间时通常会进行大量的测量。接下来的关键问题是如何浏览这些丰富的数据，并得出有意义的结论。解释结果对于做出正确的设计决策至关重要。从实验中获得的见解可能支持也可能不支持架构师的假设。如果实验支持假设，则改进实验设计，并进行额外的实验，即逐步完善设计（内环）。如果实验不支持假设，即结果完全出乎意料，那么架构师需要重新检查假设（外环），这可能导致架构师改变设计或提出新的设计。 |
| Although there are clear similarities, there are also important differences that separate systems research from scientific research. Out of practical necessity the step of picking a baseline design and workloads is typically based on the experimenters judgment and experience, rather than objectively drawing a scientific sample from a given population. This means that the architect should be well aware of the subjective human aspect of experiment design when interpreting and analyzing the results. Trusting the results produced through the experiment without a clear understanding of its design may lead to misleading or incorrect conclusions. In this book, we will focus on the scientific approach suggested by Kuhn, but we will also pay attention to making the important task of workload selection less subjective. | 虽然有明显的相似之处，但系统研究与科学研究之间也有重要的区别。出于实际需要，选择基准设计和工作负载的步骤通常是基于实验者的判断和经验，而不是客观地从给定的群体中抽取科学样本。这意味着架构师在解释和分析结果时应该充分意识到实验设计的主观性方面。相信实验结果而不清楚其设计可能导致误导或不正确的结论。在这本书中，我们将侧重于Kuhn建议的科学方法，但我们也将注意使工作负载选择这一重要任务不那么主观。 |
| 1.2 IMPORTANCE OF PERFORMANCE EVALUATION | 1.2 性能评估的重要性 |
| The structure of computer architecture evolution, as described above, clearly illustrates that performance evaluation is at the crux of computer architecture research and development. Rigorous performance evaluation is absolutely crucial in order to make correct design decisions and drive research in a fruitful direction. And this is even more so for systems research than is the case for scientific research: as argued above, in order to make meaningful and valid conclusions, it is absolutely crucial to clearly understand how the experimental design is set up and how this may affect the results. | 如上所述，计算机体系结构演变的结构清楚地说明了，性能评估是计算机体系结构研究和发展的关键。为了做出正确的设计决策，并推动研究在一个富有成效的方向上，严格的性能评估是至关重要的。与科学研究相比，系统研究更是如此：如上所述，为了做出有意义和有效的结论，清楚地理解实验设计是如何建立的以及实验设计可能如何影响结果是绝对至关重要的。 |
| The structure of computer architecture evolution also illustrates that there are multiple aspects to performance evaluation, ranging from picking workloads, picking a baseline design, picking an appropriate modeling approach, running the model, and interpreting the results in a meaningful way. And each of these aspects is equally important — a performance evaluation methodology that lacks rigor in one of these crucial aspects may fall short. For example, picking a representative set of benchmarks, picking the appropriate modeling approach, and running the experiments in a rigorous way may still be misleading if inappropriate performance metrics are used to quantify the benefit of one design point relative to another. Similarly, inadequate modeling, e.g., a simulator that models the architecture at too high a level of abstraction, may either underestimate or overestimate the performance impact of a design decision, even while using the appropriate performance metrics and benchmarks. | 计算机体系结构演进的结构也展示了性能评估的多个方面，从选择工作负载、选择基准设计、选择适当的建模方法、运行模型，以及以有意义的方式解释结果。每一个方面都是同等重要的——在这些关键方面缺乏严格规划的性能评估方法可能会失败。例如，选择一组具有代表性的基准，选择适当的建模方法，并以严格的方式运行实验，如果使用不恰当的性能度量来量化一个设计点相对于另一个设计点的好处，仍然可能会产生误导。类似地，不充分的建模，例如，在过高的抽象级别上对架构建模的仿真器，可能会低估或高估设计决策的性能影响，即使在使用适当的性能度量和基准测试程序时也是如此。 |
| Architects are generally well aware of the importance of adequate performance evaluation, and, therefore, they pay detailed attention to the experimental setup when evaluating research ideas and design innovations. | 架构师一般都很清楚性能评估的重要性。因此，他们在评估研究思路和创新设计时，会非常注意实验设置。 |
| 1.3 BOOK OUTLINE | 1.3 本书要点 |
| This book presents an overview of the current state-of-the-art in computer architecture performance evaluation, with a special emphasis on methods for exploring processor architectures.The book covers performance metrics (Chapter 2), workload design (Chapter 3), analytical performance modeling (Chapter 4), architectural simulation (Chapter 5), sampled simulation (Chapter 6), statistical simulation (Chapter 7), parallel simulation and hardware acceleration (Chapter 8). Finally, we conclude in Chapter 9. | 本书介绍了计算机体系结构性能评估领域当前先进技术的概述，特别强调了探索处理器体系结构的方法。本书涵盖了性能度量（第2章）、工作负载设计（第3章）、分析性能建模（第4章）、架构仿真（第5章）、抽样仿真（第6章）、统计仿真（第7章）、并行仿真和硬件加速（第8章）。最后，我们在第9章总结。 |

|  |  |
| --- | --- |
| CHAPTER 2 Performance Metrics | 第2章 性能度量 |
| Performance metrics are at the foundation of experimental research and development. When evaluating a new design feature or a novel research idea, the need for adequate performance metrics is paramount. Inadequate metrics may be misleading and may lead to incorrect conclusions and may steer development and research in the wrong direction. This chapter discusses metrics for evaluating computer system performance. This is done in a number of steps. We consider metrics for single-threaded workloads, multi-threaded workloads and multi-program workloads. Subsequently, we will discuss ways of summarizing performance in a single number by averaging across multiple benchmarks. Finally, we will briefly discuss the utility of partial metrics. | 性能度量是实验研究和开发的基础。当评估一个新的设计特性或一个新的研究想法时，对恰当的性能度量的需求是至关重要的。不恰当的度量可能会造成误导，并可能导致不正确的结论，进而可能将开发和研究引向错误的方向。本章讨论评估计算机系统性能的度量。这需要几个步骤来完成。我们依次考虑单线程工作负载、多线程工作负载和多程序工作负载。随后，我们将讨论通过对多个基准测试程序的度量指标取平均值，来用单个数字总结性能的方法。最后，我们将简要讨论部分度量的使用。 |
| 2.1 SINGLE-THREADED WORKLOADS | 2.1 单线程负载 |
| Benchmarking computer systems running single-threaded benchmarks is well understood.The total time T to execute a single-threaded program is the appropriate metric.For a single-threaded program that involves completing N instructions, the total execution time can be expressed using the ‘Iron Law of Performance’ [169]: | 运行单线程基准测试对计算机系统进行基准测试是很容易理解的。执行单线程程序的总时是恰当的度量。对于一个涉及完成条指令的单线程程序，总执行时间可以用“性能铁律”来表示[169]： |
|  | |
| with CPI the average number of cycles per useful instruction and f the processor’s clock frequency. Note the wording ‘useful instruction’.This is to exclude the instructions executed along mispredicted paths — contemporary processors employ branch prediction and speculative execution, and in case of a branch misprediction, speculatively executed instructions are squashed from the processor pipeline and should, therefore, not be accounted for as they don’t contribute to the amount of work done. | 其中，表示每条有效指令的平均周期数，表示处理器的时钟频率。注意措辞“有效指令”。这是为了排除沿着错误预测路径执行的指令——现代处理器采用分支预测和投机执行。如果发生分支预测错误，投机执行的指令会从处理器流水线中被排出，因此它们不贡献负载，不应该被考虑。 |
| The utility of the Iron Law of Performance is that the terms correspond to the sources of performance. The number of instructions N is a function of the instruction-set architecture (ISA) and compiler; CPI is a function of the micro-architecture and circuit-level implementation; and f is a function of circuit-level implementation and technology. Improving one of these three sources of performance improves overall performance. Justin Rattner [161], in his PACT 2001 keynote presentation, reported that x86 processor performance has improved by over 75× over a 10 years time period,between the 1.0μ technology node (early 1990s) and the 0.18μ technology node (around 2000): 13× comes from improvements in frequency, and 6× from micro-architecture enhancements, and the 50× improvement in frequency in its turn is due to improvements in technology (13×) and micro-architecture (4×). | 性能铁律的意义在于，这些术语对应于性能的来源。指令数是指令集体系结构（ISA）和编译器的函数；是一个微架构和电路级实现的函数；是电路级实现和工艺的函数。改善这三个性能来源中的任意一个都可以提高整体性能。Justin Rattner[161]在PACT 2001主题演讲中指出，从1.0μ技术节点（20世纪90年代初）到0.18μ技术节点（2000年左右），在10年的时间内，x86处理器的性能提高了超过75倍：13倍来自频率的提高，6倍来自微架构的增强，而这50倍的频率提高则来自与工艺（13×）和微架构（4×）的提高。 |
| Assuming that the amount of work that needs to be done is constant, i.e., the number of dynamically executed instructions N is fixed, and the processor clock frequency f is constant, one can express the performance of a processor in terms of the CPI that it achieves. The lower the CPI, the lower the total execution time, the higher performance. Computer architects frequently use its reciprocal, or IPC, the average number of (useful) instructions executed per cycle. IPC is a higher-is-better metric. The reason why IPC (and CPI) are popular performance metrics is that they are easily quantified through architectural simulation. Assuming that the clock frequency does not change across design alternatives, one can compare microarchitectures based on IPC. | 假设需要完成的负载是恒定的，即动态执行的指令数量是恒定的而且处理器时钟频率是恒定的，那么可以用处理器实现的CPI来表示处理器的性能。CPI越低，总执行时间越短，性能越高。计算机架构师经常使用它的倒数，或IPC，即每个周期执行的（有效的）指令数的平均值。IPC是一个越高越好的指标。IPC（和CPI）之所以是流行的性能度量，是因为它们很容易通过架构仿真量化。假设在不同的设计方案中时钟频率不会改变，那么可以根据IPC比较微架构设计方案。 |
| Although IPC seems to be more widely used than CPI in architecture studies — presumably, because it is a higher-is-better metric — CPI provides more insight. CPI is additive, and one can break up the overall CPI in so called CPI adders [60] and display the CPI adders in a stacked bar called the CPI stack. The base CPI is typically shown at the bottom of the CPI stack and represents useful work done. The other CPI adders, which reflect ‘lost’ cycle opportunities due to miss events such as branch mis-predictions and cache and TLB misses, are stacked on top of each other. | 虽然在架构研究中，IPC似乎比CPI使用得更广泛（大概是因为IPC是一个越高越好的指标），CPI提供了更多的见解。CPI是可以累加的，可以用所谓的CPI累加器[60]将CPI分解，并将CPI累加器展示在一个称为CPI栈的堆叠条中。基本CPI通常显示在CPI栈的底部，代表所做的有效工作。其他CPI累加器反映了由于缺失事件（如分支预测错误、缓存和TLB缺失）而导致的“丢失”的周期机会，它们是相互堆叠的。 |
| 2.2 MULTI-THREADED WORKLOADS | 2.2 多线程负载 |
| Benchmarking computer systems that run multi-threaded workloads is in essence similar to what we described above for single-threaded workloads and is adequately done by measuring the time it takes to execute a multi-threaded program, or alternatively, to get a given amount of work done (e.g., a number of transactions for a database workload), assuming there are no co-executing programs. | 运行多线程工作负载对计算机系统进行基准测试，在本质上类似于我们上面描述的单线程工作负载。假设没有共同执行的程序，通过测量执行多线程程序所需的时间，或者完成给定工作量（例如，数据库工作负载的一些事务）所需的时间来充分地进行。 |
| In contrast to what is the case for single-threaded workloads IPC is not an accurate and reliable performance metric for multi-threaded workloads and may lead to misleading or incorrect conclusions [2]. The fundamental reason is that small timing variations may lead to different execution paths and thread interleavings, i.e., the order in which parallel threads enter a critical section may vary from run to run and may be different across different microarchitectures. For example, different interrupt timing may cause system software to take different scheduling decisions. Also, different microarchitectures may lead to threads reaching the critical sections in a different order because threads’ performance may be affected differently by the microarchitecture. As a result of these differences in timing and thread interleavings, the total number of instructions executed may be different across different runs, and hence, the IPC may be different. For example, threads executing spin-lock loop instructions before acquiring a lock increase the number of dynamically executed instructions (and thus IPC); however, these spin-lock loop instructions do not contribute to overall execution time, i.e., the spin-lock loop instructions do not represent useful work. Moreover, the number of spin-lock loop instructions executed may vary across microarchitectures. Spin-lock loop instructions is just one source of error from using IPC. Another source relates to instructions executed in the operating system, such as instructions handling Translation Lookaside Buffer (TLB) misses, and idle loop instructions. | 与单线程工作负载的情况相反，对于多线程工作负载，IPC并不是一个准确可靠的性能度量，反而可能会导致误导或错误的结论[2]。根本原因是微小的时序变化可能导致不同的执行路径和线程交替。也就是说，并行线程进入临界区的顺序在不同次的运行中可能不同，在不同的微架构中也可能不同。例如，不同的中断时序可能导致系统软件采取不同的调度决策。此外，不同的微架构可能会导致线程以不同的顺序到达临界区，因为线程的性能可能会受到不同微架构的影响。由于这些时间和线程交替的差异，在不同的运行中执行的总指令数可能不同，因此IPC可能不同。例如，在获取锁之前执行自旋锁循环指令的线程会增加动态执行指令的数量（因此IPC也会增加）；然而，这些自旋锁循环指令对总体执行时间没有贡献，也就是说，自旋锁循环指令并不代表有效的工作量。此外，执行的自旋锁环指令的数量可能因微架构而异。自旋锁循环指令只是IPC的误差的一个来源。另一个来源与操作系统中执行的指令有关，比如处理地址转换缓存器（TLB）缺失的指令，以及空闲循环指令。 |
| Alameldeen and Wood [2] present several case studies showing that an increase in IPC does not necessarily imply better performance, or a decrease in IPC does not necessarily reflect performance loss for multi-threaded programs.They found this effect to increase with an increasing number of processor cores because more threads are spending time in spin-lock loops. They also found this effect to be severe for workloads that spend a significant amount of time in the operating system, e.g., commercial workloads such as web servers, database servers, and mail servers. | Alameldeen和Wood[2]提供的一些案例研究表明，对于多线程程序来说，IPC的增加并不一定意味着性能的提高，或者IPC的降低并不一定意味着性能的下降。他们发现，随着处理器核心数量的增加，这种效果还会增加，因为更多的线程在自旋锁循环中消耗时间。他们还发现，对于花费大量时间在操作系统上的工作负载，如web服务器、数据库服务器和邮件服务器等商业工作负载，这种影响是很严重的。 |
| Wenisch et al. [190] consider user-mode instructions only and quantify performance in terms of user-mode IPC (U-IPC). They found U-IPC to correlate well with the number of transactions completed per unit of time for their database and web server benchmarks. The intuition is that when applications do not make forward progress, they often yield to the operating system (e.g., the OS idle loop or spin-lock loops in OS code). A limitation for U-IPC is that it does not capture performance of system-level code, which may account for a significant fraction of the total execution time, especially in commercial workloads. | Wenisch等[190]只考虑用户模式指令，并根据用户模式IPC（U-IPC）来量化性能。在他们的数据库和web服务器基准测试中，他们发现U-IPC与单位时间内完成的事务数密切相关。直觉上，当应用程序没有进展时，它们通常会让位于操作系统（例如，操作系统空闲循环或操作系统代码中的自旋锁循环）。U-IPC的一个限制是它不能捕获系统级代码的性能。特别是在商业工作负载中，系统级代码可能占总执行时间的很大一部分。 |
| Emer and Clark [59] addressed the idle loop problem in their VAX-11/780 performance study by excluding the VMS Null process from their per-instruction statistics. Likewise, one could measure out spin-lock loop instructions; this may work for some workloads, e.g., scientific workloads where most of the spinning happens in user code at places that can be identified a priori. | Emer和Clark[59]在他们关于VAX-11/780的性能研究中，通过从指令统计数据中排除VMS Null进程来解决空闲循环的问题。同样地，我们可以测量出自旋锁环指令。这种方法可能适用于某些特定的工作负载，例如，科学计算工作负载，因为其中大多数自旋锁发生在用户代码中可以预先识别的地方。 |
| 2.3 MULTIPROGRAM WORKLOADS | 2.3 多程序工作负载 |
| The proliferation of multi-threaded and multicore processors in the last decade has urged the need for adequate performance metrics for multiprogram workloads. Not only do multi-threaded and multicore processors execute multi-threaded workloads, they also execute multiple independent programs concurrently. For example, a simultaneous multi-threading (SMT) processor [182] may co-execute multiple independent jobs on a single processor core. Likewise, a multicore processor or a chip-multiprocessor [151] may co-execute multiple jobs, with each job running on a separate core. A chip-multithreading processor may co-execute multiple jobs across different cores and hardware threads per core, e.g., Intel Core i7, IBM POWER7, Sun Niagara [108]. | 在过去的十年中，多线程和多核处理器的普及迫切需要为多程序工作负载提供充分的性能度量。多线程和多核处理器不仅可以执行多线程工作负载，还可以并发地执行多个独立的程序。例如，一个同时多线程（SMT）处理器[182]可以在单个处理器核心上共同执行多个独立的作业。同样，多核处理器或片上多处理器[151]也可以共同执行多个作业，每个作业在单独的核心上运行。一个片上多线程处理器可以跨不同的核心和每个核心的硬件线程共同执行多个任务，例如Intel Core i7、IBM POWER7、Sun Niagara[108]。 |
| As the number of cores per chip increases exponentially, according to Moore’s law, it is to be expected that more and more multiprogram workloads will run on future hardware. This is true across the entire compute range. Users browse, access email, and process messages and calls on their cell phones while listening to music. At the other end of the spectrum, servers and datacenters leverage multi-threaded and multicore processors to achieve greater consolidation. | 根据摩尔定律，随着芯片上的核心数量呈指数级增长，可以预期未来硬件上将运行越来越多的多程序工作负载。在整个计算范围内都是如此。用户可以一边听音乐，一边用手机浏览网页、访问电子邮件、处理信息和拨打电话。另一方面，服务器和数据中心利用多线程和多核处理器来实现更大的整合。 |
| The fundamental problem that multiprogram workloads impose to performance evaluation and analysis is that the independent co-executing programs affect each other’s performance. The amount of performance interaction depends on the amount of resource sharing. A multicore processor typically shares the last-level cache across the cores as well as the on-chip interconnection network and the off-chip bandwidth to memory. Chandra et al. [24] present a simulation-based experiment that shows that the performance of individual programs can be affected by as much as 65% due to resource sharing in the memory hierarchy of a multicore processor when co-executing two independent programs.Tuck and Tullsen [181] present performance data measured on the Intel Pentium 4 processor which is an SMT processor with two hardware threads. They report that for some programs per-program performance can be as low as 71% of the per-program performance observed when run in isolation, whereas for other programs, per-program performance may be comparable (within 98%) to isolated execution. | 多程序工作负载给性能评估和分析带来的根本问题是独立的、共同执行的程序相互影响性能。性能交互的数量取决于资源共享的数量。多核处理器通常在各个核心之间共享最后一级缓存，以及片内互连网络和片外到内存的带宽。Chandra等人[24]提出了一个基于仿真的实验，该实验表明，当共同执行两个独立的程序时，由于多核处理器的内存层次中的资源共享，单个程序的性能可能会受到高达65%的影响。Tuck和Tullsen[181]展示了在Intel Pentium 4处理器上测量的性能数据，这是一种带有两个硬件线程的SMT处理器。他们报告说，对于一些程序，每个程序的性能可能低至在单独运行时性能的71%；而对于其他程序，每个程序的性能可能与单独执行相当（在98%以内）。 |
| Intel uses the term HyperThreading rather than SMT. | Intel使用术语“超线程”（HyperThread）而不是SMT。 |
| Eyerman and Eeckhout [61] take a top-down approach to come up with performance metrics for multiprogram workloads, namely system throughput (STP) and average normalized turnaround time (ANTT).They start from the observation that there are two major perspectives to multiprogram performance: a user’s perspective and a system’s perspective. A user’s perspective cares about the turnaround time for an individual job or the time it takes between submitting the job and its completion. A system’s perspective cares about the overall system throughput or the number of jobs completed per unit of time. Of course, both perspectives are not independent of each other. If one optimizes for job turnaround time, one will likely also improve system throughput. Similarly, improving a system’s throughput will also likely improve a job’s turnaround time. However, there are cases where optimizing for one perspective may adversely impact the other perspective. For example, optimizing system throughput by prioritizing short-running jobs over long-running jobs will have a detrimental impact on job turnaround time, and it may even lead to starvation of long-running jobs. | Eyerman和Eeckhout[61]采用自顶向下的方法提出了多程序工作负载的性能度量，即系统吞吐量（STP）和平均归一化周转时间（ANTT）。他们从观察到多程序性能的两个主要的角度开始：用户的角度和系统的角度。用户的视角关注的是单个作业的周转时间，或者从提交作业到完成作业所花费的时间。系统的角度关心总体系统吞吐量或单位时间内完成的作业数量。当然，这两个角度并非相互独立。如果能够优化作业周转时间，就有可能提高系统吞吐量。同样，提高系统的吞吐量也可能提高作业的周转时间。然而，在某些情况下，对一个角度的优化可能会对另一个角度产生负面影响。例如，通过将短时间运行的作业优先于长时间运行的作业来优化系统吞吐量，这将对作业周转时间产生不利影响，甚至可能导致长时间运行的作业饿死。 |
| We now discuss the STP and ANTT metrics in more detail, followed by a discussion on how they compare against prevalent metrics. | 我们现在更详细地讨论STP和ANTT指标，然后讨论它们如何与流行的指标进行比较。 |
| 2.3.1 SYSTEM THROUGHPUT | 2.3.1 系统吞吐率 |
| We, first, define a program’s normalized progress as | 首先，我们将程序的归一化进度定义为 |
|  | |
| with TiSP and TiMP, the execution time under single-program mode (i.e., the program runs in isolation) and multiprogram execution (i.e., the program co-runs with other programs), respectively. Given that a program runs slower under multiprogram execution, normalized progress is a value smaller than one. The intuitive understanding of normalized progress is that it represents a program’s progress during multiprogram execution. For example, an NP of 0.7 means that a program makes 7 milliseconds of single-program progress during a 10 millisecond time slice of multiprogram execution. | 和分别表示单程序模式（即程序独立运行）和多程序模式（即程序与其他程序共同运行）下的执行时间。假设一个程序在多程序执行下运行得较慢，那么归一化进度的值就小于1。归一化进度的直观理解是，它表示多程序执行期间的程序进度。例如，0.7的NP意味着一个程序在10毫秒的多程序模式执行时间片中，完成了单程序模式执行7毫秒的任务量。 |
| System throughput (STP) is then defined as the sum of the normalized progress rates across all jobs in the multiprogram job mix: | 系统吞吐量（STP）定义为多程序作业组合中所有作业的归一化进度之和： |
|  | |
| In other words, system throughput is the accumulated progress across all jobs, and thus it is a higher-is-better metric. For example, running two programs on a multi-threaded or multicore processor may cause one program to make 0.75 normalized progress and the other 0.5; the total system throughput then equals 0.75 + 0.5 = 1.25. STP is typically larger than one: latency hiding increases system utilization and throughput, which is the motivation for multi-threaded and multicore processing in the first place, i.e., cache miss latencies from one thread are hidden by computation from other threads, or memory access latencies are hidden through memory-level parallelism (MLP), etc. Nevertheless, for some combinations of programs, severe resource sharing may result in an STP smaller than one. For example, co-executing memory-intensive workloads may evict each other’s working sets from the shared last-level cache resulting in a huge increase in the number of conflict misses, and hence they have detrimental effects on overall performance. An STP smaller than one means that better performance would be achieved by running all programs back-to-back through time sharing rather than through co-execution. | 换句话说，系统吞吐量是所有作业的累积进度，因此它是一个越高越好的指标。例如，在多线程或多核处理器上运行两个程序可能导致一个程序的归一化进度为0.75，另一个为0.5；系统的总吞吐量等于。STP通常大于1：延迟隐藏增加了系统利用率和吞吐量，这是多线程和多核处理的首要动机。例如，一个线程的缓存缺失延迟被其他线程的计算所隐藏，或者通过内存级并行（MLP）来隐藏内存访问延迟，等等。然而，对于某些程序组合，严重的资源共享可能导致STP小于1。例如，共同执行内存密集型工作负载可能会从共享的最后一级缓存中清除彼此的数据集，从而导致冲突缺失的数量大幅增加，进而对总体性能产生不利影响。STP小于1意味着通过背对背连续运行将获得更好的性能，而不是同时运行所有程序。 |
| 2.3.2 AVERAGE NORMALIZED TURNAROUND TIME | 2.3.2 平均归一化周转时间 |
| To define the average normalized turnaround time,we first define a program’s normalized turnaround time as | 要定义平均归一化周转时间，首先定义一个程序的归一化周转时间为 |
|  | |
| Normalized turnaround time quantifies the user-perceived slowdown during multiprogram execution relative to single-program execution, and typically is a value larger than one. NTT is the reciprocal of NP. | 归一化周转时间通常是一个大于1的值，量化了用户在多程序执行过程中感知到的相对于单程序执行的速度降低。NTT是NP的倒数。 |
| The average normalized turnaround time is defined as the arithmetic average across the programs’ normalized turnaround times: | 平均归一化周转时间定义为所有程序的归一化周转时间的算术平均值： |
|  | |
| ANTT is a lower-is-better metric. For the above example, the one program achieves an NTT of 1/0.75 = 1.33 and the other 1/0.5 = 2, and thus ANTT equals (1.33 + 2)/2 = 1.67, which means that the average slowdown per program equals 1.67. | ANTT是一个越低越好的度量指标。对于前面的例子，一个程序获得的NTT为和另一个的NTT为，因此ANTT等于，这意味着每个程序的平均减速等于1.67。 |
| 2.3.3 COMPARISON TO PREVALENT METRICS | 2.3.3 与流行度量指标的比较 |
| Prevalent metrics for quantifying multiprogram performance in the architecture community are IPC throughput, weighted speedup [174] and harmonic mean [129]. We will now discuss these performance metrics and compare them against STP and ANTT. | 在架构领域，量化多程序性能的常用指标是IPC吞吐量、加权加速比[174]和调和平均[129]。我们现在将讨论这些性能度量，并将它们与STP和ANTT进行比较。 |
| Computer architects frequently use single-threaded SPEC CPU benchmarks to compose multiprogram workloads for which IPC is an adequate performance metric. Hence, the multiprogram metrics that have been proposed over the recent years are based on IPC (or CPI). This limits the applicability of these metrics to single-threaded workloads. STP and ANTT, on the other hand, are defined in terms of execution time, which makes the metrics applicable for multi-threaded workloads as well. In order to be able to compare STP and ANTT against the prevalent metrics, we first convert STP and ANTT to CPI-based metrics based on Equation 2.1 while assuming that the number of instructions and clock frequency is constant. It easily follows that STP can be computed as | 计算机架构师经常使用单线程SPEC CPU基准测试来构成多程序工作负载，对于这些工作负载，IPC是一个充分的性能度量。因此，近年来提出的多程序度量指标是基于IPC（或CPI）的。这限制了这些指标对单线程工作负载的适用性。另一方面，STP和ANTT是根据执行时间定义的，这使得该指标也适用于多线程工作负载。为了能够比较STP和ANTT与流行的度量标准，我们首先将STP和ANTT转换为基于CPI的度量标准，并假设指令数量和时钟频率是恒定的。很容易得出STP可以用下式计算： |
|  | |
| with CPIiSP and CPIiMP the CPI under single-program and multiprogram execution, respectively. ANTT can be computed as | 其中，和分别表示单程序和多程序执行下的CPI。ANTT可以用下式计算： |
|  |  |
| IPC throughput. IPC throughput is defined as the sum of the IPCs of the co-executing programs: | IPC吞吐量。IPC吞吐量定义为共同执行程序的IPC之和： |
|  | |
| IPC throughput naively reflects a computer architect’s view on throughput; however, it doesn’t have a meaning in terms of performance from either a user perspective or a system perspective. In particular, one could optimize a system’s IPC throughput by favoring high-IPC programs; however, this may not necessarily reflect improvements in system-level performance (job turnaround time and/or system throughput). Therefore, it should not be used as a multiprogram performance metric. | IPC吞吐量直观地反映了计算机架构师对吞吐量的看法；但是，无论是从用户角度还是从系统角度来看，它都没有性能方面的意义。特别是，可以通过倾向于高IPC程序来优化系统的IPC吞吐量；但是，这不一定反映系统级性能的提高（作业周转时间和/或系统吞吐量）。因此，它不应该被用作多程序性能度量。 |
| Weighted speedup. Snavely and Tullsen [174] propose weighted speedup to evaluate how well jobs co-execute on a multi-threaded processor. Weighted speedup is defined as | 加权加速比。Snavely和Tullsen[174]提出了加权加速比来评估在多线程处理器上作业协同执行的情况。加权加速比的定义为 |
|  | |
| The motivation by Snavely and Tullsen for using IPC as the basis for the speedup metric is that if one job schedule executes more instructions than another in the same time interval, it is more symbiotic and, therefore, yields better performance; the weighted speedup metric then equalizes the contribution of each program in the job mix by normalizing its multiprogram IPC with its single-program IPC. | Snavely和Tullsen使用IPC作为加速指标的基础的动机是，如果一个作业调度在相同的时间区间内执行的指令比另一个多，那么它就会更加适合共同执行，从而产生更好的性能；然后，加权加速比通过将多程序IPC与单程序IPC归一化，使每个程序在作业组合中的贡献相等。 |
| From the above, it follows that weighted speedup equals system throughput (STP) and, in fact, has a physical meaning — it relates to the number of jobs completed per unit of time — although this may not be immediately obvious from weighted speedup’s definition and its original motivation. | 从上面可以看出，加权加速比等于系统吞吐量（STP）。事实上，加权加速比有一个物理意义——它与单位时间内完成的作业数量有关——尽管从加权加速比的定义及其最初动机来看，这可能不是很明显。 |
| Harmonic mean. Luo et al. [129] propose the harmonic mean metric, or hmean for short, which computes the harmonic mean rather than an arithmetic mean (as done by weighted speedup) across the IPC speedup numbers: | 调和平均数。Luo等人[129]提出了调和平均数指标，简称hmean，它计算IPC加速比的调和平均数，而不是算术平均值（如加权加速比）： |
|  | |
| The motivation by Luo et al. for computing the harmonic mean is that it tends to result in lower values than the arithmetic average if one or more programs have a lower IPC speedup, which they argue better captures the notion of fairness than weighted speedup. The motivation is based solely on properties of the harmonic and arithmetic means and does not reflect any system-level meaning. It follows from the above that the hmean metric is the reciprocal of the ANTT metric, and hence it has a system-level meaning, namely, it relates to (the reciprocal of) the average job’s normalized turnaround time. | Luo等人计算调和平均数的动机是，如果一个或多个程序的IPC加速较低，它往往会导致比算术平均值更低的值，他们认为调和平均数比加权加速比更能抓住公平性的概念。其动机完全反应调和平均数和算术平均数的性质，而不反映任何系统级别的意义。由上可知，hmean指标是ANTT指标的倒数，因此它具有系统级的意义，即，它与作业的平均归一化周转时间（的倒数）有关。 |
| 2.3.4 STP VERSUS ANTT PERFORMANCE EVALUATION | 2.3.4 STP与ATT的性能评估 |
| Because of the complementary perspectives, multiprogram workload performance should be characterized using both the STP and ANTT metrics in order to get a more comprehensive performance picture. Multiprogram performance, using only one metric provides an incomplete view and skews the perspective. Eyerman and Eeckhout [61] illustrate this point by comparing different SMT fetch policies: one fetch policy may outperform another fetch policy according to one metric; however, according to the other metric, the opposite may be true. Such a case illustrates that there is trade-off in user-level performance versus system-level performance. This means that one fetch policy may yield higher system throughput while sacrificing average job turnaround time, whereas the other one yields shorter average job turnaround times while reducing system throughput. So, it is important to report both the STP and ANTT metrics when reporting multiprogram performance. | 由于互补的角度，多程序工作负载性能应该同时使用STP和ANTT度量来表征，以便获得更全面的性能视图。只使用一个指标，程序性能提供不完整的视图和有偏见的视角。Eyerman和Eeckhout[61]通过比较不同的SMT取指令策略说明了这一点：根据一个度量标准，一种取指策略可能优于另一种取指策略；然而，根据另一种度量标准，情况可能恰恰相反。这种情况说明了用户级性能与系统级性能之间存在权衡。这意味着一种取指策略可以在牺牲作业平均周转时间的同时获得更高的系统吞吐量，而另一种取指策略可以在降低系统吞吐量的同时获得更短的作业平均周转时间。因此，在报告多程序性能时，同时报告STP和ANTT指标是很重要的。 |
| 2.4 AVERAGE PERFORMANCE | 2.4 平均性能 |
| So far, we discussed performance metrics for individual workloads only. However, people like to quantify what this means for average performance across a set of benchmarks. Although everyone agrees that ‘Performance is not a single number’, there has been (and still is) a debate going on about which number is better. Some argue for the arithmetic mean, others argue for the harmonic mean, yet others argue for the geometric mean. This debate has a long tradition: it started in 1986 with Fleming and Wallace [68] arguing for the geometric mean. Smith [173] advocated the opposite, shortly thereafter. Cragon [37] also argues in favor of the arithmetic and harmonic mean. Hennessy and Patterson [80] describe the pros and cons of all three averages. More recently, John [93] argued strongly against the geometric mean, which was counterattacked by Mashey [134]. | 到目前为止，我们只讨论了单个工作负载的性能度量。然而，人们喜欢量化对一系列基准测试的平均性能的含义。尽管每个人都同意“性能不是单一一个数字”，但关于哪个数字更好的争论一直存在（现在仍然存在）。有人支持算术平均数，有人支持调和平均数，还有人支持几何平均数。这一争论有着悠久的传统：它始于1986年Fleming和Wallace[68]对几何平均值的争论。不久之后，Smith[173]就提出了相反的观点。Cragon[37]也支持算术平均数和调和平均数。Hennessy和Patterson[80]描述了这三个平均数的利弊。最近，John[93]强烈反对几何平均值，但这被Mashey[134]反驳。 |
| It seems that even today people are still confused about which average to choose; some papers use the harmonic mean, others use the arithmetic mean, and yet others use the geometric mean. The goal of this section is to shed some light into this important problem, describe the two opposite viewpoints and make a recommendation. The contradictory views come from approaching the problem from either a mathematical angle or a statistical angle. The mathematical viewpoint, which leads to using the harmonic and arithmetic mean, starts from understanding the physical meaning of the performance metric, and then derives the average in a meaningful way that makes sense mathematically. The statistical viewpoint, on the other hand, assumes that the benchmarks are randomly chosen from the workload population, and the performance speedup metric is log-normally distributed, which leads to using the geometric mean. | 直到今天，人们似乎仍然不知道应该选择哪个平均数；有些论文使用调和平均数，有些论文使用算术平均数，还有一些论文使用几何平均数。本节的目标是阐明这一重要问题，描述两种相反的观点并提出建议。这种矛盾的观点来自于从数学角度或统计角度来研究这个问题。数学观点从理解性能度量的物理意义开始，然后选择在数学上有意义的平均数，导致使用调和平均和算术平均。另一方面，统计观点假设基准是从工作负载总体中随机选择的，并且性能加速指标是对数正态分布的，这导致使用几何平均值。 |
| 2.4.1 HARMONIC AND ARITHMETIC AVERAGE: MATHEMATICAL VIEWPOINT | 2.4.1 调和平均数和算术平均数：数学观点 |
| The mathematical viewpoint starts from a clear understanding of what the metrics mean, and then chooses the appropriate average (arithmetic or harmonic) for the metrics one wants to compute the average for. This approach does not assume a particular distribution for the underlying population, and it does not assume that the benchmarks are chosen randomly from the workload space. It simply computes the average performance metric for the selected set of benchmarks in a way that makes sense physically, i.e., understanding the physical meaning of the metric one needs to compute the average for, leads to which average to use (arithmetic or harmonic) in a mathematically meaningful way. | 数学观点从清晰地理解度量的含义开始，然后为需要计算平均值的度量选择恰当的平均值（算术或调和）。这种方法不假设总体的特定分布，也不假设基准是从工作负载空间中随机选择的。它只是以一种物理上有意义的方式计算所选基准测试集的平均性能度量，也就是说，理解需要计算平均值的指标的物理意义，从而以一种数学上有意义的方式使用哪个平均值（算术或调和）。 |
| In particular, if the metric of interest is obtained by dividing A by B and if A is weighed equally among the benchmarks then the harmonic mean is meaningful. Here is the mathematical derivation: | 特别是，如果感兴趣的度量指标是通过A除以B得到的，并且如果A在基准测试中是平等的，那么调和平均值是有意义的。下面是数学推导： |
|  | |
| If on the other hand, B is weighed equally among the benchmarks, then the arithmetic mean is meaningful: | 另一方面，如果B在基准测试中是平等的，那么算术平均值是有意义的： |
|  | |
| We refer to John [93] for a more extensive description, including a discussion on how to weigh the different benchmarks. | 我们参考John的工作[93]以获得更深入的描述，包括关于如何衡量不同基准的讨论。 |
| Hence, depending on the performance metric and how the metric was computed, one has to choose for either the harmonic or arithmetic mean. For example, assume we have selected a 100M instruction sample for each benchmark in our benchmark suite. The average IPC (instructions executed per cycles) needs to be computed as the harmonic mean across the IPC numbers for the individual benchmarks because the instruction count is constant across the benchmarks. The same applies to computing MIPS or million instructions executed per second. Inversely, the average CPI (cycles per instruction) needs to be computed as the arithmetic mean across the individual CPI numbers. Similarly, the arithmetic average also applies for TPI (time per instruction). | 因此，根据性能度量的定义和指标的计算方式，必须选择调和或算术平均值。例如，假设我们从基准测试套件中的每个基准测试程序选择了一个100M指令的样本。需要将平均IPC（每个周期执行的指令）计算为各个基准测试的IPC数字的调和平均值，因为各个基准测试的指令数是恒定的。这同样适用于MIPS或每秒执行百万指令的计算。相反，平均CPI（每个指令的周期）需要作为CPI数字的算术平均值来计算。同样，算术平均值也适用于TPI（每条指令的时间）。 |
| The choice for harmonic versus arithmetic mean also depends on the experimenter’s perspective. For example, when computing average speedup (execution time on original system divided by execution time on improved system) over a set of benchmarks, one needs to use the harmonic mean if the relative duration of the benchmarks is irrelevant, or, more precisely, if the experimenter weighs the time spent in the original system for each of the benchmarks equally. If on the other hand, the experimenter weighs the duration for the individual benchmarks on the enhanced system equally, or if one expects a workload in which each program will run for an equal amount of time on the enhanced system, then the arithmetic mean is appropriate. | 选择调和平均数还是算术平均数也取决于实验者的观点。例如，在一组基准测试程序上计算平均加速比（原始系统上的执行时间除以改进系统上的执行时间）时，如果基准测试程序的持续时间相对无关，或者更准确地说，如果实验人员加权平均衡量每个基准测试程序在原始系统上所花费的时间，就需要使用调和平均值。另一方面，如果实验人员在增强的系统上加权平均衡量各个基准测试的持续时间，或者如果期望每个程序在增强的系统上运行相同的时间，那么算术平均值是合适的。 |
| Weighted harmonic mean and weighted arithmetic can be used if one knows a priori which applications will be run on the target system and in what percentages — this may be the case in some embedded systems. Assigning weights to the applications proportional to their percentage of execution will provide an accurate performance assessment. | 如果预先知道哪些应用程序将在目标系统上运行以及以何种比例运行，则可以使用加权调和平均数和加权算术平均数——在一些嵌入式系统中可能有这样的情况。按照它们执行的比例，为应用程序分配权重将提供准确的性能评估。 |
| 2.4.2 GEOMETRIC AVERAGE: STATISTICAL VIEWPOINT | 2.4.2 几何平均数：统计观点 |
| The statistical viewpoint regards the set of benchmarks as being representative for a broader set of workloads and assumes that the population from which the sample (the benchmarks) are taken follows some distribution. In particular, Mashey [134] argues that speedups (the execution time on the reference system divided by the execution time on the enhanced system) are distributed following a log-normal distribution. A log-normal distribution means that the elements in the population are not normally or Gaussian distributed, but their logarithms (of any base) are. The main difference between a normal and a log-normal distribution is that a log-normal distribution is skewed, i.e., its distribution is asymmetric and has a long tail to the right, whereas a normal distribution is symmetric and has zero skew. Normal distributions arise from aggregations of many additive effects, whereas log-normal distributions arise from combinations of multiplicative effects. In fact, performance is a multiplicative effect in CPI and clock frequency for a given workload, see Equation 2.1, which may seem to suggest that a log-normal distribution is a reasonable assumption; however, CPI and clock frequency are not independent, i.e., the micro-architecture affects both CPI and clock frequency. The assumption that speedups are distributed along a log-normal distribution also implies that some programs experience much larger speedups than others, i.e., there are outliers, hence the long tail to the right. | 统计学的观点认为，一组基准测试程序是一组更广泛的工作负载的代表，并假定抽样（基准测试程序）所来自的总体遵循某种分布。特别地，Mashey[134]认为加速比（参考系统上的执行时间除以增强系统上的执行时间）遵循对数正态分布。对数正态分布意味着总体中的元素不是正态分布或高斯分布，但它们（任何底数）的对数是正态分布。正态分布和对数正态分布的主要区别在于对数正态分布是倾斜的，也就是说，它的分布是不对称的，有一个向右的长尾，而正态分布是对称的，倾斜为零。正态分布产生于许多相加效应的聚集，而对数正态分布产生于乘法效应的组合。事实上，对于给定的工作负载，性能是CPI和时钟频率的乘法效应，见公式2.1，这似乎表明对数正态分布是一个合理的假设；但是，CPI和时钟频率不是独立的，即微架构同时影响CPI和时钟频率。加速比沿对数正态分布的假设也意味着一些程序的加速比其他程序大得多，也就是说，存在异常值，因此长尾向右。 |
| Assuming that speedups follow a log-normal distribution, the geometric mean is the appropriate mean for speedups: | 假设加速比遵循对数正态分布，几何平均值是加速比的恰当平均值： |
|  | |
| In this formula, it is assumed that x is log-normally distributed, or, in other words, ln(x) is normally distributed. The average of a normal distribution is the arithmetic mean; hence, the exponential of the arithmetic mean over ln(xi) computes the average for x (see the right-hand side of the above formula). This equals the definition of the geometric mean (see the left-hand side of the formula). | 在这个公式中，假设是对数正态分布，换句话说，是正态分布。正态分布的平均值是算术平均值；因此，的算术均值指数计算的平均值（见上面公式的右边）。这等于几何平均值的定义（见公式的左边）。 |
| The geometric mean has an appealing property. One can compute the average speedup between two machines by dividing the average speedups for these two machines relative to some reference machine. In particular, having the average speedup numbers of machines A and B relative to some reference machine R, one can compute the average relative speedup between A and B by simply dividing the former speedup numbers. SPEC CPU uses the geometric mean for computing average SPEC rates, and the speedups are computed against some reference machine, namely a Sun Ultra5\_10 workstation with a 300MHz SPARC processor and 256MB main memory. | 几何平均值有一个吸引人的特性。可以通过将这两台机器相对于某个参考机器的平均加速比相除的方法，计算两台机器的平均加速比。特别是，有了机器A和B相对于参考机器R的平均加速比，可以通过简单地除以前者的加速比来计算A和B之间的平均相对加速比。SPEC CPU使用几何平均值来计算平均SPEC速率，加速比是相对于参考机器计算的，即使用300MHz SPARC处理器和256MB主存的Sun Ultra5\_10工作站。 |
| The geometric mean builds on two assumptions. For one, it assumes that the benchmarks are representative for the much broader workload space. A representative set of benchmarks can be obtained by randomly choosing benchmarks from the population, provided a sufficiently large number of benchmarks are taken. Unfortunately, the set of benchmarks is typically not chosen randomly from a well-defined workload space; instead,a benchmark suite is a collection of interesting benchmarks covering an application domain of interest picked by a committee, an individual or a marketing organization. In other words, and as argued in the introduction, picking a set of workloads is subject to the experimenter’s judgment and experience. Second, the geometric mean assumes that the speedups are distributed following a log-normal distribution. These assumptions have never been validated (and it is not clear how they can ever be validated),so it is unclear whether these assumptions holds true. Hence, given the high degree of uncertainty regarding the required assumptions, using the geometric mean for computing average performance numbers across a set of benchmarks is of questionable value. | 几何平均值建立在两个假设基础上。首先，它假设基准测试程序能够代表更广泛的工作负载空间。可以通过从总体中随机选择基准测试程序来获得具有代表性的一组基准测试程序，前提是采用足够多的基准测试程序。不幸的是，基准测试程序集通常不是从定义良好的工作负载空间中随机选择的；相反，基准测试套件是由委员会、个人或商业组织挑选的涵盖感兴趣的应用领域的基准测试程序的集合。换句话说，正如引言中提到的，选择一组工作负载取决于实验者的判断和经验。其次，几何平均值假设加速比服从对数正态分布。这些假设从未被验证过（也不清楚它们如何被验证），所以这些假设是否正确也不清楚。因此，考虑到所需假设的高度不确定性，使用几何平均值来计算一组基准的平均性能数字是可以质疑的。 |
| 2.4.3 FINAL THOUGHT ON AVERAGES | 2.4.3 关于平均值的进一步思考 |
| The above reasoning about how to compute averages reveals a basic problem. Harmonic mean (and arithmetic mean) can provide a meaningful summary for a given set of benchmarks, but then extrapolating that performance number to a full workload space is questionable. Geometric mean suggests a way of extrapolating performance to a full workload space, but the necessary assumptions are unproven (and are probably not valid). The one important exception is when the full workload space is known – as would be the case for some embedded systems. In this case, a weighted harmonic mean (or weighted arithmetic mean) would work well. | 上面关于如何计算平均值的推理揭示了一个基本问题。调和平均值（和算术平均值）可以为给定的基准测试程序集提供有意义的总结，但随后将性能数字外推到完整的工作负载空间是有问题的。几何平均值暗示了一种将性能外推到完整工作负载空间的方法，但是必要的假设没有经过验证（而且可能是无效的）。一个重要的例外情况是已知完整的工作负载空间——就像一些嵌入式系统的情况一样。在这种情况下，加权调和平均数（或加权算术平均数）可以很好地工作。 |
| 2.5 PARTIAL METRICS | 2.5 局部度量指标 |
| The metrics discussed so far consider overall performance, i.e., the metrics are based on total execution time. Partial metrics, such as cache miss ratio, branch misprediction rate, misses per thousand instructions, or bus utilization, reveal insight in what sources affect processor performance. Partial metrics allow for studying particular components of a processor in isolation. | 到目前为止讨论的度量考虑了总体性能，也就是说，指标基于总执行时间。部分度量指标，如缓存缺失率、分支预测错误率、每千条指令缺失率或总线利用率，可以揭示影响处理器性能的源头。局部度量指标允许独立地研究处理器的特定组件。 |

|  |  |
| --- | --- |
| CHAPTER 3 Workload Design | 第3章 工作负载设计 |
| Workload design is an important step in computer architecture research, as already described in the introduction: subsequent steps in the design process are subject to the selection process of benchmarks, and choosing a non-representative set of benchmarks may lead to biased observations, incorrect conclusions, and, eventually, designs that poorly match their target workloads. | 在引言中已经描述过，负载设计是计算机体系结构研究中的一个重要步骤：设计过程中的后续步骤取决于基准的选择过程，选择不具代表性的基准集可能会导致有偏差的观察结果，不正确的结论，最终，设计与目标负载不匹配。 |
| For example, Maynard et al. [137] as well as Keeton et al. [105] compare the behavior of commercial applications, including database servers, against the SPEC CPU benchmarks that are widely used in computer architecture. They find that commercial workloads typically exhibit more complex branch behavior, larger code and data footprints, and more OS as well as I/O activity. In particular, the instruction cache footprint of the SPEC CPU benchmarks is small compared to commercial workloads; also, memory access patterns for footprints that do not fit in on-chip caches are typically regular or strided. Hence, SPEC CPU benchmarks are well suited for pipeline studies, but they should be used with care for memory performance studies. Guiding processor design decisions based on the SPEC CPU benchmarks only may lead to suboptimal performance for commercial workloads. | 例如，Maynard等人[137]和Keeton等人[105]将商业应用程序（包括数据库服务器）的行为与计算机架构中广泛使用的SPEC CPU基准程程序进行了比较。他们发现，商业工作负载通常表现出更复杂的分支行为、更大的代码足迹和数据足迹，以及更多的操作系统和I/O活动。特别是，SPEC CPU基准测试的指令缓存足迹比商业工作负载小；此外，对于芯片缓存中不能容纳的足迹的内存访问模式通常是规则的或跨步的。因此，SPEC CPU基准测试非常适合流水线研究，但在使用它们时应该注意内存性能研究。仅根据SPEC CPU基准测试来指导处理器设计决策可能会导致商业工作负载的性能不佳。 |
| 3.1 FROM WORKLOAD SPACE TO REPRESENTATIVE WORKLOAD | 3.1 从负载空间到代表性负载 |
| Ideally,we would like to have a small set of benchmarks that is representative for the broader workload space, see also Figure 3.1. The workload space is defined as the collection of all possible applications. Although it may be possible to enumerate all the applications that will run on a given device in some application domains, i.e., in some embedded systems, this is not possible in general, e.g., the general-purpose domain.The reference workload then is the set of benchmarks that the experimenter believes to be representative for the broader workload space. In the embedded system example, it may be possible to map applications to benchmarks one-to-one and have a reference workload that very well represents the workload space. In the general-purpose computing domain, constructing the reference workload is subject to the experimenter’s judgment and experience. Because the number of benchmarks in the reference workload can be large, it may be appropriate to reduce the number of benchmarks towards a reduced workload. The bulk of this chapter deals with workload reduction methodologies which reduce the number of benchmarks in the reference workload to a limited number of benchmarks in the reduced workload. However, before doing so, we first discuss the difficulty in coming up with a good reference workload. | 理想情况下，我们希望有一小组基准测试，代表更广泛的工作负载空间，参见图3.1。工作负载空间（workload space）被定义为所有可能的应用程序的集合。虽然，在某些应用程序领域，例如在某些嵌入式系统中，可以枚举在给定设备上运行的所有应用程序，但在一般情况下，这是不可能的，例如在通用领域。参考工作负载（reference workload）就是实验者认为能够代表更广泛工作空间的一组基准。在嵌入式系统示例中，可以将应用程序映射到一对一的基准测试，并有一个非常好地表示工作负载空间的参考工作负载。在通用计算领域，参考工作负载的构建取决于实验人员的判断和经验。因为，参考工作负载中的基准测试数量可能很大，所以减少基准测试的数量以获得缩减工作负载（reduced workload）可能是合适的。本章主要讨论减少工作负载的方法，这些方法在缩减参考负载中，将参考工作负载中的基准数目减少到有限数目的基准。然而，在这样做之前，我们首先讨论提出一个良好的参考负载的困难。 |
|  | |
| Figure 3.1: Introducing terminology: workload space, reference workload and reduced workload. | 图3.1：术语介绍：工作负载空间，参考负载和缩减负载。 |
| 3.1.1 REFERENCE WORKLOAD | 3.1.1 参考负载 |
| Coming up with a good reference workload is a difficult problem for at least three reasons. For one, the workload space may be huge (and is typically not even well-defined).There exist many different, important, application domains, such as general-purpose computing, media (e.g., audio and video codecs), scientific computing, bioinformatics, medical, commercial (e.g., database, web servers, mail servers, e-commerce). And each of these domains has numerous applications and thus numerous potential benchmarks that can be derived from these applications. As a result, researchers have come up with different benchmark suites to cover different application domains, e.g., SPEC CPU (generalpurpose), MediaBench [124] (media), and BioPerf (bioinformatics) [8]. In addition, new benchmark suites emerge to evaluate new technologies, e.g., PARSEC [15] to evaluate multi-threaded workloads on multicore processors, DaCapo [18] to evaluate Java workloads, or STAMP [141] to evaluate transactional memory. In addition to this large number of existing application domains and benchmark suites, new application domains keep on emerging. For example, it is likely to expect that the trend towards cloud computing will change the workloads that will run on future computer systems. | 提出一个好的参考工作负载是一个困难的问题，至少有三个原因。首先，工作负载空间可能很大（而且通常没有良好定义）。存在许多不同的、重要的应用领域，如通用计算、媒体（如音频和视频编解码器）、科学计算、生物信息学、医疗、商业（如数据库、web服务器、邮件服务器、电子商务）。每一个领域都有大量的应用，因此可以从这些应用中派生出大量潜在的基准程序。因此，研究人员提出了不同的基准套件，以覆盖不同的应用领域，如SPEC CPU（通用）、MediaBench[124]（媒体）和BioPerf（生物信息学）[8]。此外，新的基准测试套件也出现了，用于评估新技术，例如PARSEC[15]用于评估多核处理器上的多线程工作负载，DaCapo[18]用于评估Java工作负载，或者STAMP[141]用于评估事务性内存。除了大量现有的应用程序域和基准测试套件之外，新的应用领域还在不断出现。例如，云计算的趋势可能会改变未来计算机系统上运行的工作负载。 |
| Second, the workload space is constantly changing, and, hence, the design may be optimized for a workload that is irrelevant (or at least, less important) by the time the product hits the market. This is a constant concern computer architects need to consider: given that it is unknown what the future’s workloads will be, architects are forced to evaluate future systems using existing benchmarks, which are often modeled after yesterday’s applications. In addition, architects also have to rely on old compiler technology for generating the binaries. To make things even worse, future instructionset architectures may be different than ones available today, which requires re-compilation and which may lead to different performance numbers; also, the workloads (e.g., transaction processing workloads such as databases) may need to be re-tuned for the system under study. In summary, one could say that architects are designing tomorrow’s systems using yesterday’s benchmarks and tools. To evaluate the impact of workload and tool drift, Yi et al. [198] describe an experiment in which they optimize a superscalar processor using SPEC CPU95 benchmarks and then evaluate the performance and energy-efficiency using SPEC CPU2000. They conclude that the CPU95 optimized processor performs 20% worse compared to the CPU2000 optimized design, the primary reason being that CPU2000 is more memory-intensive than CPU95.In conclusion,architects should be well aware of the impact workload drift may have, and they should thus anticipate future workload behavior as much as possible in order to have close to optimal performance and efficiency for future workloads running on future designs. | 其次，工作负载空间在不断变化。因此，到产品投放市场时，设计可能会针对不相关（至少不那么重要）的工作负载进行优化。这是计算机架构师需要考虑的一个持续的问题：鉴于未来的工作负载将是未知的，架构师被迫使用现有的基准来评估未来的系统，这些基准通常是模仿昨天的应用程序。此外，架构师还必须依赖旧的编译器技术来生成二进制文件。更糟糕的是，未来的指令集体系结构可能与现在可用的指令集不同，这需要重新编译，并可能导致不同的性能数据；此外，工作负载（例如，事务处理工作负载，如数据库）可能需要为所研究的系统重新调整。总而言之，可以说架构师正在使用昨天的基准程序和工具设计明天的系统。为了评估工作负载和工具漂移的影响，Yi等人[198]描述了一个实验，他们使用SPEC CPU95基准对超标量处理器进行优化，然后使用SPEC CPU2000评估性能和能效。他们得出的结论是，CPU95优化后的处理器性能比CPU2000优化后的处理器差20%，主要原因是CPU2000比CPU95更占用内存。总之，架构师应该充分意识到工作负载变化可能产生的影响，因此他们应该尽可能地预测未来的工作负载行为，以便在未来的设计上运行未来的工作负载，获得接近最佳的性能和效率。 |
| Finally, the process of selecting benchmarks to be included in a benchmark suite itself is subjective. John Henning [81] describes the selection process used by SPEC when composing the CPU2000 benchmark suite. The Standard Performance Evaluation Corporation (SPEC) is a nonprofit consortium whose members include hardware vendors, software vendors, universities, customers, and consultants.The SPEC subcommittee in charge of the CPU2000 benchmarks based its decisions on a number of criteria, ranging from portability and maintainability of the benchmarks’ source code, to performance characteristics of the benchmarks, to vendor interest. Hence, it is hard to argue that this selection process reflects an objective draw of a scientific sample from a population. | 最后，选择将基准测试包含在基准测试套件中的过程本身是主观的。John Henning[81]描述了SPEC在组成CPU2000基准套件时使用的选择过程。标准性能评估公司（SPEC）是一个非营利性组织，其成员包括硬件供应商、软件供应商、大学、客户和顾问。负责CPU2000基准的SPEC小组委员会基于一系列标准做出决定，从基准源代码的可移植性和可维护性，到基准的性能特征，再到供应商的兴趣。因此，很难说这种选择过程反映了从人群中客观抽取的科学样本。 |
| 3.1.2 TOWARDS A REDUCED WORKLOAD | 3.1.2 迈向缩减负载 |
| The large number of application domains and the large number of potential benchmarks per application domain leads to a huge number of potential benchmarks in the reference workload. Simulating all of these benchmarks is extremely time-consuming and is often impractical or infeasible. Hence, researchers have studied methods for reducing the number of benchmarks in order to save simulation time. Citron [33] performed a wide survey on current practice in benchmark subsetting. He found that selecting a limited number of benchmarks from a benchmark suite based on programming language, portability and simulation infrastructure concerns is common and may lead to misleading performance numbers. Hence, picking a reduced set of benchmarks should not be done in an ad-hoc way. | 大量的应用领域和每个应用领域的大量潜在基准测试导致参考工作负载中存在大量潜在基准测试。仿真所有这些基准极其耗时，而且往往不切实际或不可行。因此，研究人员研究了减少基准数量的方法，以节省仿真时间。Citron[33]对基准子集的当前实践进行了广泛的调查。他发现，基于编程语言、可移植性和仿真基础设施考虑，从基准测试套件中选择有限数量的基准测试是常见的，这可能会导致错误的性能数据。因此，不应该以一种临时想出来的方式来选择一组缩减的基准测试。 |
| This chapter describes two methodologies for composing a reduced but representative benchmark suite from a larger set of specified benchmarks, the reference workload. In other words, these methodologies seek to reduce the amount of work that needs to be done during performance studies while retaining meaningful results. The basic idea is to compose a reduced workload by picking a limited but representative set of benchmarks from the reference workload. The motivation for reducing the number of benchmarks in the workload is to limit the amount of work that needs to be done during performance evaluation. Having too many benchmarks only gets in the way: redundant benchmarks, i.e., benchmarks that exhibit similar behaviors and thus stress similar aspects of the design as other benchmarks, only increase experimentation time without providing any additional insight. | 本章描述了从一组更大的特定基准（参考工作负载）中组合出一个缩减的但具有代表性的基准套件的两种方法。换句话说，这些方法试图减少性能研究中需要做的工作量，同时保留有意义的结果。基本思想是通过从参考工作负载中选择一组有限但具有代表性的基准来组成一个缩减的工作负载。缩减工作负载中的基准测试数量的动机是限制在性能评估期间需要完成的工作量。过多的基准只会阻碍工作进展。冗余的基准，即表现出类似行为的基准，并因此强调与其他基准类似的设计方面。冗余基准只会增加实验时间，而不会提供任何额外的见解。 |
| The two methodologies described in this chapter are based on statistical data analysis techniques. The first approach uses Principal Component Analysis (PCA) for understanding the (dis)similarities across benchmarks, whereas the second approach uses the Plackett and Burman design of experiment. The fundamental insight behind both methodologies is that different benchmarks may exhibit similar behaviors and stress similar aspects of the design. As a result, there is no need to include such redundant benchmarks in the workload. | 本章中描述的两种方法是基于统计数据分析技术的。第一种方法使用主成分分析（PCA）来理解基准上的相似性和不相似性，而第二种方法使用Plackett和Burman的实验设计。这两种方法背后的根本观点是，不同的基准可能表现出相似的行为，强调设计的相似方面。因此，不需要在工作负载中包括这样的冗余基准测试。 |
| 3.2 PCA-BASED WORKLOAD DESIGN | 3.2 基于PCA的负载设计 |
| The first workload reduction methodology that we discuss is based on Principal Component Analysis (PCA) [96]. PCA is a well-known statistical data analysis technique that reduces the dimensionality of a data set. More precisely, it transforms a number of possibly correlated variables (or dimensions) into a smaller number of uncorrelated variables,which are called the principal components.Intuitively speaking, PCA has the ability to describe a huge data set along a limited number of dimensions. In other words, it presents the user with a lower-dimensional picture that yet captures the essence of the more-dimensional data set. | 我们讨论的第一个工作量减少方法是基于主成分分析（PCA）[96]。PCA是一种著名的统计数据分析技术，可以降低数据集的维数。更准确地说，它将一些可能相关的变量（或维度）转换成更少数量的不相关变量，这些变量被称为主成分。直观地说，PCA能够沿着有限的维数描述一个巨大的数据集。换句话说，它向用户呈现了一个低维的图像，但却捕捉到了更多维数据集的本质。 |
| Eeckhout et al. [53] describe how to leverage this powerful technique for analyzing workload behavior. They view the workload space as a p-dimensional space with p the number of important program characteristics.These program characteristics describe various behavioral aspects of a workload, such as instruction mix, amount of instruction-level parallelism (ILP), branch predictability, code footprint, memory working set size, and memory access patterns. Because the software and hardware are becoming more complex, the number of program characteristics p is large in order to capture a meaningful behavioral characterization. As a result, p is typically too large to easily visualize and/or reason about the workload space — visualizing a 20-dimensional space is not trivial. In addition, correlation may exist between these program characteristics. For example, complex program behavior may reflect in various metrics such as poor branch predictability, large code and memory footprint, and complex memory access patterns. As a result, a high value for one program characteristic may also imply a high value for another program characteristic. Correlation along program characteristics complicates the understanding: one may think two workloads are different from each other because they seem to be different along multiple dimensions; however, the difference may be due to a single underlying mechanism that reflects itself in several (correlated) program characteristics. | Eeckhout等人[53]描述了如何利用这种强大的技术来分析工作负载行为。他们认为工作负载空间是一个维空间，是重要程序特征的数量。这些程序特征描述了工作负载的各种行为方面，例如指令混合特性、指令级并行（ILP）、分支可预测性、代码足迹、内存负载集大小和内存访问模式。由于软件和硬件变得越来越复杂，为了捕获有意义的行为表征，程序特征的数量很大。结果是，通常太大，不容易可视化和/或推断工作负载空间——可视化20维空间并不是一件小事。此外，这些程序特征之间可能存在相关性。例如，复杂的程序行为可能反映在各种各样的指标中，例如较差的分支可预测性、较大的代码和内存足迹，以及复杂的内存访问模式。结果，一个程序特征的高值也可能意味着另一个程序特征的高值。沿着程序特征的相关性使理解复杂化：人们可能认为两种工作负载彼此不同，因为它们在多个维度上似乎是不同的；然而，这种差异可能是由于一种底层机制，它反映在几个（相关的）程序特征中。 |
| The large number of dimensions in the workload space and the correlation among the dimensions complicates the understanding substantially. PCA transforms the p-dimensional workload space to a q-dimensional space (with q p) in which the dimensions are uncorrelated. The transformed space thus provides an excellent opportunity to understand benchmark (dis)similarity. Benchmarks that are far away from each other in the transformed space show dissimilar behavior; benchmarks that are close to each other show similar behavior. When combined with cluster analysis (CA) as a subsequent step to pick to most diverse benchmarks in the reference workload, PCA can determine a reduced but representative workload. | 工作负载空间中的大量维度以及维度之间的相关性使理解变得非常复杂。PCA将维工作负载空间转换为维空间（），其中维度是不相关的。因此，转换后的空间提供了一个很好的机会来理解基准相似性和不相似性。在转换后的空间中，相距较远的基准显示出不同的行为；彼此接近的基准显示出相似的行为。当将聚类分析（CA）作为后续步骤来选择参考工作负载中最不同的基准时，PCA可以确定一个缩减但具有代表性的工作负载。 |
| 3.2.1 GENERAL FRAMEWORK | 3.2.1 通用框架 |
| Figure 3.2 illustrates the general framework of the PCA-based workload reduction method. It starts off from a set of benchmarks, called the reference workload. For each of these benchmarks, it then collects a number of program characteristics. This yields a large data matrix with as many rows as there are benchmarks, namely n, and with as many columns as there are program characteristics, namely p. Principal component analysis then transforms the p program characteristics into q principal components, yielding an n × q data matrix. Subsequently, applying cluster analysis identifies m representative benchmarks out of the n benchmarks (with m < n). | 图3.2展示了基于PCA的负载缩减方法的总体框架。它从一组基准测试开始，称为参考工作负载。对于每一个基准测试，它会收集许多程序特征。这产生了一个大的数据矩阵，有多少行就有多少基准，即，有多少列就有多少程序特征，即。然后主成分分析将个程序特征转换为个主成分，产生一个的数据矩阵。随后，应用聚类分析确定个基准中的个代表性基准（）。 |
| The next few sections describe the three major steps in this methodology: workload characterization, principal component analysis and cluster analysis. | 接下来的几节将描述此方法中的三个主要步骤：工作负载表征、主成分分析和聚类分析。 |
| 3.2.2 WORKLOAD CHARACTERIZATION | 3.2.2 负载表征 |
| The first step is to characterize the behavior of the various benchmarks in the workload. There are many ways for doing so. | 第一步是描述工作负载中各种基准测试的行为。有很多方法可以做到这一点。 |
| Hardware performance monitors. One way of characterizing workload behavior is to employ hardware performance monitors — in fact, hardware performance monitors are widely used (if not prevailing) in workload characterization because they are available on virtually all modern microprocessors, can measure a wide range of events, are easy to use, and allow for characterizing long-running complex workloads that are not easily simulated (i.e., the overhead is virtually zero).The events measured using hardware performance monitors are typically instruction mix (e.g., percentage loads, branches, floating-point operations, etc.), IPC (number of instructions retired per cycle), cache miss rates, branch mispredict rates, etc. | 硬件性能监视器。描述工作负载行为的一种方法是使用硬件性能监视器——实际上，硬件性能监视器在工作负载表征中被广泛使用（如果不能称为流行的话），因为它们几乎在所有现代微处理器上都可用，可以测量广泛的事件，易于使用，并允许描述不容易仿真的长时间运行的复杂工作负载（即，开销几乎为零）。使用硬件性能监视器测量的事件通常是指令混合特性（例如，load、分支、浮点操作等占的百分比）、IPC（每个周期退休指令的数量）、缓存缺失率、分支预测错误率等。 |
|  | |
| Figure 3.2: Schematic overview of the PCA-based workload reduction method. | 图3.2：基于PCA的负载缩减方法原理图 |
| Hardware performance monitors. One way of characterizing workload behavior is to employ hardware performance monitors — in fact, hardware performance monitors are widely used (if not prevailing) in workload characterization because they are available on virtually all modern microprocessors, can measure a wide range of events, are easy to use, and allow for characterizing long-running complex workloads that are not easily simulated (i.e., the overhead is virtually zero).The events measured using hardware performance monitors are typically instruction mix (e.g., percentage loads, branches, floating-point operations, etc.), IPC (number of instructions retired per cycle), cache miss rates, branch mispredict rates, etc. | 硬件性能监视器。描述工作负载行为的一种方法是使用硬件性能监视器——实际上，硬件性能监视器在工作负载表征中被广泛使用（如果不能称为流行的话），因为它们几乎在所有现代微处理器上都可用，可以测量广泛的事件，易于使用，并允许描述不容易仿真的长时间运行的复杂工作负载（即，开销几乎为零）。使用硬件性能监视器测量的事件通常是指令混合特性（例如，load、分支、浮点操作等占的百分比）、IPC（每个周期退休指令的数量）、缓存缺失率、分支预测错误率等。 |
| Inspite of its widespread use, there is a pitfall in using hardware performance monitors: they can be misleading in the sense that they can conceal the workload’s inherent behavior. This is to say that different inherent workload behavior can lead to similar behavior when measured using hardware performance monitors. As a result, based on a characterization study using hardware performance monitors one may conclude that different benchmarks exhibit similar behavior because they show similar cache miss rates,IPC,branch mispredict rates,etc.; however,a more detailed analysis based on a microarchitecture-independent characterization (as described next) shows that both benchmarks exhibit different inherent behavior. | 尽管广泛使用，但使用硬件性能监视器存在一个缺陷：它们可能会造成误导，因为它们可能会隐藏工作负载的固有行为。也就是说，在使用硬件性能监视器进行测量时，不同的固有工作负载行为可能导致类似的行为。因此，基于使用硬件性能监视器的特性研究，人们可能会得出这样的结论：不同的基准测试表现出相似的行为，因为它们显示出相似的缓存缺失率、IPC、分支预测错误率等；但是，基于与微体系结构无关的表征（如下所述）的更详细的分析表明，这两个基准测试表现出不同的内在行为。 |
| Hoste and Eeckhout [85] present data that illustrates exactly this pitfall, see also Table 3.1 for an excerpt.The two benchmarks, gzip with the graphic input from the SPEC CPU2000 benchmark suite and fasta from the BioPerf benchmark suite, exhibit similar behavior in terms of CPI and cache miss rates, as measured using hardware performance monitors. However, the working set size and memory access patterns are shown to be very different. The data working set size is an order of magnitude bigger for gzip compared to fasta, and the memory access patterns seem to be fairly different as well between these workloads. | Hoste和Eeckhout[85]提供的数据准确地说明了这一缺陷，参见表3.1的摘录。这两种基准测试（来自SPEC CPU2000测试讨论的以图片为输入的gzip程序和来自BioPerf基准测试套件的fasta程序）在CPI和缓存缺失率方面表现出了类似的行为（使用硬件性能监视器进行测量）。但是，工作集的大小和内存访问模式有很大的不同。与fasta相比，gzip的数据负载集大小要大一个数量级，而且这些工作负载的内存访问模式似乎也相当不同。 |
| Table 3.1: This case study illustrates the pitfall in using microarchitecture-dependent performance characteristics during workload characterization: although microarchitecture-dependent characteristics may suggest that two workloads exhibit similar behavior, this may not be the case when looking into the microarchitecture-independent characteristics. | 表3.1:这个案例研究说明了在工作负载描述中使用与微架构相关的性能特征的缺陷：尽管与微架构相关的特征可能意味着两个工作负载表现出相似的行为，但当研究与微架构无关的特征时，情况可能就不是这样了。 |
| |  |  |  | | --- | --- | --- | | Microarchitecture-dependent characterization | | | |  | gzip-graphic | fasta | | CPI on Alpha 21164 | 1.01 | 0.92 | | CPI on Alpha 21264 | 0.63 | 0.66 | | L1 D-cache misses per instruction | 1.61% | 1.90% | | L1 I-cache misses per instruction | 0.15% | 0.18% | | L2 cache misses per instruction | 0.78% | 0.25% | | Microarchitecture-independent characterization | | | |  | gzip-graphic | fasta | | Data working set size (# 4KB pages) | 46,199 | 4,058 | | Instruction working set size (# 4KB pages) | 33 | 79 | | Probability for two consecutive dynamic executions of the same static load to access the same data | 0.67 | 0.30 | | Probability for two consecutive dynamic executions of the same static store to access the same data | 0.64 | 0.05 | | Probability for the difference in memory addresses between two consecutive loads in the dynamic instruction stream to be smaller than 64 | 0.26 | 0.18 | | Probability for the difference in memory addresses between two consecutive stores in the dynamic instruction stream to be smaller than 64 | 0.35 | 0.93 | | |
| |  |  |  | | --- | --- | --- | | 微架构相关特性 | | | |  | gzip-graphic | fasta | | Alpha 21164架构上的CPI | 1.01 | 0.92 | | Alpha 21264架构上的CPI | 0.63 | 0.66 | | 平均每条指令的L1数据缓存缺失率 | 1.61% | 1.90% | | 平均每条指令的L1指令缓存缺失率 | 0.15% | 0.18% | | 平均每条指令的L2数据缓存缺失率 | 0.78% | 0.25% | | 微架构无关特性 | | | |  | gzip-graphic | fasta | | 数据负载大小（4KB页数量） | 46,199 | 4,058 | | 指令负载大小（4KB 页数量） | 33 | 79 | | 连续两次动态执行相同的静态load访问相同数据的概率 | 0.67 | 0.30 | | 连续两次动态执行相同的静态store访问相同数据的概率 | 0.64 | 0.05 | | 动态指令流中两个连续load的内存地址差小于64的概率 | 0.26 | 0.18 | | 动态指令流中两个连续store的内存地址差小于64的概率 | 0.35 | 0.93 | | |
| The notion of microarchitecture-independent versus microarchitecture-dependent characterization also appears in sampled simulation, see Chapter 6. | 微架构无关表征和微架构相关表征的概念也出现在采样仿真中，参见第6章。 |
| Hardware performance monitor data across multiple machines. One way for alleviating this pitfall is to characterize the workload on a multitude of machines and architectures. Rather than collecting hardware performance monitor data on a single machine, collecting data across many different machines is likely to yield a more comprehensive and more informative workload characterization because different machines and architectures are likely to stress the workload behavior (slightly) differently. As a result, an inherent behavioral difference between benchmarks is likely to show up on at least one of a few different machines. | 跨多台机器的硬件性能监视器数据。减轻这种缺陷的一种方法是在多种机器和体系结构上描述工作负载。与在单个机器上收集硬件性能监控数据不同，跨多台不同机器收集数据可能会产生更全面、信息量更大的工作负载表征，因为不同的机器和体系结构对工作负载行为的压力可能（略有）不同。因此，基准测试之间的内在行为差异可能至少会在几台不同的机器中的一台上显示出来。 |
| Phansalkar et al.[160] describe an experiment in which they characterize the SPEC CPU2006 benchmark suite on five different machines with four different ISAs and compilers (IBM Power, Sun UltraSPARC, Itanium and x86). They use this multi-machine characterization as input for the PCA-based workload analysis method, and then study the diversity among the benchmarks in the SPEC CPU2006 benchmark suite. This approach was used by SPEC for the development of the CPU2006 benchmark suite [162]: the multi-machine workload characterization approach was used to understand the diversity and similarity among the benchmarks for potential inclusion in the CPU2006 benchmark suite. | Phansalkar等人[160]描述了一个实验，他们在5台不同的机器上用4种不同的ISA和编译器（IBM Power、Sun UltraSPARC、Itanium和x86）描述了SPEC CPU2006基准套件。他们使用这种多机器特性作为基于PCA的工作负载分析方法的输入，然后研究SPEC CPU2006基准套件中各种基准之间的差异。SPEC在开发CPU2006基准套件时使用了这种方法[162]：使用多机器工作负载表征方法来了解可能包含在CPU2006基准套件中的基准之间的多样性和相似性。 |
| Detailed simulation. One could also rely on detailed cycle-accurate simulation for collecting program characteristics in a way similar to using hardware performance monitors.The main disadvantage is that is extremely time-consuming to simulate industry-standard benchmarks in a cycle-accurate manner — cycle-accurate simulation is at least five orders of magnitude slower than native hardware execution. The benefit though is that simulation enables collecting characteristics on a range of machine configurations that are possibly not (yet) available. | 详细的仿真。我们还可以依赖于详细的周期精确仿真，以类似于使用硬件性能监视器的方式收集程序特征。主要缺点是，以周期精确的方式仿真行业标准基准测试非常耗时——周期精确的仿真比本地硬件执行慢至少五个数量级。不过，这种仿真的好处是可以在一系列可能（还）不可用的机器配置上收集特征。 |
| Microarchitecture-independent workload characterization. Another way for alleviating the hardware performance monitor pitfall is to collect a number of program characteristics that are independent of a specific microarchitecture.The key benefit of a microarchitecture-independent characterization is that it is not biased towards a specific hardware implementation. Instead, it characterizes a workload’s ‘inherent’ behavior (but is still dependent on the instruction-set architecture and compiler). The disadvantage of microarchitecture-independent characterization is that most of the characteristics can only be measured through software. Although measuring these characteristics is done fairly easily through simulation or through binary instrumentation (e.g., using a tool like Atom [176] or Pin [128]), it can be time-consuming because the simulation and instrumentation may incur a slowdown of a few orders of magnitude — however, it will be much faster than detailed cycle-accurate simulation. In practice though, this may be a relatively small concern because the workload characterization is a one-time cost. | 微架构无关工作负载特性。减轻硬件性能监视缺陷的另一种方法是收集一些独立于特定微架构的程序特征。与微架构无关的特性的关键好处是它不偏向于特定的硬件实现。相反，它描述了工作负载的“固有”行为（但仍然依赖于指令集架构和编译器）。与微架构无关的表征方法的缺点是大多数特征只能通过软件来测量。虽然通过仿真或二进制插桩（例如，使用Atom[176]或Pin[128]等工具）可以很容易地测量这些特性，但这可能会很耗时，因为仿真和插桩可能会导致几个数量级的减速——然而，这比详细的周期精确的仿真快得多。但在实践中，这可能是一个相对较小的问题，因为工作负载描述是一次性成本。 |
| Example microarchitecture-independent characteristics are shown in Table 3.2. They include characteristics for code footprint,working set size,branch transition rate,and memory access patterns (both local and global). See the work by Joshi et al. [99] and Hoste and Eeckhout [85] for more detailed examples. | 与微架构无关的特征示例如表3.2所示。它们包括代码足迹、工作负载大小、分支转移速率和内存访问模式（包括本地和全局）的特征。参见Joshi等[99]和Hoste和Eeckhout[85]的研究，了解更详细的例子。 |
| Table 3.2: Example microarchitecture-independent characteristics that can be used as input for the PCA-based workload reduction method | 表3.2：与微架构无关的示例特征，可用于基于PCA的工作量减少方法的输入 |
| |  |  | | --- | --- | | Program Characteristic | Description | | instruction mix | Percentage of loads, stores, branches, integer arithmetic operations, floating-point operations. | | instruction-level parallelism (ILP) | Amount of ILP for a given window of instructions, e.g., the IPC achieved for an idealized out-of-order processor (with perfect branch predictor and caches). | | data working set size | The number of unique memory blocks or pages touched by the data stream. | | code footprint | The number of unique memory blocks or pages touched by the instruction stream. | | branch transition rate | The number of times that a branch switches between taken and not-taken directions during program execution. | | data stream strides | Distribution of the strides observed between consecutive loads or stores in the dynamic instruction stream.  The loads or stores could be consecutive executions of the same static instruction (local stride) or could be consecutive instructions from whatever load or store (global stride). | | |
| |  |  | | --- | --- | | 程序特征 | 描述 | | 指令混合特性 | Load、store、分支、整形计算、浮点操作的比例。 | | 指令级并行（ILP） | 对于给定的指令窗口，ILP的数量。例如，对于一个理想的乱序处理器（具有完美的分支预测器和缓存）所实现的IPC。 | | 数据集大小 | 数据流所接触的唯一内存块或页的数目。 | | 代码足迹 | 指令流所接触的唯一内存块或页的数目。 | | 分支转移概率 | 在程序执行期间，一个分支在发生方向和不发生方向之间切换的次数。 | | 数据流偏移 | 动态指令流中在连续load或store之间观察到的步长分布。  load或store可以是同一静态指令的连续执行（局部跨步），也可以是来自任何加载或存储的连续指令（全局跨步）。 | | |
| 3.2.3 PRINCIPAL COMPONENT ANALYSIS | 3.2.3 主成分分析 |
| The second step in the workload reduction method is to apply principal components analysis (PCA) [96]. PCA is a statistical data analysis technique that transforms a number of possibly correlated variables (i.e., program characteristics) in a smaller number of uncorrelated principal components. | 缩减负载方法的第二步是应用主成分分析（PCA）[96]。PCA是一种统计数据分析技术，它将大量可能相关的变量（即程序特征）转换为较少数量的不相关主成分。 |
| The principal components are linear combinations of the original variables, and they are uncorrelated. Mathematically speaking, PCA transforms p variables X1,X2,...,Xp into p principal components with . This transformation is done such that the first principal component shows the greatest variance, followed by the second, and so forth: Var[Z1] > Var[Z2] > ... > Var[Zp]. Intuitively speaking, this means that the first principal component Z1 captures the most ‘information’ and the final principal component Zp the least. In addition, the principal components are uncorrelated, or Cov[Zi,Zj] = 0,∀i = j, which basically means that there is no information overlap between the principal components. Note that the total variance in the data remains the same before and after the transformation, namely. Computing the principal components is done by computing the eigenvalue decomposition of a data covariance matrix. Figure 3.3 illustrates how PCA operates in a two-dimensional space on a Gaussian distributed data set. The first principal component is determined by the direction with the maximum variance; the second principal component is orthogonal to the first one. | 主成分是原始变量的线性组合，它们不相关。从数学上讲，PCA变换 变量化为个主分量，其中。这种转换是这样完成的：第一个主成分显示最大的方差，然后是第二个，依此类推：。直观地说，这意味着第一个主成分捕获了最多的“信息”，而最后一个主成分捕获了最少的“信息”。另外，主成分不相关，或者，这基本上意味着主成分之间不存在信息重叠。注意，数据中的总方差在转换前后保持不变，即。计算主成分是通过计算特征值分解的数据协方差矩阵。图3.3说明了PCA如何在二维空间的高斯分布数据集上运行。第一主成分由方差最大的方向确定；第二个主分量与第一个正交。 |
|  | |
| Figure 3.3: PCA identifies the principal components in a data set. | 图3.3：PCA在数据集中识别主成分。 |
| Because the first few principal components capture most of the information in the original data set, one can drop the trailing principal components with minimal loss of information.Typically, the number of retained principal components q is much smaller than the number of dimensions p in the original data set. The amount of variance accounted for by the retained principal components, provides a measure for the amount of information retained after PCA. Typically, over 80% of the total variance is explained by the retained principal components. | 因为前几个主成分捕获了原始数据集中的大部分信息，所以可以以最小的信息损失删除后面的主成分。通常情况下，保留的主成分数量比原始数据集中的维数要少得多。保留的主成分的方差量，提供了对于主成分分析后保留信息数量的度量。通常，80%以上的总方差是由保留的主成分解释的。 |
| PCA can be performed by most existing statistical software packages, both in commercial packages such as SPSS, SAS and S-PLUS, as well as open-source packages such as R. | 现有的大多数统计软件包都可以进行PCA分析，包括SPSS、SAS、S-PLUS等商业软件包，也包括开源软件包，如R。 |
| 3.2.4 CLUSTER ANALYSIS | 3.2.4 聚类分析 |
| The end result from PCA is a data matrix with n rows (the benchmarks) and q columns (the principal components). Cluster analysis now groups (or clusters) the n benchmarks based on the q principal components. The final goal is to obtain a number of clusters, with each cluster grouping a set of benchmarks that exhibit similar behavior. There exist two common clustering techniques, namely agglomerative hierarchical clustering and K-means clustering. | PCA的最终结果是一个有行（基准）和列（主成分）的数据矩阵。现在，聚类分析根据个主成分对个基准进行分组（或聚类）。最终目标是获得大量集群，每个集群将一组表现出类似行为的基准分组。常见的聚类技术有两种，即聚集层次聚类和k均值聚类。 |
| Agglomerative hierarchical clustering considers each benchmark as a cluster initially. In each iteration of the algorithm, the two clusters that are closest to each other are grouped to form a new cluster. The distance between the merged clusters is called the linkage distance. Nearby clusters are progressively merged until finally all benchmarks reside in a single big cluster.This clustering process can be represented in a so called dendrogram, which graphically represents the linkage distance for each cluster merge. Having obtained a dendrogram, it is up to the user to decide how many clusters to retain based on the linkage distance. Small linkage distances imply similar behavior in the clusters, whereas large linkage distances suggest dissimilar behavior. There exist a number of methods for calculating the distance between clusters — the inter-cluster distance is needed in order to know which clusters to merge. For example, the furthest neighbor method (also called complete-linkage clustering) computes the largest distance between any two benchmarks in the respective clusters; in average-linkage clustering, the inter-cluster distance is computed as the average distance. | 聚集层次聚类最初将每个基准视为一个集群。在算法的每次迭代中，将相距最近的两个聚类分组形成一个新的聚类。合并簇之间的距离称为连接距离（linkage distance）。附近的集群逐渐合并，直到最后所有基准都驻留在一个大集群中。这个聚类过程可以用一个所谓的树图（dendrogram）来表示，它图形地表示了每个聚类合并的连接距离。在得到一个树图后，由用户根据连杆距离决定保留多少簇。小的连接距离意味着在集群中类似的行为，而大的连接距离意味着不同的行为。有许多计算簇间距离的方法——簇间距离是需要知道哪些簇要合并。例如，最远邻居方法（也称为完全连接聚类）计算各自集群中任意两个基准之间的最大距离;在平均连接聚类中，簇间距离以平均距离计算。 |
| K-means clustering starts by randomly choosing k cluster centers. Each benchmark is then assigned to its nearest cluster, and new cluster centers are computed. The next iteration then recomputes the cluster assignments and cluster centers. This iterative process is repeated until some convergence criterion is met. The key advantage of K-means clustering is its speed compared to hierarchical clustering, but it may lead to different clustering results for different initial random assignments. | K均值聚类首先随机选择个聚类中心。然后，每个基准被分配到最近的集群，并计算新的集群中心。然后下一次迭代重新计算集群分配和集群中心。这个迭代过程重复，直到满足某个收敛准则。与层次聚类相比，K均值聚类的关键优势在于速度快，但对于不同的初始随机赋值，可能会导致不同的聚类结果。 |
| 3.2.5 APPLICATIONS | 3.2.5 应用 |
| The PCA-based methodology enables various applications in workload characterization. | 基于PCA的方法支持工作负载表征中的各种应用。 |
| Workload analysis. Given the limited number of principal components, one can visualize the workload space, as illustrated in Figure 3.4, which shows the PCA space for a set of SPEC CPU95 benchmarks along with TPC-D running on the postgres DBMS. (This graph is based on the data presented by Eeckhout et al. [53] and represents old and obsolete data — both CPU95 and TPC-D are obsolete — nevertheless, it illustrates various aspects of the methodology.) The graphs show the various benchmarks as dots in the space spanned by the first and second principal components, and third and fourth principal components, respectively. Collectively, these principal components capture close to 90% of the total variance, and thus they provide an accurate picture of the workload space. The different colors denote different benchmarks; the different dots per benchmark denote different inputs. | 工作负载分析。由于主成分数量有限，可以可视化工作负载空间，如图3.4所示，其中显示了一组SPEC CPU95基准测试的PCA空间，以及在postgres DBMS上运行的TPC-D。（此图基于Eeckhout等人[53]提供的数据，表示旧的和过时的数据——CPU95和TPC-D都过时了——尽管如此，它说明了该方法的各个方面。）这些图表分别以第一和第二主成分以及第三和第四个主成分所跨越的空间中的点表示各种基准。总的来说，这些主要组件捕获了近90%的总方差，因此它们提供了工作负载空间的准确图像。不同的颜色表示不同的基准；每个基准的不同点表示不同的输入。 |
|  | |
| Figure 3.4: Example PCA space as a function of the first four principal components: the first and second principal components are shown in the top graph, and the third and fourth principal components are shown in the bottom graph. | 图3.4：示例主成分空间作为前四个主成分的函数：上图显示第一和第二主成分，下图显示第三和第四个主成分。 |
| By interpreting the principal components, one can reason about how benchmarks differ from each other in terms of their execution behavior. The first principal component primarily quantifies a benchmark’s control flow behavior: benchmarks with relatively few dynamically executed branch instructions and relatively low I-cache miss rates show up with a high first principal component. One example benchmark with a high first principal component is ijpeg. Benchmarks with high levels of ILP and poor branch predictability have a high second principal component, see for example go and compress. The third and fourth primarily capture D-cache behavior and the instruction mix, respectively. | 通过解释主成分，可以推断出基准测试在执行行为方面是如何彼此不同的。第一个主成分主要量化基准测试的控制流行为：动态执行分支指令相对较少、指令缓存缺失率相对较低的基准测试会出现第一个主组件较高的情况。一个具有高第一主成分的基准测试示例是ijpeg。具有较高的ILP水平和较差的分支可预见性的基准具有较高的第二主成分，例如go和compress。第三种和第四种主要分别捕获数据缓存行为和指令混合特性。 |
| Several interesting observations can be made from these plots. Some benchmarks exhibit execution behavior that is different from the other benchmarks in the workload. For example, ijpeg, go and compress seem to be isolated in the workload space and seem to be relatively dissimilar from the other benchmarks. Also, the inputs given to the benchmark may have a big impact for some benchmarks, e.g.,TPC-D, whereas for other benchmarks the execution behavior is barely affected by its input, e.g., ijpeg.The different inputs (queries) are scattered around for TPC-D; hence, different inputs seem to lead to fairly dissimilar behavior; for ijpeg, on the other hand, the inputs seem to clustered, and inputs seem to have limited effect on the program’s execution behavior. Finally, it also suggests that this set of benchmarks only partially covers the workload space. In particular, a significant part of the workload space does not seem to be covered by the set of benchmarks, as illustrated in Figure 3.5. | 从这些图中可以观察到几个有趣的现象。一些基准测试的执行行为不同于工作负载中的其他基准测试。例如，ijpeg、go和compress在工作负载空间中似乎是孤立的，并且似乎与其他基准测试相对不同。此外，给基准测试的输入可能对某些基准测试有很大的影响，例如TPC-D，而对于其他基准测试，执行行为几乎不受输入的影响，例如ijpeg。对于TPC-D，不同的输入（查询）分散在各处；因此，不同的输入似乎会导致相当不同的行为；另一方面，对于ijpeg，输入似乎是聚集的，而且输入对程序执行行为的影响似乎有限。最后，还可以发现这组基准测试仅部分覆盖工作负载空间。特别是，工作负载空间的一个重要部分似乎没有被基准测试集覆盖，如图3.5所示。 |
| 散点图  低可信度描述已自动生成 | |
| Figure 3.5: The PCA-based workload analysis method allows for finding regions (weak spots) in the workload space that are not covered by a benchmark suite. Weak spots are shaded in the graph. | 图3.5：基于PCA的工作负载分析方法允许找到工作负载空间中没有被基准测试套件覆盖的区域（弱点）。图中用阴影标出了弱点。 |
| Workload reduction. By applying cluster analysis after PCA, one can group the various benchmarks into a limited number of clusters based on their behavior, i.e., benchmarks with similar execution behavior are grouped in the same cluster. The benchmark closest to the cluster’s centroid can then serve as the representative for the cluster. By doing so, one can reduce the workload to a limited set of representative benchmarks — also referred to as benchmark subsetting. Phansalkar et al. [160] present results for SPEC CPU2006, and they report average prediction errors of 3.8% and 7% for the subset compared to the full integer and floating-point benchmark suite, respectively, across five commercial processors. Table 3.3 summarizes the subsets for the integer and floating-point benchmarks. | 工作负载缩减。通过在PCA后应用聚类分析，可以根据不同的基准测试的行为将它们分组到有限数量的集群中，也就是说，具有相似执行行为的基准测试被分组在同一个集群中。然后，最接近集群质心的基准可以作为集群的代表。通过这样做，可以将工作负载减少到一组有限的代表性基准——也称为基准子集。Phansalkar等人[160]给出了SPEC CPU2006的结果，他们报告，在5个商业处理器上，与完整整数和浮点基准套件相比，该子集的平均预测误差分别为3.8%和7%。表3.3总结了整数和浮点基准测试的子集。 |
| Table 3.3: Representative SPEC CPU2006 subsets according to the study done by Phansalkar et al. [160]. | 图3.3：Phansalker等[160]研究得出的SPEC CPU2006的代表性子集 |
| |  |  | | --- | --- | | SPEC CINT2006 | 400.perlbench, 471.omnetpp, 429.mcf, 462.libquantum, 473.astar, 483.xalancbmk | | SPEC CFP2006 | 437.leslie3d, 454.calculix, 459.GemsFDTD, 436.cactusADM, 447.dealII, 450.soplex, 470.lbm, 453.povray | | |
| Other applications. The PCA-based methodology has been used for various other or related purposes, including evaluating the DaCapo benchmark suite [18], analyzing workload behavior over time [51],studying the interaction between the Java application,its input and the Java virtual machine (JVM) [48], and evaluating the representativeness of (reduced) inputs [52]. | 其他应用。基于PCA的方法已用于各种相关目的，包括评估DaCapo基准套件[18]、分析工作负载随时间变化的行为[51]、研究Java应用程序、程序输入和Java虚拟机（JVM）之间的交互[48]，以及评估（缩减）输入的代表性[52]。 |
| 3.3 PLACKETT AND BURMAN BASED WORKLOAD DESIGN | 3.3基于Plackett和Burman的工作负载设计 |
| Yi et al. [196] describe a simulation-friendly approach for understanding how workload performance is affected by microarchitecture parameters. They therefore employ a Plackett and Burman (PB) design of experiment which involves a small number of simulations — substantially fewer simulations than simulating all possible combinations of microarchitecture parameters — while still capturing the effects of each parameter and selected interactions. Also, it provides more information compared to a one-at-a-time experiment for about the same number of simulations: PB captures interaction effects which a one-at-a-time experiment does not. In particular, a Plackett and Burman design (with foldover) involves 2c simulations to quantify the effect of c microarchitecture parameters and all pairwise interactions. The outcome of the PB experiment is a ranking of the most significant microarchitecture performance bottlenecks. Although the primary motivation for Yi et al. to propose the Plackett and Burman design was to explore the microarchitecture design space in a simulationfriendly manner, it also has important applications in workload characterization. The ranking of performance bottlenecks provides a unique signature that characterizes a benchmark in terms of how it stresses the microarchitecture. By comparing bottleneck rankings across benchmarks one can derive how (dis)similar the benchmarks are. | Yi等[196]描述了一种仿真友好的方法来理解工作负载性能是如何受微架构参数影响的。因此，他们采用了Plackett和Burman （PB）设计的实验，其中涉及少量的仿真——显著少于仿真微架构参数的所有可能组合——同时仍然捕获每个参数和选定的交互作用的影响。此外，对于相同数量的仿真，与一次一次的实验相比，它提供了更多的信息：PB捕捉了一次一次的实验没有捕捉到的相互作用效应。特别是，Plackett和Burman设计（带有折叠式）涉及次仿真，以量化个微架构参数和所有成对交互的影响。PB实验的结果是对最重要的微架构性能瓶颈的排名。虽然Yi等提出Plackett和Burman设计的主要动机是为了以一种仿真友好的方式探索微架构设计空间，但它在工作负载表征方面也有重要的应用。性能瓶颈的排名提供了一个独特的特征，它描述了基准测试对微架构的压力。通过比较不同基准的瓶颈排名，可以得出这些基准的相似程度。 |
| The Plackett and Burman design uses a design matrix — Table 3.4 shows an example design matrix with foldover — Yi et al. and the original paper by Plackett and Burman provide design matrices of various dimensions. A row in the design matrix corresponds to a microarchitecture configuration that needs to be simulated; each column denotes a different microarchitecture parameter. A ‘+1’ and ‘−1’ value represents a high and low — or on and off — value for a parameter. For example, a high and low value could be a processor width of 8 and 2, respectively; or with aggressive hardware prefetching and without prefetching, respectively. It is advised that the high and low values be just outside of the normal or expected range of values in order to take into account the full potential impact of a parameter. The way the parameter high and low values are chosen may lead to microarchitecture configurations that are technically unrealistic or even infeasible. In other words, the various microarchitecture configurations in a Plackett and Burman experiment are corner cases in the microarchitecture design space. | Plackett和Burman设计使用了一个设计矩阵——表3.4显示了一个带有折叠的设计矩阵示例——Yi等和Plackett和Burman的原始论文提供了各种维度的设计矩阵。设计矩阵中的一行对应于需要仿真的微架构配置；每一列表示不同的微架构参数。“+1”和“−1”值表示参数的高、低或开、关值。例如，high和low值可以分别是处理器宽度为8和2的值；或者分别使用主动的硬件预取和不使用预取。建议高值和低值刚好超出正常或预期值范围，以便考虑参数的全部潜在影响。参数的高低值的选择可能导致微架构配置在技术上是不现实的，甚至是不可行的。换句话说，在Plackett和Burman实验中的各种微架构配置是微架构设计空间中的角落案例。 |
| Table 3.4: An example Plackett and Burman design  matrix with foldover | 表3.4 Plackett和Burman设计矩阵（有折叠）的示例。 |
| |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | A | B | C | D | E | F | G | exec. Time | | +1 | +1 | +1 | −1 | +1 | −1 | −1 | 56 | | −1 | +1 | +1 | +1 | −1 | +1 | −1 | 69 | | −1 | −1 | +1 | +1 | +1 | −1 | +1 | 15 | | +1 | −1 | −1 | +1 | +1 | +1 | −1 | 38 | | −1 | +1 | −1 | −1 | +1 | +1 | +1 | 45 | | +1 | −1 | +1 | −1 | −1 | +1 | +1 | 100 | | +1 | +1 | −1 | +1 | −1 | −1 | +1 | 36 | | −1 | −1 | −1 | −1 | −1 | −1 | −1 | 20 | | −1 | −1 | −1 | +1 | −1 | +1 | +1 | 77 | | +1 | −1 | −1 | −1 | +1 | −1 | +1 | 87 | | +1 | +1 | −1 | −1 | −1 | +1 | −1 | 5 | | −1 | +1 | +1 | −1 | −1 | −1 | +1 | 9 | | +1 | −1 | +1 | +1 | −1 | −1 | −1 | 28 | | −1 | +1 | −1 | +1 | +1 | −1 | −1 | 81 | | −1 | −1 | +1 | −1 | +1 | +1 | −1 | 67 | | +1 | +1 | +1 | +1 | +1 | +1 | +1 | 2 | | −31 | −129 | −44 | −44 | 47 | 71 | 7 |  | | |
| The next step in the procedure is to simulate these microarchitecture configurations, collect performance numbers, and calculate the effect that each parameter has on the variation observed in the performance numbers. The latter is done by multiplying the performance number for each configuration with its value (+1 or −1) and by, subsequently, adding these products across all configurations. For example, the effect for parameter A is | 该过程的下一步是仿真这些微架构配置，收集性能数字，并计算每个参数对观察到的性能数字变化的影响。后者是通过将每个配置的性能数字与其值（+1或−1）相乘，然后在所有配置中添加这些乘积来实现的。例如，参数的作用为 |
|  | |
| Similarly, one can compute the effect of pairwise effects by multiplying the performance number for each configuration with the product of the parameters’ values , and adding these products across all configurations. For example, the interaction effect between A and C is computed as: | 类似地，可以通过将每个配置的性能数字与参数值的乘积相乘，并将所有配置中的这些乘积相加，来计算成对效应的影响。例如，与的相互作用效应计算为: |
|  | |
| After having computed the effect of each parameter, the effects (including the interaction effects) can be ordered to determine their relative impact. The sign of the effect is meaningless, only the magnitude is. An effect with a higher ranking is more of a performance bottleneck than a lower ranked effect. For the example data in Table 3.4, the most significant parameter is B (with an effect of -129). | 在计算出每个参数的影响后，可以对这些影响（包括交互影响）进行排序，以确定它们的相对影响。效应的符号是没有意义的，只有幅度是有意义的。排名较高的效应比排名较低的效应更像是性能瓶颈。对于表3.4中的示例数据，最重要的参数是（影响为-129）。 |
| By running a Plackett and Burman experiment on a variety of benchmarks, one can compare the benchmarks against each other. In particular, the Plackett and Burman experiment yields a ranking of the most significant performance bottlenecks. Comparing these rankings across benchmarks provides a way to assess whether the benchmarks stress similar performance bottlenecks, i.e., if the top N ranked performance bottlenecks and their relative ranking is about the same for two benchmarks, one can conclude that both benchmarks exhibit similar behavior. | 通过在各种基准上运行Plackett和Burman实验，人们可以将这些基准相互比较。特别是，Plackett和Burman实验得出了最重要的性能瓶颈的排名。在不同的基准中比较这些排名提供了一种判断评估基准是否强调类似性能瓶颈的方法，即，如果排名前的性能瓶颈及其相对排名对于两个基准来说是相同的，那么可以得出两个基准表现出相似的行为的结论。 |
| Yi et al. [197] compare the PCA-based and PB-based methodologies against each other. The end conclusion is that both methodologies are equally accurate in terms of how well they can identify a reduced workload. Both methods can reduce the size of the workload by a factor of 3 while incurring an error (difference in IPC for the reduced workload compared to the reference workload across a number of processor architectures) of less than 5%. In terms of computational efficiency, the PCA-based method was found to be more efficient than the PB-based approach. Collecting the PCA program characteristics was done more efficiently than running the detailed cycle-accurate simulations needed for the PB method. | Yi等人[197]比较了基于PCA和基于PB的方法。最后的结论是，两种方法在识别减少的工作量方面都是同样准确的。这两种方法都可以将工作负载的大小减少3倍，同时误差（减少的工作负载与跨多个处理器架构的参考工作负载的IPC差异）不到5%。在计算效率方面，基于PCA的方法比基于PB的方法更高效。收集PCA程序特征比运行PB方法所需的详细的周期精确仿真更高效。 |
| 3.4 LIMITATIONS AND DISCUSSION | 3.4 局限和讨论 |
| Both the PCA-based as well as the PB-based workload design methodologies share a common pitfall, namely the reduced workload may not capture all the behaviors of the broader set of applications. In other words, there may exist important behaviors that the reduced workload does not cover. The fundamental reason is that the reduced workload is representative with respect to the reference workload from which the reduced workload is derived. The potential pitfall is that if the reference workload is non-representative with respect to the workload space, the reduced workload may not be representative for the workload space either. | 基于PCA和基于PB的工作负载设计方法都有一个共同的缺陷，即缩减工作负载可能无法捕捉更广泛的应用程序集的所有行为。换句话说，可能存在一些重要的行为是缩减工作负载所没有涵盖的。最根本的原因是缩减工作量相对于参考工作负载具有代表性，参考工作负载是缩减工作负载的来源。潜在的缺陷是，如果参考工作负载相对于工作负载空间来说不具有代表性，那么缩减工作负载可能也不具有代表性。 |
| Along the same line, the reduced workload is representative for the reference workload with respect to the input that was given to the workload reduction method. In particular, a PCA-based method considers a set of program characteristics to gauge behavioral similarity across benchmarks, and,as a result,evaluating a microarchitecture feature that is not captured by any of the program characteristics during workload reduction may be misleading. Similarly, the PB-based method considers a (limited) number of microarchitecture configurations during workload reduction; a completely different microarchitecture than the ones considered during workload reduction may yield potentially different performance numbers for the reduced workload than for the reference workload. | 按照同样的思路，缩减工作负载代表了参考工作负载，即对工作负载缩减方法所提供的输入。特别是，基于PCA的方法考虑一组程序特征来衡量跨基准的行为相似性。因此，在使用减少工作负载的时候，评估没有被任何程序特征捕获的微架构特征，可能会对工作负载产生误导。类似地，基于PB的方法在减少工作负载时考虑（有限）数量的微架构配置；与工作负载缩减过程中考虑的微架构完全不同的微架构，可能会为缩减工作负载产生与参考工作负载不同的性能数字。 |
| One concrete example to illustrate this pitfall is value prediction [127], which is a microarchitectural technique that predicts and speculates on the outcome of instructions. Assume that an architect wants to evaluate the performance potential of value prediction, and assume that the reduced workload was selected based on a set program characteristics (for the PCA-based method) or microarchitecture configurations (for the PB-based method) that do not capture any notion of value locality and predictability. Then the reduced workload may potentially lead to a misleading conclusion, i.e., the value predictability may be different for the reduced workload than for the reference workload, for the simple reason that the notion of value locality and predictability was not taken into account during workload reduction. | 说明这个缺陷的一个具体例子是值预测[127]，这是一种微架构技术，可以预测并投机指令的结果。假设架构师想要评估值预测的性能潜力，并假设缩减工作负载是基于一组程序特征（对于基于PCA的方法）或微架构配置（对于基于PB的方法）选择的，这些特征没有捕获任何值的局部性和可预测性的概念。那么，缩减工作负载可能会导致一个具有误导性的结论，即，缩减工作负载与参考工作负载的值可预测性可能不同。原因很简单，在工作负载缩减期间没有考虑到值的局部性和可预测性的概念。 |
| This pitfall may not be a major issue in practice though because microarchitectures typically do not change radically from one generation to the next. The transition from one generation to the next is typically smooth, with small improvements accumulating to significant performance differences over time. Nevertheless, architects should be aware of this pitfall and should, therefore, not only focus on the reduced workload during performance evaluation. It is important to evaluate the representativeness of the reduced workload with respect to the reference workload from time to time and revisit the reduced workload if needed. In spite of having to revisit the representativeness of the reduced workload, substantial simulation time savings will still be realized because (most of) the experiments are done using a limited number of benchmarks and not the entire reference workload. | 这个陷阱在实践中可能不是一个主要问题，因为微架构通常不会从根本上发生代际改变。从一代到下一代的过渡通常是平稳的，随着时间的推移，小的改进积累成显著的性能差异。尽管如此，架构师应该意识到这个陷阱，因此，在性能评估期间不应该只关注缩减工作负载。重要的是要不时评估缩减工作负载相对于参考工作负载的代表性，并在必要时重新审视缩减工作负载。尽管必须重新考虑缩减工作负载的代表性，但仍然可以实现大量节省仿真时间的目的，因为（大多数）实验是使用有限数量的基准而不是整个参考工作负载完成的。 |

|  |  |
| --- | --- |
| CHAPTER 4 Analytical Performance Modeling | 第4章 分析性能建模 |
| Analytical performance modeling is an important performance evaluation method that has gained increased interest over the past few years. In comparison to the prevalent approach of simulation (which we will discuss in subsequent chapters), analytical modeling may be less accurate, yet it is multiple orders of magnitude faster than simulation: a performance estimate is obtained almost instantaneously — it is a matter of computing a limited number of formulas which is done in seconds or minutes at most. Simulation, on the other hand, can easily take hours, days, or even weeks. | 分析性能建模是一种重要的性能评估方法，在过去几年中得到了越来越多的关注。与流行的仿真方法（我们将在后续章节中讨论）相比，分析建模可能不那么准确，但它比仿真快多个数量级：性能估计几乎是瞬间得到的——只是计算有限数量的公式，最多在几秒或几分钟内完成。另一方面，仿真可能需要数小时、数天甚至数周时间。 |
| Because of its great speed advantage, analytical modeling enables exploring large design spaces very quickly, which makes it a useful tool in early stages of the design cycle and even allows for exploring very large design spaces that are intractable to explore through simulation. In other words, analytical modeling can be used to quickly identify a region of interest that is later explored in more detail through simulation. One example that illustrates the power of analytical modeling for exploring large design spaces is a study done by Lee and Brooks [120], which explores the potential of adaptive miroarchitectures while varying both the adaptibility of the microarchitecture and the time granularity for adaptation — this is a study that would have been infeasible through detailed cycle-accurate simulation. | 由于其巨大的速度优势，分析建模能够非常快速地探索巨大的设计空间，这使得它在设计周期的早期阶段成为一个有用的工具，甚至允许探索通过仿真难以探索的非常大的设计空间。换句话说，分析建模可以用于快速确定感兴趣的区域，稍后通过仿真进行更详细的探索。Lee和Brooks[120]所做的一项研究表明了分析模型在探索大型设计空间方面的能量，该研究同时改变了微架构的自适应性和自适应的时间粒度，探索了自适应微架构的潜力——这是一项通过详细的精确周期仿真无法实现的研究。 |
| In addition, analytical modeling provides more fundamental insight. Although simulation provides valuable insight as well, it requires many simulations to understand performance sensitivity to design parameters. In contrast, the sensitivity may be apparent from the formula itself in analytical modeling. As an example, Hill and Marty extend Amdahl’s law towards multicore processors [82]. They augment Amdahl’s law with a simple hardware cost model, and they explore the impact of symmetric (homogeneous), asymmetric (heterogeneous) and dynamic multicore processing. In spite of its simplicity, it provides fundamental insight and reveals various important consequences for the multicore era. | 此外，分析模型提供了更基本的见解。虽然仿真也提供了有价值的见解，但它需要许多仿真来理解性能对设计参数的敏感性。相反，在分析建模中，从公式本身可以看出灵敏度。例如，Hill和Marty将Amdahl定律扩展到多核处理器[82]。他们用一个简单的硬件成本模型增强了Amdahl的定律，并探索了对称（同构）、不对称（异构）和动态多核处理的影响。尽管简单，但它提供了基本的见解，揭示了多核时代的各种重要结果。 |
| 4.1 EMPIRICAL VERSUS MECHANISTIC MODELING | 4.1 经验建模和机制建模 |
| In this chapter, we classify recent work in analytical performance modeling in three major categories, empirical modeling,mechanistic modeling and hybrid empirical-mechanistic modeling.Mechanistic modeling builds a performance model based on first principles, i.e., the performance model is built in a bottom-up fashion starting from a basic understanding of the mechanics of the underlying system. | 在本章中，我们将分析性能建模的近期工作分为三大类，经验建模、机制建模和混合经验-机制建模。机制建模基于基础原理建立性能模型，即性能模型从对底层系统的机制的基本了解开始，以自下而上的方式建立。 |
| Mechanistic modeling can be viewed of as ‘white-box’ performance modeling, and its key feature is to provide fundamental insight and, potentially, generate new knowledge. Empirical modeling, on the other hand, builds a performance model through a ‘black-box’ approach. Empirical modeling typically leverages statistical inference and machine learning techniques such as regression modeling or neural networks to automatically learn a performance model from training data. While inferring a performance model is easier through empirical modeling because of the complexity of the underlying system, it typically provides less insight than mechanistic modeling. Hybrid mechanistic-empirical modeling occupies the middle ground between mechanistic and empirical modeling, and it could be viewed of as ‘gray-box’ modeling. Hybrid mechanistic-empirical modeling starts from a generic performance formula that is derived from insights in the underlying system; however, this formula includes a number of unknown parameters. These unknown parameters are then inferred through fitting (e.g., regression), similarly to empirical modeling. The motivation for hybrid mechanisticempirical modeling is that it provides insight (which it inherits from mechanistic modeling) while easing the construction of the performance model (which it inherits from empirical modeling). | 机制建模可以被视为“白盒”性能模型，其关键特征是提供基本的洞察力，并可能产生新的知识。另一方面，经验建模通过“黑盒”方法构建性能模型。经验建模通常利用统计推断和机器学习技术，如回归模型或神经网络，从训练数据中自动学习性能模型。虽然由于底层系统的复杂性，通过经验模型推断性能模型更容易，但它提供的洞察力通常不如机制模型。机械-经验混合模型占据了机械和经验模型之间的中间地带，它可以被视为“灰盒”模型。混合机制-经验建模从一个通用的性能公式出发，该公式来源于对底层系统的洞察；然而，这个公式包含了许多未知参数。然后通过拟合（如回归）推断这些未知参数，类似于经验模型。混合机制-经验建模的动机是，它提供了洞察能力（继承自机制建模），同时减轻了性能模型的构建开销（继承自经验建模）。 |
| Although we make a distinction between empirical and mechanistic modeling, there is no purely empirical or mechanistic model [145]. A mechanistic model always includes some form of empiricism, for example, in the modeling assumptions and approximations. Likewise, an empirical model always includes a mechanistic component, for example, in the list of inputs to the model — the list of model inputs is constructed based on some understanding of the underlying system. As a result, the distinction between empirical and mechanistic models is relative, and we base our classification on the predominance of the empirical versus mechanistic component in the model. We now describe the three modeling approaches in more detail. | 虽然我们区分了经验建模和机制建模，但并没有纯粹的经验模型或机制模型[145]。机制模型总是包含某种形式的经验主义，例如，建模假设和近似。同样地，一个经验模型总是包含机制性的组成部分，例如，模型的输入列表中——模型输入列表是基于对底层系统的一些理解而构建的。因此，经验模型和机制模型之间的区别是相对的，我们的分类基于模型占主导的经验和机制成分。我们现在更详细地描述这三种建模方法。 |
| 4.2 EMPIRICAL MODELING | 4.2 经验建模 |
| While empirical modeling allows users to embed domain knowledge into the model, effective models might still be constructed without such prior knowledge. This flexibility might explain the recent popularity of this modeling technique. Different research groups have proposed different approaches to empirical modeling, which we revisit here. | 虽然经验建模允许用户将领域知识嵌入到模型中，但在没有这些先验知识的情况下仍然可以构建有效的模型。这种灵活性可能解释了最近这种建模技术的流行。不同的研究小组提出了不同的经验建模方法，我们在这里重新审视。 |
| 4.2.1 LINEAR REGRESSION | 4.2.1 线性回归 |
| Linear regression is a widely used empirical modeling approach which relates a response variable to a number of input parameters. Joseph et al. [97] apply this technique to processor performance modeling and build linear regression models that relate micro-architectural parameters (along with some of their interactions) to overall processor performance. Joseph et al. only use linear regression to test design parameters for significance, i.e., they do not use linear regression for predictive modeling. In that sense, linear regression is similar to the PCA and Plackett-Burman approaches discussed in the previous chapter. The more advanced regression techniques, non-linear and spline-based regression, which we discuss next, have been applied successfully for predictive modeling. (And because the more advanced regression methods extend upon linear regression, we describe linear regression here.) | 线性回归是一种广泛使用的经验建模方法，它将一个响应变量与多个输入参数联系起来。Joseph等人[97]将这种技术应用于处理器性能建模，并建立将微架构参数（以及它们的一些交互）与总体处理器性能关联起来的线性回归模型。Joseph等人只使用线性回归检验设计参数的显著性，即他们没有使用线性回归进行预测建模。在这个意义上，线性回归类似于前一章讨论的PCA和Plackett-Burman方法。更先进的回归技术，非线性和样条回归，我们接下来讨论，已经成功地应用于预测建模。（因为更高级的回归方法是在线性回归的基础上扩展的，所以我们在这里描述线性回归。） |
| The simplest form of linear regression is | 最简单的线性回归方程是： |
|  | |
| with y the dependent variable (also called the response variable), xi the independent variables (also called the input variables),and ϵ the error term due to lack of fit.β0 is the intercept with the y-axis and the βi coefficients are the regression coefficients. The βi coefficients represent the expected change in the response variable y per unit of change in the input variable xi; in other words, a regression coefficient represents the significance of its respective input variable. A linear regression model could potentially relate performance (response variable) to a set of microarchitecture parameters (input variables); the latter could be processor width, pipeline depth, cache size, cache latency, etc. In other words, linear regression tries to find the best possible linear fit for a number of data points, as illustrated in Figure 4.1. | 其中为因变量（也称响应变量），为自变量（也称输入变量），为由于欠拟合而产生的误差项。为与y轴的截距，为回归系数。表示每单位输入变量的变化中，响应变量的预期变化；也就是说，回归系数代表了其各自对应输入变量的显著性。线性回归模型可以潜在地将性能（响应变量）与一组微架构参数（输入变量）联系起来；后者可以是处理器宽度、流水线深度、缓存大小、缓存延迟等。换句话说，线性回归试图为许多数据点找到可能的最佳线性拟合，如图4.1所示。 |
|  | |
| Figure 4.1: Linear regression. | 图4.1：线性回归。 |
| This simple linear regression model assumes that the input variables are independent of each other, i.e., the effect of variable xi on the response y does not depend on the value of xj,j = i. In many cases, this is not an accurate assumption, especially in computer architecture. For example, the effect on performance of making the processor pipeline deeper depends on the configuration of the memory hierarchy. A more aggressive memory hierarchy reduces cache miss rates, which reduces average memory access times and increases pipelining advantages. Therefore, it is possible to consider interaction terms in the regression model: | 这个简单的线性回归模型假设输入变量彼此独立，即变量对响应的影响不依赖于的值。在很多情况下，这并不是一个准确的假设，特别是在计算机架构中。例如，加深处理器流水线对性能的影响取决于内存层次结构的配置。更激进的内存层次结构降低了缓存未命中率，从而减少了平均内存访问时间并增加了流水线优势。因此，可以在回归模型中考虑交互项： |
|  | |
| This particular regression model only includes, so called, two-factor interactions, i.e., pairwise interactions between two input variables only; however, this can be trivially extended towards higher order interactions. | 这个特定的回归模型只包括所谓的双变量交互作用，即两个输入变量之间的两两交互作用；然而，这可以扩展到更高阶的相互作用。 |
| The goal for applying regression modeling to performance modeling is to understand the effect of the important microarchitecture parameters and their interactions on overall processor performance. Joseph et al. [97] present such an approach and select a number of microarchitecture parameters such as pipeline depth, processor width, reorder buffer size, cache sizes and latencies, etc., along with a selected number of interactions. They then run a number of simulations while varying the microarchitecture parameters and fit the simulation results to the regression model. The method of least squares is commonly used to find the best fitting model that minimizes the sum of squared deviations between the predicted response variable (through the model) and observed response variable (through simulation). The fitting is done such that the error term is as small as possible. The end result is an estimate for each of the regression coefficients. The magnitude and sign of the regression coefficients represent the relative importance and impact of the respective microarchitecture parameters on overall performance. | 将回归模型应用于性能模型的目标是了解重要的微架构参数及其交互对整体处理器性能的影响。Joseph等人[97]提出了这样一种方法，选择了一些微架构参数，如流水线深度、处理器宽度、重排序缓存大小、缓存大小和延迟等，以及一些选定的交互作用。然后，他们运行了大量的仿真，同时改变微架构参数，并将仿真结果与回归模型拟合。最小二乘法通常用于寻找使预测响应变量（通过模型）与观测响应变量（通过仿真）之间的平方和偏差最小的最佳拟合模型。拟合误差项尽可能小。最终结果是每个回归系数的估计。回归系数的大小和符号代表了各自微架构参数对整体性能的相对重要性和影响。 |
| There are a number of issues one has to deal with when building a regression model. For one, the architect needs to select the set of microarchitecture input parameters, which has an impact on both accuracy and the number of simulations needed to build the model. Insignificant parameters only increase model building time without contributing to accuracy. On the other hand, crucial parameters that are not included in the model will likely lead to an inaccurate model. Second, the value ranges that need to be set for each variable during model construction depends on the purpose of the experiment. Typically, for design space exploration purposes, it is advised to take values that are slightly outside the expected parameter range — this is to cover the design space well [196]. | 在构建回归模型时，有许多问题需要处理。首先，架构师需要选择一组微架构输入参数，这对构建模型所需的准确性和仿真次数都有影响。不重要的参数只会增加仿真时间，而不会影响精度。另一方面，没有包含在模型中的关键参数可能会导致模型的不准确。其次，在构建模型时，每个变量需要设置的取值范围取决于实验的目的。通常，为了探索设计空间的目的，建议取稍微超出预期参数范围的值——这是为了很好地覆盖设计空间[196]。 |
| Table 4.1 shows the most significant variables and interactions obtained in one of the experiments done by Joseph et al.They consider six microarchitecture parameters and their interactions as the input variables, and they consider IPC as their response performance metric. Some parameters and interactions are clearly more significant than others, i.e., their respective regression coefficients have a large magnitude: pipeline depth and reorder buffer size as well as its interaction are significant, much more significant than L2 cache size. As illustrated in this case study, the regression coefficients can be both positive and negative, which complicates gaining insight. In particular, Table 4.1 suggests that IPC decreases with increasing reorder buffer size because the regression coefficient is negative. This obviously makes no sense. The negative regression coefficient is compensated for by the positive interaction terms between reorder buffer size and pipeline depth, and reorder buffer size and issue buffer size. In other words, increasing the reorder buffer size will increase the interaction terms more than the individual variable so that IPC would indeed increase with reorder buffer size. | 表4.1显示了Joseph等人在其中一项实验中获得的最重要的变量和交互作用。他们将六个微架构参数及其交互作用作为输入变量，并将IPC作为他们的响应性能度量。一些参数和相互作用显然比其他参数更重要，即它们各自的回归系数有很大的幅度：流水线深度和重排序缓冲区大小及其相互作用都很重要，比L2缓存大小重要得多。正如在这个案例研究中所说明的，回归系数可以是正的和负的，这使获得洞察复杂化。特别地，表4.1表明，IPC随着重排序缓冲区大小的增加而降低，因为回归系数是负的。这显然毫无意义。重新排序缓冲区大小与流水线深度的正交互项、以及重新排序缓冲区大小与发射缓冲区大小之间的正交互项补偿了负回归系数。换句话说，增加重排序缓冲区的大小将使交互项的增加超过单个变量的增加，因此IPC确实会随着重排序缓冲区的大小而增加。 |
| Table 4.1: Example illustrating the output of a linear regression experiment done by Joseph et al. [97]. | 表4.1： Joseph等人[97]所做的线性回归试验 |
| |  |  | | --- | --- | | Interrupt | 1.230 | | Pipeline depth | -0.566 | | Reorder buffer size | -0.480 | | Pipeline depth x reorder buffer size | 0.378 | | Issue queue size | -0.347 | | Reorder buffer size x issue queue size | 0.289 | | Pipeline depth x issue queue size | 0.274 | | Pipeline depth x reorder buffer size x issue queue size | -0.219 | | |
| |  |  | | --- | --- | | 中断 | 1.230 | | 流水线深度 | -0.566 | | 重排序缓存大小 | -0.480 | | 流水线深度 x 重排序缓存大小 | 0.378 | | 发射队列大小 | -0.347 | | 重排序缓存大小x发射队列大小 | 0.289 | | 流水线深度x发射队列大小 | 0.274 | | 流水线深度 x重排序缓存大小x 发射队列大小 | -0.219 | | |
| 4.2.2 NON-LINEAR AND SPLINE-BASED REGRESSION | 4.2.2 非线性回归和样条回归 |
| Basic linear regression as described in the previous section assumes that the response variable behaves linearly with its input variables.This assumption is often too restrictive.There exist several techniques for capturing non-linearity. | 上一节所述的基本线性回归假设响应变量与其输入变量的行为呈线性关系。这种假设往往太过局限。有几种技术可以捕捉非线性。 |
| The most simple approach is to transform the input variables or response variable or both. Typical transformations are square root, logarithmic, power, etc. The idea is that such transformations make the response more linear and thus easier to fit. For example, instruction throughput (IPC) is known to relate to reorder buffer size following an approximate square root relation [138; 166], so it makes sense to take the square root of the reorder buffer variable in an IPC model in order to have a better fit. The limitation is that the transformation is applied to an input variable’s entire range, and thus a good fit in one region may unduly affect the fit in another region. | 最简单的方法是转换输入变量，或响应变量，或两者都转换。典型的变换有平方根、对数、幂等。其思想是这样的转换使响应更线性，从而更容易拟合。例如，我们知道，指令吞吐量（IPC）与重新排序缓冲区的大小有一个近似的平方根关系[138;166]，所以在IPC模型中对重排序缓存变量取平方根是有意义的，以便有更好的拟合。其局限性在于转换应用于输入变量的整个范围，因此在一个区域的良好拟合可能会过度影响另一个区域的拟合。 |
| Lee and Brooks [119] advocate spline-based regression modeling in order to capture nonlinearity. A spline function is a piecewise polynomial used in curve fitting. A spline function is partitioned in a number of intervals with different continuous polynomials. The endpoints for the polynomials are called knots. Denoting the knots’ x-values as xi and their y-values as yi, the spline is then defined as | Lee和Brooks[119]提倡用样条回归建模来捕捉非线性。样条函数是一种用于曲线拟合的分段多项式。样条函数被划分为多个具有不同连续多项式的区间。多项式的端点称为结点。表示结点的x值为，它们的y值为，然后样条被定义为 |
|  | |
| with each Si(x) a polynomial. Higher-order polynomials typically lead to better fits. Lee and Brooks use cubic splines which have the nice property that the resulting curve is smooth because the first and second derivatives of the function are forced to agree at the knots. Restricted cubic splines constrain the function to be linear in the tails; see Figure 4.2 for an example restricted cubic spline. Lee and Brooks successfully leverage spline-based regression modeling to build multiprocessor performance models [122], characterize the roughness of the architecture design space [123], and explore the huge design space of adaptive processors [120]. | 每个都是一个多项式。高阶多项式通常会产生更好的拟合。Lee和Brooks使用三次样条曲线。它有一个很好的特性，即得到的曲线是平滑的，因为函数的一阶导数和二阶导数在结点处是一致的。限制性三次样条将函数约束为尾部线性；图4.2给出了一个限制性三次样条的示例。Lee和Brooks成功地利用基于样条的回归建模建立了多处理器性能模型[122]，刻画了架构设计空间的粗糙性[123]，并探索了自适应处理器的巨大设计空间[120]。 |
|  | |
| Figure 4.2: Spline-based regression. | 图4.2：样条回归 |
| 4.2.3 NEURAL NETWORKS | 4.2.3 神经网络 |
| Artificial neural networks are an alternative approach to building empirical models. Neural networks are machine learning models that automatically learn to predict (a) target(s) from a set of inputs. The target could be performance and/or power or any other metric of interest, and the inputs could be microarchitecture parameters. Neural networks could be viewed of as a generalized nonlinear regression model. Several groups have explored the idea of using neural networks to build performance models, see for example Ipek et al. [89], Dubach et al. [41] and Joseph et al. [98]. Lee et al. [121] compare spline-based regression modeling against artificial neural networks and conclude that both approaches are equally accurate; regression modeling provides better statistical understanding while neural networks offer greater automation. | 人工神经网络是建立经验模型的另一种方法。神经网络是一种机器学习模型，它可以从一组输入中自动学习预测目标。目标可以是性能和/或功率或任何其他感兴趣的指标，输入可以是微架构参数。神经网络可以看作是一种广义的非线性回归模型。有几个小组已经探索了使用神经网络建立性能模型的想法，例如Ipek等人[89]，Dubach等人[41]和Joseph等人[98]。Lee等人[121]比较了基于样条回归的模型与人工神经网络，并得出结论，两种方法的准确性相同；回归模型提供了更好的统计理解，而神经网络提供了更好的自动化。 |
| Figure 4.3(a) shows the basic organization of a fully connected feed-forward neural network. The network consists of one input layer and one output layer, and one or more hidden layers. The input layer collects the inputs to the model, and the output layer provides the model’s predictions. Data flows from the inputs to the outputs. Each node is connected to all nodes from the previous layer. Each edge has a weight and each node in the hidden and output layers computes the weighted sum of its inputs. The nodes in the hidden layer apply the weighted sum of its inputs to a so called activation function, see also Figure 4.3(b). A commonly used activation function is the sigmoid function, which is a mathematical function having an ‘S’ shape with two horizontal asymptotes. | 图4.3(a)展示了全连接前馈神经网络的基本组织。该网络由一个输入层和一个输出层，以及一个或多个隐藏层组成。输入层收集模型的输入，输出层提供模型的预测。数据从输入流到输出。每个节点都连接到上一层的所有节点。每条边都有一个权值，隐藏层和输出层中的每个节点都计算其输入的加权和。隐藏层中的节点将其输入的加权和应用于一个所谓的激活函数，参见图4.3(b)。一个常用的激活函数是sigmoid函数，它是一个具有两条水平渐近线的S形数学函数。 |
|  | |
| Figure 4.3: Neural networks: (a) architecture of a fully connected feed-forward network, and (b) architecture for an individual node. | 图4.3：神经网络：(a) 前连接前馈网络的结构，和(b) 一个单独节点的结构 |
| Training an artificial neural network basically boils down to a search problem that aims at finding the weights, such that the error between the network’s predictions and the corresponding measurements is minimized. Training the neural network is fairly similar to inferring a regression model: the network’s edge weights are adjusted to minimize the squared error between the simulation results and the model predictions. During training, examples are repeatedly presented at the inputs, differences between network outputs and target values are calculated, and weights are updated by taking a small step in the direction of steepest decrease in error. This is typically done through a well-known procedure called backpropagation. | 训练一个人工神经网络基本上可以归结为一个搜索问题，其目的是寻找使网络的预测值和相应测量值的误差最小化的权值。训练神经网络与推断回归模型相当相似：调整网络的边缘权值，以最小化仿真结果与模型预测之间的平方误差。在训练过程中，在输入处反复展示示例，计算网络输出与目标值的差值，并向误差下降幅度最大的方向迈出一小步更新权值。这通常通过众所周知的反向传播过程来完成。 |
| A limitation for empirical modeling, both neural networks and regression modeling, is that it requires a number of simulations to infer the model. This number of simulations typically varies between a couple hundreds to a few thousands of simulations. Although this is time consuming to do, it is a one-time cost. Once the simulations are run and once the model is built, making performance predictions is done instantaneously. | 经验建模，无论是神经网络还是回归建模，都有一个限制，那就是它需要大量的仿真来推断模型。仿真的数量通常在几百到几千个仿真之间变化。虽然这样做很耗时，但这是一次性成本。一旦运行了仿真并建立了模型，就可以立即进行性能预测。 |
| 4.3 MECHANISTIC MODELING: INTERVAL MODELING | 4.3 机制模型：区间建模 |
| Mechanistic modeling takes a different approach: it starts from a basic understanding of the underlying system from which a performance model is then inferred. One could view mechanistic modeling as a bottom-up approach, in contrast to empirical modeling which is as a top-down approach. | 机制模型采用了不同的方法：它从对底层系统的基本理解出发，然后从底层系统推断出性能模型。人们可以将机制模型看作是一种自下而上的方法，与经验模型相反。经验模型是一种自上而下的方法。 |
| Building mechanistic models for early processors was simple. Measuring the instruction mix and adding a constant cost per instruction based on the instruction’s execution latency and memory access latency was sufficient to build an accurate model. In contrast, contemporary processors are much more complicated and implement various ways of latency hiding techniques (instruction-level parallelism, memory-level parallelism, speculative execution, etc.), which complicates mechanistic performance modeling. Interval analysis is a recently developed mechanistic model for contemporary superscalar out-of-order processors presented by Eyerman et al. [64].This section describes interval modeling in more detail. | 为早期处理器建立机制模型很简单。测量指令组合，并根据指令的执行延迟和内存访问延迟为每条指令添加一个固定的代价，这足以构建一个准确的模型。相比之下，当代处理器要复杂得多，并实现各种延迟隐藏技术（指令级并行、内存级并行、投机执行等），这使机制性能建模变得复杂。区间分析是最近发展起来现代超标量乱序处理器的的机制模型[64]，由Eyerman等人提出的。本节将更详细地描述区间建模。 |
| 4.3.1 INTERVAL MODEL FUNDAMENTALS | 4.3.1 区间模型基础 |
| Figure 4.4(a) illustrates the fundamentals of the model: under optimal conditions, i.e., in the absence of miss events, a balanced superscalar out-of-order processor sustains instructions per cycle (IPC) performance roughly equal to its dispatch width D — dispatch refers to the movement of instructions from the front-end pipeline into the reorder and issue buffers of a superscalar out-of-order processor. However, when a miss event occurs, the dispatching of useful instructions eventually stops. There is then a period when no useful instructions are dispatched, lasting until the miss event is resolved, and then instructions once again begin flowing. Miss events divide execution time into intervals, which begin and end at the points where instructions just begin dispatching following recovery from the preceding miss event. | 图4.4(a)说明了该模型的基本原理：在最优条件下，即在没有未命中事件的情况下，一个平衡的超标量乱序处理器维持的每周期指令数（IPC）大致等于它的分发宽度——分发是指指令从前端流水线移动到超标量乱序处理器的重排序和发布缓冲区。然而，当未命中事件发生时，有效指令的分发最终会停止。然后会有一个周期，没有有效的指令被分发，持续到未命中事件被解决，然后指令再次开始流动。未命中事件将执行时间划分为多个时间区间，这些时间区间的起始点和结束点是指令刚刚开始分发的点，此时指令刚刚从前面的未命中事件中恢复并开始分发。 |
| Each interval consists of two parts, as illustrated in Figure 4.4(b). The first part performs useful work in terms of dispatching instructions into the window: if there are N instructions in a given interval (interval length of N) then it will take N/D cycles to dispatch them into the window. The second part of the interval is the penalty part and is dependent on the type of miss event.The exact mechanisms which cause the processor to stop and re-start dispatching instructions into the window, and the timing with respect to the occurrence of the miss event are dependent on the type of miss event, so each type of miss event must be analyzed separately. | 每个区间由两部分组成，如图4.4(b)所示。第一部分进行有效的工作，向窗口分发指令：如果在给定的时间区间内有条指令（区间长度为），那么将需要周期将它们分发到窗口。间隔的第二部分是代价部分，与未命中事件的类型有关。导致处理器停止和重新启动分发指令进入窗口的确切机制，以及未命中事件发生的时间取决于未命中事件的类型，所以每种类型的未命中事件必须单独分析。 |
|  | |
| Figure4.4: Interval behavior: (a) overall execution can be split up in intervals; (b) an interval consists of a base part where useful work gets done and a penalty part. | 图4.4：区间行为：(a) 整个执行可以切分为区间；(b) 一个区间包含一个基本部分（完成有效工作）和代价部分。 |
| 4.3.2 MODELING I-CACHE AND I-TLB MISSES | 4.3.2 建模指令缓存和指令TLB缺失 |
| L1 I-cache misses, L2 instruction cache misses and I-TLB misses are the easiest miss events to handle, see also Figure 4.5. At the beginning of the interval, instructions begin to fill the window at a rate equal to the maximum dispatch width. Then, at some point, an instruction cache miss occurs. Fetching stops while the cache miss is resolved, but the front-end pipeline is drained, i.e., the instructions already in the front-end pipeline are dispatched into the reorder and issue buffers, and then dispatch stops. After a delay for handling the I-cache miss, the pipeline begins to re-fill and dispatch is resumed. The front-end pipeline re-fill time is the same as the drain time — they offset each other. Hence, the penalty for an I-cache (and I-TLB) miss is its miss delay. | L1指令缓存缺失、L2指令缓存缺失和指令TLB 缺失是最容易处理的缺失事件，见图4.5。在区间开始时，指令开始以与最大分发宽度相等的速度填充窗口。然后，在某个时刻，发生指令缓存缺失。在解决缓存缺失的时候，取指停止，但是前端流水线被清空。也就是说，已经在前端流水线中的指令被分发到重排序和发射缓存中，然后分发停止。在处理指令缓存缺失的延迟之后，流水线开始重新填充并恢复指令分发。前端流水线重新填充的时间与排空时间相同——它们相互抵消。因此，指令缓存（和指令TLB）缺失的代价是它的缺失延迟。 |
|  | |
| Figure 4.5: Interval behavior of an I-cache/TLB miss. | 图4.5：指令缓存/TLB缺失的区间行为 |
| 4.3.3 MODELING BRANCH MISPREDICTIONS | 4.3.3 建模分支预测错误 |
| Figure 4.6 shows the timing for a branch misprediction interval. At the beginning of the interval, instructions are dispatched, until, at some point, the mispredicted branch is dispatched. Although wrong-path instructions continue to be dispatched (as displayed with the dashed line in Figure 4.6), dispatch of useful instructions stops at that point. Then, useful dispatch does not resume until the mispredicted branch is resolved, the pipeline is flushed, and the instruction front-end pipeline is re-filled with correct-path instructions. | 图4.6展示了分支预测错误的区间的时序。在区间的开始，指令被分发，直到在某一点，预测错误的分支被分发。尽管错误路径的指令会继续被分发（如图4.6中的虚线所示），但有效指令的分发会在此时停止。然后，在预测错误的分支被解决、流水线被清空、并且用正确路径指令重新填充前端流水线之期，有效分发不会恢复。 |
|  | |
| Figure 4.6: Interval behavior of a mispredicted branch. | 图4.6：分支预测错误的区间行为 |
| The overall performance penalty due to a branch misprediction thus equals the difference between the time the mispredicted branch enters the window and the time the first correct-path instruction enters the window following discovery of the misprediction. In other words, the overall performance penalty equals the branch resolution time, i.e., the time between the mispredicted branch entering the window and the branch being resolved, plus the front-end pipeline depth. Eyerman et al. [65] found that the mispredicted branch often is the last instruction to be executed; and hence,the branch resolution time can be approximated by the ‘window drain time’,or the number of cycles needed to empty a reorder buffer with a given number of instructions. For many programs, the branch resolution time is the main contributor to the overall branch misprediction penalty (not the pipeline re-fill time). And this branch resolution time is a function of the dependence structure of the instructions in the window, i.e., the longer the dependence chain and the execution latency of the instructions leading to the mispredicted branch, the longer the branch resolution time [65]. | 由于分支预测错误造成的总体性能损失等于预测错误的分支进入窗口的时间与发现预测错误后第一个正确路径指令进入窗口的时间之差。换句话说，总的性能损失等于分支解决的时间，即从预测错误的分支进入窗口到分支被解决之间的时间，加上前端流水线的深度。Eyerman等[65]发现，预测错误的分支往往是最后执行的指令；因此，分支解析时间可以用“窗口耗尽时间”来近似，或者用给定数量的指令清空重排序缓存的周期数。对于许多程序来说，分支解析时间是导致整个分支预测错误损失的主要原因（而不是流水线重新填充时间）。而这个分支解析时间是窗口中指令依赖结构的函数，即导致错误预测分支的指令依赖链越长，执行延迟越长，分支解析时间就越长[65]。 |
| 4.3.4 MODELING SHORT BACK-END MISS EVENTS USING LITTLE’S LAW | 4.3.4 利用LITTLE定理建模短的后端缺失事件 |
| An L1 D-cache miss is considered to be a ‘short’ back-end miss event and is modeled as if it is an instruction that is serviced by a long-latency functional unit, similar to a multiply or a divide. In other words, it is assumed that the miss latency can be hidden by out-of-order execution, and this assumption is incorporated into the model’s definition of a balanced processor design. In particular, the ILP model for balanced processor designs includes L1 D-cache miss latencies as part of the average instruction latency when balancing reorder buffer size and issue width in the absence of miss events.The ILP model is based on the notion of a window (of a size equal to the reorder buffer size) that slides across the dynamic instruction stream, see Figure 4.7. This sliding window computes the critical path length or the longest data dependence chain of instructions (including their execution latencies) in the window. Intuitively, the window cannot slide any faster than the processor can issue the instructions belonging to the critical path. | L1数据缓存缺失被认为是一个“短”的后端缺失事件，并被建模为一个由长延迟功能单元服务的指令，类似于乘法或除法。换句话说，我们假设缺失延迟可以通过乱序执行来隐藏，这个假设被纳入了平衡处理器的模型的设计定义中。特别是，在没有缺失事件的情况下，当重排序缓存大小和发射宽度平衡时，平衡处理器设计的ILP模型将L1数据缓存缺失延迟作为平均指令延迟的一部分。ILP模型基于在动态指令流上滑动的窗口（大小等于重排序缓存）的概念，参见图4.7。此滑动窗口计算窗口中的关键路径长度或最长的数据依赖指令链（包括它们的执行延迟）。直观地说，窗口滑动的速度不能超过处理器发射属于关键路径的指令的速度。 |
|  | |
| Figure 4.7: ILP model: window cannot slide any faster than determined by the critical path. | 图4.7：ILP模型：窗口的滑动速度不能超过关键路径决定的速度。 |
| The ILP model is based on Little’s law, which states that the throughput through a system equals the number of elements in the system divided by the average time spent for each element in the system. When applied to the current context, Little’s law states that the IPC that can be achieved over a window of instructions equals the number of instructions in the reorder buffer (window) W divided by the average number of cycles an instruction spends in the reorder buffer (between dispatch and retirement): | ILP模型基于Little定律。该定律指出，通过一个系统的吞吐量等于系统中元素的数量除以系统中每个元素花费的平均时间。当应用于当前上下文时，利特尔定律表明，在一个指令窗口内可以实现的IPC等于重排序缓存（窗口）指令数量除以一条指令在重排序缓存中花费的平均周期数（在分发和退役之间）：: |
|  | |
| The total time an instruction spends in the reorder buffer depends on the instruction’s execution latency and the dependency chain leading to the instruction, i.e., the critical path determines the achievable instruction-level parallelism (ILP).This ILP model has important implications.Knowing the critical path length as a function of window size, enables computing the achievable steady-state IPC for each window size. Or, in other words, the ILP model states which reorder buffer size is needed in order to have a balanced design and achieve an IPC close to the designed processor width in the absence of miss events. | 一条指令在重排序缓存中花费的总时间取决于指令的执行延迟和通向该指令的依赖链，即关键路径决定了可实现的指令级并行度（ILP）。该ILP模型具有重要的意义。已知关键路径长度与窗口大小的函数关系，就可以计算出每个窗口大小可达到的稳态IPC。或者，换句话说，ILP模型说明需要的重排序缓存的大小，以便在没有缺失事件的情况下实现一个接近设计处理器宽度的IPC。 |
| The ILP model is only one example illustrating the utility of Little’s law — Little’s law is widely applicable in systems research and computer architecture. It can be applied as long as the three parameters (throughput, number of elements in the system, latency of each element) are longterm (steady-state) averages of a stable system. There are multiple examples of how one could use Little’s law in computer architecture. One such example relates to computing the number of physical registers needed in an out-of-order processor. Knowing the target IPC and the average time between acquiring and releasing a physical register,one can compute the required number of physical registers. Another example relates to computing the average latency of a packet in a network. Tracking the latency for each packet may be complex to implement in an FPGA-based simulator in an efficient way. However, Little’s law offers an easy solution: it suffices to count the number of packets in the network and the injection rate during steady-state, and compute the average packet latency using Little’s law. | ILP模型只是一个例子来说明Litle定律的作用——Little定律在系统研究和计算机体系结构中有广泛的应用。只要这三个参数（吞吐量、系统中元素的数量、每个元素的延迟）是稳定系统的长期（稳态）平均值，就可以应用。关于如何在计算机架构中使用Little定律，有很多例子。一个例子涉及到计算乱序处理器中所需物理寄存器的数量。知道了目标IPC以及获取和释放物理寄存器之间的平均时间，就可以计算所需的物理寄存器数量。另一个例子与计算网络中数据包的平均延迟有关。在基于FPGA的仿真器中，高效实现跟踪每个包的延迟可能是复杂的。然而，Little定律提供了一个简单的解决方案：在稳定阶段，统计网络中的数据包数量和注入速率，并使用Little定律计算平均数据包延迟是足够的。 |
| 4.3.5 MODELING LONG BACK-END MISS EVENTS | 4.3.4建模长的后端缺失事件 |
| When a long data cache miss occurs, i.e., from the L2 cache to main memory, the memory delay is typically quite large — on the order of hundreds of cycles. Similar behavior is observed for D-TLB misses. | 当长时间的数据缓存丢失发生时，即从L2缓存到主存，内存延迟通常是相当大的——数百个周期。对于数据TLB缺失也观察到了类似的行为。 |
| On an isolated long data cache miss due to a load, the reorder buffer fills because the load blocks the reorder buffer head [102], and then dispatch stalls, see Figure 4.8. After the miss data returns from memory, the load can be executed and committed, which unblocks the reorder buffer, and as a result, instruction dispatch resumes. The total long data cache miss penalty equals the time between the ROB fill, and the time data returns from memory. The penalty for an isolated long back-end miss thus equals the main memory access latency minus the number of cycles where useful instructions are dispatched under the long-latency miss. These useful instructions are dispatched between the time the long-latency load dispatches and the time the ROB blocks after the longlatency load reaches its head; this is the time it takes to fill the entire ROB minus the time it takes for the load to issue after it has been dispatched — this is the amount of useful work done underneath the memory access. Given that this is typically much smaller than the memory access latency, the penalty for an isolated miss is assumed to equal the memory access latency. | 因为load发生一个孤立的长的数据缓存缺失时，重排序缓存填充，因为load阻塞了重排序缓存的队头[102]，然后指令分发停止，见图4.8。当缺失的数据从内存中返回后，可以执行并提交load，这将解除重排序缓存的阻塞，从而恢复指令分发。长数据缓存缺失的总损失等于ROB填充和数据从内存返回之间的时间。因此，一个孤立的后端长时延的代价，等于主存访问延迟，减去在长时延下分发有用指令的周期数。这些有用指令在长时延load分发的时刻和长时延load到达队头后ROB阻塞的时刻之间分发;这也是填充整个ROB所花费的时间减去load在分发后到发射所花费的时间——这是在内存访问下完成的有用的工作量。由于这通常比内存访问延迟小得多，因此假定一次孤立的缺失的代价等于内存访问延迟。 |
|  | |
| Figure 4.8: Interval behavior of an isolated long-latency load. | 图4.8：孤立的长延迟load的区间行为 |
| For multiple long back-end misses that are independent of each other and that make it in the reorder buffer at the same time, the penalties overlap [31; 76; 102; 103] — this is referred to as memory-level parallelism (MLP).This is illustrated in Figure 4.9. After the first load receives its data and unblocks the ROB, S more instructions dispatch before the ROB blocks for the second load, and the time to do so, S/D, offsets an equal amount of the second load’s miss penalty.This generalizes to any number of overlapping misses, so the penalty for a burst of independent long-latency back-end misses equals the penalty for an isolated long-latency load. | 对于多个相互独立的长的后端缺失，并同时使其进入重排序缓存，代价交叠在一起[31;76;102]；这被称为内存级并行(MLP)。图4.9说明了这一点。在第一个load接收到它的数据并解除ROB阻塞后，在load第二次阻塞ROB之前分发了条指令，并且这样做的时间抵消了第二次load的缺失代价的等量。这可以推广到任意数量的重叠缺失中。因此，一系列独立的长延迟后端的缺失等于一个孤立的长延迟负载的损失。 |
|  | |
| Figure 4.9: Interval behavior of two independent overlapping long-latency loads. | 图4.9 两个相互独立的交叠的长延迟load的区间行为。 |
| 4.3.6 MISS EVENT OVERLAPS | 4.3.6 缺失事件交叠 |
| So far, we considered the various miss event types in isolation. However, the miss events may interact with each other.The interaction between front-end miss events (branch mispredictions and I-cache/TLB misses) is limited because they serially disrupt the flow of instructions and thus their penalties serialize. Long-latency back-end miss events interact frequently and have a large impact on overall performance as discussed in the previous section, but these can be modeled fairly easily by counting the number of independent long-latency back-end misses that occur within an instruction sequence less than or equal to the reorder buffer size.The interactions between front-end miss events and long-latency back-end miss events are more complex because front-end miss events can overlap back-end misses; however, these second-order effects do not occur often (at most 5% of the total run time according to the experiments done by Eyerman et al. [64]), which is why interval modeling simply ignores them. | 到目前为止，我们孤立地考虑了各种缺失事件类型。但是，缺失事件可能会相互影响。前端缺失事件（分支预测错误和指令缓存/TLB 缺失）之间的影响是有限的，因为它们会串行地干扰指令流，因此它们的代价是串行的。如前一节所述，长延迟后端缺失事件频繁相互影响，对整体性能有很大影响，但是可以通过计算在小于或等于重排序缓存大小的指令序列中发生的独立长延迟后端缺失的数量，对这些事件进行建模。前端缺失事件和长延迟的后端缺失事件之间的相互影响更加复杂，因为前端缺失事件可能会重叠后端缺失事件；然而，这些二阶效应并不经常出现（根据Eyerman等人[64]所做的实验，它们最多只占总运行时间的5%），这就是区间建模忽略它们的原因。 |
| 4.3.7 THE OVERALL MODEL | 4.3.7 整体模型 |
| When put together, the model estimates the total execution time in the number of cycles C on a balanced processor as: | 当把它们放在一起时，该模型估计在平衡处理器上以周期数为单位的总执行时间为: |
|  | |
| The various parameters in the model are summarized in Table 4.2. The first line of the model computes the total number of cycles needed to dispatch all the intervals. Note there is an inherent dispatch inefficiency because the interval length Nk is not always an integer multiple of the processor dispatch width D, i.e., fewer instructions may be dispatched at the trailer of an interval than the designed dispatch width, simply because there are too few instructions to the next interval to fill the entire width of the processor’s front-end pipeline. The subsequent lines in Equation 4.5 represent I-cache misses, branch mispredictions and long back-end misses, respectively. (The TLB misses are not shown here to increase the formula’s readability.) The I-cache miss cycle component is the number of I-cache misses times their penalty. The branch misprediction cycle component equals the number of mispredicted branches times their penalty, the window drain time plus the front-end pipeline depth. Finally, the long back-end miss cycle component is computed as the number of non-overlapping misses times the memory access latency. | 模型中的各项参数汇总在表4.2中。模型的第一行计算分发所有间隔所需的周期总数。注意，由于区间长度并不总是处理器调度宽度的整数倍，因此存在固有的分发效率低下，也就是说，在一个间隔的尾部可能会比设计的分发宽度分发更少的指令，这仅仅是因为到下一个间隔的指令太少，无法填满处理器前端流水线的整个宽度。公式4.5的后面几行分别表示指令缓存缺失、分支预测错误和长后端错误。（这里没有显示TLB失误，以增加公式的可读性。）指令缓存缺失周期分量是指令缓存缺失次数乘以它们的代价。分支预测错误期分量等于预测分支错误的次数乘以它们的分发、窗口耗尽时间加上前端流水线深度。最后，将长后端缺失周期分量计算为非重叠缺失次数乘以内存访问延迟。 |
| Table 4.2: Parameters to the interval model. | 表4.2：区间模型的参数。 |
| |  |  | | --- | --- | | C | Total number of cycles | | Nk | Interval length for interval k | | D | Designed processor dispatch width | | miL1 and miL2 | Number of L1 and L2 I-cache misses | | ciL1 and ciL2 | L1 and L2 I-cache miss delay | | mbr | Number of mispredicted branches | | cdr | Window drain time | | cf e | Front-end pipeline depth | | m∗dL2(W ) | Number of non-overlapping L2 cache load misses for a given reorder buffer size W | | cL2 | L2 D-cache miss delay | | |
| |  |  | | --- | --- | | C | 总周期数 | | Nk | 区间k的长度 | | D | 设计的处理器调度宽度 | | miL1和miL2 | L1和L2指令缓存缺失数量 | | ciL1和ciL2 | L1 和L2指令缓存缺失延迟 | | mbr | 分支预测错误的数量 | | cdr | 窗口排空时间 | | cf e | 前端流水线深度 | | m∗dL2(W ) | 对于给定的重排序缓存大小W，非交叠的L2缓存load缺失的数量 | | cL2 | L2数据缓存缺失延迟 | | |
| 4.3.8 INPUT PARAMETERS TO THE MODEL | 4.3.8 模型的输入参数 |
| The model has two sets of program characteristics. The first set of program characteristics are related to a program’s locality behavior and include the miss rates and interval lengths for the various miss events. The second program characteristic relates to the branch resolution time which is approximated by the window drain time cdr. Note that these two sets of program characteristics are the only program characteristics needed by the model; all the other parameters are microarchitecturerelated. The locality metrics can be obtained through modeling or through specialized trace-driven simulation (i.e., cache, TLB and branch predictor simulation). The window drain time is estimated through the ILP model. | 该模型具有两组程序特征。第一组程序特征与程序的局部性行为有关，包括各种缺失事件的未命中率和间隔长度。第二组程序特性与分支解析时间有关，分支解析时间近似于窗口排空时间。注意，这两组程序特征是模型唯一需要的程序特征；所有其他参数都与微架构相关。局部性度量可以通过建模或通过专门的记录驱动仿真（即缓存、TLB和分支预测器仿真）获得。利用ILP模型估计窗口排空时间。 |
| 4.3.9 PREDECESSORS TO INTERVAL MODELING | 4.3.9 区间建模的前身 |
| A number of primary efforts led to the development of the interval model. Michaud et al. [138] build a mechanistic model of the instruction window and issue mechanism in out-of-order processors for gaining insight in the impact of instruction fetch bandwidth on overall performance. Karkhanis and Smith [103; 104] extend this simple mechanistic model to build a complete performance model that assumes sustained steady-state performance punctuated by miss events. Taha and Wills [180] propose a mechanistic model that breaks up the execution into so called macro blocks, separated by miss events. These earlier models focus on the issue stage of a superscalar out-of-order processor — issue refers to selecting instructions for execution on the functional units. The interval model as described here (see also [64] for more details) focuses on dispatch rather than issue, which makes the model more elegant.Also,the ILP model in [64] eliminates the need for extensive micro-architecture simulations during model construction, which the prior works needed to determine ‘steady-state’ performance in the absence of miss events. | 一系列先前的努力促成了区间模型的发展。Michaud等人[138]在乱序处理器中建立了指令窗口和发射机制的机制模型，以了解取指带宽对整体性能的影响。Karkhanis和Smith [103]扩展了这个简单的机制模型，建立了一个完整的性能模型，该模型假设持续的稳态性能被缺失事件打断。Taha和Wills[180]提出了一个机制模型，将执行分解为所谓的宏块，由缺失事件分隔。这些早期的模型集中在超标量乱序处理器的发射阶段——发射指的是选择在功能单元上执行的指令。这里描述的区间模型（更多细节请参见[64]）侧重于分发而不是发射，这使得模型更加优雅。此外，[64]中的ILP模型在模型构建过程中消除了对大量微架构仿真的需要，而之前的工作需要在没有缺失事件的情况下确定“稳态”性能。 |
| 4.3.10 FOLLOW-ON WORK | 4.3.10 后续工作 |
| Eyerman et al. [63] use the interval model to provide the necessary insights to develop a hardware performance counter architecture that can compute accurate CPI components and CPI stacks in superscalar out-of-order architectures. CPI stacks are stacked bars with the base cycle component typically shown at the bottom and the other CPI components stacked on top of it. CPI stacks are useful for guiding software and system optimization because it visualizes where the cycles have gone. Genbrugge et al. [73] replace the core-level cycle-accurate simulation models in a multicore simulator by the interval model; the interval model then estimates the progress for each instruction in the core’s pipeline. The key benefits are that the interval model is easier to implement than a cycleaccurate core simulator, and in addition, it runs substantially faster. Karkhanis and Smith [104] use the interval model to explore the processor design space automatically and identify processor configurations that represent Pareto-optimal design points with respect to performance, energy and chip area for a particular application or set of applications.Chen and Aamodt [27] extend the interval model by proposing ways to include hardware prefetching and account for a limited number of miss status handling registers (MSHRs). Hong and Kim [84] present a first-order model for GPUs which shares some commonalities with the interval model described here. | Eyerman等人[63]利用区间模型提供了必要的见解，开发了一种硬件性能计数器架构，可以在超标量乱序架构中精确计算CPI组件和CPI栈。CPI栈是堆叠的条形图，基本周期分量通常显示在底部，其他CPI分量则放在上面。CPI栈对于指导软件和系统优化非常有用，因为它可以可视化周期的去向。Genbrugge等人[73]在多核仿真器中用区间模型取代了处理器核级周期精确的仿真模型；然后，区间模型估计每条指令在处理器核流水线中的进度。关键的好处是，区间模型比周期精确的处理器核仿真器更容易实现，此外，它运行速度快得多。Karkhanis和Smith[104]使用区间模型来自动探索处理器设计空间，并确定针对特定应用程序或一组应用程序，在性能、能耗和芯片面积方面代表Pareto最优设计点的处理器配置。Chen和Aamodt[27]扩展了区间模型，提出了包括硬件预取和考虑有限数量的缺失状态处理寄存器（MSHR）的方法。Hong和Kim[84]提出了GPU的一阶模型，该模型与本文描述的区间模型有一些共同点。 |
| 4.3.11 MULTIPROCESSOR MODELING | 4.3.11 多处理器建模 |
| The interval model discussed so far focuses on modeling individual cores, and it does not target shared memory multiprocessor nor chip-multiprocessors. Much older work by Sorin et al. [175] proposed an analytical model for shared memory multiprocessors. This model assumes a black-box model for the individual processors — a processor is considered to generate requests to the memory system and intermittently block after a (dynamically changing) number of requests — and models the memory system through mean value analysis, which is a white-box model. (This model illustrates what we stated earlier, namely, a model is not purely mechanistic or empirical, and a mechanistic model may involve some form of empiricism.) The models views the memory system as a system consisting of queues (e.g., memory bus, DRAM modules, directories, network interfaces) and delay centers (e.g., switches in the interconnection network). Mean value analysis is a technique within queueing theory that estimates the expected queue lengths in a closed system of queues. The model then basically estimates the total time for each request to the memory system by adding the request’s mean residence time in each of the resources that it visits (e.g., processor, network interface at the sender, network, network interface at the receiver, bus at the receiver, memory and directory at the receiver side). | 到目前为止讨论的区间模型集中于对单个核心建模，它不针对共享内存多处理器或片上多处理器。Sorin等人[175]较早的工作提出了共享内存多处理器的分析模型。该模型假设单个处理器采用黑箱模型——一个处理器被认为生成对内存系统的请求，并在请求数量（动态变化）后间歇性阻塞——并通过平均值分析对内存系统建模，这是一个白箱模型。（这个模型说明了我们前面所说的，即一个模型不是纯粹的机制的或经验的，机制模型可能包含某种形式的经验主义。）该模型将存储系统视为一个由队列（如存储总线、DRAM模块、目录、网络接口）和延迟中心（如互连网络中的交换单元）组成的系统。均值分析是排队理论中的一种技术，用于估计封闭队列系统中期望的队列长度。然后，该模型通过添加请求在其访问的每个资源中的平均驻留时间（例如，处理器、发送端网络接口、网络、接收端网络接口、接收端总线、接收端内存和目录）来估计每个请求到内存系统的总时间。 |
| 4.4 HYBRID MECHANISTIC-EMPIRICAL MODELING | 4.4 混合机制-经验建模 |
| Hybrid mechanistic-empirical modeling combines the best of both worlds: it combines the insight from mechanistic modeling with the ease of model development from empirical modeling. Hybrid mechanistic-empirical modeling starts from a performance model that is based on some insight from the underlying system; however, there are a number of unknowns. These unknowns are parameters that are subsequently fit using training data, similar to what is done in empirical modeling. | 混合机制-经验建模结合了两个世界的优点:它结合了机制建模的洞察力和经验建模的模型开发的简易性。混合机制-经验建模从基于对底层系统的一些洞察的性能模型开始;然而，还有许多未知因素。这些未知数是随后使用训练数据拟合的参数，类似于在经验建模中所做的。 |
| Hartstein and Puzak [78] propose a hybrid mechanistic-empirical model for studying optimum pipeline depth. The model is parameterized with a number of parameters that are fit through detailed, cycle-accurate micro-architecture simulations. Hartstein and Puzak divide the total execution time in busy time TBZ and non-busy time TNBZ. The busy time refers to the time that the processor is doing useful work, i.e., instructions are issued; the non-busy time refers to the time that execution is stalled due to miss events. Hartstein and Puzak derive that the total execution time equals busy time plus non-busy time: | Hartstein和Puzak[78]提出了一种研究最佳流水线深度的混合机制-经验模型。通过详细的、周期精确的微结构仿真，对模型进行了参数化。Hartstein和Puzak将总执行时间划分为繁忙时间TBZ和非繁忙时间TNBZ。繁忙时间是指处理器正在做有用工作的时间，即指令发射的时间；非忙时间是指由于缺失事件而导致执行停滞的时间。Hartstein和Puzak推导出总执行时间等于繁忙时间加非繁忙时间: |
|  | |
| with Ntotal the total number of dynamically executed instructions and NH the number of hazards or miss events; to the latch overhead for a given technology, tp the total logic (and wire) delay of a processor pipeline, and p the number of pipeline stages. The α and γ parameters are empirically derived by fitting the model to data generated with detailed simulation. | 其中表示动态执行指令的总数，用表示竞争或缺失事件的数量；是给定工艺的锁存器开销，是处理器流水线的总逻辑（和线路）延迟，是流水线级数。将该模型与详细仿真得到的数据进行拟合，得到和参数。 |

|  |  |
| --- | --- |
| CHAPTER 5 Simulation | 第5章 仿真 |
| Simulation is the prevalent and de facto performance evaluation method in computer architecture. There are several reasons for its widespread use. Analytical models, in spite of the fact that they are extremely fast to evaluate and in spite of the deep insight that they provide,incur too much inaccuracy for many of the design decisions that an architect needs to make. One could argue that analytical modeling is valuable for making high-level design decisions and identifying regions of interest in the huge design space. However, small performance variations across design alternatives are harder to evaluate using analytical models. At the other end of the spectrum, hardware prototypes, although they are extremely accurate, are too time-consuming and costly to develop. | 仿真是计算机体系结构中普遍存在的、事实上的性能评估方法。仿真被广泛使用有几个原因。尽管分析模型的评估速度非常快，尽管它们提供了深刻的见解，但对于需要做出的许多设计决策的架构师来说，分析模型还是引入了太多的不精确性。有人可能会说，分析建模对于制定高级设计决策和在巨大的设计空间中确定感兴趣的区域是有价值的。然而，不同设计方案之间的小性能变化很难使用分析模型进行评估。另一方面，硬件原型虽然非常准确，但开发起来太耗时、太昂贵。 |
| A simulator is a software performance model of a processor architecture. The processor architecture that is modeled in the simulator is called the target architecture; running the simulator on a host architecture, i.e., a physical machine, then yields performance results. Simulation has the important advantage that development is relatively cheap compared to building hardware prototypes, and it is typically much more accurate than analytical models. Moreover, the simulator is flexible and easily parameterizable which allows for exploring the architecture design space — a property of primary importance to computer architects designing a microprocessor and researchers evaluating a novel idea. For example, evaluating the impact of cache size, latency, processor width, branch predictor configuration is easily done through parameterization, i.e., by changing some of the simulator’s parameters and running a simulation with a variety of benchmarks, one can evaluate what the impact is of an architecture feature. Simulation even enables evaluating (very) different architectures than the ones in use today. | 仿真器是处理器体系结构的软件性能模型。在仿真器中建模的处理器体系结构称为目标体系结构；在主机（即物理机器）上运行仿真器，然后产生性能结果。与构建硬件原型相比，仿真的开发成本相对较低，而且通常比分析模型更准确。此外，该仿真器是灵活的和容易参数化的，这允许探索架构设计空间——对于设计微处理器的计算机架构师和评估新想法的研究人员，这是主要的重要性质。例如，通过参数化，可以很容易地评估缓存大小、延迟、处理器宽度、分支预测器配置的影响，也就是说，通过更改仿真器的一些参数并使用各种基准运行仿真，可以评估参数架构特性的影响。仿真甚至能够评估与目前使用的架构（非常）不同的架构。 |
| 5.1 THE COMPUTER ARCHITECT’S TOOLBOX | 5.1 计算机架构师的工具箱 |
| There exist many flavors of simulation, each representing a different trade-off in accuracy, evaluation time, development time and coverage. Accuracy refers to the fidelity of the simulation model with respect to real hardware, i.e., how accurate is the simulation model compared to the real hardware that it models. Evaluation time refers to how long it takes to run a simulation. Development time refers to how long it takes to develop the simulator. Finally, coverage relates to what fraction of the design space a simulator can explore, e.g., a cache simulator can only be used to evaluate cache performance and not overall processor performance. | 有许多种不同的仿真，每一种都代表在准确性、评估时间、开发时间和覆盖率方面的不同权衡。精确度是指仿真模型相对于真实硬件的保真度，即仿真模型相对于它所建模的真实硬件的精确度。评估时间指的是运行仿真所需的时间。开发时间是指开发仿真器所需的时间。最后，覆盖率指的是仿真器能够探索的设计空间的百分比，例如，缓存仿真器只能用于评估缓存性能，而不能用于评估整体处理器性能。 |
| This simulation trade-off can be represented as a diamond, see Figure 5.1. (The trade-off is often represented as a triangle displaying accuracy, evaluation time and development time; a diamond illustrates a fourth crucial dimension.) Each simulation approach (or a modeling effort in general) can be characterized along these four dimensions. These dimensions are not independent of each other, and, in fact, are contradictory. For example, more faithful modeling with respect to real hardware by modeling additional features, i.e., increasing the simulator’s coverage, is going to increase accuracy, but it is also likely to increase the simulator’s development and evaluation time — the simulator will be more complex to build, and because of its increased complexity, it will also run slower, and thus simulation will take longer. In contrast, a simulator that only models a component of the entire system, e.g., a branch predictor or cache, has limited coverage with respect to the entire system; nevertheless, it is extremely valuable because its accuracy is good for the component under study while being relatively simple (limited development time) and fast (short evaluation time). | 这种仿真权衡可以用菱形表示，见图5.1。（权衡通常表示为一个三角形，显示精度、评估时间和开发时间；钻石说明了第四个关键维度。）每种仿真方法（或通常建模工作）都可以沿着这四个维度进行描述。这些方面不是彼此独立的，事实上，是相互矛盾的。例如，通过建模额外的特性，更忠实真实硬件的建模（比如增加仿真器的覆盖范围），将会提高精度，但也可能增加仿真器的开发和评估时间——仿真器的构建将更复杂。并且由于增加的复杂性，仿真器也将运行得更慢，因此仿真将花费更长的时间。相比之下，只对整个系统的一个组件（如分支预测器或缓存）建模的仿真器对整个系统的覆盖范围是有限的；尽管如此，它还是非常有价值的，因为它的准确性对于所研究的组件是好的，同时相对简单（开发时间有限）和快速（评估时间短）。 |
|  | |
| Figure 5.1: Simulation diamond illustrates the trade-offs in simulator accuracy, coverage, development time and evaluation time. | 图5.1：仿真菱形表示仿真器精确度、覆盖率、开发时间和评估时间的权衡。 |
| The following sections describe several commonly used simulation techniques in the computer architect’s toolbox, each representing a different trade-off in accuracy, coverage, development time and evaluation time. We will refer to Table 5.1 throughout the remainder of chapter; it summarizes the different simulation techniques along the four dimensions. | 以下部分描述了计算机架构师工具箱中几种常用的仿真技术，每种技术都代表了准确性、覆盖率、开发时间和评估时间方面的不同权衡。我们将在本章的其余部分参考表5.1。表格从四个维度总结了不同的仿真技术。 |
| Table 5.1: Comparing functional simulation, instrumentation, specialized cache and predictor simulation, full trace-driven simulation and full execution-driven simulation in terms of model development time, evaluation time, accuracy in predicting overall performance, and level of detail or coverage. | 表5.1：从仿真开发时间、评估时间、预测总体性能的准确性以及细节或覆盖率水平方面比较功能仿真、插桩工具、专用缓存和预测器仿真、完全记录驱动仿真和全执行驱动仿真。 |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | Develop time | Evaluation time | Accuracy | Coverage | | Function simulation | Excellent | Good | Poor | poor | | Instrumentation | Excellent | Very good | Poor | Poor | | Specialized cache and predictor simulation | Good | Good | Good | limited | | Full trace-driven simulation | poor | Poor | Very good | Excellent | | Full execution-driven simulation | Very poor | Very poor | excellent | Excellent | | |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 开发时间 | 评估时间 | 精确度 | 覆盖率 | | 功能仿真 | 非常好 | 好 | 不好 | 不好 | | 插桩 | 非常好 | 很好 | 不好 | 不好 | | 专门的缓存和预测器仿真 | 好 | 好 | 好 | 有限制 | | 完全记录驱动仿真 | 不好 | 不好 | 很好 | 非常好 | | 完全执行驱动仿真 | 非常不好 | 非常不好 | 非常好 | 非常好 | | |
| 5.2 FUNCTIONAL SIMULATION | 5.2 功能仿真 |
| Functional simulation models only the functional characteristics of an instruction set architecture (ISA) — it does not provide any timing estimates. That is, instructions are simulated one at a time, by taking input values and computing output values. Therefore, functional simulators are also called instruction-set emulators.These tools are typically most useful for validating the correctness of a design rather than evaluating its performance characteristics. Consequently, the accuracy and coverage with respect to performance and implementation detail are not applicable. However, development time is rated as excellent because a functional simulator is usually already present at the time a hardware development project is undertaken (unless the processor implements a brand new instruction set). Functional simulators have a very long lifetime that can span many development projects. Evaluation time is good because no microarchitecture features need to be modeled. Example functional simulator are SimpleScalar’s sim-safe and sim-fast [7]. | 功能仿真只仿真指令集体系结构（ISA）的功能特征，它不提供任何时序评估。也就是说，通过获取输入值和计算输出值，一条指令一条指令地仿真。因此，功能仿真器也被称为指令集模拟器。这些工具通常用于验证设计的正确性，而不是评估其性能特征。因此，对于性能和实现细节的准确性和覆盖率要求是不适用的。然而，开发时间被评为优秀，因为在硬件开发项目进行时，功能仿真器通常已经存在（除非处理器实现一个全新的指令集）。功能仿真器的生命周期非常长，可以跨越许多开发项目。因为不需要建模微架构特征，评估时间很好。典型的功能仿真器是SimpleScalar的sim-safe和sim-fast[7]。 |
| From a computer architect’s perspective, functional simulation is most useful because it can generate instruction and address traces. A trace is the functionally correct sequence of instructions and/or addresses that a benchmark program produces. These traces can be used as inputs to other simulation tools — so called (specialized) trace-driven simulators. | 从计算机架构师的角度来看，功能仿真最为有用，因为它可以生成指令和地址记录。记录是基准测试程序生成的功能正确的指令和/或地址序列。这些记录可以用作其他仿真工具的输入——所谓的（专门的）记录驱动的仿真器。 |
| 5.2.1 ALTERNATIVES | 5.2.1 不同实现方式 |
| An alternative to functional simulation is instrumentation, also called direct execution. Instrumentation takes a binary and adds code to it so that when running the instrumented binary on real hardware the property of interest is collected. For example, if the goal is to generate a trace of memory addresses, it suffices to instrument (i.e., add code to) each instruction referencing memory in the binary to compute and print the memory address; running the instrumented binary on native hardware then provides a trace of memory addresses. The key advantage of instrumentation compared to functional simulation is that it incurs less overhead. Instrumentation executes all the instructions natively on real hardware; in contrast, functional simulation emulates all the instructions and hence executes more host instructions per target instruction. There exist two flavors of instrumentation, static instrumentation, which instruments the binary statically, and dynamic instrumentation, which instruments the binary at run time. An example tool for static instrumentation is Atom [176], or EEL [114] (used in the Wisconsin Wind Tunnel II simulator [142]); Embra [191], Shade [34] and Pin [128] support dynamic instrumentation. A limitation of instrumentation compared to functional simulation is that the target ISA is typically the same as the host ISA, and thus an instrumentation framework is not easily portable. A dynamic binary translator that translates host ISA instructions to target ISA instructions can address this concern (as is done in the Shade framework); however, the simulator can only run on a machine that implements the target ISA. Zippy, a static instrumentation system at Digital in the late 1980s, reads in an Alpha binary, and adds ISA emulation and modeling code in MIPS code. | 功能仿真的另一种实现方式是插桩，也称为直接执行。插桩方法接受二进制文件并向其中添加代码，以便在实际硬件上运行插桩后的二进制文件时收集感兴趣的属性。例如，如果目的是生成内存地址的记录信息，那么在二进制文件中对每条引用内存的指令进行插桩（即添加代码）就足够了，以计算和打印内存地址。在本机硬件上运行插桩后的二进制文件，然后提供内存地址的记录。与功能仿真相比，插桩的主要优点是它产生的开销更少。插桩在真正的硬件本机上执行所有指令；相比之下，功能仿真器模拟所有的指令，因此每个目标指令执行了更多的主机指令。存有两种插桩方法：静态插桩方法（静态插桩二进制文件），和动态插桩方法（在运行时插桩二进制文件）。静态插桩的示例工具是Atom[176]，或EEL[114]（用于威斯康星Wind Tunnel II仿真器[142]）；Embra [191]， Shade[34]和Pin[128]支持动态插桩。与功能仿真相比，插桩的一个限制是目标ISA通常与主机ISA相同，因此插桩框架不容易移植。将主机ISA指令转换为目标ISA指令的动态二进制转换器可以解决这个问题（正如在Shade框架中所做的那样）。然而，插桩仿真器只能在实现目标ISA的机器上运行。Zippy是Digital公司在20世纪80年代后期开发的静态插桩系统，它读取Alpha二进制文件，并在代码中添加MIPS代码实现的ISA模拟和建模代码。 |
| An approach that combines the speed of instrumentation with the portability of functional simulation was proposed by Burtscher and Ganusov [21], see also Figure 5.2. They propose a functional simulator synthesizer which takes as input a binary executable as well as a file containing C definitions (code snippets) of all the supported instructions. The synthesizer then translates instructions in the binary to C statements. If desirable, the user can add simulation code to collect for example a trace of instructions or addresses. Compiling the synthesized C code generates the customized functional simulator. | Burtscher和Ganusov[21]提出了一种将插桩的速度与功能仿真的可移植性相结合的方法，见图5.2。他们提出了一个功能仿真器生成器，将一个二进制可执行文件和一个包含所有支持指令的C定义（代码片段）的文件作为输入。然后，合成器将二进制中的指令转换为C语句。如果需要，用户可以添加仿真代码来收集如指令或地址的序列。编译生成的C代码会生成定制的功能仿真器。 |
|  | |
| Figure 5.2: Functional simulator synthesizer proposed by Burtscher and Ganusov [21]. | 图5.2：Burtscher和Ganusov[21]等提出的功能仿真器生成器。 |
| 5.2.2 OPERATING SYSTEM EFFECTS | 5.2.2 操作系统的影响 |
| Functional simulation is often limited to user-level code only,i.e.,application and system library code, however, it does not simulate what happens upon an operating system call or interrupt. Nevertheless, in order to have a functionally correct execution, one needs to correctly emulate the system effects that affect the application code. A common approach is to ignore interrupts and emulate the effects of system calls [20]. Emulating system calls is typically done by manually identifying the input and output register and memory state to a system call and invoking the system call natively. This needs to be done for every system call, which is tedious and labor-intensive, especially if one wants to port a simulator to a new version of the operating system or a very different operating system. | 功能仿真通常仅限于用户级代码，即应用程序和系统库代码。但是，它不能仿真操作系统调用或中断时发生的情况。然而，为了实现功能上正确的执行，需要正确地模拟影响应用程序代码的系统影响。一种常见的方法是忽略中断并模拟系统调用的效果[20]。模拟系统调用通常是通过手动标识系统调用的输入和输出寄存器和内存状态，并调用本机的系统调用来完成的。每一次系统调用都需要这样做，这是一项繁琐而费力的工作，特别是如果想要将仿真器移植到操作系统的新版本或完全不同的操作系统。 |
| Narayanasamy et al. [144] present a technique that automatically captures the side effects of operating system interactions. An instrumented binary collects for each system call executed, interrupt and DMA transfer, how it changes register state and memory state. The memory state change is only recorded if the memory location is later read by a load operation. This is done as follows. The instrumented binary maintains a user-level copy of the application’s address space. A system effect (e.g., system call, interrupt, DMA transfer) will only affect the application’s address space, not the user-level copy. A write by the application updates both the application’s address space and the user-level copy. A read by the application verifies whether the data read in the application’s address space matches the data in the user-level copy; upon a mismatch, the system knows that the application’s state was changed by a system effect, and thus it knows that the load value in the application’s address space needs to be logged. These state changes are stored in a so called system effect log. During functional simulation, the system effect log is read when reaching a system call, and the state change which is stored in the log is replayed, i.e., the simulated register and memory state is modified to emulate the effect of the system call. Because this process does not depend on the semantics of system calls, it is completely automatic, which eases developing and porting user-level simulators. A side-effect of this technique is that it enables deterministic simulation, i.e., the system effects are the same across runs. While this facilitates comparing design alternatives, it also comes with its pitfall, as we will discuss in Section 5.6.2. | Narayanasamy等人[144]提出了一种自动捕获操作系统交互的副作用的技术。插桩的二进制程序收集执行的每个系统调用、中断和DMA传输，以及它如何改变寄存器状态和内存状态。只有在稍后由load操作读取的内存位置，才会记录内存状态变化。这一点是这样实现的。插桩的二进制文件维护应用程序地址空间的用户级副本。系统影响（例如，系统调用、中断、DMA传输）只会影响应用程序的地址空间，而不会影响用户级的副本。应用程序的写入同时更新应用程序的地址空间和用户级的副本。应用程序读取数据时，验证在应用程序地址空间中读取的数据是否与用户级副本中的数据相匹配；在不匹配的情况下，系统知道应用程序的状态是由于系统影响而改变的，因此它知道需要记录应用程序地址空间中的负载值。这些状态变化存储在所谓的系统影响日志中。在功能仿真过程中，当到达系统调用时读取系统影响日志，并回放存储在日志中的状态变化，即修改仿真的寄存器和内存状态，以模拟系统调用的效果。因为这个过程不依赖于系统调用的语义，所以它是完全自动的，这就简化了用户级仿真器的开发和移植。这种技术的一个副作用是，它使确定性仿真成为可能，也就是说，系统响应在不同次运行中是相同的。虽然这有助于比较不同的设计方案，但它也有其缺陷，我们将在第5.6.2节中讨论。 |
| 5.3 FULL-SYSTEM SIMULATION | 5.3 全系统仿真 |
| User-level simulation is sufficiently accurate for some workloads,e.g.,SPEC CPU benchmarks spend little time executing system-level code, hence limiting the simulation to user-level code is sufficient. However,for other workloads,e.g.,commercial workloads such as database servers,web servers,email servers, etc., simulating only user-level code is clearly insufficient because these workloads spend a considerable amount of time executing system-level code, and hence these workloads require fullsystem simulation. Also, the proliferation of multicore hardware has increased the importance of full-system simulation because multi-threaded workload performance is affected by OS scheduling decisions; not simulating the OS may lead to inaccurate performance numbers because it does not account for OS effects. | 用户级仿真对于某些工作负载是足够准确的，例如，SPEC CPU基准测试花费很少的时间执行系统级代码，因此将仿真限制为用户级代码就足够了。然而，对于其他工作负载，比如商业化负载（如数据库服务器、web服务器、电子邮件服务器等），只仿真用户级代码显然是不够的，因为这些工作负载需要相当多的时间来执行系统级代码，因此这些工作负载需要全系统仿真。此外，多核硬件的普及增加了全系统仿真的重要性，因为多线程工作负载性能受操作系统调度决策的影响；不仿真操作系统可能会导致不准确的性能数字，因为它没有考虑操作系统的影响。 |
| Full-system simulation refers to simulating an entire computer system such that complete software stacks can run on the simulator. The software stack includes application software as well as unmodified, commercial operating systems, so that the simulation includes I/O and OS activity next to processor and memory activity. In other words, a full-system simulation could be viewed of as a system emulator or a system virtual machine which appears to its user as virtual hardware, i.e., the user of the full-system simulator is given the illusion to run on real hardware. Well-known examples of full-system simulators are SimOS [167], Virtutech’s SimICs [132], AMD’s SimNow (x86 and x86-64) [12], M5 [16], Bochs (x86 and x86-64) [139], QEMU, Embra [191], and IBM’s Mambo (PowerPC) [19]. | 全系统仿真是指仿真整个计算机系统，使完整的软件堆栈可以在仿真器上运行。软件堆栈包括应用程序软件以及未经修改的商业操作系统，因此仿真包括I/O和OS活动，以及处理器和内存活动。换句话说，一个完整的系统仿真可以被看作是一个系统模拟器或一个系统虚拟机，它在用户看来是虚拟硬件，也就是说，完整系统仿真器的用户被赋予了在真实硬件上运行的错觉。全系统仿真器的知名例子有SimOS[167]、Virtutech的SimICs[132]、AMD的SimNow（x86和x86-64）[12]、M5[16]、Bochs（x86和x86-64）[139]、QEMU、Embra[191]和IBM的Mambo（PowerPC）[19]。 |
| The functionality provided by a full-system simulator is basically the same as for a userlevel functional simulator — both provide a trace of dynamically executed instructions — the only difference being that functional simulation simulates user-level code instructions only, whereas fullsystem simulation simulates both user-level and system-level code. Full-system simulation thus achieves greater coverage compared to user-level simulation; however, developing a full-system simulator is far from trivial. | 全系统仿真器提供的功能与用户级功能仿真器基本相同——都提供动态执行指令的记录——唯一的区别是功能仿真只仿真用户级代码指令，而全系统仿真同时仿真用户级和系统级代码。因此，与用户级仿真相比，全系统仿真的覆盖范围更大；然而，开发一个全系统仿真器绝非易事。 |
| 5.4 SPECIALIZED TRACE-DRIVEN SIMULATION | 5.4 专用的记录驱动的仿真器 |
| Specialized trace-driven simulation takes instruction and address traces — these traces may include user-level instructions only or may contain both user-level and system-level instructions — and simulates specific components, e.g., cache or branch predictor, of a target architecture in isolation. Performance is usually evaluated as a ‘miss rate’. A number of these tools are widely available, especially for cache simulation, see for example Dinero IV [44] from the University of Wisconsin– Madison. In addition, several proposals have been made for simulating multiple cache configurations in a single simulation run [35; 83; 135; 177]. While development time and evaluation time are both good, coverage is limited because only certain components of a processor are modeled. And, while the accuracy in terms of miss rate is quite good, overall processor performance accuracy is only roughly correlated with these miss rates because many other factors come into play. Nevertheless, specialized trace-driven simulation has its place in the toolbox because it provides a way to easily evaluate specific aspects of a processor. | 专门的记录驱动仿真采用指令和地址记录——这些跟踪可能只包含用户级指令，也可能同时包含用户级和系统级指令——并单独仿真目标架构的特定组件，例如缓存或分支预测器。性能通常利用“未命中率”来评估。这些工具中有许多是广泛可用的，特别是用于缓存仿真，例如威斯康星大学麦迪逊分校的Dinero IV[44]。此外，还有一些研究可以在一次仿真运行中仿真多个缓存配置[35;83;135;177]。虽然开发时间和评估时间都很好，但覆盖范围有限，因为只建模处理器的某些组件。而且，虽然在未命中率方面的准确性相当好，但总体处理器性能的准确性仅与这些未命中率大致相关，因为还有许多其他因素也在发挥作用。然而，专门的记录驱动仿真在工具箱中占有一席之地，因为它提供了一种轻松评估处理器特定方面的方法。 |
| 5.5 TRACE-DRIVEN SIMULATION | 5.5 记录驱动仿真 |
| Full trace-driven simulation, or trace-driven simulation for short, takes program instruction and address traces, and feeds the full benchmark trace into a detailed microarchitecture timing simulator. A trace-driven simulator separates the functional simulation from the timing simulation.This is often useful because the functional simulation needs to be performed only once, while the detailed timing simulation is performed many times when evaluating different microarchitectures. This separation reduces evaluation time somewhat. Overall, full trace-driven simulation requires a long development time and requires long simulation run times, but both accuracy and coverage are very good. | 全记录驱动仿真，或简称为记录驱动仿真，以程序指令和地址记录为驱动，并将完整的基准程序记录输入到详细的微架构时序仿真器中。记录驱动仿真器将功能仿真与时序仿真分离开来。这通常是有用的，因为功能仿真只需要执行一次，而详细的时序仿真在评估不同的微架构时要执行多次。这种分离在一定程度上减少了评估时间。总的来说，完整的记录驱动仿真需要很长的开发时间和很长的仿真运行时间，但是准确性和覆盖率都非常好。 |
| One obvious disadvantage of this approach is the need to store the trace files, which may be huge for contemporary benchmarks and computer programs with very long run times. Although disk space is cheap these days, trace compression can be used to address this concern; several approaches have been made to computer trace compression [22; 95]. | 这种方法的一个明显缺点是需要存储记录文件，这对于当前的基准测试和运行时间很长的计算机程序来说可能非常大。虽然磁盘空间现在很廉价，但记录压缩可以用来解决这个问题。计算机记录压缩已经有几种方法[22;95]。 |
| Another disadvantage for modern superscalar processors is that they predict branches and execute many instructions speculatively — speculatively executed instructions along mis-predicted paths are later nullified. These nullified instructions do not show up in a trace file generated via functional simulation, although they may affect cache and/or predictor contents [11; 143]. Hence, trace-driven simulation will not accurately model the effects along mis-predicted paths. | 现代超标量处理器的另一个缺点是它们预测分支并投机性地执行许多指令——投机地沿着预测错误的路径执行的指令后来会被作废。这些无效指令不会出现在通过功能仿真生成的记录文件中，尽管它们可能会影响缓存和/或预测器的上下文[11;143]。因此，记录驱动仿真并不能准确地仿真沿错误预测路径执行的影响。 |
| An additional limitation when simulating multi-threaded workloads is that trace-driven simulation cannot model the interaction between inter-thread ordering and the target microarchitecture. The reason is that the trace is fixed and imposes a particular ordering. However, the ordering and inter-thread dependences may be different across microarchitectures. For some studies, this effect may be limited; however, for other studies, it may be significant. All depends on the type of optimization and the workloads being evaluated. The key problem is that changes in some microarchitecture structure (e.g., branch predictor, cache, prefetcher, etc.) — this could even be small changes — may cause threads to acquire locks in a different order. This may lead to different conflict and contention behavior in shared resources (e.g., caches, memory, interconnection network, etc.), which, in its turn, may affect the inter-thread interleaving. Hence, even small changes in the microarchitecture can lead to (very) different performance numbers, and these changes may lead to big differences for particular benchmarks only; hence, a big change for one particular benchmark may not be representative of other workloads. Because trace-driven simulation simulates a single ordering, it cannot capture these effects. Moreover, a trace may reflect a particular ordering that may not even occur on the target microarchitecture. | 另一个限制是，仿真多线程工作负载时，记录驱动仿真不能仿真线程间保序和目标微架构之间的交互。原因是记录是固定的，并强加了特定的顺序。然而，保序和线程间的依赖关系可能在不同的微架构中有所不同。对于一些研究来说，这种影响可能是有限的；然而，对于其他研究来说，这可能意义重大。这一切都取决于优化的类型和正在评估的工作负载。关键问题是，某些微架构结构的变化（例如，分支预测器、缓存、预取器等）——这甚至可能是很小的变化——可能会导致线程以不同的顺序获取锁。这可能会导致共享资源（如缓存、内存、互连网络等）中不同的冲突和竞争行为，这反过来又可能影响线程间交替。因此，即使是微架构上很小的变化也可能导致（非常）不同的性能数据，而这些变化可能只会导致特定基准测试的巨大差异；因此，一个特定基准测试的较大变化可能并不代表其他工作负载。由于记录驱动仿真仿真的是单个保序，因此无法捕捉这些效果。此外，记录可能反映一种特定的顺序，这种顺序甚至可能不会出现在目标微体系结构上。 |
| 5.6 EXECUTION-DRIVEN SIMULATION | 5.6 执行驱动仿真 |
| In contrast to trace-driven simulation, execution-driven simulation combines functional with timing simulation. By doing so, it eliminates the disadvantages of trace-driven simulation: trace files do not need to be stored, speculatively executed instructions get simulated accurately, and the inter-thread ordering in multi-threaded workloads is modeled accurately. For these reasons, execution-driven simulation has become the de-facto simulation approach. Example execution-driven simulators are SimpleScalar [7], RSIM [88], Asim [57], M5 [16], GEMS [133], Flexus [190], and PTLSim [199]. Although execution-driven simulation achieves higher accuracy than trace-driven simulation, it comes at the cost of increased development time and evaluation time. | 与记录驱动仿真相比，执行驱动仿真结合了功能仿真和时序仿真。通过这样做，它消除了记录驱动仿真的缺点：不需要存储记录文件，可以准确地仿真投机性执行的指令，并准确地建模多线程工作负载中的线程间排序。由于这些原因，执行驱动仿真已经成为事实上的仿真方法。示例执行驱动仿真器有SimpleScalar [7]， RSIM [88]， Asim [57]， M5 [16]， GEMS [133]， Flexus[190]和PTLSim[199]。尽管执行驱动仿真比记录驱动仿真获得更高的精度，但它以增加开发时间和评估时间为代价。 |
| 5.6.1 TAXONOMY | 5.6.1 分类法 |
| Mauer et al. [136] present a useful taxonomy of execution-driven simulators, see also Figure 5.3.  The taxonomy reflects four different ways of how to couple the functional and timing components in order to manage simulator complexity and development time. An execution-driven simulator that tightly integrates the functional and timing components, hence called integrated execution-driven simulator (see Figure 5.3(a)), is obviously harder to develop and maintain. An integrated simulator is not flexible, is harder to extend (e.g., when evaluating a new architectural feature), and there is a potential risk that modifying the timing component may accidentally introduce an error in the functional component. In addition, the functional model tends to change very little, as mentioned before; however, the timing model may change a lot during architecture exploration. Hence, it is desirable from a simulator complexity and development point of view to decouple the functional part from the timing part. There are a number of ways of how to do the decoupling, which we discuss now. | Mauer等人[136]提出了一种有用的执行驱动仿真器分类方法，参见图5.3。分类法反映了如何耦合功能和时序组件的四种不同方法，以管理仿真器的复杂性和开发时间。紧密集成了功能和时序组件的执行驱动仿真器，被称为集成的执行驱动仿真器（参见图5.3(a)），显然更难以开发和维护。集成的仿真器不灵活，更难以扩展（例如，在评估一个新的架构特性时），并且存在修改时序组件可能意外地在功能组件中引入错误的潜在风险。另外，如前所述，功能模型往往变化很小；然而，在体系结构探索期间，时序模型可能会发生很大的变化。因此，从仿真器的复杂性和开发角度来看，将功能部分与时序部分解耦是可取的。解耦有很多种方法，我们现在来讨论一下。 |
| Timing-directed simulation. A timing-directed simulator lets the timing simulator direct the functional simulator to fetch instructions along mispredicted paths and select a particular thread interleaving (Figure 5.3(b)). The Asim simulator [57] is a timing-directed simulator. The functional models keeps track of the architecture state such as register and memory values. The timing model has no notion of values; instead, it gets the effective addresses from the functional model, which it uses to determine cache hits and misses, access the branch predictor, etc. The functional model can be viewed of as a set of function calls that the timing model calls to perform specific functional tasks at precisely the correct simulated time. The functional model needs to be organized such that it can partially simulate instructions. In particular, the functional simulator needs the ability to decode, execute, perform memory operations, kill, and commit instructions.The timing model then calls the functional model to perform specific tasks at the correct time in the correct order. For example, when simulating the execution of a load instruction on a load unit, the timing model asks the functional model to compute the load’s effective address. The address is then sent back to the timing model, which subsequently determines whether this load incurs a cache miss. Only when the cache access returns or when a cache miss returns from memory,according to the timing model,will the functional simulator read the value from memory. This ensures that the load reads the exact same data as the target architecture would. When the load commits in the target architecture, the instruction is also committed in the functional model. The functional model also keeps track of enough internal state so that an instruction can be killed in the functional model when it turns out that the instruction was executed along a mispredicted path. | 时序引导仿真。时序引导仿真器允许时序仿真器引导功能仿真器沿着错误预测的路径获取指令，并选择特定的线程交错（图5.3(b)）。Asim仿真器[57]是一个时序引导仿真器。功能模型跟踪架构状态，比如寄存器和内存值。时序模型没有值的概念；相反，它从功能模型中获取有效地址，用于确定缓存命中和未命中、访问分支预测器等。功能模型可以被视为一组函数调用，时序模型调用这些函数调用来在准确的仿真时间执行特定的功能任务。需要对功能模型进行组织，使其能够部分仿真指令。特别是，功能仿真器需要解码、执行、进行内存操作、终止和提交指令的能力。时序模型然后调用功能模型，以在正确的时间以正确的顺序执行特定的任务。例如，当在负载单元上仿真load指令的执行时，时序模型要求功能模型计算load的有效地址。然后，地址被发送回时序模型，时序模型随后确定此负载是否会导致缓存未命中。只有当缓存访问返回或缓存缺失从内存返回时，根据时序模型，功能仿真器才会从内存读取值。这确保了load读取与目标体系结构完全相同的数据。当load在目标体系结构中提交时，指令也会在功能模型中提交。功能模型还会追踪足够多的内部状态，以便当发现某条指令沿着预测错误的路径执行时，可以在功能模型中终止该指令。 |
|  | |
| Figure 5.3: Taxonomy of execution-driven simulation. | 图5.3：执行驱动仿真的分类法 |
| Functional-first simulation. In the functional-first simulation model (Figure 5.3(c)), a functional simulator feeds an instruction trace into a timing simulator.This is similar to trace-driven simulation, except that the trace need not to be stored on disk; the trace may be fed from the functional simulator into the timing simulator through a UNIX pipe. | 功能优先仿真。在功能优先仿真模型(图5.3(c))中，功能仿真器向时序仿真器提供指令记录。这类似于记录驱动仿真，不同的是记录不需要存储在磁盘上；跟踪可以通过UNIX管道从功能仿真器输入到时序仿真器。 |
| In order to be able to simulate along mispredicted paths and model timing-dependent interthread orderings and dependences, the functional model provides the ability to roll back to restore prior state [178]. In particular, when executing a branch, the functional model does not know whether the branch is mispredicted — only the timing simulator knows — and thus, it will execute only correct-path instructions. When a mispredicted branch is detected in the fetch stage of the timing simulator, the functional simulator needs to be redirected to fetch instructions along the mispredicted branch. This requires that the functional simulator rolls back to the state prior to the branch and feeds instructions along the mispredicted path into the timing simulator. When the mispredicted branch is resolved in the timing model, the functional model needs to roll back again, and then start feeding correct-path instructions into the timing model. In other words, the functional model is speculating upon which path the branch will take, i.e., it speculates that the branch will be correctly predicted by the timing model. | 为了能够沿着预测错误的路径进行仿真，并对依赖于时序的线程间顺序和依赖进行建模，功能模型提供了回滚恢复之前状态的能力[178]。特别是，在执行分支时，功能模型不知道该分支是否被错误预测——只有时序仿真器知道——因此，它将只执行正确路径指令。当在时序仿真器的取指阶段检测到预测错误的分支时，需要将功能仿真器重定向到沿着预测错误的分支获取指令。这要求功能仿真器回滚到分支之前的状态，并沿着预测错误的路径向时序仿真器提供指令。当在时序模型中解决了预测错误的分支时，功能模型需要再次回滚，然后开始向时序模型提供正确路径指令。换句话说，功能模型正在推测分支将采取的路径，也就是说，它推测时序模型将正确地预测分支。 |
| As mentioned earlier, inter-thread dependences may depend on timing, i.e., small changes in timing may change the ordering in which threads acquire a lock and thus may change functionality and performance. This also applies to a functional-first simulator: the timing may differ between the functional model and the timing model, and as a result, the ordering in which the functional model acquires a lock is not necessarily the same as the ordering observed in the timing model. This ordering problem basically boils down to whether loads ‘read’ the same data in the functional and timing models. Functional-first simulation can thus handle this ordering problem by keeping track of the data read in the functional model and the timing model. The simulator lets the functional model run ahead; however, when the timing model detects that the data a load would read in the target architecture differs from the data read in the functional model, it rolls back the functional model and requests the functional model to re-execute the load with the correct data — this is called speculative functional-first simulation [29]. Addressing the ordering problem comes at the cost of keeping track of the target memory state in the timing simulator and comparing functional/timing simulator data values. | 正如前面提到的，线程间的依赖关系可能依赖于时序，也就是说，时序上的微小变化可能会改变线程获得锁的顺序，从而可能会改变功能和性能。这也适用于功能优先的仿真器：功能模型和时序模型之间的时序可能不同，因此，功能模型获得锁的顺序不一定与时序模型中观察到的顺序相同。这个保序问题基本上归结为load是否在功能和时序模型中“读取”相同的数据。因此，功能优先仿真可以通过跟踪功能模型和时序模型中读取的数据来处理这个排序问题。仿真器让功能模型提前运行；但是，当时序模型检测到load将在目标架构中读取的数据与在功能模型中读取的数据不同时，它会回滚功能模型并请求功能模型使用正确的数据重新执行load—这称为投机功能优先仿真[29]。解决排序问题的代价是跟踪时序仿真器中的目标内存状态，并比较功能/时序仿真器的数据值。 |
| Argollo et al. [4] present the COTSon simulation infrastructure which employs AMD’s SimNow functional simulator to feed a trace of instructions into a timing simulator. The primary focus for COTSon is to simulate complex benchmarks, e.g., commodity operating systems and multi-tier applications, as well as have the ability to scale out and simulate large core counts. In the interest of managing simulator complexity and achieving high simulation speeds, COTSon does not provide roll-back functionality but instead implements timing feedback, which lets the timing simulator adjust the speed of the functional simulator to reflect the timing estimates. | Argollo等人[4]提出了COTSon仿真基础框架，该仿真框架使用AMD的SimNow功能仿真器将指令记录输入到时序仿真器。COTSon的主要关注点是仿真复杂的基准测试，例如，商业操作系统和多层应用程序，以及扩展和仿真大核心数系统的能力。为了管理仿真器的复杂性和实现快的仿真速度，COTSon不提供回滚功能，而是实现了时序反馈，这让时序仿真器能够调整功能仿真器的速度，以反映时序估计。 |
| In summary, the key advantage of functional-first simulation is that it allows the functional simulator to run ahead of the timing simulator and exploit parallelism, i.e., run the functional and timing simulator in parallel.Relative to timing-directed simulation in which the timing model directs the functional model at every instruction and/or cycle,functional-first simulation improves simulator performance (i.e., reduces evaluation time) and reduces the complexity of the overall simulator. | 总之，功能优先仿真的关键优势是它允许功能仿真器先于时序仿真器运行，并利用并行性，即并行运行功能仿真器和时序仿真器。相对于时序导向仿真，时序模型在每个指令和/或周期指导功能模型，功能优先仿真提高了仿真器的性能（即，减少了评估时间），并降低了整个仿真器的复杂性。 |
| Timing-first simulation. Timing-first simulation lets the timing model run ahead of the functional model [136], see Figure 5.3. The timing simulator models architecture features (register and memory state), mostly correctly, in addition to microarchitecture state. This allows for accurately (though not perfectly) modeling speculative execution along mispredicted branches as well as the ordering of inter-thread events. When the timing model commits an instruction, i.e., when the instruction becomes non-speculative, the functional model verifies whether the timing simulator has deviated from the functional model. On a deviation, the timing simulator is repaired by the functional simulator. This means that the architecture state be reloaded and microarchitecture state be reset before restarting the timing simulation. In other words, timing-first simulation consists of an almost correctly integrated execution-driven simulator (the timing simulator) which is checked by a functionally correct functional simulator. | 时序优先仿真。时序优先仿真允许时序模型先于功能模型运行[136]，见图5.3。除了微架构状态之外，时序仿真器对架构特性（寄存器和内存状态）的建模基本上是正确的。这允许沿着错误预测的分支精确（尽管不是完美）仿真投机执行，以及线程间事件的排序。当时序模型提交一条指令时，即当指令变为非投机指令时，功能模型验证时序仿真器是否偏离了功能模型。在出现偏差时，时序仿真器由功能仿真器修复。这意味着在重新启动时序仿真之前，重新加载架构状态并重置微架构状态。换句话说，时序优先仿真包括一个几乎正确集成的执行驱动仿真器（时序仿真器），由功能正确的功能仿真器进行检查。 |
| A timing-first simulator is easier to develop than a fully integrated simulator because the timing simulator does not need to implement all the instructions. A subset of instructions that is important to performance and covers the dynamically executed instructions well is sufficient. Compared to timing-directed simulation, timing-first simulation requires less features in the functional simulator while requiring more features in the timing simulator. | 时序优先仿真器比完全集成的仿真器更容易开发，因为时序仿真器不需要实现所有指令。一个对性能很重要，并且很好地覆盖了动态执行指令的指令子集就足够了。与时序引导仿真相比，时序优先仿真对功能仿真器的特性要求较低，而对时序仿真器的特性要求较高。 |
| 5.6.2 DEALING WITH NON-DETERMINISM | 5.6.2 处理非确定性 |
| An important challenge one has to deal with when simulating multi-threaded workloads on execution-driven simulators is non-determinism — Alameldeen and Wood present a comprehensive evaluation on non-determinism [1]. Non-determinism refers to the fact that small timing variations can cause executions that start from the same initial state to follow different execution paths. Non-determinism occurs both on real hardware as well as in simulation. On real hardware, timing variations arise from a variety of sources such as interrupts, I/O, bus contention with direct memory access (DMA), DRAM refreshes, etc. One can also observe non-determinism during simulation when comparing architecture design alternatives. For example, changes in some of the design parameters (e.g., cache size, cache latency, branch predictor configuration, processor width, etc.) can cause non-determinism for a number of reasons. | 当在执行驱动仿真器上仿真多线程工作负载时，一个必须处理的重要挑战是非确定性——Alameldeen和Wood提出了一项关于非确定性的综合评估[1]。不确定性指的是这样一个事实：小的时序变化可能导致从相同初始状态开始的执行遵循不同的执行路径。不确定性既发生在真实的硬件上，也发生在仿真中。在真正的硬件上，时间变化来自于各种各样的来源，如中断、I/O、与直接内存访问(DMA)的总线争用、DRAM刷新等。在比较架构设计方案的仿真过程中，还可以观察到不确定性。例如，一些设计参数的改变（例如缓存大小、缓存延迟、分支预测器配置、处理器宽度等）会由于许多原因导致不确定性。 |
| The operating system might make different scheduling decisions across different runs. For example, the scheduling quantum might end before an I/O event in one run but not in another. | 操作系统可能在不同的运行过程中做出不同的调度决策。例如，在一次运行中，调度时间可能在I/O事件之前结束，但在另一次运行中不会。 |
| Threads may acquire locks in a different order. This may cause the number of cycles andinstructions spent executing spin-lock loop instructions to be different across different architectures. | 线程可以以不同的顺序获得锁。这可能会导致，在不同的架构中，执行自旋锁循环指令的周期数和指令数是不同的。 |
| Threads that run at different relative speeds may incur different cache coherence traffic aswell as conflict behavior in shared resources. For example, the conflict behavior in shared multicore caches may be different across different architecture designs; similarly, contention in the interconnection network may be different. | 以不同的相对速度运行的线程可能导致不同的缓存一致性负载以及共享资源中的冲突行为。例如，在不同的架构设计中，共享多核缓存中的冲突行为可能是不同的；同样，在互连网络中可能会有不同的竞争。 |
| Non-determinism severely complicates comparing design alternatives during architecture exploration. The timing differences may lead the simulated workload to take different execution paths with different performance characteristics. As a result, it becomes hard to compare design alternatives. If the variation in the execution paths is significant, comparing simulations becomes unreliable because the amount and type of work done differs across different executions. The fundamental question is whether the performance differences observed across the design alternatives are due to differences in the design alternatives or due to differences in the workloads executed. Not addressing this question may lead to incorrect conclusions. | 在架构探索阶段，不确定性使得比较设计方案变得非常复杂。时序差异可能导致仿真工作负载采取具有不同性能特征的不同执行路径。因此，很难比较不同的设计方案。如果执行路径的变化很明显，那么比较将变得不可靠，因为不同的执行所做的工作的数量和类型是不同的。基本问题是，不同设计方案观察到的性能差异是由于设计方案的差异，还是由于执行的工作负载的差异。不解决这个问题可能会导致错误的结论。 |
| Alameldeen and Wood [1] present a simple experiment that clearly illustrates the need for dealing with non-determinism in simulation. They consider an OLTP workload and observe that performance increases with increasing memory access time, e.g., 84-ns DRAM leads to 7% better performance than 81-ns DRAM. Of course, this does not make sense — no computer architect will conclude that slower memory leads to better performance. Although this conclusion is obvious in this simple experiment, it may be less obvious in more complicated and less intuitive design studies. Hence, we need appropriate counteractions. | Alameldeen和Wood[1]提出了一个简单的实验，清楚地说明了在仿真中处理非确定性的必要性。他们考虑OLTP负载，并观察到性能随着内存访问时间的增加而增加，例如，84-ns DRAM比81-ns DRAM的性能提高了7%。当然，这是没有意义的——没有计算机架构师会得出这样的结论：内存越慢，性能越好。虽然这个结论在这个简单的实验中很明显，但在更复杂、更不直观的设计研究中可能就不那么明显了。因此，我们需要适当的应对措施。 |
| There are three potential solutions for how to deal with non-determinism. | 对于如何处理非确定性，有三种可能的解决方案。 |
| Long-running simulations. One solution is to run the simulation for a long enough period, e.g., simulate minutes of simulated times rather than seconds. Non-determinism is likely to largely vanish for long simulation experiments; however, given that architecture simulation is extremely slow, this is not a viable solution in practice. | 长时间运行仿真。一种解决方案是运行仿真足够长的时间，例如，仿真几分钟而不是几秒的仿真时间。在长时间的仿真实验中，不确定性很可能在很大程度上消失；然而，考虑到架构仿真极其缓慢，这在实践中不是可行的解决方案。 |
| Eliminate non-determinism. A second approach is to eliminate non-determinism. Lepak et al. [125] and Pereira et al. [156] present approaches to provide reproducible behavior of multithreaded programs when simulating different architecture configurations on execution-driven simulators; whereas Lepak et al. consider full-system simulation, Pereira et al. focus on user-level simulation.These approaches eliminate non-determinism by guaranteeing that the same execution paths be executed: they enforce the same order of shared memory accesses across simulations by introducing artificial stalls; also, interrupts are forced to occur at specific points during the simulation. Both the Lepak et al. and Pereira et al. approaches propose a metric and method for quantifying to what extent the execution was forced to be deterministic. Introducing stalls implies that the same amount of work be done in each simulation; hence, one can compare design alternatives based on a single simulation. The pitfall of enforcing determinism is that it can lead to executions that may never occur in a real system. In other words, for workloads that are susceptible to non-determinism, this method may not be useful. So, as acknowledged by Lepak et al., deterministic simulation must to be used with care. | 消除不确定性。第二种方法是消除不确定性。Lepak等人[125]和Pereira等人[156]提出了在执行驱动仿真器上仿真不同架构配置时，提供多线程程序可重现行为的方法；Lepak等人考虑的是全系统仿真，而Pereira等人关注的是用户级仿真。这些方法通过保证执行相同的执行路径来消除不确定性：它们通过引入人工延迟来强制不同仿真的共享内存访问的相同顺序；此外，在仿真过程中，中断被迫发生在特定的点。Lepak等人和Pereira等人的方法都提出了一种度量和方法来量化在何种程度上强制执行是确定性的。引入阻塞意味着在每次仿真中所做的工作是相同的；因此，我们可以基于单一的仿真来比较设计方案。强制确定性的缺陷在于，它可能导致在真实系统中永远不会发生的执行。换句话说，对于易受不确定性影响的工作负载，这种方法可能并不有用。因此，正如Lepak等人所承认的，确定性仿真必须谨慎使用。 |
| Note that trace-driven simulation also completely eliminates non-determinism because the instructions in the trace are exactly the same across different systems. However, trace-driven simulation suffers from the same limitation: it cannot appropriately evaluate architectural designs that affect thread interactions. | 注意，记录驱动的仿真也完全消除了不确定性，因为记录中的指令在不同的系统中是完全相同的。然而，记录驱动仿真也受到同样的限制：它不能适当地评估影响线程交互的架构设计。 |
| Statistical methods. A third approach is to use (classical) statistical methods to draw valid conclusions. Alameldeen and Wood [1] propose to artificially inject small timing variations during simulation. More specifically, they inject small changes in the memory system timing by adding a uniformly distributed random number between 0 and 4 ns to the DRAM access latency. These randomly injected perturbations create a range of possible executions starting from the same initial condition — note that the simulator is deterministic and will always produce the same timing, hence the need for introducing random perturbations. They then run the simulation multiple times and compute the mean across these runs along with its confidence interval. | 统计方法。第三种方法是使用（经典）统计方法来得出有效的结论。Alameldeen和Wood[1]提出在仿真程中人工注入小的时序变化。更具体地说，它们通过在DRAM访问延迟中添加一个均匀分布在0到4ns之间的随机数，在内存系统时序上注入了小的变化。这些随机注入的扰动产生了从相同初始条件开始的一系列可能的执行——请注意，仿真器是确定性的，并且总是产生相同的时序，因此需要引入随机扰动。然后，他们多次运行仿真，并计算这些运行的平均值及其置信区间。 |
| An obvious drawback of this approach is that it requires multiple simulation runs, which prolongs total simulation time. This makes this approach more time-consuming compared to the approaches that eliminate non-determinism. However, this is the best one can do to obtain reliable performance numbers through simulation. Moreover, multiple (small) simulation runs is likely to be more time-efficient than one very long-running simulation. | 这种方法的一个明显缺点是它需要多次仿真运行，这延长了整个仿真时间。这使得这种方法比消除不确定性的方法更耗时。然而，这是通过仿真获得可靠性能数据的最佳方法。此外，多次（小型）仿真运行可能比一次长时间运行的仿真更有时间效率。 |
| 5.7 MODULAR SIMULATION INFRASTRUCTURE | 5.7 模块化仿真组件 |
| Cycle-accurate simulators are extremely complex pieces of software, on a par with the microarchitectures they are modeling. Like in any other big software project, making sure that the software is well structured is crucial in order to keep the development process manageable. Modularity and reusability are two of the key goals in order to improve manageability and speed of model development. Modularity refers to breaking down the performance modeling problem into smaller pieces that can be modeled separately. Reusability refers to reusing individual pieces in different contexts. Modularity and reusability offer many advantages. It increases modeling productivity as well as fidelity in the performance models (because the individual pieces may have been used and validated before in a different contexts); it allows for sharing individual components across projects, even across products and generations in an industrial environment; it facilitates architectural experimentation (i.e., one can easily exchange components while leaving the rest of the performance model unchanged). All of these benefits lead to a shorter overall development time. | 周期精确的仿真器是极其复杂的软件，与它们正在建模的微架构是一样的。与任何其他大型软件项目一样，为了保持开发过程的可管理性，确保软件架构良好是至关重要的。为了提高模型开发的可管理性和速度，模块化和可重用性是两个关键目标。模块化是指将性能建模问题分解为可以单独建模的更小的部分。可重用性指的是在不同的上下文中重用各个部分。模块化和可重用性提供了许多优点。它提高了建模的效率以及性能模型的保真度（因为单个部件可能在不同的上下文中被使用和验证过）；它允许跨项目共享单个组件，甚至在工业环境中跨产品和代际；它促进了架构实验（例如，可以在保持性能模型其余部分不变的情况下轻松地替换组件）。所有这些好处都会缩短整体开发时间。 |
| Several simulation infrastructures implement the modularity principle, see for example Asim [57] by Digital/Compaq/Intel, Liberty [183] at Princeton University, MicroLib [159] at INRIA, UNISIM [6] at INRIA/Princeton, and M5 [16] at the University of Michigan. A modular simulation infrastructure typically provides a simulator infrastructure for creating many performance models rather than having a single performance model. In particular, Asim [57] considers modules, which are the basic software components. A module represents either a physical component of the target design (e.g., a cache, branch predictor, etc.) or a hardware algorithm’s operation (e.g., cache replacement policy). Each module provides a well-defined interface, which enables module reuse. Developers can contribute new modules to the simulation infrastructure as long as they implement the module interface, e.g., a branch predictor should implement the three methods for a branch predictor: get a branch prediction, update the branch predictor and handle a mispredicted branch. Asim comes with the Architect’s Workbench [58] which allows for assembling a performance model by selecting and connecting modules. | 一些仿真基础框架实现了模块化原理，例如，Digital/Compaq/Intel的Asim[57]，普林斯顿大学的Liberty [183]， INRIA的MicroLib [159]， INRIA/Princeton的UNISIM[6]，以及密歇根大学的M5[16]。模块化仿真基础设施通常提供用于创建许多性能模型的仿真器基础设施，而不是拥有单个性能模型。特别地，Asim[57]考虑了模块，模块是基本的软件组件。一个模块可以代表目标设计的一个物理组件（如缓存、分支预测器等），也可以代表硬件算法的操作（如缓存替换策略）。每个模块都提供了定义良好的接口，从而支持模块重用。开发人员只要实现模块接口，就可以为仿真基础设施贡献新的模块，例如，分支预测器应该为分支预测器实现三种方法：获得分支预测结果、更新分支预测器和处理预测错误的分支。Asim附带了Architect 's Workbench[58]，它允许通过选择和连接模块来组装性能模型。 |
| 5.8. NEED FOR SIMULATION ACCELERATION | 5.8 仿真加速的需求 |
| Cycle-accurate simulation is extremely slow,which is a key concern today in architecture research and development.The fundamental reason is that the microarchitectures they are modeling are extremely complex. Today’s processors consist of hundreds of millions or even billions of transistors, and they implement complex functionality such as memory hierarchies, speculative execution, out-of-order execution,prefetching,etc.Moreover,the trend towards multicore processing has further exacerbated this problem because multiple cores now need to be simulated as well as their interactions in the shared resources. This huge number of transistors leads to a very large and complex design space that needs to be explored during the design cycle of new microarchitectures. | 周期精确的仿真非常慢，这是当今架构研究和开发的一个关键问题。其根本原因是他们所建模的微架构极其复杂。今天的处理器由数亿甚至数十亿个晶体管组成，它们实现了复杂的功能，如内存层次结构、投机执行、乱序执行、预取等。此外，多核处理的趋势进一步加剧了这个问题，因为现在需要仿真多个核以及它们在共享资源中的交互。这种巨大数量的晶体管导致了，在新微架构的设计周期中，需要探索一个非常大而复杂的设计空间。 |
| Although intuition and analytical modeling can help guide the design process, eventually architects have to rely on detailed cycle-accurate simulation in order to make correct design decisions in this complex design space. The complexity of microarchitectures obviously reflects itself in the complexity and speed of the performance models. Cycle-accurate simulation models are extremely slow.Chiou et al.[30] give an overview of the simulation speeds that are typical for today’s simulators, see also Table 5.2. The speed of academic simulators ranges between 69 KIPS and 740 KIPS, and they are typically faster than the simulators used in industry which operate in the 1 KHz to 200 KIPS speed range. In other words, simulating only one second of real time (of the target system) may lead to multiple hours or even days of simulation time, even on today’s fastest simulators running on today’s fastest machines. And this is to simulate a single design point only. Architects typically run many simulations in order to get insight in the design space. Given that the design space is huge, the number of simulations that need to be run is potentially very large, which may make design space exploration quickly become infeasible. | 尽管直觉和分析建模可以帮助指导设计过程，但最终架构师必须依赖于详细的周期精确仿真，以便在这个复杂的设计空间中做出正确的设计决策。微架构的复杂性明显地体现在性能模型的复杂性和速度上。周期精确的仿真模型非常慢。Chiou等人[30]概述了当今仿真器的典型仿真速度，参见表5.2。学术界仿真器的速度范围在69 KIPS和740 KIPS之间，他们通常比工业中使用的仿真器在1 KIPS到200 KIPS的速度范围更快。换句话说，只仿真（目标系统的）一秒的实时时间可能会导致几个小时甚至几天的仿真时间，即使是在当今最快的机器上运行的当今最快的仿真器。这只是仿真一个设计点。架构师通常会运行许多仿真，以深入了解设计空间。考虑到设计空间巨大，需要运行的仿真数量可能非常大，这可能会使设计空间探索很快变得不可行的。 |
| Table 5.2: Comparing different simulators in terms of speed, the architectures and microarchitectures they support, and whether they support full-system simulation [30]. | 表5.2:比较不同仿真器的速度，它们支持的架构和微架构，以及它们是否支持全系统仿真[30]。 |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Industry simulators | | | | | | Simulator | ISA | Uarch | Speed | OS | | Intel | X86-64 | Core2 | 1-10KHz | Yes | | AMD | X86-64 | Opteron | 1-10KHz | Yes | | IBM | Power | Power5 | 200KIPS | Yes | | Freescale | PowerPC | E500 | 80KIPS | No | | Academic simulators | | | | | | PTLSim[199] | X86-64 | AMD Athlon | 270KIPS | Yes | | Sim-outorder[20] | Alpha | Alpha21654 | 740KIPS | No | | GEMS | Sparc | Generic | 69KIPS | Yes | | |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 工业仿真器 | | | | | | 仿真器 | 指令集 | 微架构 | 速度 | 支持OS | | Intel | X86-64 | Core2 | 1-10KHz | Yes | | AMD | X86-64 | Opteron | 1-10KHz | Yes | | IBM | Power | Power5 | 200KIPS | Yes | | Freescale | PowerPC | E500 | 80KIPS | No | | 学术仿真器 | | | | | | PTLSim[199] | X86-64 | AMD Athlon | 270KIPS | Yes | | Sim-outorder[20] | Alpha | Alpha21654 | 740KIPS | No | | GEMS | Sparc | Generic | 69KIPS | Yes | | |
| To make things even worse, the benchmarks that are being simulated grow in complexity as well. Given that processors are becoming more and more powerful, the benchmarks need to grow more complex. In the past, before the multicore era, while single-threaded performance was increasing exponentially, the benchmarks needed to execute more instructions and access more data in order to stress contemporary and future processors in a meaningful way. For example, the SPEC CPU benchmarks have grow substantially in complexity: the dynamic instruction count has increased from an average 2.5 billion dynamically executed instructions per benchmark in CPU89 to 230 billion instructions in CPU2000 [99],and an average 2,500 billion instructions per benchmark in CPU2006 [160]. Now, in the multicore era, benchmarks will need to be increasingly multi-threaded in order to stress multicore processors under design. | 更糟糕的是，被仿真的基准测试的复杂性也在增加。考虑到处理器越来越强大，基准测试需要变得更加复杂。在过去，在多核时代之前，当单线程性能呈指数级增长时，基准测试需要执行更多的指令和访问更多的数据，以有意义的方式给当代和未来的处理器施加压力。例如，SPEC CPU基准测试的复杂性大幅增加：动态指令数从CPU89的平均每个基准测试动态执行25亿条指令增加到CPU2000[99]的2300亿条指令，CPU2006[160]的动态指令数平均每个基准测试执行25000亿条指令。现在，在多核时代，基准测试将需要越来越多的多线程，以便在设计中对多核处理器施加压力。 |
| The slow speed of cycle-accurate simulation is a well-known and long-standing problem, and researchers have proposed various solutions for addressing this important issue. Because of its importance, the rest of this book is devoted to techniques that increase simulation speed. Sampled simulation is covered in Chapter 6 and probably is the most widely used simulation acceleration technique and reduces simulation time by simulating only (a) small snippet(s) from a much longer running benchmark. Statistical simulation, which we revisit in Chapter 7, takes a different approach by generating small synthetic workloads that are representative for long running benchmarks, while at the same time reducing the complexity of the simulator — the purpose for statistical simulation is merely to serve as a fast design space exploration technique that is complementary to detailed cycleaccurate simulation. Finally, in Chapter 8, we describe three approaches that leverage parallelism to speed up simulation: (i) distribute simulation runs across a parallel machine, (ii) parallelize the simulator to benefit from parallelism in the host machine, and (iii) exploit fine-grain parallelism by mapping (parts of) a simulator on reconfigurable hardware (i.e., FPGAs). | 周期精确仿真的速度慢是一个众所周知和长期存在的问题，研究人员提出了各种解决方案来解决这一重要问题。由于它的重要性，本书的其余部分致力于提高仿真速度的技术。采样仿真将在第6章中介绍，它可能是最广泛使用的仿真加速技术，通过只仿真较长运行基准的小片段来减少仿真时间。我们将在第7章重新讨论统计仿真，它采用了一种不同的方法，生成小型合成工作负载，代表长时间运行的基准，同时降低了仿真器的复杂性——统计仿真的目的仅仅是作为一种快速设计空间探索技术，对详细的周期精确仿真是一种补充。最后，在第8章中，我们描述了三种利用并行性来加速仿真的方法:(i)在并行机上分布仿真运行，(ii)并行化仿真器以受益于主机上的并行性，以及(iii)通过在可重构硬件（即FPGA）上映射仿真器（部分）来利用细粒度的并行性。 |

|  |  |
| --- | --- |
| CHAPTER 6 Sampled Simulation | 第6章 采样仿真 |
| The most prevalent method for speeding up simulation is sampled simulation. The idea of sampled simulation is to simulate the execution of only a small fraction of a benchmark’s dynamic instruction stream, rather than its entire stream. By simulating only a small fraction, dramatic simulation speedups can be achieved. | 加速仿真最常用的方法是采样仿真。采样仿真的想法是，只仿真测试程序中动态指令流中的一小部分指令的执行，而不是仿真完整指令流的指令。通过只仿真一小部分，可以实现显著的仿真加速。 |
| Figure 6.1 illustrates the concept of sampled simulation. Sampled simulation simulates one or more so called sampling units selected at various places from the benchmark execution.The collection of sampling units is called the sample. We refer to the pre-sampling units as the parts between two sampling units. Sampled simulation only reports performance metrics of interest for the instructions in the sampling units and discards the instructions in the pre-sampling units. And this is where the dramatic performance improvement comes from: only the sampling units, which is a small fraction of the total dynamic instruction count, are simulated in a cycle-accurate manner. | 图6.1描述了采样仿真的概念。采样仿真仿真一个或多个所谓的采样单元（sample unit），采样单元从测试程序执行的不同位置选取。采样单元的集合称为采样（sample）。我们将两个采样单元之间的部分称为预采样单元（per-sampling unit）。采样仿真只报告采样单元中感兴趣的指令的性能度量，丢弃预采样单元中的指令。这就是显著性能提升的来源：只有采样单元用时钟精确方法仿真，而采样单元只占整个动态指令流的一小部分。 |
|  | |
| Figure 6.1: General concept of sampled simulation. | 图6.1: 采样仿真的基本概念。 |
| There are three major challenges for sampled simulation to be accurate and fast: | 想要采样仿真即准确又快速，有三个主要挑战： |
| 1. What sampling units to select? | 1. 选择哪些采样单元？ |
| The first challenge is to select a sample that is representative for the entire simulation. Selecting sampling units from the initialization phase of a program execution is unlikely to be representative. This is a manifestation of the more general observation that a program typically goes through various phases of execution. Sampling should reflect this. In other words, a sample needs to be chosen such that all major phases of a program execution are represented. | 第一个挑战是选择能够代表整个仿真的采样。从程序执行的初始阶段选择采样单元通常是没有代表性的。更普适的观察发现的现象，程序通常会经历不同的执行阶段。采样应反映这一点。换句话说，采样需要从程序执行的所有主要阶段选取，这样才有代表性。 |
| 2. How to initialize a sampling unit’s architecture starting image? | 2. 如果初始化采样单元的架构起始镜像？ |
| The sampling unit’s Architecture Starting Image (ASI) is the architecture state, i.e., register and memory state, needed to correctly functionally simulate the sampling unit’s execution. This is a non-issue for trace-driven simulation because the pre-sampling unit instructions can simply be discarded from the trace, i.e., need not to be stored on disk. For execution-driven simulation, on the other hand, getting the correct ASI in an efficient manner is challenging. It is important to establish the architecture state (registers and memory) as fast as possible so that the sampled simulation can quickly jump from one sampling unit to the next. | 采样单元的架构起始镜像（ASI）是正确仿真采样单元功能执行所需要的架构态，比如寄存器和内存状态。对于记录驱动的仿真不是问题，因为预采样单元的指令可以直接从记录中丢弃，即不需要保存到磁盘上。另一方面，对于执行驱动仿真，用高效的方法获得正确的ASI是有挑战的。尽可能快地建立架构状态（寄存器和内存）很重要，这样采样仿真才能快速地从一个采样单元跳转到下一个采样单元。 |
| 3. How to accurately estimate a sampling unit’s microarchitecture starting image? | 3. 如何正确估计采样单元的微架构起始镜像？ |
| The sampling unit’s Microarchitecture Starting Image (MSI) is the microarchitecture state (content of caches,branch predictor,processor core structures,etc.) at the beginning of the sampling unit. Obviously, the MSI should be close to the microarchitecture state should the whole dynamic instruction stream prior to the sampling unit be simulated in detail. Unfortunately, during sampled simulation, the sampling unit’s MSI is unknown. This is well known in the literature as the cold-start problem. | 采样单元的微架构起始镜像（MSI）是采样单元开始时刻的微架构状态（包括缓存、分支预测、处理器核心结构等）。显然，MSI应该尽可能接近，对采样单元之前的整个动态指令流进行详细仿真得到的微架构状态。遗憾的是，在采样仿真中，采样单元的MSI是未知的。文献中称为冷启动（cold-start）问题。 |
| Note the subtle but important difference between getting the architecture state correct and getting the microarchitecture state accurate. Getting the architecture state correct is absolutely required in order to enable a functionally correct execution of the sampling unit. Getting the microarchitecture state as accurate as possible compared to the case where the entire dynamic instruction stream would have been executed up to the sampling unit is desirable if one wants accurate performance estimates. | 请注意，获得正确的架构状态和使微架构状态准确无误之间存在细微但重要的区别。为了能够正确执行采样单元，获得正确的架构状态是绝不能含糊的。如果想要获得准确的性能估计，使得微架构状态尽可能与整个动态指令流一直执行到采样单元的微架构状态相当是理想的。 |
| The following subsections describe each of these three challenges in more detail. | 下面章节详细描述这方面的挑战。 |
| 6.1 WHAT SAMPLING UNITS TO SELECT? | 6.1 选择哪些采样单元？ |
| There are basically two major ways for selecting sampling units, statistical sampling and targeted sampling. | 选择采样单元主要有两种主要的方法，统计采样和定向采样。 |
| 6.1.1 STATISTICAL SAMPLING | 6.1.1 统计采样 |
| Statistical sampling can select the sampling units either randomly or periodically. Random sampling selects sampling units randomly from the entire instruction stream in an attempt to provide a unbiased and representative sample, see Figure 6.2(a). Laha et al. [113] pick randomly chosen sampling units for evaluating cache performance. Conte et al. [36] generalized this concept to sampled processor simulation. | 统计采样可以随机地或者周期地选择采样单元。随机采样从整个指令流中随机地选择采样单元，试图提供无偏执而且有代表性的样本，见图6.2（a）。Laha等[113]提取随机选择的采样单元来评估缓存性能。Conte等人[36]将这一概念推广到采样处理器仿真。 |
| The SMARTS (Sampling Microarchitecture Simulation) approach by Wunderlich et al. [193;194] proposes systematic sampling, also called periodic sampling, which selects sampling units periodically across the entire program execution, see also Figure 6.2(b): the pre-sampling unit size is fixed, as opposed to random sampling, e.g., a sampling unit of 10,000 instructions is selected every 1 million instructions. | Wunderlich等人的SMARTS（采样微架构仿真）方法[193;194]提出了系统采样方法，也称为定期采样，它在整个程序执行过程中定期选择采样单元，另见图6.2（b）：预采样单元大小是固定的，与随机采样相反。例如，每100万条指令选择10,000条指令的采样单元。 |
| The key advantage of statistical sampling is that it builds on statistics theory, and allows for computing confidence bounds on the performance estimates through the central limit theorem [126]. Assume we have a performance metric of interest (e.g., CPI) for n sampling units: xi,1 ≤ i ≤ n. The mean of these measurements x¯ is computed as | 统计采样的主要优点是，统计采样构建在统计理论之上，而且允许通过中心极限定理[126]计算性能估计的置信度。假设个采样单元中感兴趣的性能度量（例如CPI）：。这些度量的期望的计算公式为 |
|  | |
|  | |
| Figure 6.2: Three ways for selecting sampling units: (a) random sampling, (b) periodic sampling, and (c) representative sampling. | 图6.2: 选择采样单元的三种方法：(a) 随机采样，(b) 周期性采样，和 (c) 代表性采样。 |
| The central limit theory then states that, for large values of n (typically n ≥ 30), x¯ is approximately Gaussian distributed provided that the samples xi are (i) independent, and (ii) come from the same population with a finite variance. Statistics then states that we can compute the confidence interval [c1,c2] for the mean as | 中心极限理论指出，对于足够大时（通常为），近似为高斯分布，前提是样本是（i）独立的，并且（ii）来自具有有限方差的同一总体。统计理论还声称，我们可以计算这个期望的置信区间为 |
|  | |
| 1 One could argue whether the condition of independent measurements is met because the sampling units are selected from a single program execution, and thus these measurements are not independent.This is even more true for periodic sampling because the measurements are selected at fixed intervals. | 人们可以争论是否满足独立测量的条件，因为采样单位是从单个程序执行中选择的，因此这些测量不是独立的。对于定期采样来说更是如此，因为测量值是以固定的时间间隔选择的。 |
| 2 Note that the central limit theory does not impose a particular distribution for the population from which the sample is taken. The population may not be Gaussian distributed (which is most likely to be the case for computer programs), yet the sample mean is Gaussian distributed. | 请注意，中心极限理论不要求被取样的总体必须服从特定的分布。总体可能不是高斯分布的（这很可能是计算机程序的情况），但样本均值是高斯分布的。 |
| with s the sample’s standard deviation. The value z1−α/2 is typically obtained from a precomputed table; z1−α/2 equals 1.96 for a 95% confidence interval. A 95% confidence interval [c1,c2] basically means that the probability for the true mean μ to lie between c1 and c2 equals 95%. In other words, a confidence interval gives the user some confidence that the true mean (e.g., the average CPI across the entire program execution) can be approximated by the sample mean x¯. | 其中是采样的标准差。的值通常从预先计算好的表格得出；对于95%的置信度，等于1.96。95%的置信区间（confidence interval）基本上意味着真均值介于 和 之间的概率等于 95%。换句话说，置信区间为用户提供了一些信心，即真实的期望（例如，整个程序执行的平均 CPI）可以通过样本平均值 近似。 |
| SMARTS [193; 194] goes one step further and leverages the above statistics to determine how many sampling units are required to achieve a desired confidence interval at a given confidence level. In particular, the user first determines a particular confidence interval size (e.g., a 95% confidence interval within 3% of the sample mean). The benchmark is then simulated and n sampling units are collected, n being some initial guess for the number of sampling units. The mean and its confidence interval is computed for the sample, and, if it satisfies the above 3% rule, this estimate is considered to be good. If not, more sampling units (> n) must be collected, and the mean and its confidence interval must be recomputed for each collected sample until the accuracy threshold is satisfied. This strategy yields bounded confidence interval sizes at the cost of requiring multiple (sampled) simulation runs. | SMARTS [193; 194]更进一步，利用上述统计理论来确定在给定的置信水平下，实现所需置信区间所需的采样单位数量。特别是，用户首先确定特定的置信区间大小（例如，样本均值的3%以内的95%置信区间）。然后仿真测试程序，并收集个采样单元。是对采样单元数量的初始猜测。计算样本的均值及其置信区间，如果它满足上述 3%的要求，则此估计值被认为是好的。否则，必须收集更多采样单位（），并且必须为每个收集的样本重新计算平均值及其置信区间，直到满足精度阈值。此策略产生有界置信区间大小，但代价是需要多次（采样）仿真运行。 |
| SMARTS [193; 194] uses a fairly small sampling unit size of 1,000 instructions for SPEC CPU workloads; Flexus [190] uses sampling units of a few 10,000 instructions for full-system server workloads. The reason for choosing a small sampling unit size is to minimize measurement (reduce number of instructions simulated in detail) while keeping into account measurement practicality (i.e., measure IPC or CPI over a long enough time period) and bias (i.e., make sure microarchitecture state is warmed up, as we will discuss in Section 6.3).The use of small sampling units implies that we need lots of them, typically on the order of 1000 sampling units.The large number of sampling units implies in its turn that statistical simulation becomes embarrassingly parallel, i.e., one can distribute the sampling units across a cluster of machines, as we will discuss in Section 8.1. In addition, it allows for throttling simulation turnaround time on-the-fly based on a desired error and confidence. | SMARTS[193; 194]针对SPEC CPU负载使用相当小的采样单元大小（1000 条指令）；Flexus[190]对全系统服务器工作负载使用由几万条指令组成的采样单元。选择小采样单元尺寸的原因是尽量减少测量（减少详细仿真的指令数量），同时考虑测量实用性（即，在足够长的时间段内测量IPC或CPI）和偏差（即，确保微架构状态预热，我们将在第6.3节中讨论）。使用小采样单元意味着我们需要很多采样单元，通常为1000个采样单元数量级。大量的采样单元反过来意味着统计仿真变得高度并行，即我们可以将采样单元分布在一组机器上，正如我们将在第8.1节中讨论的那样。此外，它还允许根据所需的误差和置信度，动态调节仿真周转时间。 |
| The potential pitfall of systematic sampling compared to random sampling is that the sampling units may give a skewed view in case the periodicity present in the program execution under measurement equals the sampling periodicity or its higher harmonics. For populations with low homogeneity though, periodic sampling is a good approximation of random sampling. Wunderlich et al. [193; 194] showed this to be the case for their workloads. This also agrees with the intuition that the workloads do not have sufficiently regular cyclic behavior at the periodicity relevant to sampled simulation (tens of millions of instructions). | 与随机采样相比，系统采样的潜在缺陷是，如果被测量的程序执行中表现出周期性，而且周期性等于采样周期性或其更高次谐波，则采样单元可能会给出有偏置的视角。然而，对于同质性低的群体，定期抽样是随机抽样的一个很好的近似值。Wunderlich等人[193; 194]表明他们的工作负载就表现为这种情况。这也符合这样一种直觉，即工作负载在与采样仿真相关的周期性（数千万条指令）上没有足够规则的周期性行为。 |
| 6.1.2 TARGETED SAMPLING | 6.1.2 定向采样 |
| Targeted sampling contrasts with statistical sampling in that it first analyzes the program’s execution to pick a sampling unit for each unique behavior in the program’s execution, see also Figure 6.2(c): targeted sampling selects a single sampling unit from each program phase and then weighs each sampling unit to provide an overall performance number. The key advantage of targeted sampling relative to statistical sampling is that it may lead to potentially fewer sampling units and thus a shorter overall simulation time as well as an overall simpler setup. The reason is that it leverages program analysis and intelligently picks sampling units. The major limitation for targeted sampling is its inability to provide a confidence bound on the performance estimates, unlike statistical sampling. A number of approaches have been proposed along this line, which we briefly revisit here. | 目标抽样与统计抽样形成鲜明对比。定向采样首先分析程序的执行，为程序执行中的每个独特行为选择一个抽样单元，见图6.2（c）：目标抽样从每个程序阶段选择一个采样单元，然后权衡每个采样单元的权重，以提供总体性能数字。相对于统计抽样，目标抽样的主要优点是，它可能导致抽样单元更少，从而缩短整体仿真时间，而且整体设置更简单。原因是它利用程序分析并智能地选取采样单元。与统计抽样不同，目标抽样的主要限制是它无法为性能估计提供置信度限制。沿着这条路线已经提出了一些办法，我们在这里简要地回顾一下。 |
| Skadron et al. [172] select a single sampling unit of 50 million instructions for their microarchitectural simulations. To this end, they first measure branch misprediction rates, data cache miss rates and instruction cache miss rates for each interval of 1 million instructions. By plotting these measurements as a function of the number of instructions simulated, they observe the time-varying program execution behavior, i.e., they can identify the initialization phase and/or periodic behavior in a program execution. Based on these plots, they manually select a contiguous sampling unit of 50 million instructions.Obviously,this sampling unit is chosen after the initialization phase.The validation of the 50 million instruction sampling unit is done by comparing the performance characteristics (obtained through detailed architectural simulations) of this sampling unit to 250 million instruction sampling units. The selection and validation of a sampling unit is done manually. The potential pitfall of this approach is that although the sampling unit is representative for a larger instruction sequence for this particular microarchitecture, it may not be representative on other microarchitectures. The reason is that the similarity analysis was done for a particular microarchitecture. In other words, phase analysis and picking the sampling unit is done based on microarchitecture-dependent characteristics and performance metrics only. | Skadron等[172]为他们的微架构仿真选择一个5000万条指令的采样单元。为此，他们首先按照100万条指令的区间测量分支预测错误率，数据缓存未命中率和指令缓存未命中率。通过将这些测量结果绘制为仿真的指令数的函数，他们发展程序执行的时变行为，即，他们可以在程序执行中区分初始化阶段和/或周期性行为。根据这些图线，他们手动选择5000万条指令的连续采样单元。显然，这个采样单元在初始化阶段后选择。5000万指令采样单元的验证是通过比较该采样单元的性能特征（通过详细的架构仿真获得）与2.5亿指令采样单元来实现的。采样单元的选择和验证是手动完成的。这种方法的潜在缺陷是，尽管采样单元在某种特定微架构下，对于较大指令序列具有代表性，但它在其他微架构上可能不具有代表性。原因是，相似性分析是针对特定的微架构进行的。换句话说，执行阶段分析和采样单元选取只根据微架构依赖的特性和性能评估完成的。 |
| Lafage and Seznec [112] use cluster analysis to detect and select sampling units that exhibit similar behavior. In their approach, they first measure two microarchitecture-independent metrics for each instruction interval of one million instructions. These metrics quantify the temporal and spatial behavior of the data reference stream in each instruction interval. Subsequently, they perform cluster analysis and group intervals that exhibit similar temporal and spatial behavior into so called clusters. For each cluster, the sampling unit that is closest to the center of the cluster is chosen as the representative sampling unit for that cluster. The performance characteristics per representative sampling unit are then weighted with the number of sampling units it represents, i.e., with the number of intervals grouped in the cluster the sampling unit represents. The microarchitecture independent metrics proposed by Lafage and Seznec are limited to quantifying data stream locality only. | Lafage和Seznec [112]使用聚类分析来检测和选择表现出相似行为的采样单元。在他们的方法中，他们首先按照100条指令的区间测量两个与微架构无关的指标。这些指标量化了每个指令间隔内数据访问流的时间和空间行为。随后，他们执行聚类分析，将表现出相似时间和空间行为的指令区间聚合到所谓的聚类中。对于每个聚类，选择最接近分类中心的采样单元作为该分类的代表性采样单元。然后，为每个代表性采样单元的性能特征，根据其代表的采样单元数进行加权，即，使用采样单元表示的聚类中的指令区间数。Lafage和Seznec提出的微架构独立指标仅限于量化数据流局部性。 |
| SimPoint [171] is the most well known targeted sampling approach, see Figure 6.3 for an overview of the approach. SimPoint breaks a program’s execution into intervals, and for each interval, it creates a code signature (step 1 in Figure 6.3). (An interval is a sequence of dynamically executed instructions.) The code signature is a so called Basic Block Vector (BBV) [170], which counts the number of times each basic block is executed in the interval, weighted with the number of instructions per basic block. Because a program may have a large static code footprint, and thus may touch (i.e., execute at least once) a large number of basic blocks, the BBVs may be large. SimPoint therefore reduces the dimensionality of the BBVs through random projection (step 2 in Figure 6.3), i.e., the dimensionality of the BBVs is reduced to a 15-dimensional vector, in order to increase the effectiveness of the next step. SimPoint then performs clustering, which aims at finding the groups of intervals that have similar BBV behavior (step 3 in Figure 6.3). Intervals from different parts of the program execution may be grouped into a single cluster. A cluster is also referred to as a phase. The key idea behind SimPoint is that intervals that execute similar code, i.e., have similar BBVs, have similar architecture behavior (e.g., cache behavior, branch behavior, IPC performance, power consumption, etc.). And this has been shown to be the case, see [116]. Therefore, only one interval from each phase needs to be simulated in order to recreate a complete picture of the program’s execution. They then choose a representative sampling unit from each phase and perform detailed simulation on that interval (step 4 in Figure 6.3). Taken together, these sampling units (along with their respective weights) can represent the complete execution of a program. In SimPoint terminology, a sampling unit is called a simulation point. Each simulation point is an interval of on the order of millions, or tens to hundreds of millions of instructions. Note that the simulation points were selected by examining only a profile of the code executed by a program. In other words, the profile is microarchitecture-independent. This suggests that the selected simulation points can be used across microarchitectures (which people have done successfully); however, there may be a potential pitfall in that different microarchitecture features may lead to a different performance impact for the simulation points compared to the parts of the execution not selected by SimPoint, and this performance impact is potentially unbounded — statistical sampling, on the other hand, bounds the error across microarchitectures. | SimPoint[171]是最广为人知的定向采样方法，这种方法的简介见图6.3。SimPoint将程序的执行切分为区间。对于每一个区间，SimPoint创建一个代码签名（图6.3中的第1步）。（一个区间是一段动态执行的指令流。）代码签名称为基本块向量（BBV）[170]，利用每个基本块的指令数加权统计区间中的每一个基本块的执行时间。由于程序可能具有较大的静态代码痕迹，因此可能接触（即至少执行一次）大量的基本块，因此BBV可能很大。因此，SimPoint通过随机投影（图6.3中的步骤2）来降低BBV的维数，即BBV的维数减少到15维向量，以提高下一步的效率。接着，SimPoint进行聚类，目的是寻找具有相同BBV行为的区间的分组（图6.3中第3步）。来自程序执行中的不同部分的区间可能被聚合到同一个分组中。聚类也被称为阶段（phase）。SimPoint背后的关键想法是，执行相似代码的区间，即具有相似的BBV，具有相似的架构行为（例如，缓存行为，分支行为，IPC性能，功耗等）。这些可以通过示例展示，见[116]。因此，为了重建程序执行的完整图景，只需要仿真每个阶段中的一个区间。接下来，他们从每一个阶段选择一个有代表性的采样单元，对这些区间进行详细仿真（图6.3中第4步）。放到一起，这些采样单元（以及他们代表的权重）可以代表程序的完整执行过程。在SimPoint术语中，采样单元称为仿真点（simulation point）。每一个仿真点是一个具有几百万条到几亿条指令的扫描点。换句话说，性能分析是微架构独立的。这意味着，选择的仿真点可以用于不同的微架构（已经成功实现）；然而，潜在的缺陷是，相对于没有被SimPoint选择的程序执行，不同的微架构特性可能导致不同的性能影响，并且这种性能影响可能是无限的——另一方面，统计抽样限制了跨微体系结构的错误。 |
| The SimPoint group extended SimPoint in a number of ways. They proposed techniques for finding simulation points early in the dynamic instruction stream in order to reduce the time needed to functionally simulate to get to these simulation points [157]. They considered alternative program characteristics to BBVs to find representative sampling units, such as loops and method calls [115; 117]. They correlate a program’s phase behavior to its control flow behavior, and by doing so, they identify cross binary simulation points so that simulation points can be used by architects and compiler builders when studying ISA extensions and evaluating compiler and software optimizations [158]. SimPoint v3.0 [77] improves the efficiency of the clustering algorithm, which enables applying the clustering to large data sets containing hundreds of thousands of intervals; the end result is that the SimPoint procedure for selecting representative simulation points can be applied in a couple minutes. PinPoint [153], developed by Intel researchers, implements BBV collection as a Pin tool, which allows for finding representative simulation points in x86 workloads. Yi et al. [195] compared the SimPoint approach against the SMARTS approach, and they conclude that SMARTS is slightly more accurate than SimPoint, however, SimPoint has a better speed versus accuracy trade-off. Also, SMARTS provides a confidence on the error, which SimPoint does not. | SimPoint研究组以多种方式扩展SimPoint。他们提出了早期从动态指令流中寻找仿真点的技术，从而降低为了获取这些仿真点而进行功能性仿真需要的时间[157]。他们考虑利用其他的BBV程序特征来表征采样单元，比如循环和方法调用[115; 117]。他们将程序的阶段性行为与程序的控制流行为相关联。通过这种做法，他们确定了二进制仿真点，使得仿真点可以被架构师和编译器工程师用来研究ISA扩展和评估编译器和软件优化[158]。SimPoint v3.0[77]改进了聚类算法的效率，使其能够用于包含成百上千个区间的大数据集的聚类；结果是，SimPoint可以在几分钟内完成选择达标性仿真点的过程。由Intel研究者开发的PinPoint[153]实现BBV收集，类似于Pin工具，允许在x86负载中寻找有代表性的仿真点。Yi等[195]对比SimPoint方法和SMARTS方法，他们的结论是，SMARTS明显比SimPoint准确，但是SimPoint对于速度和准确度进行了更好的折中。另外，SMARTS提供了对于误差的信心，但是SimPoint没有。 |
|  | |
| Figure 6.3: Overview of the SimPoint approach. | 图6.3 SimPoint方法示意图。 |
| Other approaches in the category of targeted sampling include early work by Dubey and Nair [42], which uses basic block profiles; Lauterbach [118] evaluates the representativeness of a sample using its instruction mix, the function execution frequency and cache statistics; Iyengar et al. [91] consider an instruction’s history (cache and branch behavior of the instructions prior to the instruction in the dynamic instruction stream) to quantify a sample’s representativeness. Eeckhout et al. [51] consider a broad set of microarchitecture-independent metrics to find representative sampling units across multiple programs; the other approaches are limited to finding representative sampling unit within a single program. | 定向采样分类中的其他方法包括，Dubey和Nair的早期工作[42]，使用了基本块分析；Lauterbach[118]利用指令混合特性、功能执行频率和缓存特性评估采样单元的代表性；Iyengar等[91]考虑指令历史（在动态指令流中先于这条指令的那些指令的缓存和分支行为）以量化采样的代表性；Eeckhout等人[51]考虑了一组广泛的微体系结构独立的指标，以在多个程序中找到代表性的采样单元；其他程序局限于在单个程序中寻找有代表性的采样点。 |
| 6.1.3 COMPARING DESIGN ALTERNATIVES THROUGH SAMPLED SIMULATION | 6.1.3 通过采样仿真比较设计方法 |
| In computer architecture research and development, comparing design alternatives, i.e., determining the relative performance difference between design points,is often more important than determining absolute performance in a single design point. Luo and John [130], and Ekman and Stenström [54] make the interesting observation that fewer sampling units are needed when comparing design points than when evaluating performance in a single design point. They make this observation based on a matched-pair comparison, which exploits the phenomenon that the performance difference between two designs tends to be (much) smaller than the variability across sampling units. In other words, it is likely the case that a sampling unit yielding high (or low) performance for one design point will also yield high (or low) performance for another design point. In other words, performance is usually correlated across design points, and the performance ratio between design points does not change as much as performance varies across sampling units. By consequence, we need to evaluate fewer sampling units to get an accurate performance estimate for the relative performance difference between design alternatives. Matched-pair comparisons can reduce simulation time by an order of magnitude [54; 130]. | 在计算机架构研究和开发中，比较设计方案，即确定设计点的相对性能，一般比确定单个采样点的绝对性能更加重要。Luo和John[130]，以及Ekman和Stenström[54]进行了有趣的观察，相比于评估单个设计点的性能，当比较设计点时需要更少的采样单元。他们是在配对比较的基础上进行这一观察的，这种比较利用了这样一种现象：两种设计之间的性能差异往往（远）小于抽样单位之间的差异。换句话说，最可能的情况是，一个设计点在一个采样点表现为高（或低）性能，那么在另一个采样点也会表现为高（或低）性能。换句话说，不同的设计点之间的性能通常是相关的，设计点之间的性能的变化并不像在不同的采样单元之间的性能变化那么大。因此，我们只需要评估更少的采样单元就能获得设计方案之间相对性能差异的准确性能估计。配对比较可以将仿真时间减少一个数量级[54;130]。 |
| 6.2 HOW TO INITIALIZE ARCHITECTURE STATE? | 6.2 如何确认初始架构状态？ |
| The second issue to deal with in sampled processor simulation is how to accurately provide a sampling unit’s architecture starting image (ASI). The ASI is the architecture state (register and memory content) needed to functionally simulate the sampling unit’s execution to achieve the correct output for that sampling unit. This means that the register and memory state needs to be established at the beginning of the sampling unit just as if all instructions prior to the sampling unit would have been executed.The two approaches for constructing the ASI are fast-forwarding and checkpointing, which we discuss in more detail. | 采样处理器仿真需要处理的第二个问题是，如何准确提供采样单元的架构起始镜像（ASI）。ASI是采样单元进行功能仿真得到正确输出所需要的架构状态（寄存器和内存上下文）。这表示，寄存器和内存状态需要在采样单元启动前建立，就好像采样单元之前的所有指令都已经被执行过。构建ASI的两种方式分别是快进和检查点，后文详细讨论。 |
| 6.2.1 FAST-FORWARDING | 6.2.1 快进 |
| The principle of fast-forwarding between sampling units is illustrated in Figure 6.4(a). Starting from either the beginning of the program or the prior sampling unit, fast-forwarding constructs the architecture starting image through functional simulation. When it reaches the beginning of the next sampling unit, the simulator switches to detailed execution-driven simulation. When detailed simulation reaches the end of the sampling unit, the simulator switches back to functional simulation to get to the next sampling unit, or in case the last sampling unit is executed, the simulator quits. | 图6.4(a)展示了采样点之间快进的原则。从程序的开始或者前一个采样点，快进通过功能仿真创建架构起始镜像。当到达下一个采样点的开始时，仿真器切换到详细的执行驱动仿真。当详细仿真到达采样点结尾时，仿真器切换回功能仿真，以获得下一个采样点；或者，如果是执行的最后一个采样点，仿真器退出。 |
| The main advantage is that it is relatively straightforward to implement in an execution-driven simulator — an execution-driven simulator comes with a functional simulator,and switching between functional simulation and detailed execution-driven simulation is not that hard to implement. The disadvantage is that fast-forwarding can be time-consuming for sampling units that are located deep in the dynamic instruction stream. In addition, it also serializes the simulation of all of the sampling units, i.e., one needs to simulate all prior sampling units and fast-forward between the sampling units in order to construct the ASI for the next sampling unit. Because fast-forwarding can be fairly time-consuming, researchers have proposed various techniques to speed up fast-forwarding. | 主要优势在于，在执行驱动仿真器中，实现快进比较简单——执行驱动仿真器本身具有功能仿真器，功能仿真和详细执行驱动仿真的切换实现起来不难。劣势在于，对于位于动态指令流深处的采样单元，快进会是消耗时间的。另外，快进还将所有的采样点仿真串行化，需要仿真之前所有的采样点并且在采样点之间快进，从而构造下一个采样点的ASI。因为快进相当耗时的，研究者提出了多种方法来加速快进。 |
|  | |
| Figure 6.4: Three approaches to initialize the architecture starting image: (a) fast-forwarding through functional simulation, (b) fast-forwarding through direct execution, and (c) checkpointing. | 图6.4: 初始化架构起始镜像的三种方法: (a) 通过功能仿真快进；(b) 通过直接执行跨进；和 (c) 检查点。 |
| Szwed et al. [179] propose to fast-forward between sampling units through native hardware execution, called direct execution, rather than through functional simulation, see also Figure 6.4(b). Because native hardware execution is much faster than functional simulation, substantial speedups can be achieved. Direct execution is employed to quickly go from one sampling unit to the next. When the next sampling unit is reached, checkpointing is used to communicate the architecture state from the real hardware to the simulator. Detailed execution-driven simulation of the sampling unit is done starting from this checkpoint. When the end of the sampling unit is reached, the simulator switches back to native hardware execution to quickly reach the next sampling unit. Many ways to incorporate direct hardware execution into simulators for speeding up simulation and emulation systems have been proposed, see for example [43; 70; 109; 163; 168]. | Szwed等[179]提出，在采样点之间通过原生硬件执行进行快进，称为直接执行，而不是通过功能模型，见图6.4(b)。因为原生硬件执行比功能模型快很多，可以获得明显的加速。直接执行用于快速地从一个采样单元到下一个采样单元。当到达下一个采样点的时候，使用检查点从真实硬件向仿真器通报架构状态。从检查点开始，执行采样单元的详细的执行驱动仿真。当到达采样点结尾时，仿真器切换回原生硬件执行，从而快速到达下一个采样点。已经提出了很多可以将直接硬件执行整合到仿真器中的方法，从而加速仿真和模拟系统，如示例[43; 70; 109; 163; 168]。 |
| One requirement for fast-forwarding through direct execution is that the simulation needs to be done on a host machine with the same instruction-set architecture (ISA) as the target machine. Fast-forwarding on a host machine with a different ISA than the target machine cannot be sped up through direct execution. This is a serious concern for studies that explore ISA extensions, let alone an entirely novel ISA. This would imply that such studies would need to fall back to relatively slow functional simulation. One possibility to overcome this limitation is to employ techniques from dynamic binary translation methods such as just-in-time (JIT) compilation and caching of translated code, as is done in Embra [191]. A limitation with dynamic binary translation though is that it makes the simulator less portable to host machines with different ISAs. An alternative approach is to resort to so called compiled instruction-set simulation as proposed by [21; 147; 164]. The idea of compiled instruction-set simulation is to translate each instruction in the benchmark by C code that decodes the instruction. Compiling the C code yields a functional simulator. Given that the generated functional simulator is written in C, it is easily portable across platforms. (We already discussed these approaches in Section 5.2.1.) | 通过直接执行的方法进行快速仿真的要求是，仿真需要在与目标机器具有相同指令集架构（ISA）的主机上完成。在不同ISA的主机上进行快进，不能通过直接执行来加速。这对于探索ISA扩展的研究来说是一个严重的问题，更不用说一个完全新颖的ISA了。这意味着这类研究将需要退回到相对较慢的功能仿真。克服这种限制的一种可能方法是，利用动态二进制翻译方法提供的技术，比如just-in-time（JIT）编译和缓存翻译代码，例如Embra所做[191]。动态二进制转换的一个限制是，它使仿真器对具有不同ISA的主机的可移植性降低。另一种方法是采用所谓的编译指令集仿真，如[21;147;164]。编译指令仿真的想法是，编译指令集仿真的思想是用C代码对基准中的每条指令进行解码。编译C代码会产生一个功能仿真器。由于生成的功能仿真器是用C语言编写的，所以它很容易跨平台移植。（我们已经在第5.2.1节讨论了这些方法。） |
| 6.2.2 CHECKPOINTING | 6.2.2 检查点 |
| Checkpointing takes a different approach and stores the ASI before a sampling unit. Taking a checkpoint is similar to storing a core dump of a program so that it can be replayed at that point in execution. A checkpoint stores the register contents and the memory state prior to a sampling unit. During sampled simulation, getting the architecture starting image initialized is just a matter of loading the checkpoint from disk and updating the register and memory state in the simulator, see Figure 6.4(c).The advantage of checkpointing is that it allows for parallel simulation, in contrast to fast-forwarding, i.e., checkpoints are independent of each other and enables simulating multiple sampling units in parallel. | 检查点采用不同的方法：在采样单元之前存储ASI。采用检查点类似于载入程序的核心转储（core dump），以便在执行时可以重现。检查点存储在采样单元之前的寄存器内容和存储器状态。在采样仿真期间，初始化架构起始镜像只是从磁盘加载检查点，并且更新仿真器中的寄存器和内存状态的问题，参见图6.4(c)。检查点的优点是它允许并行仿真，这与快进相反，即检查点彼此独立，并允许并行仿真多个采样单元。 |
| There is one major disadvantage to checkpointing compared to fast-forwarding and direct execution, namely, large checkpoint files need to be stored on disk. Van Biesbrouck et al. [184] report checkpoint files up to 28 GB for a single benchmark. Using many sampling units could be prohibitively costly in terms of disk space. In addition, the large checkpoint file size also affects total simulation time due to loading the checkpoint file from disk when starting the simulation of a sampling unit and transferring over a network during parallel simulation. | 与快进和直接执行相比，检查点有一个主要缺点，即需要将大容量的检查点文件存储在磁盘上。Van Biesbrouck等人[184]报告了在单个基准测试中需要高达28 GB的检查点文件。就磁盘空间而言，使用许多采样单元的代价可能会过高。此外，大的检查点文件也会影响总仿真时间，这是由于在开始仿真采样单元时从磁盘加载检查点文件，并在并行仿真期间通过网络传输检查点文件。 |
| Reduced checkpointing addresses the large checkpoint concern by limiting the amount of information stored in the checkpoint. The main idea behind reduced checkpointing is to only store the registers along with the memory words that are read in the sampling unit — a naive checkpointing approach would store the entire memory state. The Touched Memory Image (TMI) approach [184] and the live-points approach in TurboSMARTS [189] implement this principle. The checkpoint only stores the chunks of memory that are read during the sampling unit. This is a substantial optimization compared to full checkpointing which stores the entire memory state for each sampling unit. An additional optimization is to store only the chunks of memory that are read before they are written — there is no need to store a chunk of memory in the checkpoint in case that chunk of memory is written prior to being read in the sampling unit. At simulation time, prior to simulating the given sampling unit, the checkpoint is loaded from disk and the chunks of memory in the checkpoint are written to their corresponding memory addresses. This guarantees a correct ASI when starting the simulation of the sampling unit. A small file size is further achieved by using a sparse image representation, so regions of memory that consist of consecutive zeros are not stored in the checkpoint. | 通过限制检查点中存储的信息量，缩减检查点解决大检查点问题。缩减检查点背后的主要思想是只存储寄存器和在采样单元中读取的内存字——直接的检查点方法将存储整个内存状态。触及存储镜像（TMI）方法[184]和TurboSMARTS[189]中的活跃点方法实现了这一原理。检查点只存储采样单元期间读取的内存块。与存储每个采样单元的整个内存状态的完全检查点相比，这是一个显著的优化。另一种优化是只存储在写入之前读取的内存块——如果在采样单元中读取内存块之前写入内存块，则不需要在检查点中存储内存块。在仿真时，在仿真给定的采样单元之前，从磁盘加载检查点，并将检查点中的内存块写入相应的内存地址。这保证了在开始仿真采样单元时有一个正确的ASI。通过使用稀疏的镜像表示，进一步实现了较小的文件大小，因此由连续零组成的内存区域不会存储在检查点中。 |
| Van Biesbrouck et al.[184] and Wenisch et al.[189] provide a comprehensive evaluation of the impact of reduced ASI checkpointing on simulation accuracy, storage requirements, and simulation time. These studies conclude that the impact on error is marginal (less than 0.2%) — the reason for the inaccuracy due to ASI checkpointing is that the data values for loads along mispredicted paths may be incorrect. Reduced ASI checkpointing reduces storage requirements by two orders of magnitude compared to full ASI checkpointing. For example, for SimPoint using one-million instruction sampling units, an average (compressed) full ASI checkpoint takes 49.3 MB whereas a reduced ASI checkpoint takes only 365 KB. Finally, reduced ASI checkpointing reduces the simulation time by an order of magnitude (20×) compared to fast-forwarding and by a factor 4× compared to full checkpointing. | Van biesbrock等人[184]和Wenisch等人[189]提供了缩减ASI检查点大小对仿真精度、存储要求和仿真时间影响的综合评估。这些研究得出结论，对误差的影响是微不足道的（小于0.2%）——ASI检查点造成误差的原因是沿着错误预测路径的load的数据值可能是不正确的。与完全ASI检查点相比，缩减ASI检查点减少了两个数量级的存储需求。例如，对于使用100万个指令采样单元的SimPoint，平均（压缩）完整的ASI检查点占用49.3 MB，而缩减ASI检查点只占用365 KB。最后，与快进的仿真速度相比，缩减ASI检查点减少了一个数量级（20倍），与完全检查点相比减少了4倍。 |
| Ringenberg and Mudge [165] present intrinsic checkpointing which basically stores the checkpoint in the binary itself. In other words, intrinsic checkpointing brings the ASI up to state by providing fix-up checkpointing code consisting of store instructions to put the correct data values in memory — again, only memory locations that are read in the sampling unit need to be updated; it also executes instructions to put the correct data values in registers. Intrinsic checkpointing has the limitation that it requires binary modification for including the checkpoint code in the benchmark binary. On the other hand, it does not require modifying the simulator, and it even allows for running sampling units on real hardware. Note though that the checkpoint code may skew the performance metrics somewhat; this can be mitigated by considering large sampling units. | Ringenberg和Mudge[165]提出了内在检查点，它基本上将检查点存储在二进制文件本身中。换句话说，内部检查点通过提供由store指令组成的修正检查点代码将正确的数据值放入内存中，从而将ASI带到状态——同样，只有在采样单元中读取的内存位置需要更新；它还执行指令，将正确的数据值放入寄存器。内在检查点具有一定的局限性，它需要修改二进制代码才能将检查点代码包含在基准二进制代码中。另一方面，它不需要修改仿真器，甚至允许在真实的硬件上运行采样单元。请注意检查点代码可能会在一定程度上影响性能度量；这可以通过考虑大容量的采样单元来缓解。 |
| 6.3 HOW TO INITIALIZE MICROARCHITECTURE STATE? | 6.3 如何初始化微架构状态？ |
| The third issue in sampled simulation is to establish an as accurate as possible microarchitecture starting image (MSI), i.e., cache state, predictor state, processor core state, etc., for the sampling unit to be simulated. The MSI for the sampling unit should be as accurate as possible compared to the MSI that would have been obtained through detailed simulation of all the instructions prior to the sampling unit. An inaccurate MSI introduces error and will sacrifice the error bound and confidence in the estimates (for statistical sampling). | 采样仿真的第三个问题是为待仿真的采样单元建立尽可能精确的微架构起始镜像 （MSI），即缓存状态、预测器状态、处理器核心状态等。与采样单元之前通过详细仿真所有指令获得的MSI相比，采样单元的MSI应尽可能准确。不准确的MSI会引入误差，并会牺牲估计的误差范围和置信度（用于统计抽样）。 |
| The following subsections describe MSI approaches related to cache structures, branch predictors, and processor core structures such as the reorder buffer, issue queues, store buffers, functional units, etc. | 下面的小节描述了与缓存结构、分支预测器和处理器核心结构（如重排序缓存、发射队列、store缓冲区、功能单元等）相关的MSI方法。 |
| 6.3.1 CACHE STATE WARMUP | 6.3.1 缓存状态热身 |
| Caches are probably the most critical aspect of the MSI because caches can be large (up to several MBs) and can introduce a long history. In this section, we use the term ‘cache’ to collectively refer to a cache, a Translation Lookaside Buffers (TLB) and a Branch Target Buffers (BTB) because all of these structures have a cache-like structure. | 缓存可能是MSI最关键的方面，因为缓存可以很大（多达几个MB），并且可以引入很长的历史。在本节中，我们使用术语“缓存”来统称缓存、翻译后备缓冲区（Translation Lookaside Buffers, TLB）和分支目标缓冲区（Branch Target Buffers, BTB），因为所有这些结构都具有类似缓存的结构。 |
| A number of cache state warmup strategies have been proposed over the past 15 years. We now discuss only a selection. | 在过去的15年里，已经提出了许多缓存状态预热策略。我们现在只讨论一个选择。 |
| Nowarmup. The cold or no warmup scheme [38; 39; 106] assumes an empty cache at the beginning of each sampling unit. Obviously, this scheme will overestimate the cache miss rate. However, the bias can be small for large sampling unit sizes.Intel’s PinPoint approach [153],for example,considers a fairly large sampling unit size, namely 250 million instructions, and does not employ any warmup approach because the bias due to an inaccurate MSI is small. | 不热身。冷启动（cold）或不热身（no warmup）方案[38;39;106]假设在每个采样单元开始时有一个空的缓存。显然，这种方案会高估缓存未命中率。然而，对于大的采样单元，这种偏差可以很小。例如，Intel的PinPoint方法[153]考虑一个相当大的采样单位大小，即2.5亿条指令，并且没有采用任何预热方法，因为MSI不准确导致的偏差很小。 |
| Continuous warmup. Continuous warmup, as the name says, continuously keeps the cache state warm between sampling units. This means that the functional simulation between sampling units needs to be augmented to also access the caches. This is a very accurate approach but increases the time spent between sampling units. This approach is implemented in SMARTS [193; 194]: the tiny sampling units of 1,000 instructions used in SMARTS require a very accurate MSI, which is achieved through continuous warmup; this is called functional warming in the SMARTS approach. | 持续热身。顾名思义，持续预热是指在采样单元之间持续保持缓存状态温暖。这意味着采样单元之间的功能仿真需要增强，也需要访问缓存。这是一种非常精确的方法，但增加了采样单元之间的时间。该方法在SMARTS中实现[193;194]：SMARTS中使用的1000条指令的微小采样单元需要一个非常精确的MSI，这是通过持续的预热来实现的。这在SMARTS方法中被称为功能热身（functional warming）。 |
| Stitch. Stitch or stale state [106] approximates the microarchitecture state at the beginning of a sampling unit with the hardware state at the end of the previous sampling unit. An important disadvantage of the stitch approach is that it cannot be employed for parallel sampled simulation. | 缝补。缝补（stitch）或过时状态（stale state）[106]利用前一个采样单元结束时的硬件状态近似采样单元开始时的微架构状态。该方法的一个重要缺点是不能用于并行采样仿真。 |
| Cache miss rate estimation. Another approach is to assume an empty cache at the beginning of each sampling unit and to estimate which cold-start misses would have missed if the cache state at the beginning of the sampling unit was known. This is the so called cache miss rate estimator approach [106; 192]. A simple example cache miss estimation approach is hit-on-cold or assume-hit. Hit-on-cold assumes that the first access to a cache line is always a hit.This is an easy-to-implement technique which is fairly accurate for programs with a low cache miss rate. | 缓存未命中率估计。另一种方法是，假设每个采样单元开始时缓存是空的，并估计哪些冷启动未命中是真实的未命中，如果采样单元开始时的缓存状态是已知的。这就是所谓的缓存未命中率估计器方法[106;192]。缓存未命中估计方法的一个简单例子是冷命中或假设命中。冷命中假设对缓存行的第一次访问总是命中的。这是一种易于实现的技术，而且对于缓存未命中率较低的程序来说是相当准确的。 |
| Self-monitored adaptive (SMA) warmup. Luo et al. [131] propose a self-monitored adaptive (SMA) cache warmup scheme in which the simulator monitors the warmup process of the caches and decides when the caches are warmed up. This warmup scheme is adaptive to the program being simulated as well as to the cache being simulated — the smaller the application’s working set size or the smaller the cache, the shorter the warmup phase. One limitation of SMA is that it is unknown a priori when the caches will be warmed up and when detailed simulation should get started. This may be less of an issue for random statistical sampling (although the sampling units are not selected in a random fashion anymore), but it is a problem for periodic sampling and targeted sampling. | 自我监测自适应（SMA）热身。Luo等人[131]提出了一种自监测自适应（SMA）缓存预热方案，其中仿真器监控缓存的预热过程，并决定缓存何时被预热。这种热身方案适用于被仿真的程序和被仿真的缓存——应用程序的工作集越小或缓存越小，热身阶段越短。SMA的一个限制是，它事先不知道缓存何时会被预热，以及何时应该开始详细的仿真。对于随机统计抽样来说，这可能不是问题（尽管抽样单位不再以随机方式选择），但对于周期性抽样和定向采样来说，这是一个问题。 |
| Memory Reference Reuse Latency (MRRL). Haskins and Skadron [79] propose the MRRL warmup strategy. The memory reference reuse latency is defined as the number of instructions between two consecutive references to the same memory location. The MRRL warmup approach computes the MRRL for each memory reference in the sampling unit, and collects these MRRLs in a distribution. A given percentile, e.g., 99%, then determines when cache warmup should start prior to the sampling unit. The intuition is that a sampling unit with large memory reference reuse latencies also needs a long warmup period. | 内存引用重用延迟（MRRL）。Haskins和Skadron[79]提出了MRRL预热策略。内存引用重用延迟被定义为对同一内存位置的两个连续引用之间的指令数。MRRL预热方法计算采样单元中每个内存引用的MRRL，并在一个分布中收集这些MRRL。然后，给定的百分比（例如99%）决定在采样单元之前何时应该开始缓存预热。直觉上，具有较大内存引用重用延迟的采样单元也需要很长的预热期。 |
| Boundary Line Reuse Latency (BLRL). Eeckhout et al. [49] only look at reuse latencies that ‘cross’ the boundary line between the pre-sampling unit and the sampling unit, hence the name boundary line reuse latency (BLRL). In contrast, MRRL considers all the reuse latencies which may not be an accurate picture for the cache warmup required for the sampling unit. Relative to BLRL, MRRL may result in a warmup period that is either too short to be accurate or too long for the attained level of accuracy. | 边界线复用延迟（BLRL）。Eeckhout等人[49]只考察“跨越”预采样单元和采样单元之间的边界线的重用延迟，因此将其称为边界线重用延迟（BLRL）。相反，MRRL考虑了所有重用延迟，这对于采样单元所需的缓存预热可能不是一个准确的图像。相对于BLRL，MRRL可能会导致一个预热期要么太短而不能达到准确水平，要么对于需要的精确度来说太长。 |
| Checkpointing. Another approach to the cold-start problem is to checkpoint or to store the MSI at the beginning of each sampling unit. Checkpointing yields perfectly warmed up microarchitecture state. On the flipside, it is specific to a particular microarchitecture, and it may require excessive disk space for storing checkpoints for a large number of sampling units and different microarchitectures. Since this is infeasible to do in practice, researchers have proposed more efficient approaches to MSI checkpointing. | 检查点。另一种解决冷启动问题的方法是在每个采样单元开始时进行检查点（checkpint）或存储MSI。检查点产生了完美预热的微架构状态。另一方面，它是特定于特定微架构的，它可能需要过多的磁盘空间来存储大量采样单元和不同微架构的检查点。由于这在实践中是不可行的，研究人员提出了更有效的MSI检查点方法。 |
| One approach is the No-State-Loss (NSL) approach [35; 118]. NSL scans the pre-sampling unit and records the last reference to each unique memory location. This is called the least recently used (LRU) stream. For example, the LRU stream of the following reference stream ‘ABAACDABA’ is ‘CDBA’.The LRU stream can be computed by building the LRU stack: it is easily done by pushing an address on top of the stack when it is referenced. NSL yields a perfect warmup for caches with an LRU replacement policy. | 一种方法是No-State-Loss（NSL）方法[35;118]。NSL扫描预采样单元，并记录对每个内存位置的最后一次引用。这被称为最近最少使用（least recently used stream, LRU）流。例如，以下引用流' ABAACDABA '的LRU流是' CDBA '。LRU流可以通过构建LRU栈来计算：很容易通过当地址被引用时将地址压入栈顶来实现。NSL和LRU替换策略为缓存提供了一个完美的热身。 |
| Barr and Asanovic [10] extended this approach for reconstructing the cache and directory state during sampled multiprocessor simulation. In order to do so, they keep track of a timestamp per unique memory location that is referenced. In addition, they keep track of whether a memory location is read or written. This information allows them to quickly reconstruct the cache and directory state at the beginning of a sampling unit. | Barr和Asanovic[10]扩展了这种方法，用于在采样多处理器仿真期间，重建缓存和目录状态。为了做到这一点，它们跟踪每个被引用的内存位置的时间戳。此外，它们还跟踪内存位置是读取还是写入。这些信息允许他们在采样单元开始时快速重建缓存和目录状态。 |
| Van Biesbrouck et al. [185] and Wenisch et al. [189] proposed a checkpointing approach in which the largest cache of interest is simulated once for the entire program execution.The SimPoint project refers to this technique as ‘memory hierarchy state’; the TurboSMARTS project proposes the term ‘live points’. At the beginning of each sampling unit, the cache content is stored on disk as a checkpoint. The content of smaller sized caches can then be derived from the checkpoint. Constructing the content of a cache with a smaller associativity is trivial to from the checkpoint: the most recently accessed cache lines need to be retained per set, see Figure 6.5(a). Reducing the number of sets in the cache is slightly more complicated: the new cache set retains the most recently used cache lines from the merging cache sets — this requires keeping track of access times to cache lines during checkpoint construction, see Figure 6.5(b). | Van biesbrock等[185]和Wenisch等[189]提出了一种检查点方法，在整个程序执行过程中仿真感兴趣的缓存中容量最大的缓存。SimPoint项目将这种技术称为“内存层次状态”；TurboSMARTS项目提出了术语“激活点”。在每个采样单元开始时，缓存内容作为检查点存储在磁盘上。然后可以从检查点派生较小大小缓存的内容。构造一个关联性较小的缓存内容对于检查点来说是很简单的：每个组最近访问的缓存块需要保留，见图6.5(a)。减少缓存中组的数量稍微复杂一些：新的缓存组保留合并缓存组中最近使用的缓存组——这需要在检查点构建期间跟踪对缓存线的访问时间，见图6.5(b)。 |
| 6.3.2 PREDICTOR WARMUP | 6.3.2 预测器热身 |
| Compared to the amount of work done on cache state warmup, little work has been done on predictor warmup — it is an overlooked problem. In particular, accurate branch predictor warmup is required for accurate sampled simulation, even for fairly large sampling units of 1 to 10 million instructions [107]. There is no reason to believe that this observation made for branch predictors does not generalize to other predictors, such as next cache line predictors, prefetchers, load hit/miss predictors, load/store dependency predictors, etc. | 与缓存状态预热所做的大量工作相比，预测器预热所做的工作很少——这是一个被忽视的问题。特别是，即使是对于相当大的采样单位（100 - 1000万条指令） ，精确的采样仿真需要精确的分支预测器预热[107]。我们没有理由不相信对分支预测器的观察不会推广到其他预测器，例如下一个缓存行预测器、预取器、加载命中/未命中预测器、load/store依赖关系预测器等。 |
| One approach is to employ stitch, i.e., the sampling unit’s MSI is assumed to be the same as the state at the end of the prior sampling unit. Another approach is to consider a fixed-length warmup, e.g., start warming the branch predictor at for example 1 million instructions prior to the sampling unit, as proposed by Conte et al. [36]. Barr and Asanovic [9] propose warming the branch predictor using all the instructions prior to the sampling unit. In order not to have to store huge branch trace files to be stored on disk, they propose branch trace compression. Kluyskens and Eeckhout [107] propose Branch History Matching (BHM) which builds on a similar principle as MRRL and BLRL. BHM considers the reuse latency between dynamic branch instances that have share the same (or at least a very similar) branch history. Once the reuse latency distribution is computed, it is determined how long predictor warmup should take prior to the sampling unit. | 一种方法是采用缝合，即假设采样单元的MSI与前一个采样单元结束时的状态相同。另一种方法是考虑固定长度的预热，例如，像Conte等[36]提出的那样，在采样单元之前的100万条指令开始预热分支预测器。Barr和Asanovic[9]建议使用采样单元之前的所有指令来预热分支预测器。为了不需要在磁盘上存储大量的分支跟踪文件，他们提出了分支跟踪压缩。Kluyskens和Eeckhout[107]提出了分支历史匹配（BHM），其原理与MRRL和BLRL类似。BHM考虑具有相同（或至少非常相似）分支历史的动态分支实例之间的重用延迟。一旦计算出重用延迟分布，就可以确定预测器在采样单元之前的预热时间。 |
|  | |
| Figure 6.5: Constructing the content of a smaller sized cache from a checkpoint, when (a) reducing associativity and (b) reducing the number of sets. Each cache line in the checkpoint is tagged with a timestamp that represents the latest access to the cacheline. | 图6.5: 从检查点创建更小容量的cache的内容，(a) 减小组相连和(b)减小set的数量。检查点中的每一个缓存行都用时间戳进行标记，表示最后访问缓存行的时间。 |
| 6.3.3 PROCESSOR CORE STATE | 6.3.3 处理器核心状态 |
| So far, we discussed MSI techniques for cache and branch predictor structures. The processor core consists of a reorder buffer,issue queues,store buffers,functional units,etc.,which may also need to be warmed up.This is not a major concern for large sampling units because events in the processor core do not incur an as long history as in the cache hierarchy and branch predictors. However, for small sampling units, it is crucial to accurately warmup the processor core.Therefore, SMARTS [193; 194] considers small sampling units of 1,000 instructions and proposes fixed-length warming of the processor core of 2,000 to 4,000 instructions prior to each sampling unit — warmup length can be fixed because of the bounded history of the processor core as opposed to the unbounded history for caches, TLBs and predictors. | 到目前为止，我们讨论了用于缓存和分支预测器结构的MSI技术。处理器核心由重排序缓存、发布队列、存储缓冲区、功能单元等组成，可能也需要预热。对于大型采样单元来说，这不是一个主要的问题，因为处理器核心中的事件不会像缓存层次结构和分支预测器中那样产生很长的历史。然而，对于小的采样单元，准确地预热处理器核心是至关重要的。因此, SMARTS[193;194]考虑1000条指令的小采样单元，并建议在每个采样单元之前对处理器核心进行2000到4000条指令固定长度的预热——预热长度可以是固定的，因为处理器核心的历史是有边界的，而不是缓存、TLB和预测器的无边界历史。 |
| 6.4 SAMPLED MULTIPROCESSOR AND MULTI-THREADED PROCESSOR SIMULATION | 6.4 采样多处理器和多线程处理器仿真 |
| Whereas sampled simulation for single-threaded processors can be considered mature technology, and significant progress has been made towards sampled simulation of multi-threaded server workloads (see the Flexus project [190]), sampling multi-threaded workloads in general remains an open problem. One key problem to address in sampled simulation for multi-threaded, multicore and multiprocessor architectures running multi-threaded workloads relates to resource sharing. When two or more programs or threads share a processor’s resource such as a shared L2 cache or interconnection network — as is the case in many contemporary multi-core processors — or even issue queues and functional units — as is the case in Simultaneous Multithreading (SMT) processors — the performance of both threads becomes entangled. In other words, co-executing programs and threads affect each other’s performance. And, changing a hardware parameter may change which parts of the program execute together, thereby changing their relative progress rates and thus overall system performance. This complicates both the selection of sampling units and the initialization of the ASI and MSI. | 虽然用于单线程处理器的采样仿真可以被认为是成熟的技术，而且在多线程服务器工作负载的采样仿真方面已经取得了重大进展（参见Flexus项目[190]），但对多线程工作负载进行采样仍然是一个普遍的开放问题。在运行多线程工作负载的多线程、多核和多处理器架构的采样仿真中，要解决的一个关键问题与资源共享有关。当两个或两个以上的程序或线程共享一个处理器的资源时，比如共享的L2缓存或互连网络（这是许多当代多核处理器的情况），甚至发射队列和功能单元（这是同步多线程（SMT）处理器的情况），线程的性能就会发生纠缠。换句话说，共同执行的程序和线程会影响彼此的性能。而且，改变一个硬件参数可能会改变程序的哪些部分一起执行，从而改变它们的相对进度，进而改变整个系统的性能。这使得取样单位的选择和ASI和MSI的初始化复杂化。 |
| Van Biesbrouck et al. [188] propose the co-phase matrix approach, which models the impact of resource sharing on per-program performance when running independent programs (i.e., from a multi-program workload) on multithreaded hardware. The basic idea is to first use SimPoint to identify the program phases for each of the co-executing threads and keep track of the performance data of previously executed co-phases in a so called co-phase matrix;whenever a co-phase gets executed again, the performance data is easily picked from the co-phase matrix. By doing so, each unique co-phase gets simulated only once, which greatly reduces the overall simulation time. The co-phase matrix is an accurate and fast approach for estimating multithreaded processor performance both when the co-executing threads start at a given starting point as well as when multiple starting points are considered for the co-executing threads,see [186].The multiple starting points approach provides a much more representative overall performance estimate than a single starting point. Whereas the original co-phase matrix work focuses on two or four independent programs co-executing on a multithreaded processor, the most recent work by Van Biesbrouck et al. [187] studies how to select a limited number of representative co-phase combinations across multiple benchmarks within a benchmark suite. | Van Biesbrouck等[188]提出了同相矩阵方法，该方法仿真了在多线程硬件上运行独立程序（即多程序工作负载）时，资源共享对每个程序性能的影响。基本思想是，首先使用SimPoint来确定每个共同执行线程的程序阶段，并在所谓的同相矩阵（co-phase matrix）中跟踪先前执行的同相阶段的性能数据；每当一个同相再次执行时，性能数据很容易从同相矩阵中提取出来。这样一来，每个独特的同相阶段只仿真一次，大大缩短了整个仿真时间。当共同执行的线程从一个给定的起始点开始时，以及当共同执行的线程考虑多个起始点时，同相矩阵是一种精确而快速的估计多线程处理器性能的方法，参见[186]。与单一起点相比，多重起点方法提供了更有代表性的总体性能估计。最初的共相矩阵研究的重点是两个或四个独立程序在多线程处理器上共同执行，而Van Biesbrouck等人[187]的最新研究是如何在一个基准套件中的多个基准中选择有限数量的代表性同相组合。 |
| As mentioned earlier, Barr and Asanovic [10] propose an MSI checkpointing technique to reconstruct the directory state at the beginning of a sampling unit in a multiprocessor system. State reconstruction for a multiprocessor is harder than for a single-core processor because the state is a function of the relative speeds of the programs or threads running on the different cores, which is hard to estimate without detailed simulation. | 如前所述，Barr和Asanovic[10]提出了一种MSI检查点技术来重建多处理器系统中采样单元开始时的目录状态。多处理器的状态重建比单核处理器要困难，因为状态是运行在不同核上的程序或线程的相对速度的函数，如果没有详细的仿真，很难估计。 |
| Whereas estimating performance of a multi-threaded workload through sampled simulation is a complex problem, estimating overall system performance for a set if independent threads or programs is much simpler. Ekman and Stenström [54] observed that the variability on overall system throughput is smaller than the per-thread performance variability when running multiple independent threads concurrently. The intuitive explanation is that there is a smoothing effect of different threads executing high-IPC and low-IPC phases simultaneously. As such, if one is interested in overall system throughput, a relatively small sample size will be enough to obtain accurate average performance estimates and performance bounds; Ekman and Stenström experimentally verify that a factor N fewer sampling units are needed when simulating a system with N cores compared to single-core simulation.This is true only if the experimenter is interested in average system throughput only. If the experimenter is interested in the performance for individual programs, he/she will need to collect more sampling units. In addition, this smoothing effect assumes that the various threads are independent. This is the case, for example, in commercial transaction-based workloads where transactions, queries and requests arrive randomly, as described by Wenisch et al. [190]. | 虽然通过采样仿真估计多线程工作负载的性能是一个复杂的问题，但如果是一组独立线程或程序，则估计系统的整体性能要简单得多。Ekman和Stenström[54]观察到，当并发运行多个独立线程时，总体系统吞吐量的可变性要小于每个线程的性能可变性。直观的解释是，不同线程同时执行高IPC和低IPC阶段会产生平滑效果。因此，如果一个人对整个系统吞吐量感兴趣，相对较少的样本量将足以获得准确的平均性能估计和性能边界；Ekman和Stenström通过实验验证，与单核仿真相比，在仿真个处理器核心的系统时，所需的采样单元减少了倍。只有当实验者只对平均系统吞吐量感兴趣时，这才成立。如果实验人员对个别程序的性能感兴趣，需要收集更多的抽样单位。此外，这种平滑效果假定各个线程是独立的。这就是Wenisch等人所描述的情况[190]，例如，在基于商业事务的工作负载中，事务、查询和请求随机到达。 |

|  |  |
| --- | --- |
| CHAPTER 7 Statistical Simulation | 第7章 统计仿真 |
| Statistical modeling has a long history. Researchers typically employ statistical modeling to generate synthetic workloads that serve as proxies for realistic workloads that are hard to capture. For example, collecting traces of wide area networks (WAN) or even local area networks (LAN) (e.g., a cluster of machines) is non-trivial, and it requires a large number of disks to store these huge traces. Hence, researchers often resort to synthetic workloads (that are easy to generate) to exhibit the same characteristics (in a statistical sense) as the real network traffic. As another example, researchers studying commercial server systems may employ statistical workload generators to generate load for the server systems under study. Likewise, researchers in the area of interconnection networks frequently use synthetic workloads in order to evaluate a network topology and/or router design across a range of network loads. | 统计模型历史悠久。研究者一般利用统计模型来产生合成负载，作为不易捕捉的实际负载的代表。比如，统计广域网（WAN）或局域网（LAN）的记录是不容易获得的，需要大量的硬盘来存储这些巨大的记录。因此，研究者经常求助于合成负载（容易生成）来展示与实际网络流量相同的特征（在统计意义上）。另一个例子，研究商用服务器系统的研究者可以使用统计负载产生器为所研究的服务器系统生成负载。同样，互联网络领域的研究者经常使用合成负载来产生一系列网络负载，评估网络拓扑和路由器设计。 |
| Synthetic workload generation can also be used to evaluate processor architectures. For example, Kumar and Davidson [111] used synthetically generated workloads to evaluate the performance of the memory subsystem of the IBM 360/91; the motivation for using synthetic workloads is that they enable investigating the performance of a computer system as a function of the workload characteristics. Likewise, Archibald and Baer [3] use synthetically generated address streams to evaluate cache coherence protocols. The paper by Carl and Smith [23] renewed recent interest in synthetic workloads for evaluating modern processors and coined the term ‘statistical simulation’. The basic idea of statistical simulation is to collect a number of program characteristics in the form of distributions and then generate a synthetic trace from it that serves as a proxy for the real program. Simulating the synthetic trace then yields a performance estimate for the real program. Because the synthetic trace is much shorter than the real program trace, simulation is much faster. Several research projects explored this idea over the past decade [13; 50; 87; 149; 152]. | 合成负载生成还可以用来评估处理器架构。比如， Kumar和Davidson[111]利用合成生成负载来评估IBM 360/91的存储子系统的性能；使用合成负载的动机是，合成负载能够将计算机系统的性能作为工作负载特征的函数。同样， Archibald和Baer[3]利用合成产生的地址流来评估缓存一致性协议。 Carl和 Smith的论文[23]重新燃起了最近对用于评估现在处理器的合成工作负载的兴趣，并创造了“统计仿真”一词。统计仿真的基本想法是，以统计分布的形式收集一系列程序特征，然后根据程序特征产生合成序列，作为实际程序的代表。仿真合成序列可以读到实际程序的性能估计值。因为合成序列比实际程序序列短很多，仿真更快。过去十年，一些研究项目探索了这样的想法[13; 50; 87; 149; 152]。 |
| 7.1 METHODOLOGY OVERVIEW | 7.1 方法学简介 |
| Figure 7.1 illustrates the general framework of statistical simulation, which consists of three major steps. The first step collects a number of execution characteristics for a given workload in the form of distributions, hence the term statistical profiling. The statistical profile typically consists of a set of characteristics that are independent of the microarchitecture (e.g., instruction mix, instruction dependencies, control flow behavior) along with a set of microarchitecture-dependent characteristics (typically locality events such as cache and branch prediction miss rates). Statistical profiling is done only once and is done relatively quickly through specialized simulation, which is much faster (typically more than one order of magnitude) compared to cycle-accurate simulation.The second step generates a synthetic trace based from this statistical profile.The characteristics of the trace reflect the properties in the statistical profile and thus the original workload,by construction.Finally,simulating the synthetic trace on a simple trace-driven statistical simulator yields performance numbers. The hope/goal is that, if the statistical profile captures the workload’s behavior well and if the synthetic trace generation algorithm is able to translate these characteristics into a synthetic trace, then the performance numbers obtained through statistical simulation should be accurate estimates for the performance numbers of the original workload. | 图7.1描述了统计仿真的通用框架，包括三个主要的步骤。第一个步骤，按照统计分布的形式，从给定的负载收集一系列执行特性，因此，这个步骤称为统计分析。统计分析一般包括一系列与微架构无关的特征（例如，指令混合特性，指令依赖，控制流行为）和一系列与微架构有关的特征（同上是局部性事件，如缓存和分支预测错误率）。统计特性只进行一次，并且通过专门的仿真快速完成。与周期精确的仿真相比，这个仿真要快很多（通常超过一个数量级）。第二个步骤，根据统计分析产生合成记录。记录的特征反映统计分析的属性，从而反映了原始工作负载。最后，在简化的记录驱动的统计仿真器上仿真合成记录，得到性能数字。这里的希冀和目标是，如果统计分析很好地捕获了负载的行为，而且如果合成记录的生成算法能够将这些特征翻译为一个合成记录，那么通过统计仿真获得的性能数字应该是原始工作负载性能数字的准确估计。 |
|  | |
| Figure 7.1: Statistical simulation framework. | 图7.1 统计仿真框架。 |
| The key idea behind statistical simulation is that capturing a workload’s execution behavior in the form of distributions enables generating short synthetic traces that are representative for long-running real-life applications and benchmarks. Several researchers have found this property to hold true: it is possible to generate short synthetic traces with on the order of a few millions of instructions that resemble workloads that run for tens to hundreds of billions of instructions — this implies a simulation speedup of at least four orders of magnitude. Because the synthetic trace is generated based on distributions, its performance characteristics quickly converge, typically after one million (or at most a few millions) of instructions. And this is obviously where the key advantage lies for statistical simulation: it enables predicting performance for long-running workloads using short running synthetic traces. This is likely to speed up processor architecture design space exploration substantially. | 统计仿真背后的核心想法是，按照统计分布获取负载的执行行为，能够产生简短的合成负载，代表长时间运行的真实的应用和测试程序。一些研究者已经证明这种观点是正确的：可以产生在几百万条指令数量级的较短的合成负载，这些合成负载与数千亿条指令的负载相类似——这意味着仿真加速至少4个数量级。因为合成负载是按照概率分布产生的，其性能特性会很快收敛，大约在一百万（或至多一百万）条指令之后。显然，这是统计仿真的关键优势：统计仿真可以利用短运行时间的负载来预测长运行时间负载的性能。这很可能大大加快处理器架构设计空间的探索。 |
| 7.2 APPLICATIONS | 7.2 应用 |
| Statistical simulation has a number of potential applications. | 统计仿真有许多潜在的应用。 |
| Designspaceexploration. The most obvious application for statistical simulation is processor design space exploration. Statistical simulation does not aim at replacing detailed cycle-accurate simulation, primarily because it is less accurate — e.g., it does not model cache accesses along mispredicted paths, it simulates an abstract representation of a real workload, etc., as we will describe later. Rather, statistical simulation aims at providing a tool that enables a computer architect to quickly make high-level design decisions, and it quickly steers the design exploration towards a region of interest, which can then be explored through more detailed (and thus slower) simulations. Steering the design process in the right direction early on in the design cycle is likely to reduce the overall design process and time to market. In other words, statistical simulation is to be viewed of as a useful complement to the computer architect’s toolbox to quickly make high-level design decisions early in the design cycle. | 设计空间探索。统计仿真最明显的应用是处理器设计空间探索。统计仿真的目标不是替换详细的时钟精确仿真，主要是因为它不精确——比如，统计仿真不能建模预测错误通路上的缓存访问。如后文所述，统计仿真可以仿真真实负载的抽象描述。相反，统计仿真旨在提供一种能够让计算机架构师快速进行高层设计决策的工具，并快速将设计探索引向感兴趣的领域，然后通过更详细的（因此更慢）仿真来探索该领域。换句话说，统计仿真被视为计算机架构师工具箱中的有效组件，在设计周期早期快速进行高层设计决策。 |
| Workloadspaceexploration. Statistical simulation can also be used to explore how program characteristics affect performance. In particular, one can explore the workload space by varying the various characteristics in the statistical profile in order to understand how these characteristics relate to performance. The program characteristics that are part of the statistical profile are typically hard to vary using real benchmarks and workloads, if at all possible. Statistical simulation, on the other hand, allows for easily exploring this space. Oskin et al. [152] provide such a case study in which they vary basic block size, cache miss rate, branch misprediction rate, etc. and study its effect on performance. They also study the potential of value prediction. | 负载空间探索。统计仿真还可以用来探索程序特征如何影响性能。特别的，为了理解负载特性和性能的关系，可以通过改变统计分析中的不同特性参数来探索负载。程序特性作为统计分布的一部分，一般很难使用真实的测试程序和负载来改变，如果可能。另一方面，统计仿真允许很容易的探索负载空间。Oskin等[152]进行了示例研究，他们改变基本块大小、缓存未命中率、分支预测错误率等参数，研究参数对性能的影响。他们还研究了值预测的潜力。 |
| Stresstesting. Taking this one step further, one can use statistical simulation for constructing stressmarks, or synthetic benchmarks that stress the processor for a particular metric, e.g., max power consumption, max temperature, max peak power, etc. Current practice is to manually construct stressmarks which is both tedious and time-consuming. An automated stressmark building framework can reduce this overhead and cost. This can be done by integrating statistical simulation in a tuning framework that explores the workload space (by changing the statistical profile) while searching for the stressmark of interest. Joshi et al. [101] describe such a framework that uses a genetic algorithm to search the workload space and automatically generate stressmarks for various stress conditions. | 压力测试。更进一步，统计仿真可以用来构造为了特定度量指标对处理器施压的压力标记或合成测试程序，比如最大功耗、最高温度、最大漏功率等。目前的做法是手动构建压力标记，这既繁琐又耗时。自动的压力标记构建框架可以减少这方面的开销和代价。在搜索感兴趣领域的压力标记时，可以通过将统计仿真集成到探索负载空间的优化框架（通过改变统计分析的方式）中实现。Joshi等[101]描述了这样的框架，框架使用一个通用算法搜索负载空间，并自动为不同的压力条件产生压力标记。 |
| Program behavior characterization. Another interesting application for statistical simulation is program characterization. When validating the statistical simulation methodology in general and the characteristics included in the statistical profile more in particular, it becomes clear which program characteristics must be included in the profile for attaining good accuracy. That is, this validation process distinguishes program characteristics that influence performance from those that do not. | 程序行为特性。统计仿真的另一个有意思的应用是程序特征。当一般性验证统计仿真方法时以及特定验证统计分析中包含的特性时，可以清楚地知道，哪些程序特征必须包含在配置文件中才能获得良好的准确性。也就是说，此验证过程区分影响性能的程序特征与不影响性能的程序特征。 |
| Workload characterization. Given that the statistical profile captures the most significant program characteristics, it can be viewed of as an abstract workload model or a concise signature of the workload’s execution behavior [46]. In other words, one can compare workloads by comparing their respective statistical profiles. | 负载特性。鉴于统计分析捕获了最重要的程序特征，因此可以将统计分析视为负载执行行为的抽象模型或者简要说明。换句话说，可以通过比较统计分析来比较工作负载。 |
| Large system evaluation. Finally, current state-of-the-art in statistical simulation addresses the time-consuming simulation problem of single-core and multi-core processors. However, for larger systems containing several processors, such as multi-chip servers, clusters of computers, datacenters, etc., simulation time is even a bigger challenge. Statistical simulation may be an important and interesting approach for such large systems. | 大系统评估。当前最先进的统计仿真技术解决了单核和多核处理器的仿真问题。然而，对于更大的包含多个处理器的系统，比如多芯片服务器，计算机集群，数据中心等，仿真时间甚至面临更大的问题。对于这样的大型系统，统计仿真可能是一种重要而且吸引人的方法。 |
| 7.3 SINGLE-THREADED WORKLOADS | 7.3 单线程负载 |
| We now describe the current state-of-the-art in statistical simulation, and we do that in three steps with each step considering a progressively more difficult workload type, going from single-threaded, to multi-program and multi-threaded workloads. This section considers single-threaded workloads; the following sections discuss statistical simulation for multi-program and multi-threaded workloads. Figure 7.2 illustrates statistical simulation for single-threaded workloads in more detail. | 现在，我们描述当前最先进的统计仿真方法。我们分成三步进行，每个步骤都考虑了越来越困难的工作负载类型，从单线程负载到多程序和多线程负载。本节考虑单线程负载；后续的章节讨论多程序和多线程负载的统计仿真。图7.2详细描述了单线程负载的统计仿真。 |
| 7.3.1 STATISTICAL PROFILING | 7.3.1 统计分析 |
| Statistical profiling takes a workload (program trace or binary) as input and computes a set of characteristics. This profiling step is done using a specialized simulator. This could be a specialized trace-driven simulator, a modified execution-driven simulator, or the program binary could even be instrumented to collect the statistics during a program run (see Chapter 5 for a discussion on different forms of simulation). The statistical profile captures the program execution for a specific program input, i.e., the trace that serves as input was collected for a specific input, or the (instrumented) binary is given a specific input during statistics collection. Statistical profiling makes a distinction between characteristics that are microarchitecture-independent versus microarchitecture-dependent. | 统计分析以负载（程序流或二进制可执行文件）为输入，计算一组特征。分析步骤使用专门的仿真器完成。这可能是一个专门的流驱动仿真器，或者一个修改后的执行驱动仿真器，或者二进制可执行文件甚至也可以在程序运行期间收集统计信息（见第5章关于不同类型仿真的讨论）。统计分析捕获特定程序输入的程序执行，即，为特定输入收集用作输入的记录，或者在统计信息收集期间为（检测的）二进制文件提供特定输入。统计分析区分与微架构无关的特性与微架构相关的特性。 |
| Microarchitecture-independent characteristics. A minimal statistical model would collect the instruction mix, i.e., percentage loads, stores, branches, integer operations, floating-point operations, etc. in the dynamic instruction stream. The number of instruction types is typically limited — the goal is only to know to which functional unit to steer the instruction during simulation. In addition, it includes a distribution that characterizes the inter-instruction dependences (through both registers and memory). Current approaches are typically limited to modeling read-after-write (RAW) dependencies, i.e., they do not model write-after-write (WAW) and write-after-read (WAR) dependencies, primarily because these approaches are targeting superscalar out-of-order processor architectures in which register renaming removes WAW and WAR dependences (provided that sufficient rename registers are available) and in which load bypassing and forwarding is implemented. Targeting statistical simulation towards simpler in-order architectures is likely to require modeling WAR and WAW dependences as well. | 微架构无关的特性。最小化的统计模型会收集指令混合特性，即，load/store/分支/整形操作/浮点操作等的比例。指令类型的数量是有限——目的只是获知，在仿真过程中，哪个仿真单元承载了指令的压力。另外，统计模型包含了描述指令间依赖（通过寄存器和存储）的分布。目前的方法一般限制为建模读后写（RAW）依赖，即，不建模写后写（WAW）和写后读（WAR）依赖。主要是因为，这些方法的目标是超标量乱序处理器架构，这种架构通过寄存器重命名消除WAW和WAR依赖（前提是提供足够的重命名寄存器可用），而且架构实现了load bypassing和forwarding。面向简单的顺序架构的统计仿真可能还需要建模WAR和WAW依赖。 |
|  | |
| Figure 7.2: Statistical simulation framework for single-threaded workloads. | 图7.2：单线程负载的统计仿真框架 |
| Inter-instruction dependencies can either be modeled as downstream dependences (i.e., an instruction produces a value that is later consumed) or upstream dependences (i.e., an instruction consumes a value that was produced before it in the dynamic instruction stream). Downstream dependences model whether an instruction depends on the instruction immediately before it, two instructions before it, etc. A problem with downstream dependencies occurs during synthetic trace generation when an instruction at the selected dependence distance does not produce a value, e.g., a store or a branch does not produce a register value. One solution is to go back to the distribution and try again [47; 152]. Upstream dependences model have a complementary problem. Upstream dependences model whether an instruction at distance d in the future will be dependent on the currently generated instruction. This could lead to situations where instructions have more (or less) incoming dependences than they have input operands.One solution is to simply let this happen [149]. | 指令间依赖可以建模为下游依赖关系（即，指令提供后面需要消费的值）或者上游依赖关系（即，指令使用之前在动态指令流中产生的值）。下游依赖建模，指令是否依赖于紧挨着的前一条指令，或者依赖于前两条指令等。在生成合成记录的阶段，当选定依赖距离处的指令不生成值的时候，例如，store或者分支不提供寄存器值，下游依赖关系就发生了问题。一种解决方法是返回统计分布并且重新尝试[47; 152]。上游依赖模型有互补的问题。上游依赖建模，在未来距离为的指令将会依赖于当前产生的指令。这可能会导致，指令比他们的输入操作数具有更多（或更少）的输入依赖。一种解决方法是，简单地允许这种情况发生。 |
| In theory, inter-instruction dependence distributions have infinite size (or at least as large as the instruction trace size) because long dependence distances may exist between instructions. Fortunately, the distributions that are stored on disk as part of a statistical profile can be limited in size because long dependence distances will not affect performance.For example, a RAW dependence between two instructions that are further away from each other in the dynamic instruction stream than the processor’s reorder buffer size is not going to affect performance anyway, so there is no need to model such long dependences. Hence, the distribution can be truncated. | 理论上，指令间依赖分布具有无穷大的大小（或者至少和指令记录大小一样大），因为指令之间可能存在长依赖距离。幸好，因为长依赖距离不影响性能，作为统计分析的一部分存储在硬盘上的概率分布可以限制大小。比如，在动态指令流中的很远的两条指令之间的RAW依赖，指令距离比处理器的重排序缓存大小还要大，那么不会影响性能，所以不需要建模如此的长距离。因此，分布可以截断。 |
| The initial statistical simulation methods did not model any notion of control flow behavior [23;47]:the program characteristics in the statistical profile are simply aggregate metrics,averaged across all instructions in the program execution. Follow-on work started modeling control flow behavior in order to increase the accuracy of statistical simulation. Oskin et al. [152] propose the notion of a control flow graph with transition probabilities between the basic blocks. However, the program characteristics were not correlated to these basic blocks, i.e., they use aggregate statistics, and hence did not increase accuracy much. Nussbaum and Smith [149] correlate program characteristics to basic block size in order to improve accuracy.They measure a distribution of the dynamic basic block size and compute instruction mix, dependence distance distributions and locality events (which we describe next) conditionally dependent on basic block size. | 最初的统计仿真方法没有建模任何控制流行为的概念[23;47]：统计分析只是简化的聚合指标，对程序执行过程中的所有指令的特性取平均。后续工作开始建模控制流行为以增加统计模型的准确性。Oskin等[152]提出控制流图的概念，并且基本块之间具有转移概率。然而，程序特征与这些基本块没有关系，即，他们使用聚合度量指标，因此没有明显增加准确度。Nussbaum和Smith [149]将程序特征与基本块大小关联，从而增加准确度。他们测量动态基本块的大小的分布，然后计算指令混合特性分布、依赖距离分布和局部性时间（我们在后面描述）有条件地依赖于基本块大小。 |
| Eeckhout et al. [45] propose the statistical flow graph (SFG) which models the control flow using a Markov chain; various program characteristics are then correlated to the SFG. The SFG is illustrated in Figure 7.3 for the AABAABCABC example basic block sequence. In fact, this example shows a first-order SFG because it shows transition probabilities between nodes that represent a basic block along with the basic block executed immediately before it. (Extending towards higherorder SFGs is trivial.) The nodes here are A|A, B|A, A|B, etc.: A|B refers to the execution of basic block A given that basic block B was executed immediately before basic block A. The percentages at the edges represent the transition probabilities between the nodes. For example, there is a 33.3% and 66.6% probability to execute basic block A and C, respectively, after having executed basic block A and then B (see the outgoing edges from B|A node). The idea behind the SFG and the reason why it improves accuracy is that, by correlating program characteristics along the SFG, it models execution path dependent program behavior. | Eeckhout等[45]提出用马尔科夫链建模控制流的统计流程图（SFG）；变化的程序特征与SFG有关。示例基本块序列AABAABCABC如图7.3所示。实际上，这个示例展示了一个一阶SFG，因为节点之间的转移概率，表示基本块在紧挨着的前一个基本块之后。（扩展到高阶SFG是平凡的。）这里的节点包括：A|A，B|A，A|B等：A|B指的是，基本块A在基本块B后立即执行。边上的百分比表示节点之间的转移概率。例如，在先后执行了基本块A和B之后，按照33.3%和66.6%的概率分别执行基本块A和C（见节点B|A的输出边）。SFG背后的想法，以及其提高准确度的原因是，通过关联SFG和程序特性，SFG建模指令路径依赖的程序行为。 |
| All of the characteristics discussed so far are independent of any microarchitecture-specific organization. In other words, these characteristics do not rely on assumptions related to processor issue width, window size, number of ALUs, instruction execution latencies, etc. They are, therefore, called microarchitecture-independent characteristics. | 当目前为止，讨论的所有特性都与微架构无关。换句话说，这些特性不依赖于对于处理器发射宽度、窗口大小、ALU数量、指令执行延迟等的假设。因此，他们被称为微架构无关特性。 |
|  | |
| Figure 7.3: An example statistical flow graph (SFG). | 图7.3: 统计流程图（SFG）示例 |
| Microarchitecture-dependent characteristics. In addition to the above characteristics, we also measure a number of characteristics that are related to locality events, such as cache and branch predictor miss rates. Locality events are hard to model in a microarchitecture-independent way. Therefore, a pragmatic approach is taken and characteristics for specific branch predictors and specific cache configurations are computed through specialized cache and branch predictor simulation. Note that although this approach requires the simulation of the complete program execution for specific branch predictors and specific cache structures, this does not limit its applicability. In particular, multiple cache configuration can be simulated in parallel using a single-pass algorithm [83; 135]. An alternative approach is to leverage cache models that predict miss rates. For example, Berg and Hagersten [14] propose a simple statistical model that estimates cache miss rates for a range of caches based on a distribution of reuse latencies (the number of memory references between two references to the same memory location). In other words, rather than computing the miss rates through specialized simulation, one could also use models to predict the miss rates. While this may reduce the time needed to collect the statistical profile, it may come at the cost of increased inaccuracy. | 微架构有关特性。除了上面这些特性，我们还测量一些与局部性事件有关的特性。比如缓存未命中率和分支预测错误率。局部性事件很难用微架构无关的方式建模。因此，使用的方法是，通过专门的缓存和分支预测仿真器计算特定的分支预测器和缓存配置的特性。注意，虽然这种方法需要在特定分支预测器和特定缓存结构上完成程序执行的仿真，但是并不影响这种方法的适用性。特别是，可以使用单通道算法并行仿真多个缓存配置[83; 135]。另一种方法是利用缓存模型预测未命中率。例如，Berg和Hagersten[14]提出一个简单的统计模型，根据重用延迟的分布（对于相同内存地址的两次访问之间的全部内存访问次数）估计一组缓存的缓存未命中率。换句话说，除了通过专门仿真器计算未命中率，也可以利用模型来预测未命中率。在减少收集统计分析的时间的同时，这种方法也会付出降低准确度的代价。 |
| The locality events captured in the initial frameworks are fairly simple [23; 47; 149; 152]. For the branches, it comprises (i) the probability for a taken branch, which puts a limit on the number of taken branches that are fetched per clock cycle; (ii) the probability for a fetch redirection, which corresponds to a branch target misprediction in conjunction with a correct taken/not-taken prediction for conditional branches — this typically results in (a) bubble(s) in the pipeline; and (iii) the probability for a branch misprediction: a BTB miss for indirect branches and a taken/nottaken misprediction for conditional branches. The cache statistics typically consist of the following probabilities: (i) the L1 instruction cache miss rate, (ii) the L2 cache miss rate due to instructions, (iii) the L1 data cache miss rate, (iv) the L2 cache miss rate due to data accesses only, (v) the instruction translation lookaside buffer (I-TLB) miss rate, and (vi) the data translation lookaside buffer (D-TLB) miss rate. (Extending to additional levels of caches and TLBs is trivial.) | 在最初的框架中，捕获的局部性事件是很简单的[23; 47; 149; 152]。对于分支，包括(i)跳转分支的概率，限制每个周期取指的跳转指令；（ii）取指重定向的概率，对应于条件分支的跳转/不跳转预测正确但是分支目标地址的预测错误——典型会导致流水线上的气泡；(iii)分支预测错误的概率：间接分支的BTB未命中，以及条件分支跳转/不跳转预测错误。缓存统计指标一般包括这些概率：(i)L1指令缓存未命中率，(ii)指令访问L2缓存未命中率；(iii)L1数据缓存未命中率；(iv)数据访问L2缓存未命中率；(v)指令地址转换查找缓存（I-TLB）未命中率；和(vi)数据地址转换查找缓存（D-TLB）未命中率。（扩展到额外层次的缓存和TLB是平凡的。） |
| At least two major problems with these simple models occur during synthetic trace simulation. First, it does not model delayed hits or hits to outstanding cache lines — there are only hits and misses. The reason is that statistical profiling only sees hits and misses because it basically is a (specialized) functional simulation that processes one memory reference at a time, and it does not account for timing effects. As a result, a delayed hit is modeled as a cache hit although it should see the latency of the outstanding cache line. Second, it does not model the number of instructions in the dynamic instruction stream between misses. However, this has an important impact on the available memory-level parallelism (MLP). Independent long-latency load misses that are close enough to each other in the dynamic instruction stream to make it into the reorder buffer together, potentially overlap their execution, thereby exposing MLP. Given the significant impact of MLP on out-oforder processor performance [31; 102], a performance model lacking adequate MLP modeling may yield large performance prediction errors. Genbrugge and Eeckhout [71] address these problems by modeling the cache statistics conditionally dependent on the (global) cache hit/miss history — by doing so, they model the correlation between cache hits and misses which enables modeling MLP effects. They also collect cache line reuse distributions (i.e., number of memory references between two references to the same cache lines) in order to model delayed hits to outstanding cache lines. | 在合成记录仿真的时候，这些简单的模型至少导致两个主要问题。首先，没有建模延迟的命中或对于未完成的缓存行的命中——只有命中和未命中。原因在于，统计分析只能看到命中和未命中，由于统计分析使用的是一次处理一个内存访问的（专门的）功能仿真模型，没有考虑时序作用。结果是，尽管应该看到未完成的缓存行的延迟，延迟的命中还是建模为命中。第二，没有建模在未命中之间的，动态指令流上的指令数。然而，这对于可用的存储级并行（MLP）有很重要的影响。相互独立的长延迟未命中load，如果他们在动态指令流上足够近，使得他们一起进入重排序缓存，很可能相互覆盖他们的指令，从而利用MLP。鉴于MLP对于乱序处理器性能的显著影响[31; 102]，缺少充足的MLP模型的性能模型会引起较大的性能预测错误。Genbrugge和Eeckhout [71]通过将缓冲统计指标建模为条件性依赖于（全局）缓存命中/未命中历史，从而满足这个问题——通过这样做，他们建模缓存命中和未命中之间的关联，使能对于MLP影响的建模。他们还收集缓存块重用分布（即，对于同一缓存行的两次仿问之间的存储访问次数），从而建模未完成缓存块的延迟命中。 |
| 7.3.2 SYNTHETIC TRACE GENERATION | 7.3.2 合成记录产生 |
| The second step in the statistical simulation methodology is to generate a synthetic trace from the statistical profile. The synthetic trace generator takes as input the statistical profile and produces a synthetic trace that is fed into the statistical simulator. Synthetic trace generation uses random number generation for generating a number in [0,1]; this random number is then used with the inverse cumulative distribution function to determine a program characteristic, see Figure 7.4. The synthetic trace is a linear sequence of synthetically generated instructions. Each instruction has an instruction type,a number of source operands,an inter-instruction dependence for each register input (which denotes the producer for the given register dependence in case of downstream dependence distance modeling), I-cache miss info, D-cache miss info (in case of a load), and branch miss info (in case of a branch).The locality miss events are just labels in the synthetic trace describing whether the load is an L1 D-cache hit, L2 hit or L2 miss and whether the load generates a TLB miss. Similar labels are assigned for the I-cache and branch miss events. | 统计仿真方法的第二个步骤是根据统计分布产生合成记录。合成记录生成器将统计分析作为输入，并且产生提供给统计仿真器的合成记录。合成记录生成使用随机数生成器产生[0,1]之间的一个数；然后利用这个随机数和逆累计分布函数确定程序特征，如图7.4所示。合成记录是一个生成指令的线性序列。每条指令提供指令类型，一些源操作数和每个输入寄存器的指令间依赖（以下游依赖举例，表示给定寄存器依赖的生产者），指令缓存未命中信息，数据缓存未命中信息（如果是load），和分支预测错误信息（如果是分支）。局部性未命中事件直接标记在合成记录中，描述load是否L1数据缓存命中，L2命中或L2未命中，以及load是否产生TLB未命中。类似的标记可以标记指令缓存和分支未命中事件。 |
|  | |
| Figure 7.4: Synthetic trace generation: determining a program characteristic through random number generation. | 图7.4: 合成记录产生：通过随机数生成确定程序特性。 |
| 7.3.3 SYNTHETIC TRACE SIMULATION | 7.3.3 合成记录仿真 |
| Synthetic trace simulation is very similar to trace-driven simulation of a real program trace. The simulator takes as input a (synthetic) trace and simulates the timing for each instruction as it moves through the processor pipeline: a number of instructions are fetched, decoded and dispatched each cycle; instructions are issued to a particular functional unit based on functional unit availability and depending on instruction type and whether its dependences have been resolved.When a mispredicted branch is fetched, the pipeline is filled with instructions (from the synthetic trace) as if they were from the incorrect path.When the branch gets executed, the synthetic instructions down the pipeline are squashed and synthetic instructions are re-fetched (and now they are considered from the correct path). In case of an I-cache miss, the fetch engine stops fetching instructions for a number of cycles. The number of cycles is determined by whether the instruction causes an L1 I-cache miss, an L2 cache miss or a D-TLB miss. For a load, the latency will be determined by whether this load is an L1 D-cache hit, an L1 D-cache miss, an L2 cache miss, or a D-TLB miss. For example, in case of an L2 miss, the access latency to main memory is assigned. For delayed hits, the latency assigned is the remaining latency for the outstanding cache line. | 合成记录仿真与真实程序的记录驱动仿真非常相似。仿真器以（合成）记录为输入，仿真每一条指令沿着处理器流水线移动的时序：每个周期一些指令被取指、译码和分发；根据功能单元占用情况、指令类型、以及依赖关系是否解除，指令被发射给特定的功能单元。当发生分支预测错误的时候，流水线被指令（来自合成记录）填充，如同这些指令来自不正确路径。当分支执行的时候，流水线中的合成指令被放弃，合成指令重新取指（现在它们被认为来自正确路径）。如果发生指令缓存缺失，取指引擎停止取指一些周期。周期数取决于指令是否引起L1指令缓存未命中还是L2缓存未命中，或者指令TLB未命中。对于load，延迟取决于load是否引起L1数据缓存未命中，L1数据缓存未命中、L2缓存未命中、或者数据TLB未命中。例如，如果发生L2未命中，需要设置访问主存的延迟。对于延迟命中，设置的延迟是未完成的缓存块的剩余延迟。 |
| Although synthetic trace simulation is fairly similar to trace-driven simulation of a real program trace,the simulator itself is less complex because there are many aspects a trace-driven simulator needs to model that a synthetic trace simulator does not need to model. For example, the instruction decoder in the synthetic trace simulator is much less complex as it needs to discern only a few instruction types. It does not need to model register renaming because the RAW dependencies are already part of the synthetic trace. Likewise, the synthetic trace simulator does not need to model the branch predictor and the caches because the locality events are modeled as part of the synthetic trace. The fact that the synthetic trace simulator is less complex has two important implications. It is easier to develop, and, in addition, it runs faster, i.e., evaluation time is shorter. This is part of the trade-off that statistical simulation offers: it reduces evaluation time as well as development time while incurring some inaccuracy. | 尽管合成记录仿真与真实程序的记录驱动仿真非常相似，合成记录仿真器本身还是要更简化，因为很多记录驱动仿真器需要建模的特性，合成记录仿真器不需要建模。例如，合成序列仿真器中的指令译码器简化很多，因为它只需要辨别几种指令类型。合成序列仿真器不需要建模寄存器重命名，因为RAW依赖关系已经是合成记录的一部分。类似的，合成序列仿真器不需要建模分支预测和缓存，因为局部性事件已经建模为合成记录的一部分。合成记录仿真器更加简化的情况，有两个重要的影响。合成记录仿真器更容易开发，而且运行更快，即，评估时间更短。这是统计仿真提供的折中：统计仿真减少了评估时间和开发时间，同时产生一些不准确性。 |
| 7.4 MULTI-PROGRAM WORKLOADS | 7.4 多程序负载 |
| The approach to statistical simulation as described above models cache behavior through miss statistics. Although this is appropriate for single-core processors, it is inadequate for modeling multicore processors with shared resources, such as shared L2 and/or L3 caches, shared off-chip bandwidth, interconnection network and main memory. Co-executing programs on a chip multiprocessor affect each other’s performance through conflicts in these shared resources, and the level of interaction between co-executing programs is greatly affected by the microarchitecture — the amount of interaction can be (very) different across microarchitectures. Hence, cache miss rates profiled from single-threaded execution are unable to model conflict behavior in shared resources when co-executing multiple programs on multicore processors. Instead, what we need is a model that is independent of the memory hierarchy so that conflict behavior among co-executing programs can be derived during multicore simulation of the synthetic traces. | 上面描述的统计仿真方法，通过未命中统计来建模缓存行为。尽管这种方法对于单核心处理器是合适的，但对于使用共享资源（例如共享 L2 和/或 L3 高速缓存、共享片外带宽、互连网络和主存）的多核处理器进行建模是不够的。在一个片上多处理器中共同执行的程序通过这些共享资源中的冲突相互影响性能，而且共同执行程序之间的相互作用的程度很大程度上受到微架构的影响——不同微架构下，相互影响的程度非常不同。因此，从单线程执行获取的缓存未命中率并不能建模，当多程序在多核处理器上共同执行的时候，共享资源中的冲突行为。取而代之，我们需要一个与存储层次无关的模型，从而可以在合成记录的多核仿真期间产生出共同执行程序之间的冲突行为。 |
| Genbrugge and Eeckhout [72] propose two additional program characteristics, namely, the cache set profile and the per-set LRU stack depth profile for the largest cache of interest (i.e., the largest cache the architect is interested in during design space exploration). The cache set profile basically is a distribution that captures the fraction of accesses to each set in the cache. The per-set LRU stack depth profile stores the fraction of accesses to each position in the LRU stack. These program characteristics solve two problems: it makes the cache locality profile largely microarchitectureindependent — it only depends on cache line size — which enables estimating cache miss rates for smaller sized caches (i.e., caches with smaller associativity and/or fewer sets). (The underlying mechanism is very similar to the MSI checkpointing mechanism explained in Section 6.3, see also Figure 6.5.) In addition, the cache set and per-set LRU stack depth profiles allows for modeling conflict behavior in shared caches. The only limitation though is that synthetic trace simulation requires that the shared cache(s) is (are) simulated in order to figure out conflict misses among co-executing programs/threads. | 因为对最大缓存容量的研究兴趣（即在设计空间探索阶段，架构师感兴趣的最大缓存），Genbrugge和Eeckhout [72]提供两种额外的程序特征，称为缓存组相连分析（cache set profile）和组分LRU栈深度分析（per-set LRU stack depth profile）。缓存组相连分析基本上是捕捉缓存中每一个组的访问比例的分布。组分LRU栈深度分析保存LRU栈每一个位置的访问比例。这些程序特征解决两个问题：它是的缓存局部性分析很大程度上与微架构无关——只取决于缓存块大小——使得能够对更小缓存（即，缓存具有更小的组相连和/或更少的组）的未命中率进行估计。（其基本机制与第 6.3 节中介绍的 MSI 检查点机制非常相似，请参见图 6.5。）另外，缓存组相连分析和组分LRU栈深度分析使能建模共享缓存中的模型冲突行为。唯一的限制是合成记录仿真要求共享缓存是经过仿真的，从而找出共同执行的程序/线程的冲突。 |
| A critical issue to multicore processor performance modeling is that the synthetic trace should accurately capture the program’s time-varying execution behavior. The reason is that a program’s behavior is greatly affected by the behavior of its co-executing program(s), i.e., the relative progress of a program is affected by the conflict behavior in the shared resources [24; 188]. In particular, cache misses induced through cache sharing may slow down a program’s execution, which in its turn may result in different sharing behavior. A simple but effective solution is to model a program’s timevarying behavior by dividing the program trace in a number of instruction intervals and generating a synthetic mini-trace for each of these instruction intervals [72]. Coalescing the mini-traces then yields a synthetic trace that captures the original program’s time-varying execution behavior. | 对于多核处理器性能建模的关键问题是，合成记录应该准确捕获程序随时序变化的执行行为。这个原因是，程序的行为很大程度上受到他共同执行的程序的行为的影响，即，相关程序的进展被共享资源的冲突行为影响[24; 188]。特别是，通过缓存共享引起的缓存未命中可能会减慢程序的执行速度，进而可能导致不同的共享行为。一种简单但是有效的解决方案是，通过将程序记录分成一系列指令间隔，并为每个指令间隔生成一个合成的迷你记录，从而对程序的时变行为进行建模。合并微型记录生成一个合成记录，用于捕获原始程序的时变执行行为。 |
| 7.5 MULTI-THREADED WORKLOADS | 7.5 多线程负载 |
| Nussbaum and Smith [150] extended the statistical simulation methodology towards symmetric multiprocessor (SMP) systems running multi-threaded workloads. This requires modeling interthread synchronization and communication. More specifically, they model cache coherence events, sequential consistency events, lock accesses and barrier distributions. For modeling cache coherence events and sequential consistency effects, they model whether a store writes to a cache line that it does not own, in which case it will not complete until the bus invalidation has reached the address bus. Also, they model whether a sequence of consecutive stores access private versus shared memory pages. Consecutive stores to private pages can be sent to memory when their input registers are available; consecutive stores to shared pages can only be sent to memory if the invalidation of the previous store has reached the address bus in order to preserve sequential consistency. | Nussbaum和Smith [150]将统计仿真方法学扩展到运行多线程负载的同构多线程处理器（SMP）系统。这需要仿真线程间的同步和通信。更加确切的是，他们仿真了缓存一致性事件、序列一致性事件、锁访问和barrier分布。为了建模缓存一致性事件和序列一致性事件，他们建模了，一个store是否写入它本不占有的缓存块。在这种情况下，store只有到总线失效事件到达地址总线之后才能完成。而且，他们建模了，一系列连续store是否访问私有或者共享存储页面。对于私有页面的连续store，可以在输入寄存器有效的时候发送给存储。对于共享页面的连续store，只有在前一个store的失效事件已经到达地址总线的时候才能发送给存储，从而保证序列一致性。 |
| Lock accesses are modeled through acquire and release instructions in the statistical profile and synthetic trace. More specifically, for an architecture that implements critical sections through load-linked and store-conditional instructions, the load-linked instruction is retried until it finds the lock variable is clear. It then acquires the lock through a store-conditional to the same lock variable. If the lock variable has been invalidated since the load-linked instruction, this indicates that another thread entered the critical section first. Statistical simulation models all instructions executed between the first load-linked — multiple load-linked instructions may need to be executed before it sees the lock variable is clear — and the successful store-conditional as a single acquire instruction. When a thread exits the critical section, it releases the lock by storing a zero in the lock variable through a conventional store instruction; this is modeled through a single release instruction in statistical simulation. A distribution of lock variables is also maintained in order to be able to discern different critical sections and have different probabilities for entering each critical section. During statistical simulation, a random number along with the lock variable distribution then determines which critical section a thread is entering. Separate statistical profiles are computed for code executed outside versus inside critical sections. | 通过统计分析和合成记录中的获取和释放指令对锁访问进行建模。更具体地说，对于通过load-linked和条件store指令实现临界区的体系结构，将load-linked指令只有在它发现锁标量是干净的时候才能退休。如果锁变量自load-linked指令开始已失效，则表示另一个线程首先进入了临界区。统计仿真将在第一个load-linked指令（可能需要执行多个load-linked指令，才能看到锁定变量释放）和成功的条件store指令之间的所有指令仿真为单个获取指令。当线程退出临界区时，它通过常规store指令在锁变量中写入0来释放锁；在统计仿真中，通过单个释放指令建模。还需要维护锁变量的分布，从而能够区分不同的临界区，并且进入每个临界区有不同的概率。在统计仿真中，一个随机数和锁变量分布决定线程进入哪一个临界区。在临界区内部和外部执行代码，分别计算独立的统计分布。 |
| Finally, modeling barrier synchronization is done by counting the number of instructions executed per thread between two consecutive barriers.During synthetic trace generation,the number of instructions between barriers is then scaled down proportionally to the number of instructions executed during detailed execution relative to the number of instructions during statistical simulation. The idea is to scale down the amount of work done between barriers in proportion to the length of the statistical simulation. | 最后，通过每一个线程两个相邻barrier之间执行的指令数建模barrier同步。在生成合成记录的时候，barrier之间的指令数量与详细执行期间执行的指令数量成比例地缩小，相对于统计仿真期间的指令数量。这个想法是按照统计仿真的长度成比例地减少barrier之间完成的工作量。 |
| A limitation of the Nussaum and Smith approach is that many of the characteristics are microarchitecture-dependent; hence, the method requires detailed simulation instead of specialized functional simulation during statistical profiling. Therefore, Hughes and Li [87] propose the concept of a synchronized statistical flow graph (SSFG), which is a function only of the program under study and not the microarchitecture. The SSFG is illustrated in Figure 7.5. Thread T0 is the main thread, and T1 and T2 are two child threads There is a separate SFG for each thread, and thread spawning is marked explicitly in the SSFG, e.g., T1 is sapwaned in node B of T0 and T2 is spawned in node C of T0. In addition, the SSFG also models critical sections and for each critical section which lock variable it accesses. For example, node F in T1 accesses the same critical section as node N in T2; they both access the same lock variable L1. | Nussaum和Smith方法的限制是，很多特性是与微架构有关的；因此，在统计分析时，这种方法需要详细仿真，而不是专用的功能仿真器。所以，Hughes和Li [87]提出来同步统计流程图（SSFG）的概念，作为被研究的程序的函数，而不是微架构的函数。SSFG的示例如图7.5所示。线程T0是主线程，T1和T2是两个子线程。每个线程有单独的SFG，并且在 SSFG 中显式标记了线程生成，例如，T1在线程T0的节点B生成，T2在T0的节点C生成。另外，SSFG还建模了临界区以及每个临界区访问的锁变量。例如，T1的节点F和T2的阶段N访问相同的临界区；他们都访问相同的锁变量L1。 |
|  | |
| Figure 7.5: Synchronized statistical flow graph (SSFG) illustrative example. | 图7.5: 同步统计流程图（SSFG）示例。 |
| 7.6. OTHERWORK IN STATISTICAL MODELING | 7.6. 统计仿真中的其他工作 |
| There exist a number of other approaches that use statistics in performance modeling. Noonburg and Shen [148] model a program execution as a Markov chain in which the states are determined by the microarchitecture and the transition probabilities by the program. This approach works well for simple in-order architectures because the state space is relatively small. However, extending this approach towards superscalar out-of-order architectures explodes the state space and results in a far too complex Markov chain. | 在性能评估中利用统计还存在一些其他的方法。Noonburg和Shen [148]将程序的指令建模成马尔科夫链，其中状态由微架构决定，转移概率由程序决定。这种方法对于简单的顺序处理器工作很好，因为状态空间相对小。然而，将这种方法扩展到超标量乱序架构师扩展了状态空间，结果得到一个复杂很多的马尔科夫链。 |
| Iyengar et al. [91] present SMART , which generates representative synthetic traces based on the concept of a fully qualified basic block. A fully qualified basic block is a basic block along with its context of preceding basic blocks. Synthetic traces are generated by coalescing fully qualified basic blocks of the original program trace so that they are representative for the real program trace while being much shorter.The follow-on work in [90] shifted the focus from the basic block granularity to the granularity of individual instructions. A fully qualified instruction is determined by its preceding instructions and their behavior, i.e., instruction type, I-cache hit/miss, and, if applicable, D-cache and branch misprediction behavior. As a result, SMART makes a distinction between two fully qualified instructions having the same sequence preceding instructions, except that, the behavior may be slightly different, e.g., in one case, a preceding instruction missed in the cache, whereas in the other case it did not. As a result, collecting all these fully qualified instructions during statistical profiling results in a huge amount of data to be stored in memory. For some benchmarks, the authors report that the amount of memory that is needed can exceed the available memory in a machine, so that some information needs to be discarded from the graph. | Iyengar等[91]提出了SMART方法，它基于完全限定的基本块的概念生成了具有代表性的合成记录。完全限定的基本块是基本块及其前面的基本块的上下文。合成记录是通过合并原始程序记录的完全限定的基本块来生成的，因此它们代表了真正的程序记录，同时要短得多。后续工作[90]将重点从基本块粒度转移到单个指令的粒度。完全限定指令由其前面的指令及其行为确定，即指令类型、指令缓存命中/未命中，以及数据缓存和分支预测错误行为（如果适用）。因此，SMART区分了具有相同的序列的两个完全限定的指令，除了行为可能略有不同，例如，在一种情况下，缓存中缺少前面的指令，而在另一种情况下则没有。结果，在统计分析期间收集所有这些完全限定的指令会导致大量数据存储在内存中。对于某些基准测试，作者报告说，所需的内存量可能超过计算机中的可用内存，因此需要从图景中丢弃一些信息。 |
| SMART should not be confused with SMARTS [193; 194], which is a statistical sampling approach, as described in Chapter 6. | SMART不应该与SMARTS[193; 194]混淆，后者是一种统计采样方法，如第6章所述。 |
| Recent work also focused on generating synthetic benchmarks rather than synthetic traces. Hsieh and Pedram [86] generate a fully functional program from a statistical profile. However, all the characteristics in the statistical profile are microarchitecture-dependent, which makes this technique useless for microprocessor design space explorations. Bell and John [13] generate short synthetic benchmarks using a collection of microarchitecture-independent and microarchitecturedependent characteristics similar to what is done in statistical simulation as described in this chapter. Their goal is performance model validation of high-level architectural simulators against RTL-level cycle-accurate simulators using small but representative synthetic benchmarks. Joshi et al. [100] generate synthetic benchmarks based on microarchitecture-independent characteristics only, and they leverage that framework to automatically generate stressmarks (i.e., synthetic programs that maximize power consumption, temperature, etc.), see [101]. | 最近的工作还侧重于生成合成基准而不是合成记录。Hsieh和Pedram[86]从统计分析产生功能完全的程序。然而，统计分析中所有特性都是微架构依赖的，使得这项技术对于微架构设计空间探索没有用处。Bell和John[13]利用一组微架构无关和微架构有关特性产生短的合成基准测试，与本章描述的统计仿真的做法类似。他们的目标是使用小而具有代表性的合成基准，使用RTL级别周期精确仿真器对高级架构仿真器进行性能模型验证。Joshi等人[100]仅基于与微架构无关的特征生成合成基准，并且他们利用该框架自动生成压力标记（即最大化功耗，温度等的合成程序），参见[101]。 |
| A couple other research efforts model cache performance in a statistical way. Berg and Hagersten [14] propose light-weight profiling to collect a memory reference reuse distribution at low overhead and then estimate cache miss rates for random-replacement caches. Chandra et al. [24] propose performance models to predict the impact of cache sharing on co-scheduled programs. The output provided by the performance model is an estimate of the number of extra cache misses for each thread due to cache sharing. | 其他几项研究工作以统计方式对缓存性能进行了建模。Berg和Hagersten [14]提出了轻量级分析，以低开销收集内存参考重用分布，然后估计随机替换缓存的缓存未命中率。Chandra等人[24]提出了性能模型来预测缓存共享对共同调度程序的影响。性能模型提供的输出是对由于缓存共享而导致的每个线程的额外缓存未命中数的估计值。 |

|  |  |
| --- | --- |
| CHAPTER 8 Parallel Simulation and Hardware Acceleration | 第8章并行仿真和硬件加速 |
| Computer architects typically have to run a large number of simulations when sweeping across the design space to understand performance sensitivity to specific design parameters.They therefore distribute their simulations on a large computing cluster — all of these simulations are independent of each other and are typically run in parallel. Distributed simulation achieves high simulation throughput — the increase in throughput is linear in the number of simulation machines. Unfortunately, this does not reduce the latency of obtaining a single simulation result. This is a severe problem for simulation runs that take days or even weeks to run to completion. Waiting for these simulations to finish is not productive and slows down the entire design process. In practice, individual simulations that finish in a matter of hours (e.g., overnight) or less is preferred. In other words, obtaining a single simulation result quickly is important because it may be a critical result that moves research and development forward. | 为了扫描设计空间以理解特定设计参数对于性能的敏感度，计算机架构师通常需要运行海量的仿真。因此，架构师将仿真分配到庞大的计算集群上——通常每一个仿真都是相互独立的，并且并行执行。分布式仿真可以获得高的仿真吞吐率——吞吐率随着仿真机器的数量线性增加。遗憾的是，分布式仿真并不能减少观察到单个仿真结果的延迟。仿真耗费几天甚至几周时间完成是很严重的问题。等待这些仿真结果的产出很低，而且会拖慢整个设计流程。实践中，单个仿真推荐在几小时（比如过夜）或者更少的时间完成。换句话说，快速观察到单个仿真的结果是很重要的，因为它可能是推进结果和开发的重要结果。 |
| One way to speed up individual simulation runs is to exploit parallelism.This chapter describes three approaches for doing so:(i) sampled simulation,which distributes sampling units across a cluster of machines, (ii) parallel simulation, which exploits coarse-grain parallelism to map a software simulator on parallel hardware, e.g., a multicore processor, SMP, cluster of machines, etc., and (iii) FPGA-accelerated simulation, which exploits fine-grain parallelism by mapping a simulator on FPGA hardware. | 加速单个仿真运行的一种方式是利用并行性。本章描述三种利用并行性的方式：(i) 采样仿真，在一组机器上分布样本单元；（ii）并行仿真，通过在并行硬件上映射软件仿真器，利用粗粒度的并行，例如，多核处理器、SMP和机器集群等；（iii）FPGA加速仿真，通过将仿真器映射到FPGA硬件，利用细粒度并行。 |
| 8.1 PARALLEL SAMPLED SIMULATION | 8.1 并行采样仿真 |
| As mentioned in Chapter 6, sampled simulation leans well towards parallel simulation, provided that checkpointing is employed for constructing the architecture and microarchitecture starting images for each sampling unit. By loading the checkpoints in different instances of the simulators across a cluster of machines, one can simulate multiple sampling units in parallel, which greatly reduces turnaround time of individual simulation runs. Several researchers explored this avenue, see for example Lauterbach [118] and Nguyen et al. [146]. More recently, Wenisch et al. [190] report a thousand fold reduction in simulation turnaround time through parallel simulation: they distribute thousands of sampling units (along with their checkpoints) across a cluster of machines. | 如第6章所述，采样仿真倾向于并行仿真。检查点用来构造每一个样本单元的架构和微架构起始镜像。通过在一组机器上的不同仿真器实体上载入不同的检查点，可以并行仿真很多采样单元，从而显著减少单个仿真运行的完整时间。有一些工作利用这种路线，比如 Lauterbach [118]和Nguyen等[146]的工作。最近， Wenisc等[190]报告通过并行仿真将仿真完整时间减少了一千倍：他们将几千个采样单元（以及他们的检查点）分布到一组机器上。 |
| Girbal et al.[75] propose a different approach to parallel simulation:they partition the dynamic instruction stream in so called chunks, and these chunks are distributed across multiple machines, see also Figure 8.1 for an illustrative example. In particular, there are as many chunks as there are machines, and each chunk consists of 1/Nth of the total dynamic instruction stream with N the number of machines. This is different from sampled simulation because sampled simulation simulates sampling units only, in contrast to the Girbal et al. approach which eventually simulates the entire benchmark. Each machine executes the benchmark from the start. The first machine starts detailed simulation immediately; the other machines employ fast-forwarding (or, alternatively, employ checkpointing),and when the beginning of the chunk is reached,they run detailed simulation. The idea now is to continue the simulation of each chunk past its end point (and thus simulate instructions for the next chunk). In other words, there is overlap in simulation load between adjacent machines. By comparing the post-chunk performance numbers against the performance numbers for the next chunk (simulated on the next machine), one can verify whether microarchitecture state has been warmed up on the next machine. (Because detailed simulation of a chunk starts from a cold state on each machine, the performance metrics will be different than for the post-chunk on the previous machine — this is the cold-start problem described in Chapter 6.) Good similarity between the performance numbers will force the simulation to stop on the former machine. The performance numbers at the beginning of each chunk are then discarded and replaced by post-chunk performance numbers from the previous chunk.The motivation for doing so is to compute an overall performance score from performance numbers that were collected from a warmed up state. | Girbal等[75]提供了并行仿真的另一种方式：他们将动态指令流拆分为多个碎片（chunk），这些分布在多台机器上，如图8.1示例所示。特别地，碎片的数量与机器数量相同，而且每一个碎片包含全部动态指令流的第分之一，表示机器的数量。这与采样仿真不同，因为采样仿真只仿真采样单元，与Girbal等使用的仿真整个测试程序的方法不同。每台机器从开始执行测试程序。第一台机器立即启动详细仿真；其他机器启动快速仿真（或，使用检查点作为替代方法）。当到达碎片开始点的时候，这些机器开始运行详细仿真。这里的想法是，继续仿真每一个碎片知道碎片结束点（而且仿真下一个碎片的指令）。换句话说，相邻机器之间的仿真负载是有重叠的。通过比较后碎片的性能数字和（下一台机器上仿真的）下一个碎片的性能数字，可以验证下一台机器上的微架构状态是否已经被热身。（因为每一个机器上的详细仿真都是从冷状态开始的，性能度量会与前一台机器上的后碎片不同——这是第6章描述的冷启动问题。）性能参数的优秀匹配会强制前一台机器的仿真停止。每一个碎片起点的性能数字被丢弃，并且用前一个碎片的后碎片性能数字替代。这样做的目的是为了从热身状态统计的性能数字开始计算完整性能得分。 |
|  | |
| Figure 8.1: Chunk-based parallel simulation. | 图8.1：基于碎片的并行仿真 |
| 8.2 PARALLEL SIMULATION | 8.2 并行仿真 |
| Most software simulators are single-threaded, which used to be fine because simulation speed benefited from advances in single-thread performance. Thanks to Moore’s law and advances in chip technology and microarchitecture, architects were able to improve single-thread performance exponentially. However, this trend has changed over the past decade because of power issues, which has led to the proliferation of chip-multiprocessors (CMPs) or multicore processors. As a result of this multicore trend, it is to be expected that single-thread performance may no longer improve much — and may even deteriorate as one moves towards more power and energy efficient designs. A chip-multiprocessors integrates multiple processor cores on a single chip, and thus a software simulator needs to simulate each core along with their interactions. A simulator that is single-threaded (which is typically the case today) thus is not a scalable solution: whereas Moore’s law predicts that the number of cores will double every 18 to 24 months, single-threaded simulator performance will become exponentially slower relative to native processor performance. In other words, the already very wide gap in simulation speed versus native execution speed will grow exponentially as we move forward. | 大部分软件仿真器是单线程的。在过去这样做可以是因为仿真速度受益于单线程性能的软件。归功于摩尔定律、芯片工艺的进步以及微架构，架构师能够指数级提高单线程性能。然而，由于功耗问题，这种趋势在过去十年发生了变化，进而导致了片上多处理器（CMP）或多核处理器的快速增长。这种多核趋势的结果是，可以预期，当面向更加强大和更高能效的设计时，单线程性能不会继续明显增加——甚至可能会恶化。片上多处理器在单个芯片上集成多个处理器核心，因此软件仿真器需要能够仿真这些核心以及他们之间的交互。单线程仿真器（当下的典型场景）并不是一个可扩展的解决方法：鉴于摩尔定律预测处理器核心数量会在18到24个月翻倍，相对于原生处理器性能，单线程仿真器性能会指数级落后。换句话说，随着设计演进，仿真速度和原生执行速度之间的差距会指数级增长。 |
| One obvious solution is to parallelize the simulator so that it can exploit the multiple thread contexts in the host multicore processor to speed up simulation. This could potentially lead to a scalable solution in which future generation multicore processors can be simulated on today’s multicore processors — simulation would thus scale with each generation of multicore processors. The basic idea is to partition the simulator across multiple simulator threads and have each simulator thread do some part of the simulation work. One possible partitioning is to map each target core onto a separate simulator thread and have one or more simulator threads simulate the shared resources. The parallel simulator itself can be implemented as a single address space shared-memory program, or, alternatively, it can be implemented as a distributed program that uses message passing for communication between the simulator threads. | 一种显然的解决方法是并行化仿真器，从而可以利用物理主机多核处理器中的多线程环境来加速仿真。这样可能产生一个可扩展的解决方案：未来的多核处理器可以在今天的多核处理器上仿真——仿真可以在每一个多核处理器上伸缩。基本想法是，将仿真器切分成多个仿真线程，每一个仿真线程进行仿真工作的一部分。一个可能的划分是将目标核心映射到一个单独的仿真器线程，而且有一个或多个仿真器线程仿真共享资源。并行仿真器本身可以被实现为一个单地址空间的共享存储的程序，或者也可以实现为一个利用消息在仿真线程之间通信的分布式程序。 |
| Parallel simulation is not a novel idea — it has been studied for many years [142; 163] — however, interest has renewed recently given the multicore era [4; 26; 140; 155]. One could even argue that parallel simulation is a necessity if we want simulation to be scalable with advances in multicore processor technology. Interestingly, multicore processors also provide an opportunity for parallel simulation because inter-core communication latencies are much shorter,and there are higher bandwidths between the cores than it used to be case when simulating on symmetric multiprocessors or clusters of machines. | 并行仿真器不是一个新的想法——相关研究已经持续多年[142; 163]——但是，在多线程时代，研究热度重新提升[4; 26; 140; 155]。如果我们想要仿真对于多核处理器技术是可扩展的，那么可以主张并行仿真是必须的。有趣的是，由于更短的核间通信延迟，多核处理器也为并行仿真器提供了机会。相对于在同构多处理器或机器集群上仿真，多核处理器提供更高的核间通信带宽。 |
| One of the key issues in parallel simulation is to balance accuracy versus speed. Cycle-by-cycle simulation advances one cycle at a time, and thus the simulator threads simulating the target threads need to synchronize every cycle, see Figure 8.2(a). Whereas this is a very accurate approach, its performance may not be that great because it requires barrier synchronization at every simulated cycle. If the number of simulator instructions per simulated cycle is low, parallel cycle-by-cycle simulation is not going to yield substantial simulation speed benefits and scalability will be poor. | 并行仿真的关键问题是平衡精确度和速度。时钟精确仿真每次只推进一个周期的，而且每一个周期都需要在目标线程之间进行同步，见图8.2（a）。虽然这是精确度最好的方式，但是仿真器性能不是很好，因为每一个仿真的时钟周期都需要进行同步。如果每一个仿真周期的指令数很低，并行的时钟精确仿真不会产生显著的速度收益，而且可扩展性差。 |
|  | |
| Figure 8.2: Three approaches for synchronizing a parallel simulator that simulates a parallel machine: (a) barrier synchronization at every cycle, (b) relaxed or no synchronization, and (c) quantum-based synchronization. | 图8.2：仿真并行机器的并行仿真器的三个同步方法：(a)每个周期进行同步；(b)放松或没有同步，(c)基于量程的同步。 |
| In order to achieve better simulation performance and scalability, one can relax the cycle-bycycle condition. In other words, the simulated cores do not synchronize every simulated cycle, which greatly improves simulation speed, see Figure 8.2(b). The downside is that relaxing the synchronization may introduce simulation error. The fundamental reason is that relaxing the cycle-by-cycle condition may lead to situations in which a future event may affect state in the past or a past event does not affect the future — a violation of temporal causality. Figure 8.3 illustrates this. Assume a load instruction to a shared memory address location A is executed at simulated time x + 1 and a store instruction to that same address is executed at simulated time x; obviously, the load should see the value written by the store. However, if the load instruction at cycle x would happen to be simulated before the store at cycle x − 1, the load would see the old value at memory location A. | 为了获得更好的性能和可扩展性，可以放松时钟精确的条件。换句话说，仿真核心必须要每一个周期都进行同步，这会显著提升仿真速度，见图8.2（b）。缺陷是，放松同步会导致仿真错误。这基本的原因是，放松时钟精确条件可能导致这样的情况：未来的事件可能影响过去的状态，或者一个过去的时间没有影响到未来——违反了时间上的因果性。图8.3解释了这种情况。假设一条对共享存储地址A的load指令在仿真时间x+1执行，而对于相同地址的 store指令在仿真时间x执行。显然，load执行应该能够看到store写入的值。但是，如果时钟周期x的load指令在时钟周期x-1的store指令之前仿真，load指令就会看到内存地址A的旧值。 |
|  | |
| Figure8.3: Violation of temporal causality due to lack of synchronization: the load sees the old value at memory location A. | 图8.3：由于没有同步引起的时序因果性违例：load看到了内存位置A的旧值。 |
| There exist two approaches to relax the synchronization imposed by cycle-by-cycle simulation [69].The optimistic approach takes periodical checkpoints and detects timing violations. When a timing violation is detected, the simulation is rolled back and resumes in a cycle-by-cycle manner until after the timing violation and then switches back to relaxed synchronization. The conservative approach avoids timing violations by processing events only when no other event could possibly affect it. A popular and effective conservative approach is based on barrier synchronization, while relaxing the cycle-by-cycle simulation. The entire simulation is divided into quanta, and each quantum comprises multiple simulated cycles. Quanta are separated through barrier synchronization, see Figure 8.2(c). In other words, simulation threads can advance independently from each other between barriers, and the simulated events become visible to all threads at each barrier. Provided that the time intervals are smaller than the latency for an inter-thread dependence (e.g., to propagate an event from one core to another), temporal causality will be preserved. Hence, quantum-based synchronization achieves cycle-by-cycle simulation accuracy while greatly improving simulation speed compared to cycle-by-cycle simulation.The Wisconsin Wind Tunnel projects [142; 163] implement this approach; the quantum is 100 cycles when simulating shared memory multiprocessors. Chandrasekaran and Hill [25] aim at overcoming the quantum overhead through speculation; Falsafi and Wood [67] leverage multiprogramming to hide the quantum overhead. Falcon et al. [66] propose an adaptive quantum-based synchronization scheme for simulating clusters of machines and the quantum can be as large as 1,000 cycles. When simulating multicore processors, the quantum needs to be smaller because of the relatively small communication latencies between cores: for example, Chidester and George [28] employ a quantum of 12 cycles. A small quantum obviously limits simulation speed. Therefore, researchers are looking into relaxing even further, thereby potentially introducing simulation inaccuracy. Chen et al. [26] study both unbounded slack and bounded slack schemes; Miller et al. [140] study similar approaches. Unbounded slack implies that the slack, or the cycle count difference between two target cores in the simulation, can be as large as the entire simulated execution time. Bounded slack limits the slack to a preset number of cycles, without incurring barrier synchronization. | 有两种方法来减弱时钟精确仿真要求的同步。乐观的方法使用周期性的检查点并且检查时序违例。如果发生了时序违例，仿真会回滚，并且按照时钟精确的方式继续直到发生时序违例的地方。之后，切换回弱同步方式。悲观的方式通过在没有其他事件影响的时候处理事件的方法来避免时序违例。常用和有效的悲观方法是在放松时钟精确的同时使用barrier同步。整个仿真切分为多个量程，每个量程打包多个仿真周期。量程通过barrier同步分割，见图8.2（c）。也就是说，在barrier之间，仿真线程可以相互独立地推进，在每个barrier，仿真的时间对于所有线程可见。只要保证时间间隔小于线程之间依赖的延迟（例如从一个核心传播事件到另一个核心），就可以保证时序因果性。因此，基于量程的同步可以达到与时钟精确仿真相同的准确度，同时又可以显著提高仿真速度，相对于时钟精确仿真。Wisconsin Wind Tunnel项目[142; 163]使用这种方法；当仿真共享内存多处理器时，量程是100个周期。Chandrasekaran和Hill[25]目标是通过投机执行来消除量程开销；Falsafi和Wood [67]充分利用多程序设计来隐藏量程开销。Falcon等[66]提出了一种自适应的量程机制用来仿真机器集群，量程大小可以达到1000周期。当仿真多核处理器时，由于核间相对小的通信延迟，量程需要比较小：例如，Chidester和George[28]采用12个周期的量程。显然小量程限制仿真速度。因此，研究者寻求进一步放松约束，然而可能引入仿真不精确。Chen等[26]研究了无限松弛和有限松弛机制；Miller等[140]研究相似的机制。无限松弛意味着松弛，或者目标核之间的时钟数差距，可以达到整个仿真周期。有限松弛将送至限制在一个预设的周期数之内，而不招致barrier同步。 |
| 8.3 FPGA-ACCELERATED SIMULATION | 8.3 FPGA加速仿真 |
| Although parallelizing a simulator helps simulator performance, it is to be expected that the gap between simulation speed and target speed is going to widen, for a number of reasons. First, simulating an n-core processor involves n times more work than simulating a single core for the same simulated time period. In addition, the uncore, e.g., the on-chip interconnection network, tends to grow in complexity as the number of cores increases. As a result, parallel simulation does not yield a linear speedup in the number of host cores, which makes the simulation versus target speed gap grow wider. To make things even worse, it is expected that cache sizes are going to increase as more cores are put on the chip — larger caches reduce pressure on off-chip bandwidth. Hence, ever longer-running benchmarks need to be considered to fully exercise the target machine’s caches. | 尽管并行化仿真器可以提高仿真性能，但是由于一些原因，仿真速度和目标速度之间的差距仍然预期进一步扩大。首先，相对于仿真单个处理器核心，对于相同的仿真周期，仿真n个处理器核心需要n倍的负载。此外，核外部分（比如片上互联网络）的复杂度随着核心数量增加而增加。结果，并行仿真的速度并没有随着主机核心数量增加而线性增加，这使得仿真速度和目标速度的差距更加扩大。使得事情更糟糕的是，缓存大小随着芯片上的核心数量增加和增加——更大的缓存减少了片外带宽的压力。因此，为了完全检验目标机器的缓存，测试程序需要更长。 |
| Therefore, interest has grown over the past few years for FPGA-accelerated simulation. The basic idea is to implement (parts of) the simulator on an FPGA (Field Programmable Gate Array). An FPGA is an integrated circuit that can be configured by its user after manufacturing. The main advantage of an FPGA is its rapid turnaround time for new hardware because building a new FPGA design is done quickly: it only requires synthesizing the HDL code (written in a Hardware Description Language, such as VHDL or Verilog) and loading the FPGA code onto the FPGA — much faster than building an ASIC. Also, FPGAs are relatively cheap compared to ASIC designs as well as large SMPs and clusters of servers. In addition, as FPGA density grows with Moore’s law, it will be possible to implement more and more processor cores in a single FPGA and hence keep on benefiting from advances in technology, i.e., it will be possible to design next-generation systems on today’s technology. Finally, current FPGA-accelerated simulators are substantially faster than software simulators, and they are even fast enough (on the order of tens to hundreds of MIPS, depending on the level of accuracy) to run standard operating systems and complex large-scale applications. | 所以，对于FPGA加速仿真的研究兴趣在过去几年持续增长。基本想法是在FPGA（现场可编程逻辑门阵列）上实现全部（或者局部）仿真器。FPGA是一种可以被用户在制造完成后进行配置的集成电路。FPGA的主要优点是，FPGA加速了新硬件的实现时间，因为在FPGA上完成一个新设计是非常快：FPGA只需要综合HDL代码（利用硬件描述语言编写，例如VHDL或Verilog）和将FPGA代码载入到FPGA——比建造一个ASIC快很多。而且，相对于ASIC设计、更大的SMP和服务器集群，FPGA也相对便宜。此外，FPGA的集成密度随着摩尔定律增加，在单个FPGA上可能实现更多的处理器核心，因此可以持续从工艺进步中获益。也就是说，可以用今天的技术设计下一代系统。最后，当前FPGA加速仿真器大幅度领先于软件仿真器，而且速度快到（数量级达到几千MIPS，取决于精确度）可以运行标准操作系统和复杂的大规模应用。 |
| FPGA-accelerated simulation thus clearly addresses the evaluation time axis in the simulation diamond (see Chapter 5). Simulation time is greatly reduced compared to software simulation by leveraging the fine-grain parallelism available in the FPGA: work that is done in parallel on the target machine may also be done in parallel on the FPGA, whereas software simulation typically does this in a sequential way. However, simulator development time may be a concern. Because the simulator needs to be synthesizable to the FPGA, it needs to be written in a hardware description language (HDL) like Verilog or VHDL, or at a higher level of abstraction, for example, in Bluespec. This, for one, is likely to increase simulator development time substantially compared to writing a software simulator in a high-level programming language such as C or C++. In addition, software simulators are easily parameterizable and most often do not need recompilation to evaluate a design with a different configuration. An FPGA-accelerated simulator, on the other hand, may need to rerun the entire FPGA synthesis flow (which may take multiple hours, much longer than recompiling a software simulator, if at all needed) when varying parameters during a performance study. Modular simulation infrastructure is crucial for managing FPGA-accelerated simulator development time. | FPGA加速仿真器明显满足仿真菱形的评估时间维度（见第5章）。与软件仿真器相比，通过利用FPGA中的细粒度并行，仿真时间显著减少：在目标机器上并行执行的工作，在FPGA上也可以并行执行，但是软件仿真一般只能以串行方式执行。然而，仿真器的开发时间会是关注点。因为，仿真器需要能够综合为FPGA，仿真器需要使用硬件描述语言（HDL）开发，例如Verilog或VHDL，或者更高层次的抽象，例如Bluespec。与用高级编程语言（C或C++）开发软件仿真器，FPGA加速仿真器很可能会显著增加仿真器开发时间，此外，软件仿真器更容易参数化，而且在评估不同配置的设计时不需要重新编译。相反，在性能研究中改变参数时，FPGA加速仿真器可能需要重新运行整个FPGA综合流程（可能需要几小时时间，比软件仿真器重启编译的时间长很多）。模块仿真基础设施对于管理FPGA加速仿真器开发时间至关重要。 |
| 8.3.1 TAXONOMY | 8.3.1 分类法 |
| Many different FPGA-accelerated simulation approaches have been proposed by various research groups both in industry and academia. In order to understand how these approaches differ from each other, it is important to classify them in terms of how they operate and yield performance numbers. Joel Emer presents a useful taxonomy on FPGA-accelerated simulation [55] and discerns three flavors: | 在工业界和学术界的研究组提出了很多不同的FPGA加速仿真方法。为了理解这些方法相互的区别，按照他们操作方式和产出的性能数字进行分类就很重要。Joel Emer提出了有效的分类方法[55]，并且辨别三种方式： |
| • A functional emulator is a circuit that is functionally equivalent to a target design, but does not provide any insight on any specific design metric. A functional emulator is similar to a functional simulator, except that it is implemented in FPGA hardware instead of software.The key advantage of an FPGA-based functional emulator is that it can execute code at hardware speed (several orders of magnitude faster than software simulation), which allows architects and software developers to run commercial software in a reasonable amount of time. | • 功能模拟器（functional emulator）与目标设计在功能上等价，但是并不提供任何特定设计度量的观察。功能模拟器与功能仿真器类似，除了不是在软件而是在FPGA硬件上实现。基于FPGA的功能模拟器的关键优势是，可以按照硬件速度执行代码（比软件仿真快几个数量级），从而允许架构师和软件开发者能够按照合理的实现运行商业软件。 |
| • A prototype (or structural emulator) is a functionally equivalent and logically isomorphic representation of the target design. Logically isomorphic means that the prototype implements the same structures as in the target design, and its timing may be scaled with respect to the target system. Hence, a prototype or structural emulator can be used to project performance. For example, a prototype may be a useful vehicle for making high-level design decisions, e.g., study the scalability of software code and/or architecture proposal. | • 原型（prototype）（或结构模拟器）与目标设计器在功能等效，而且逻辑上同构。逻辑同构表示原型实现了目标设计中的相同结构，而且相对于目标系统，原型的时序进行了缩放。因此，原型或结构模拟器能够用于项目性能的评估。例如，原型系统可以用于高层设计决定，比如研究软件代码的缩放性，或者架构提议。 |
| • A model is a representation that is functionally equivalent and logically isomorphic with the target design, such that a design metric of interest (of the target design), e.g., performance, power and/or reliability, can be faithfully quantified. The advantage of a model compared to a prototype is that it allows for some abstraction which simplifies model development and enables modularity and easier evaluation of target design alternatives.The issue of representing time faithfully then requires that a distinction is made between a simulated cycle (of the target system) versus an FPGA cycle. | • 模型（model）是目标系统的功能等效和逻辑同构的表达，例如目标系统关注的设计度量可以被准确度量，比如，性能、功耗和/或可靠性。相对于原型，模型的优势在于允许进行抽象以简化模型开发，同时使能模块化设计，而且简化目标设计的评估。为了忠实地表示时序，需要区分（目标系统的）仿真周期和FPGA周期。 |
| 8.3.2 EXAMPLE PROJECTS | 8.3.2 示例项目 |
| As mentioned before, there are several ongoing projects that focus on FPGA-accelerated simulation. The RAMP (Research Accelerator for Multiple Processors) project [5] is a multi-university collaboration that develops open-source FPGA-based simulators and emulators of parallel architectures. The RAMP infrastructure includes a feature that allows for having the inter-component communication channels run as fast as the underlying FPGA hardware will allow, which enables building a range of FPGA simulators from functional emulators, to prototypes and models.The RAMP project has built a number of prototypes, such as RAMP Red and RAMP Blue, and models such as RAMP Gold.The purpose of these designs ranges from evaluating architectures with transactional memory support, message passing machines, distributed shared memory machines, and manycore processor architectures. | 如前文所属，一些正在进行的项目关注FPGA加速仿真。RAMP（多处理器加速器研究）项目[5]由多所学校合作，为并行架构开发基于FPGA的开源仿真器和模拟器。RAMP的基础设施包括这样的特性，允许组件间通信通道可以以底层FPGA硬件允许的速度运行，这允许构建从功能模拟器到原型和模型的一系列FPGA仿真器。RAMP项目已经建立了一些原型，例如RAMP Red和RAMP Blue，以及模型，例如RAMP Gold。这些设计的目的包括评估具有事务内存支持、消息传递机器、分布式共享内存机器和多核处理器架构的架构。 |
| A number of other projects are affiliated with RAMP as well, such as Protoflex, FAST and HAsim. Protoflex from Carnegie Mellon University takes a hybrid simulation approach [32] and implements the functional emulator in hardware to fast-forward between sampling units;the detailed cycle-accurate simulation of the sampling units is done in software. Protoflex also augments the functional emulator with cache simulation (e.g., to keep cache state warm between sampling units). Both FAST and HAsim are cycle-level models that partition the functional and timing parts. A principal difference is in the implementation strategy for the functional part. FAST [30], at the University of Texas at Austin, implements a speculative functional-first simulation strategy, see  Section 5.6.A functional simulator generates a trace of instructions that is fed into a timing simulator. The functional simulator is placed in software and the timing simulator is placed on the FPGA.  The functional simulator speculates on branch outcomes [30; 178] and the memory model [29], and rolls back to an earlier state when mis-speculation is detected. HAsim [154], a collaboration effort between Intel and MIT, is a timing-directed execution-driven simulator, in the terminology of Section 5.6, which requires tight coupling between the functional and timing models: the functional part performs an action in response to a request from the timing part. The functional and timing models are placed on the FPGA. Rare events, such as system calls, are handled in software, alike what is done in Protoflex. | 其他一些项目也与RAMP有关联，例如Protoflex，FAST和Hasim。卡内基梅隆大学的Protoflex项目使用混合仿真方法[32]，而且在硬件上实现了功能模拟器，从而加速样本单元之间的快进；样本单元的详细时钟精确仿真由软件完成。Protoflex还通过缓存仿真器增强功能模拟器（例如，在样本单元之间保持缓存状态启动）。FAST和Hasim都是时钟级模型，分为功能和时序部分。原则上的区别在于功能部分的实现方法。在奥斯丁的德州大学，FAST[30]实现了一种投机的功能优先的策略，见5.6节。功能仿真器产生指令流的记录，再灌入时序仿真器。功能仿真器在软件中实现，时序仿真器在FPGA中实现。功能仿真器对于分支结果[30; 178]和存储模型[29]进行投机，并且在检测到错误投机的时候，仿真器回归到更早的状态。Hasim[154]是Intel和MIT的合作产物，是一个时序导向的执行驱动仿真器。用5.6节的术语来说，Hasim需要功能和时序模型的紧耦合：功能模型执行一个动作以响应来自时序部分的请求。功能模型和时序模型都用于FPGA。少数事件在软件中处理，如系统调用，就像Protoflex一样。 |
| An earlier study using FPGAs to accelerate simulation was done at Princeton University. Penry et al. [155] employ FPGAs as an accelerator for the modular Liberty simulator. | 普林斯顿大学完成了使用FPGA加速仿真的早期研究。Penry等[155]使用FPGA作为模块化的Liberty仿真器的加速器。 |
| A current direction of research in FPGA-accelerated simulation is to time-division multiplex multiple modes. This allows for simulating more components, e.g., cores, on a single FPGA. Protoflex does this for functional emulation; RAMP Gold and Hasim multiplex the timing modes. | FPGA加速器仿真的当前方向是，时分复用模式。时分复用模式允许在单个FPGA上仿真更多的组件（例如核心）。Protoflex使用时分复用模式用于功能模拟；RAMP Gold和Hasim复用时序模型。 |

|  |  |
| --- | --- |
| CHAPTER 9 Concluding Remarks | 第9章 结论 |
| This book covered a wide spectrum of performance evaluation topics, ranging from performance metrics to workload design, to analytical modeling and various simulation acceleration techniques such as sampled simulation, statistical simulation, and parallel and hardware-accelerated simulation. However, there are a number of topics that this book did not cover. We will now briefly discuss a couple of these. | 本书覆盖了性能评估话题的大部分，从性能度量到工作负载设计，再到分析模型和不同的仿真加速技术，比如采样仿真、统计仿真、以及并行和硬件加速仿真。然而，仍然有一些话题没有覆盖到。我们会在这里简单讨论一下。 |
| 9.1 TOPICS THAT THIS BOOK DID NOT COVER (YET) | 9.1 内容拾遗 |
| 9.1.1 MEASUREMENT BIAS | 9.1.1 测量偏置 |
| Mytkovicz et al. [160] study the effect of measurement bias on performance evaluation results. Measurement bias occurs when the experimental setup is biased. This means that when comparing two design alternatives, the experimental setup favors one of the two alternatives so that the benefit of one alternative may be overstated; it may even be the case that the experiment states that one alternative outperforms the other one, even if it is not. In other words, the conclusion may be a result of a biased experimental setup and not because of one alternative being better than the other. | Mytkovicz等[160]研究了测量偏置对于性能评估结果的影响。当试验设置有偏置的时候，会发生测量偏置。这意味着，当比较两种设计方案的时候，试验设置对于两个方案之一有利，因此，一个方案的优势会被放大；甚至可能是这样的情况：实验表明，一种选择优于另一种，但是事实并非如此。换句话说，这个结论可能是有偏见的实验设置的结果，而不是因为其中一种选择比另一种更好。 |
| The significance of the work by Mytkovicz et al. is that they have shown that seemingly innocuous aspects of an experimental setup can lead systems researchers to draw incorrect conclusions in practical studies (including simulation studies), and thus, measurement bias cannot be ignored. Mytkowicz et al. show that measurement bias is commonplace across architectures, compilers and benchmarks. They point out two sources of measurement bias. The first source is due to the size of the UNIX environment, i.e., the number of bytes required to store the environment variables. The environment size may have an effect on the layout of the data stored on the stack, which in its turn may affect performance-critical characteristics such as cache and TLB behavior. For example, running the same simulation in different directories may lead to different performance numbers because the directory name affects the environment size. The second source is due to the link order, i.e., the order of object files (with .o extension) given to the linker. The link order may affect code and data layout, and thus overall performance. The fact that there may be other unknown sources of measurement bias complicates the evaluation process even further. | Mytkovicz等的工作的意义在于，他们已经表明，实验设置上看似无害的方面，可能会导致系统研究人员在实际研究（包括仿真研究）中得出不正确的结论。因此，测量偏差不能被忽视。Mytkowicz等表明，测量偏差在架构、编译器和基准测试中普遍存在。他们指出了测量偏差的两个来源。第一个来源与UNIX环境的大小有关，即存储环境变量所需的字节数。环境大小可能会影响存储在堆栈上的数据的布局，而这又可能影响性能关键特征，如缓存和TLB行为。例如，在不同的目录中运行相同的仿真可能会导致不同的性能数字，因为目录名称会影响环境大小。第二个来源是由于链接顺序，即给链接器的对象文件（扩展名为.o）的顺序。链接顺序可能会影响代码和数据布局，从而影响整体性能。事实上，可能还有其他未知的测量偏差的来源使评估过程更加复杂。 |
| Mytkovicz et al. present two possible solutions. The first solution is to randomize the experimental setup, run each experiment in a different experimental setup, and summarize the performance results across these experimental setups using statistical methods (i.e., compute average and confidence intervals).The obvious downside is that experimental setup randomization drastically increases the number of experiments that need to be run. The second solution is to establish confidence that the outcome is valid even in the presence of measurement bias. | Mytkovicz等提出了两种可能的解决方案。第一个解决方案是随机化实验设置。在不同的实验设置中运行每个实验，并使用统计方法（即计算平均值和置信区间）总结这些实验设置的性能结果。明显的缺点是实验设置的随机性大大增加了需要运行的实验数量。第二个解决方案是建立置信度，即使在存在测量偏差的情况下，结果也是有效的。 |
| 9.1.2 DESIGN SPACE EXPLORATION | 9.1.2 设计空间探索 |
| We briefly touched upon the fact that computer designers need to explore a large design space when developing the next generation processor. Also, researchers need to run a huge number of simulations in order to understand the performance sensitivity of a new feature in relation to other microarchitecture design parameters. | 我们简要地谈到了这样一个事实，即计算机设计师在开发下一代处理器时需要探索很大的设计空间。此外，研究人员需要进行大量的仿真，以了解新特性对于其他微架构设计相关参数的性能敏感性。 |
| A common approach is the one-parameter-at-a-time method, which keeps all microarchitecture parameters constant while varying one parameter of interest. The pitfall may be though that some constant parameter may introduce measurement bias, i.e., the effect of the parameter of interest may be overstated or understated. Also, using the one-parameter-at-a-time approach during design space exploration may lead to a suboptimal end result, i.e., the search process may end up in a local optimum, which may be substantially worse than the global optimum. | 一种常见的方法是控制变量的方法，它保持所有微架构参数不变，同时只改变一个感兴趣的参数。这种方法的陷阱可能是一些常数参数可能会引入测量偏差，也就是说，感兴趣参数的影响可能被夸大或低估。此外，在设计空间探索中使用控制变量法可能会导致最终的次优结果，即搜索过程可能会陷入局部最优，而局部最优的性能可能会大大低于全局最优。 |
| Yi et al. [160] propose the Plackett and Burman design of experiment to identify the key microarchitecture parameters in a given design space while requiring a small number of simulations only. (We discussed the Plackett and Burman design of experiment in Chapter 3 as a method for finding similarities across benchmarks; however, the method was originally proposed for identifying key microarchitecture parameters.) The end result of a Plackett and Burman design of experiment is a ranking of the most significant microarchitecture parameters. This ranking can be really helpful during design space exploration: it guides the architects to first explore along the most significant microarchitecture parameters before considering the other parameters. This strategy gives higher confidence of finding a good optimum with a limited number of simulations. | Yi等[160]提出了Plackett和Burman实验设计，在给定的设计空间中确定关键的微架构参数，同时只需要少量的仿真。（我们在第3章讨论了Plackett和Burman的实验设计，作为在不同基准中寻找相似点的方法；然而，该方法最初是为了识别关键微架构参数而提出的。）通过Plackett和Burman的实验设计，对最重要的微架构参数进行了排序。这个排名在设计空间探索过程中可能非常有用：它指导架构师在考虑其他参数之前先沿着最重要的微架构参数进行探索。该策略在有限的仿真次数下具有较高的置信度。 |
| Eyerman et al. [62] consider various search algorithms to explore the microarchitecture design space, such as tabu search and genetic algorithms. These algorithms were found to be better at avoiding local optima than the one-parameter-at-a-time strategy at the cost of more simulations. They also propose a two-phase simulation approach in which statistical simulation is first used to identify a region of interest which is then further explored through detailed simulation. | Eyerman等[62]考虑了各种搜索算法来探索微架构设计空间，如禁忌搜索和遗传算法。研究发现，这些算法在避免局部最优方面优于控制变量的策略，但代价是更多的仿真。他们还提出了一种两阶段仿真方法，其中首先将统计仿真用于确定感兴趣的区域，然后通过详细仿真进一步探索该区域。 |
| Karkhanis and Smith [104] and Lee and Brooks [120] use analytical models to exhaustively explore a large design space. Because an analytical model is very fast (i.e., a performance prediction is obtained instantaneously), a large number of design points can be explored in an affordable amount of time. | Karkhanis和Smith[104]以及Lee和Brooks[120]使用分析模型详尽地探索一个大的设计空间。由于分析模型的速度非常快（即可以瞬间获得性能预测），因此可以在可承受的时间内探索大量的设计点。 |
| Design space exploration is also crucial in embedded system design. The constraints on the time-to-market are tight, and there is great need for efficient exploration techniques. Virtual prototyping, high-abstraction models, and transaction-level modeling (TLM), which focuses on data transfer functionality rather than implementation, are widely used techniques in embedded system design during early stages of the design cycle. | 在嵌入式系统设计中，设计空间的探索也是至关重要的。对上市时间的限制是严格的，进而对有效的探索技术有很大的需求。在设计周期的早期阶段，虚拟原型、高抽象模型和事务级建模（TLM）是嵌入式系统设计中广泛使用的技术，它们关注的是数据传输功能而不是实现。 |
| 9.1.3 SIMULATOR VALIDATION | 9.1.3 仿真器验证 |
| A nagging issue to architectural simulation relates to validation. Performance simulators model a computer architecture at some level of detail, and they typically do not model the target architecture in a cycle-accurate manner; this is especially true for academic simulators.(Although these simulators are often referred to as cycle-accurate simulators, a more appropriate term would probably be cyclelevel simulators.) The reason for the lack of validation is threefold. For one, although the high-level design decisions are known, many of the details of contemporary processors are not well described so that academics can re-implement them in their simulators. Second, implementing those details is time-consuming and is likely to severely slowdown the simulation. Third, research ideas (mostly) target future processors for which many of the details are not known anyway, so a generic processor model is a viable option. (This is even true in industry during the early stages of the design cycle.) | 架构仿真的一个棘手问题与验证有关。性能仿真器在一定程度上对计算机架构进行了详细的建模，它们通常不会以周期精确的方式对目标架构进行建模；对于学术仿真器来说尤其如此。（虽然这些仿真器通常被称为周期精确的仿真器，但更合适的术语可能是周期级（cycle-level）仿真器。）缺乏验证的原因有三点。首先，虽然高级设计决策是已知的，但当代处理器的许多细节并没有被很好地描述，因此学者们可以在他们的仿真器中重新实现它们。其次，实现这些细节是非常耗时的，并且可能会严重减慢仿真的速度。第三，研究思路（主要是）针对未来的处理器，这些处理器的许多细节都不为人知，因此通用处理器仿真是一个可行的选择。（甚至在工业设计周期的早期阶段也是如此。） |
| Desikan et al. [40] describe the tedious validation process of the sim-alpha simulator against the Alpha 21264 processor. They were able to get the mean error between the hardware and the simulator down to less than 2% for a set of microbenchmarks; however, the errors were substantially higher for the SPEC CPU2000 benchmarks (20% on average). | Desikan等[40]描述了sim-alpha仿真器针对Alpha 21264处理器的冗长验证过程。在一组微基准测试中，他们能够将硬件和仿真器之间的平均误差降低到低于2%；但是，SPEC CPU2000基准测试的误差要高得多（平均20%）。 |
| Inspite of the fact that simulator validation is time-consuming and tedious, it is important to validate the simulators against real hardware in order to gain more confidence in the performance numbers that the simulators produce. Again, this is a matter of measurement bias: inadequate modeling of the target system may lead to misleading or incorrect conclusions in practical research studies. | 尽管仿真器验证是耗时且乏味，但为了获得对仿真器产生的性能数字的更多信心，将仿真器与真实的硬件进行验证是很重要的。再次说明，这是测量偏差的问题：目标系统建模不充分可能导致实际研究中产生误导或不正确的结论。 |
| 9.2 FUTURE WORK IN PERFORMANCE EVALUATION METHODS | 9.2 性能评估方法的未来 |
| Performance evaluation methods are at the foundation of computer architecture research and development. Hence, it will remain to be an important topic to both academia and industry in the future. Both researchers in academia and industry, as well as practitioners will care about rigorous performance evaluation, because it is one of the key elements that drives research and development forward. There are many challenges ahead of us in performance evaluation. | 性能评估方法是计算机体系结构研究与开发的基础。因此，它仍将是未来学术界和产业界的一个重要课题。无论是学术界和工业界的研究人员，还是实践者，都会关注严格的绩效评估，因为它是推动研发和发展的关键因素之一。在性能评估方面，我们面临着许多挑战。 |
| 9.2.1 CHALLENGES RELATED TO SOFTWARE | 9.2.1 与软件相关的挑战 |
| Software stacks are becoming more and more complex. A modern software stack may consist of a hypervisor (virtual machine monitor or VMM), multiple guest virtual machines each running an operating system, process virtual machines (Java Virtual Machine or Microsoft’s .NET framework), middleware, libraries, application software, etc. Whereas simulating application software, like the SPEC CPU benchmarks, is well understood and fairly trivial, simulating more complex workloads, e.g., virtual machines, is much more difficult and is a largely unsolved problem. For example, recent work in Java performance evaluation [17; 18; 48; 74] reveals that Java performance depends on the virtual machine (including the Just-In-Time compiler),the garbage collector,heap size,Java program input, etc. As a result, choosing the right set of system parameters to obtain meaningful simulation results is non-trivial. Moreover, if one is interested in steady-state performance rather than startup performance, a huge number of instructions may need to be simulated. | 软件栈变得越来越复杂。现代的软件栈可能包含一个管理程序（虚拟机监视器或VMM）、多个运行一个操作系统的客户虚拟机、进程虚拟机（Java虚拟机或Microsoft的. net框架）、中间件、库、应用程序软件等。虽然仿真应用程序软件（如SPEC CPU基准测试）很容易理解，而且相当琐碎，但仿真更复杂的工作负载（如虚拟机）则要困难得多，而且在很大程度上是一个没有解决的问题。例如，最近在Java性能评估[17;18;48;74]展示，Java的性能依赖于虚拟机（包括即时编译器）、垃圾收集器、堆大小、Java程序输入等。因此，选择合适的系统参数集来获得有意义的仿真结果是非常有意义的。此外，如果对稳态性能而不是启动性能感兴趣，可能需要仿真大量指令。 |
| In addition, given the trend towards multicore and manycore architectures, more and more applications will co-execute on the same processor, and they will share resources, and they will thus affect each other’s performance. This implies that we will have to simulate consolidated workloads, and this is going to increase the simulation requirements even further. Moreover, for a given set of applications of interest, the number of possible combinations of these applications (and their starting points) that need to be simulated quickly explodes in the number of cores and applications. | 此外，考虑到多核和众核架构的趋势，越来越多的应用程序将在同一个处理器上共同执行，它们将共享资源，从而相互影响性能。这意味着我们将不得不仿真统一的工作负载，这将进一步增加仿真需求。此外，对于一组给定的感兴趣的应用程序，需要仿真的这些应用程序的可能组合（及其起点）的数量，随着核心和应用程序的数量中迅速增加。 |
| Finally, different applications have different properties. For example, some applications are best-effort applications, whereas others may have (soft) real-time requirements, or need to deliver some level of quality-of-service (QoS) and need to fulfill a service-level agreement (SLA). The mixture of different application types co-executing on the same platform raises the question how to quantify overall system performance and how to compare different systems. | 最后，不同的应用程序具有不同的属性。例如，一些应用程序是最佳性能程序，而其他应用程序可能有（软）实时需求，或需要交付某种级别的服务质量（QoS），并需要实现服务级别协议（SLA）。不同类型的应用程序混合在同一个平台上共同执行，提出了如何量化总体系统性能以及如何比较不同系统的问题。 |
| 9.2.2 CHALLENGES RELATED TO HARDWARE | 9.2.2 与硬件相关的挑战 |
| The multicore and manycore era poses several profound challenges to performance evaluation methods. Simulating a multicore or manycore processor is fundamentally more difficult than simulating an individual core. As argued before, simulating an n-core processor involves n times more work than simulating a single core for the same simulated time period; in addition, the uncore is likely to grow in complexity. In other words, the gap in simulation speed versus real hardware speed is likely to grow wider. Given Moore’s law, it is to be expected that this gap will grow exponentially over time. Scalable solutions that efficiently simulate next-generation multicore processors on today’s hardware are a necessity. Parallel simulation is a promising approach; however, balancing simulation speed versus accuracy is a non-trivial issue, and it is unclear whether parallel simulation will scale towards large core counts at realistic levels of accuracy. FPGA-accelerated simulation may be a more scalable solution because it may benefit from Moore’s law to implement more and more cores on a single FPGA and thus keep on benefiting from advances in chip technology. | 多核和众核时代对性能评估方法提出了一些深刻的挑战。从根本上来说，仿真多核或众核处理器比仿真单个核心要困难得多。如前所述，在相同的仿真时间段内，仿真n核处理器所涉及的工作量是仿真单个核心的n倍；此外，核外结构可能会变得更加复杂。换句话说，仿真速度与真实硬件速度之间的差距可能会越来越大。根据摩尔定律，可以预期这一差距将随着时间的推移呈指数增长。在今天的硬件上有效地仿真下一代多核处理器的可伸缩解决方案是必要的。并行仿真是一种很有前途的方法；然而，平衡仿真速度和精度是一个重要的问题，目前还不清楚并行仿真是否能在相同的精度水平上扩展到很大的核心数。FPGA加速仿真可能是一个更可扩展的解决方案，因为它可能受益于摩尔定律，在单个FPGA上实现越来越多的核心，从而继续受益于芯片技术的进步。 |
| In addition to the challenges related to scale-up (i.e., more and more cores per chip), scale-out (i.e., more and more compute nodes in a datacenter) comes with its own challenges. The trend towards cloud computing in which users access services in ‘the cloud’, has let to so called warehousescale computers or datacenters that are optimized for a specific set of large applications (or Internet services). Setting up a simulation environment to study warehouse-scale computers is far from trivial because of the large scale — a warehouse-scale computers easily hosts thousands of servers — and because the software is complex — multiple layers of software including operating systems, virtual machines, middleware, networking, application software, etc. | 除了与纵向扩展相关的挑战（即每个芯片有越来越多的核心），横向扩展（即数据中心中有越来越多的计算节点）也带来了自己的挑战。用户在“云”中访问服务的云计算趋势，催生了所谓的仓库规模计算机或数据中心，这些计算机或数据中心针对特定的大型应用程序（或互联网服务）进行了优化。建立一个仿真环境来研究仓库规模的计算机远非小事，因为规模很大——仓库规模的计算机很容易托管数千台服务器——而且因为软件很复杂——包括操作系统、虚拟机、中间件、网络、应用软件等多层软件。 |
| Given this trend towards scale-up and scale-out it is inevitable that we will need higher levels of abstraction in our performance evaluation methods. Cycle-accurate simulation will still be needed to fine-tune individual cores; however, in order to be able to simulate large multicore and manycore processors as well as large datacenters and exascale supercomputers, performance models at higher levels of abstraction will be sorely needed. Analytical models and higher-abstraction simulation models (e.g., statistical simulation approaches) will be absolutely necessary in the near future in order to be able to make accurate performance projections. | 鉴于这种纵向和横向扩展的趋势，我们将不可避免地需要在我们的性能评估方法中使用更高层次的抽象。对单个内核进行微调，仍然需要周期精确的仿真仿真；然而，为了能够仿真大型多核和众核处理器，以及大型数据中心和百亿亿次超级计算机，将迫切需要更高抽象层次的性能模型。在不久的将来，为了能够作出准确的性能预测，分析模型和更高抽象的仿真模型（例如统计仿真方法）将是绝对必要的。 |
| 9.2.3 FINAL COMMENT | 9.2.3 最后的建议 |
| This book did provide a — hopefully — comprehensive overview of the current state-of-the-art in computer architecture performance evaluation methods. And, hopefully, this book did provide the required background to tackle the big challenges ahead of us which will require non-trivial advances on multiple fronts, including performance metrics, workloads, modeling, and simulation paradigms and methodologies. | 这本书确实（希望）提供在计算机架构性能评估方法的当前最先进的全面概述。而且，这本书确实提供了必要的背景，以解决我们面前的巨大挑战，这将需要在多个方面取得重大进展，包括性能度量、工作负载、建模和仿真范式和方法。 |

|  |
| --- |
| Bibliography |
| 1. A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. In Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA), pages 7–18, February 2003. DOI: 10.1109/HPCA.2003.1183520 58, 59, 60 |
| 1. A. R. Alameldeen and D. A. Wood. IPC considered harmful for multiprocessor workloads. IEEE Micro, 26(4):8–17, July 2006. DOI: 10.1109/MM.2006.73 6 |
| 1. J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. ACM Transactions on Computer Systems (TOCS), 4(4):273–298, November 1986. DOI: 10.1145/6513.6514 81 |
| 1. E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: Infrastructure for full system simulation. SIGOPS Operating System Review, 43(1):52–61, January 2009. DOI: 10.1145/1496909.1496921 58, 97 |
| 1. Arvind, K. Asanovic, D. Chiou, J. C. Hoe, C. Kozyrakis, S.-L. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek. RAMP: Research accelerator for multiple processors — a community vision for a shared experimental parallel HW/SW platform. Technical report, University of California, Berkeley, 2005. 102 |
| 1. D. I. August, S. Girbal J. Chang, D. G. Pérez, G. Mouchard, D. A. Penry, O. Temam, and N Vachharajani. UNISIM:An open simulation environment and library for complex architecture design and collaborative development. IEEE Computer Architecture Letters, 6(2):45–48, February 2007. DOI: 10.1109/L-CA.2007.12 61 |
| 1. T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. IEEE Computer, 35(2):59–67, February 2002. 51, 55 |
| 1. D. A. Bader, Y. Li, T. Li, and V. Sachdeva. BioPerf: A benchmark suite to evaluate highperformance computer architecture on bioinformatics applications. In Proceedings of the 2005 IEEE International Symposium on Workload Characterization (IISWC), pages 163–173, October 2005. DOI: 10.1109/IISWC.2005.1526013 16 |
| 1. K. C. Barr and K. Asanovic. Branch trace compression for snapshot-based simulation. In Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 25–36, March 2006. 76 |
| 1. K. C. Barr, H. Pan, M. Zhang, and K. Asanovic. Accelerating multiprocessor simulation with a memory timestamp record. In Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 66–77, March 2005. DOI: 10.1109/ISPASS.2005.1430560 76, 78 |
| 1. C. Bechem, J. Combs, N. Utamaphetai, B. Black, R. D. Shawn Blanton, and J. P. Shen. An integrated functional performance simulator. IEEE Micro, 19(3):26–35, May/June 1999. DOI: 10.1109/40.768499 55 |
| 1. R. Bedichek. SimNow: Fast platform simulation purely in software. In Proceedings of the Symposium on High Performance Chips (HOT CHIPS), August 2004. 54 |
| 1. R. Bell, Jr. and L. K. John. Improved automatic testcase synthesis for performance model validation. In Proceedings of the 19th ACM International Conference on Supercomputing (ICS), pages 111–120, June 2005. DOI: 10.1145/1088149.1088164 81, 93 |
| 1. E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS), pages 169–180, June 2005. DOI: 10.1145/1064212.1064232 88, 93 |
| 1. C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 72–81, October 2008. DOI: 10.1145/1454115.1454128 16 |
| 1. N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. IEEE Micro, 26(4):52–60, 2006. DOI: 10.1109/MM.2006.82 54, 55, 61 |
| 1. S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS), pages 25–36, June 2004. 105 |
| 1. S.M.Blackburn,R.Garner,C.Hoffmann,A.M.Khan,K.S.McKinley,R.Bentzur,A.Diwan, |
| D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. Eliot B.Moss,A.Phansalkar,D.Stefanovic,T.VanDrunen,D.von Dincklage,and B.Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pages 169–190, October 2006. 16, 27, 105 |
| 1. P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang. Mambo: a full system simulator for the PowerPC architecture. ACM SIGMETRICS Performance Evaluation Review, 31(4):8–12, March 2004. DOI: 10.1145/1054907.1054910 54 |
| 1. D. C. Burger and T. M. Austin. The SimpleScalar Tool Set. Computer Architecture News, 1997. See also http://www.simplescalar.com for more information. DOI: 10.1145/268806.268810 52, 61 |
| 1. M. Burtscher and I. Ganusov. Automatic synthesis of high-speed processor simulators. In Proceedings of the 37th IEEE/ACM Symposium on Microarchitecture (MICRO), pages 55–66, December 2004. 52, 72 |
| 1. M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. The VPC trace-compression algorithms. IEEE Transactions on Computers, 54(11):1329–1344, November 2005. DOI: 10.1109/TC.2005.186 54 |
| 1. R. Carl and J. E. Smith. Modeling superscalar processors via statistical simulation. In Workshop on Performance Analysis and its Impact on Design (PAID), held in conjunction with the |
| 25th Annual International Symposium on Computer Architecture (ISCA), June 1998. 81, 86, 88 |
| 1. D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip-multiprocessor architecture. In Proceedings of the Eleventh International Symposium on High Performance Computer Architecture (HPCA), pages 340–351, February 2005. 7, 91, 93 |
| 1. S. Chandrasekaran and M. D. Hill. Optimistic simulation of parallel architectures using program executables. In Proceedings of theTenth Workshop on Parallel and Distributed Simulation (PADS), pages 143–150, May 1996. 100 |
| 1. J. Chen, M. Annavaram, and M. Dubois. SlackSim: A platform for parallel simulation of CMPs on CMPs. ACM SIGARCH Computer Architecture News, 37(2):20–29, May 2009. DOI: 10.1145/1577129.1577134 97, 100 |
| 1. X. E. Chen and T. M. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. In Proceedings of the International Symposium on Microarchitecture (MICRO), pages 59–70, December 2008. 46 |
| 1. M. Chidester and A. George. Parallel simulation of chip-multiprocessor architectures. ACM Transactions on Modeling and Computer Simulation, 12(3):176–200, July 2002. DOI: 10.1145/643114.643116 100 |
| 1. D. Chiou, H. Angepat, N. A. Patil, and D. Sunwoo. Accurate functional-first multicore simulators. IEEE Computer Architecture Letters, 8(2):64–67, July 2009. DOI: 10.1109/L-CA.2009.44 57, 102 |
| 1. D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycleaccurate simulators. In Proceedings of the Annual IEEE/ACM International Symposium on |
| Microarchitecture (MICRO), pages 249–261, December 2007. 61, 62, 102 |
| 1. Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memorylevel parallelism. In Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA), pages 76–87, June 2004. 43, 88 |
| 1. E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi. ProtoFlex: Towards scalable, full-system multiprocessor simulations using FPGAs. ACM Transactions on Reconfigurable Technology and Systems, 2(2), June 2009. Article 15. 102 |
| 1. D. Citron. MisSPECulation: Partial and misleading use of SPEC CPU2000 in computer architecture conferences. In Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA), pages 52–59, June 2003. DOI: 10.1109/ISCA.2003.1206988 17 |
| 1. B. Cmelik and D. Keppel. SHADE: A fast instruction-set simulator for execution profiling. |
| In Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 128–137, May 1994. DOI: 10.1145/183018.183032 51 |
| 1. T. M. Conte, M. A. Hirsch, and W. W. Hwu. Combining trace sampling with single pass methods for efficient cache simulation. IEEE Transactions on Computers, 47(6):714–720, June 1998. DOI: 10.1109/12.689650 54, 76 |
| 1. T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In Proceedings of the International Conference on Computer Design (ICCD), pages 468–477, October 1996. DOI: 10.1109/ICCD.1996.563595 64, 76 |
| 1. H. G. Cragon. Computer Architecture and Implementation. Cambridge University Press, 2000. 11 |
| 1. P. Crowley and J.-L. Baer. Trace sampling for desktop applications on Windows NT. In Proceedings of the First Workshop on Workload Characterization (WWC) held in conjunction with the 31st ACM/IEEE Annual International Symposium on Microarchitecture (MICRO), November 1998. 74 |
| 1. P. Crowley and J.-L. Baer. On the use of trace sampling for architectural studies of desktop applications. In Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pages 208–209, June 1999. DOI: 10.1145/301453.301573 74 |
| 1. R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA), pages 266–277, July 2001. DOI: 10.1145/379240.565338 105 |
| 1. C. Dubach, T. M. Jones, and M. F. P. O’Boyle. Microarchitecture design space exploration using an architecture-centric approach. In Proceedings of the IEEE/ACM Annual International |
| Symposium on Microarchitecture (MICRO), pages 262–271, December 2007. 36 |
| 1. P. K. Dubey and R. Nair. Profile-driven sampled trace generation. Technical Report RC 20041, IBM Research Division, T. J. Watson Research Center, April 1995. 70 |
| 1. M. Durbhakula, V. S. Pai, and S. V. Adve. Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors. In Proceedings of the Fifth International Symposium on High-Performance Computer Architecture (HPCA), pages 23–32, January 1999. DOI: 10.1109/HPCA.1999.744317 71 |
| 1. J. Edler and M. D. Hill. Dinero IV trace-driven uniprocessor cache simulator. Available through http://www.cs.wisc.edu/∼markhill/DineroIV, 1998. 54 |
| 1. L. Eeckhout, R. H. Bell Jr., B. Stougie, K. De Bosschere, and L. K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA), pages 350–361, June 2004. 87 |
| 1. L. Eeckhout and K. De Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 25–34, September 2001. DOI: 10.1109/PACT.2001.953285 84 |
| 1. L. Eeckhout, K. De Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In The IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 1–6, April 2000. 86, 88 |
| 1. L. Eeckhout, A. Georges, and K. De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Languages, Applications and Systems (OOPSLA), pages 169–186, October 2003. 27, 105 |
| 1. L. Eeckhout, Y. Luo, K. De Bosschere, and L. K. John. BLRL: Accurate and efficient warmup for sampled processor simulation. The Computer Journal, 48(4):451–459, May 2005. DOI: 10.1093/comjnl/bxh103 75 |
| 1. L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere. Statistical simulation: Adding efficiency to the computer designer’s toolbox. IEEE Micro, 23(5):26–38, Sept/Oct 2003. DOI: 10.1109/MM.2003.1240210 81 |
| 1. L. Eeckhout, J. Sampson, and B. Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In Proceedings of the 2005 IEEE International Symposium on Workload Characterization (IISWC), pages 2–12, October 2005. DOI: 10.1109/IISWC.2005.1525996 27, 70 |
| 1. L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Designing workloads for computer architecture research. IEEE Computer, 36(2):65–71, February 2003. 27 |
| 1. L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. Journal of Instruction-Level Parallelism, 5, February 2003. http://www.jilp.org/vol5. 18, 25 |
| 1. M. Ekman and P. Stenström. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 89–99, March 2005. DOI: 10.1109/ISPASS.2005.1430562 70, 79 |
| 1. J. Emer. Accelerating architecture research. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), April 2009. Keynote address. 1, 101 |
| 1. J. Emer. Eckert-Mauchly Award acceptance speech. June 2009. 1 |
| 1. J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. IEEE Computer, 35(2):68–76, February 2002. 55, 56, 61 |
| 1. J.Emer,C.Beckmann,and M.Pellauer. AWB:The Asim architect’s workbench. In Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), held in conjunction with ISCA, June 2007. 61 |
| 1. J. S. Emer and D. W. Clark. A characterization of processor performance in the VAX-11/780. In Proceedings of the International Symposium on Computer Architecture (ISCA), pages 301–310, June 1984. 7 |
| 1. P. G. Emma. Understanding some simple processor-performance limits. IBM Journal of Research and Development, 41(3):215–232, May 1997. DOI: 10.1147/rd.413.0215 6 |
| 1. S. Eyerman and L. Eeckhout. System-level performance metrics for multi-program workloads. IEEE Micro, 28(3):42–53, May/June 2008. DOI: 10.1109/MM.2008.44 8, 11 |
| 1. S. Eyerman, L. Eeckhout, and K. De Bosschere. Efficient design space exploration of high performance embedded out-of-order processors. In Proceedings of the 2006 Conference on Design Automation and Test in Europe (DATE), pages 351–356, March 2006. 104 |
| 1. S. Eyerman, L. Eeckhout,T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In Proceedings ofTheTwelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 175–184, October 2006. 45 |
| 1. S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. ACM Transactions on Computer Systems (TOCS), 27(2), May 2009. 38, 44, 45 |
| 1. S. Eyerman, James E. Smith, and L. Eeckhout. Characterizing the branch misprediction penalty. In IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 48–58, March 2006. 40 |
| 1. A. Falcón, P. Faraboschi, and D. Ortega. An adaptive synchronization technique for parallel simulation of networked clusters. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 22–31, April 2008. DOI: 10.1109/ISPASS.2008.4510735 100 |
| 1. B. Falsafi and D. A. Wood. Modeling cost/performance of a parallel computer simulator. ACM Transactions on Modeling and Computer Simulation (TOMACS), 7(1):104–130, January 1997. DOI: 10.1145/244804.244808 100 |
| 1. P. J. Fleming and J. J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. Communications of the ACM, 29(3):218–221, March 1986. DOI: 10.1145/5666.5673 11 |
| 1. R.M.Fujimoto. Parallel discrete event simulation. Communications of the ACM,33(10):30–53, October 1990. DOI: 10.1145/84537.84545 99 |
| 1. R. M. Fujimoto and W. B. Campbell. Direct execution models of processor behavior and performance. In Proceedings of the 19th Winter Simulation Conference, pages 751–758, December 1987. 71 |
| 1. D. Genbrugge and L. Eeckhout. Memory data flow modeling in statistical simulation for the efficient exploration of microprocessor design spaces. IEEE Transactions on Computers, 57(10):41–54, January 2007. 88 |
| 1. D. Genbrugge and L. Eeckhout. Chip multiprocessor design space exploration through statistical simulation. IEEE Transactions on Computers, 58(12):1668–1681, December 2009. DOI: 10.1109/TC.2009.77 90, 91 |
| 1. D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In Proceedings of the International Symposium on HighPerformance Computer Architecture (HPCA), pages 307–318, January 2010. 45 |
| 1. A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, |
| Languages, Applications and Systems (OOPSLA), pages 57–76, October 2007. 105 |
| 1. S. Girbal, G. Mouchard, A. Cohen, and O. Temam. DiST: A simple, reliable and scalable method to significantly reduce processor architecture simulation time. In Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pages 1–12, June 2003. DOI: 10.1145/781027.781029 95 |
| 1. A. Glew. MLP yes! ILP no! In ASPLOS Wild and Crazy Idea Session, October 1998. 43 |
| 1. G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and more flexible program analysis. Journal of Instruction-Level Parallelism, 7, September 2005. 70 |
| 1. A. Hartstein and T. R. Puzak. The optimal pipeline depth for a microprocessor. In Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA), pages 7–13, May 2002. DOI: 10.1109/ISCA.2002.1003557 46 |
| 1. J. W. Haskins Jr. and K. Skadron. Accelerated warmup for sampled microarchitecture simulation. ACM Transactions on Architecture and Code Optimization (TACO), 2(1):78–108, March 2005. DOI: 10.1145/1061267.1061272 75 |
| 1. J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, third edition, 2003. 11 |
| 1. J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. IEEE Computer, 33(7):28–35, July 2000. 17 |
| 1. M. D. Hill and M. R. Marty. Amdahl’s law in the multicore era. IEEE Computer, 41(7):33–38, July 2008. 31 |
| 1. M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. IEEE Transactions on Computers, 38(12):1612–1630, December 1989. DOI: 10.1109/12.40842 54, 87 |
| 1. S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In Proceedings of the International Symposium on Computer Architecture (ISCA), pages 152–163, June 2008. 46 |
| 1. K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. IEEE Micro, 27(3):63–72, May 2007. DOI: 10.1109/MM.2007.56 20, 23 |
| 1. C. Hsieh and M. Pedram. Micro-processor power estimation using profile-driven program synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 17(11):1080–1089, November 1998. DOI: 10.1109/43.736182 93 |
| 1. C. Hughes and T. Li. Accelerating multi-core processor design space evaluation using automatic multi-threaded workload synthesis. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), pages 163–172, September 2008. DOI: 10.1109/IISWC.2008.4636101 81, 92 |
| 1. C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: Simulating shared-memory multiprocessors with ILP processors. IEEE Computer, 35(2):40–49, February 2002. 55 |
| 1. E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Efficiently exploring architectural design spaces via predictive modeling. In Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 195–206, October 2006. 36 |
| 1. V. S. Iyengar and L. H. Trevillyan. Evaluation and generation of reduced traces for benchmarks. Technical Report RC 20610, IBM Research Division, T. J. Watson Research Center, October 1996. 93 |
| 1. V. S. Iyengar, L. H. Trevillyan, and P. Bose. Representative traces for processor models with infinite cache. In Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA), pages 62–73, February 1996. DOI: 10.1109/HPCA.1996.501174 70, 93 |
| 1. R. K. Jain. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley, 1991. xi |
| 1. L. K. John. More on finding a single number to indicate overall performance of a benchmark suite. ACM SIGARCH Computer Architecture News, 32(4):1–14, September 2004. 11, 12 |
| 1. L. K. John and L. Eeckhout, editors. Performance Evaluation and Benchmarking. CRC Press, Taylor and Francis, 2006. xi |
| 1. E. E. Johnson, J. Ha, and M. B. Zaidi. Lossless trace compression. IEEE Transactions on Computers, 50(2):158–173, February 2001. DOI: 10.1109/12.908991 54 |
| 1. R. A. Johnson and D. W. Wichern. Applied Multivariate Statistical Analysis. Prentice Hall, fifth edition, 2002. 18, 23 |
| 1. P. J. Joseph, K. Vaswani, and M. J.Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis. In Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA), pages 99–108, February 2006. 32, 34, 35 |
| 1. P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. A predictive performance model for superscalar processors. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 161–170, December 2006. 36 |
| 1. A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. IEEETransactions on Computers, 55(6):769–782, June 2006. DOI: 10.1109/TC.2006.85 23, 62 |
| 1. A. M. Joshi, L. Eeckhout, R. Bell, Jr., and L. K. John. Distilling the essence of proprietary workloads into miniature benchmarks. ACM Transactions on Architecture and Code Optimization (TACO), 5(2), August 2008. 93 |
| 1. A. M. Joshi, L. Eeckhout, L. K. John, and C. Isen. Automated microprocessor stressmark generation. In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), pages 229–239, February 2008. 84, 93 |
| 1. T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In Proceedings of the 2nd Annual Workshop on Memory Performance Issues (WMPI) held in conjunction with ISCA, May 2002. 42, 43, 88 |
| 1. T. Karkhanis and J. E. Smith. A first-order superscalar processor model. In Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA), pages 338–349, June 2004. 43, 45 |
| 1. T. Karkhanis and J. E. Smith. Automated design of application specific superscalar processors: An analytical approach. In Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA), pages 402–411, June 2007. DOI: 10.1145/1250662.1250712 45, 104 |
| 1. K.Keeton,D.A.Patterson,Y.Q.He,R.C.Raphael,and W.E.Baker. Performance characterization of a quad Pentium Pro SMP using OLTP workloads. In Proceedings of the International Symposium on Computer Architecture (ISCA), pages 15–26, June 1998. 15 |
| 1. R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. IEEE Transactions on Computers, 43(6):664–675, June 1994. DOI: 10.1109/12.286300 74, 75 |
| 1. S. Kluyskens and L. Eeckhout. Branch predictor warmup for sampled simulation through branch history matching. Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC), 2(1):42–61, January 2007. 76 |
| 1. P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. IEEE Micro, 25(2):21–29, March/April 2005. DOI: 10.1109/MM.2005.35 7 |
| 1. V. Krishnan and J. Torrellas. A direct-execution framework for fast and accurate simulation of superscalar processors. In Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 286–293, October 1998. DOI: 10.1109/PACT.1998.727263 71 |
| 1. T. Kuhn. The Structure of Scientific Revolutions. University Of Chicago Press, 1962. 1 |
| 1. B. Kumar and E. S. Davidson. Performance evaluation of highly concurrent computers by deterministic simulation. Communications of the ACM, 21(11):904–913, November 1978. DOI: 10.1145/359642.359646 81 |
| 1. T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In IEEE 3rd Annual Workshop on Workload Characterization (WWC-2000) held in conjunction with the International Conference on Computer Design (ICCD), September 2000. 67 |
| 1. S. Laha, J. H. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. IEEE Transactions on Computers, 37(11):1325–1336, November 1988. DOI: 10.1109/12.8699 64 |
| 1. J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 291–300, June 1995. 51 |
| 1. J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. In Proceedings of the International Symposium on Code Generation and Optimization (CGO), pages 135–146, March 2006. DOI: 10.1109/CGO.2006.32 68 |
| 1. J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 236–247, March 2005. DOI: 10.1109/ISPASS.2005.1430578 68 |
| 1. J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In Proceedings of the 2004 International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 57–67, March 2004. DOI: 10.1109/ISPASS.2004.1291356 68 |
| 1. G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. Technical Report SMLI TR-93-22, Sun Microsystems Laboratories Inc., December 1993. 70, 76, 95 |
| 1. B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 185–194, October 2006. 35 |
| 1. B. Lee and D. Brooks. Efficiency trends and limits from comprehensive microarchitectural adaptivity. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 36–47, March 2008. DOI: 10.1145/1346281.1346288 31, 36, 104 |
| 1. B. Lee, D. Brooks, Bronis R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), pages 249–258, March 207. 36 |
| 1. B. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models. In Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 270–281, November 2008. 36 |
| 1. B. C. Lee and D. M. Brooks. Illustrative design space studies with microarchitectural regression models. In Proceedings of the International Symposium on High Performance Computer Architecture (HPCA), pages 340–351, February 2007. 36 |
| 1. C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In Proceedings of the 30th Annual IEEE/ACM Symposium on Microarchitecture (MICRO), pages 330–335, December 1997. 16 |
| 1. K. M. Lepak, H. W. Cain, and M. H. Lipasti. Redeeming IPC as a performance metric for multithreaded programs. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 232–243, September 2003. 59 |
| 1. D. J. Lilja. Measuring Computer Performance: A Practitioner’s Guide. Cambridge University Press, 2000. DOI: 10.1017/CBO9780511612398 xi, 65 |
| 1. M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 138–147, October 1996. 30 |
| 1. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI), pages 190–200, June 2005. DOI: 10.1145/1065010.1065034 22, 51 |
| 1. K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 164–171, November 2001. 9, 10 |
| 1. Y. Luo and L. K. John. Efficiently evaluating speedup using sampled processor simulation. Computer Architecture Letters, 4, September 2004. 70 |
| 1. Y. Luo, L. K. John, and L. Eeckhout. SMA: A self-monitored adaptive warmup scheme for microprocessor simulation. International Journal on Parallel Programming, 33(5):561–581, October 2005. DOI: 10.1007/s10766-005-7305-9 75 |
| 1. P. S. Magnusson, M. Christensson, Jesper Eskilson, D. Forsgren, G. Hallberg, J. Högberg nad F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. IEEE |
| Computer, 35(2):50–58, February 2002. 53 |
| 1. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset. ACM SIGARCH Computer Architecture News, 33(4):92–99, November 2005. DOI: 10.1145/1105734.1105747 55, 61 |
| 1. J. R. Mashey. War of the benchmark means: Time for a truce. ACM SIGARCH Computer Architecture News, 32(4):1–14, September 2004. DOI: 10.1145/1040136.1040137 11, 13 |
| 1. R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. IBM Systems Journal, 9(2):78–117, June 1970. DOI: 10.1147/sj.92.0078 54, 87 |
| 1. C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In Proceedings of the 2002 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 108–116, June 2002. DOI: 10.1145/511334.511349 55, 58 |
| 1. A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 145–156, October 1994. DOI: 10.1145/195473.195524 15 |
| 1. P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 2–10, October 1999. DOI: 10.1109/PACT.1999.807388 35, 45 |
| 1. D.Mihocka and S.Schwartsman. Virtualization without direct execution or jitting:Designing a portable virtual machine infrastructure. In Proceedings of the Workshop on Architectural and Microarchitectural Support for Binary Translation, held in conjunction with ISCA, June 2008. 54 |
| 1. J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distribuyted parallel simulator for multicores. In Proceedings of the |
| International Symposium on High Performance Computer Architecture (HPCA), pages 295–306, January 2010. 97, 100 |
| 1. C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), pages 35–46, September 2008. DOI: 10.1109/IISWC.2008.4636089 16 |
| 1. S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. HussLederman, and J. R. Larus. Wisconsin wind tunnel II: A fast, portable parallel architecture simulator. IEEE Concurrency, 8(4):12–20, October 2000. DOI: 10.1109/4434.895100 51, 97, 100 |
| 1. O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. Understanding the effects of wrongpath memory references on processor performance. In Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI) held in conjunction with the 31st International Symposium on Computer Architecture (ISCA), pages 56–64, June 2005. 55 |
| 1. S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application level architecture simulation. In Proceedings of the ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), pages 216–227, June 2006. 53 |
| 1. I. Nestorov, M. Rowland, S. T. Hadjitodorov, and I. Petrov. Empirical versus mechanistic modelling: Comparison of an artificial neural network to a mechanistically based model for quantitative structure pharmacokinetic relationships of a homologous series of barbiturates. The AAPS Journal, 1(4):5–13, December 1999. 32 |
| 1. A.-T. Nguyen, P. Bose, K. Ekanadham, A. Nanda, and M. Michael. Accuracy and speed-up of parallel trace-driven architectural simulation. In Proceedings of the 11th International Parallel Processing Symposium (IPPS), pages 39–44, April 1997. DOI: 10.1109/IPPS.1997.580842 95 |
| 1. A. Nohl, G. Braun, O. Schliebusch, R. Leupers, and H. Meyr. A universal technique for fast and flexible instruction-set architecture simulation. In Proceedings of the 39th Design Automation Conference (DAC), pages 22–27, June 2002. DOI: 10.1145/513918.513927 72 |
| 1. D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance. In Proceedings of the Third International Symposium on High-Performance Computer Architecture (HPCA), pages 298–309, February 1997. DOI: 10.1109/HPCA.1997.569691 92 |
| 1. S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 15–24, September 2001. DOI: 10.1109/PACT.2001.953284 81, 86, 88 |
| 1. S. Nussbaum and J. E. Smith. Statistical simulation of symmetric multiprocessor systems. In Proceedings of the 35th Annual Simulation Symposium 2002, pages 89–97, April 2002. DOI: 10.1109/SIMSYM.2002.1000093 91 |
| 1. K. Olukotun, B. A. Nayfeh, L. Hammond, K.Wilson, and K.-Y. Chang. The case for a singlechip multiprocessor. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 2–11, October 1996. 7 |
| 1. M. Oskin, F.T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor design. In Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA), pages 71–82, June 2000. DOI: 10.1145/339647.339656 81, 83, 86, 88 |
| 1. H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO), pages 81–93, December 2004. 70, 74 |
| 1. M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. S. Emer. Quick performance models quickly: Closely-coupled partitioned simulation on FPGAs. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 1–10, April 2008. DOI: 10.1109/ISPASS.2008.4510733 102 |
| 1. D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors. |
| Exploiting parallelism and structure to accelerate the simulation of chip multi-processors. In Proceedings of the Twelfth International Symposium on High Performance Computer Architecture (HPCA), pages 27–38, February 2006. 97, 102 |
| 1. C. Pereira, H. Patil, and B. Calder. Reproducible simulation of multi-threaded workloads for architecture design space exploration. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), pages 173–182, September 2008. DOI: 10.1109/IISWC.2008.4636102 59 |
| 1. E.Perelman,G.Hamerly,and B.Calder. Picking statistically valid and early simulation points. In Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 244–256, September 2003. 68 |
| 1. E. Perelman, J. Lau, H. Patil, A. Jaleel, G. Hamerly, and B. Calder. Cross binary simulation points. In Proceedings of the Annual International Symposium on Performance Analysis of Systems and Software (ISPASS), March 2007. 70 |
| 1. D. G. Perez, G. Mouchard, and O.Temam. MicroLib: A case for the quantitative comparison of micro-architecture mechanisms. In Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO), pages 43–54, December 2004. 61 |
| 1. A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In Proceedings of the Annual International Symposium on Computer Architecture (ISCA), pages 412–423, June 2007. 21, 25, 27, 62, 103, 104 |
| 1. J. Rattner. Electronics in the internet age. Keynote at the International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2001. 5 |
| 1. J. Reilly. Evolve or die: Making SPECâŁ™s CPU suite relevant today and tomorrow. IEEE International Symposium on Workload Characterization (IISWC), October 2006. Invited presentation. DOI: 10.1109/IISWC.2006.302735 21 |
| 1. S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The wisconsin wind tunnel: Virtual prototyping of parallel computers. In Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 48–60, May 1993. DOI: 10.1145/166955.166979 71, 97, 100 |
| 1. M. Reshadi, P. Mishra, and N. D. Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In Proceedings of the 40th Design Automation Conference (DAC), pages 758–763, June 2003. DOI: 10.1145/775832.776026 72 |
| 1. J. Ringenberg, C. Pelosi, D. Oehmke, and T. Mudge. Intrinsic checkpointing: A methodology for decreasing simulation time through binary modification. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 78– 88, March 2005. DOI: 10.1109/ISPASS.2005.1430561 73 |
| 1. E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. IEEE Transactions on Computers, C-21(12):1405–1411, December 1972. DOI: 10.1109/T-C.1972.223514 35 |
| 1. M.Rosenblum,E.Bugnion,S.Devine,and S.A.Herrod. Using the SimOS machine simulator to study complex computer systems. ACM Transactions on Modeling and Computer Simulation (TOMACS), 7(1):78–103, January 1997. DOI: 10.1145/244804.244807 53 |
| 1. E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. In Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 283–294, October 1998. DOI: 10.1145/291069.291063 71 |
| 1. J. P. Shen and M. H. Lipasti. Modern Processor Design: Fundamentals of Superscalar Processors. McGraw-Hill, 2007. 5 |
| 1. T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 3–14, September 2001. DOI: 10.1109/PACT.2001.953283 68 |
| 1. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 45–57, October 2002. 68 |
| 1. K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instructionwindow size, and cache size: Performance tradeoffs and simulation techniques. IEEE Transactions on Computers, 48(11):1260–1281, November 1999. DOI: 10.1109/12.811115 67 |
| 1. J. E. Smith. Characterizing computer performance with a single number. Communications of the ACM, 31(10):1202–1206, October 1988. DOI: 10.1145/63039.63043 11 |
| 1. A. Snavely and D. M.Tullsen. Symbiotic jobscheduling for simultaneous multithreading processor. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 234–244, November 2000. 9, 10 |
| 1. D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ILP processors. In Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA), pages 380–391, June 1998. 46 |
| 1. A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report 94/2, Western Research Lab, Compaq, March 1994. 22, 51 |
| 1. R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In Proceedings of the 1993 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), pages 24–35, 1993. DOI: 10.1145/166955.166974 54 |
| 1. D. Sunwoo, J. Kim, and D. Chiou. QUICK: A flexible full-system functional model. In Proceedings of the Annual International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 249–258, April 2009. 57, 102 |
| 1. P.K.Szwed,D.Marques,R.B.Buels,S.A.McKee,and M.Schulz. SimSnap: Fast-forwarding via native execution and application-level checkpointing. In Proceedings of the Workshop on the Interaction between Compilers and Computer Architectures (INTERACT), held in conjunction with HPCA, February 2004. DOI: 10.1109/INTERA.2004.1299511 71 |
| 1. T. M.Taha and D. S.Wills. An instruction throughput model of superscalar processors. IEEE Transactions on Computers, 57(3):389–403, March 2008. DOI: 10.1109/TC.2007.70817 45 |
| 1. N.Tuck and D.M.Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 26–34, September 2003. 7 |
| 1. D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA), pages 392–403, June 1995. DOI: 10.1109/ISCA.1995.524578 7 |
| 1. M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In Proceedings of the 35th International Symposium on Microarchitecture (MICRO), pages 271–282, November 2002. 61 |
| 1. M. Van Biesbrouck, B. Calder, and L. Eeckhout. Efficient sampling startup for SimPoint. IEEE Micro, 26(4):32–42, July 2006. DOI: 10.1109/MM.2006.68 73 |
| 1. M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. In 2005 International Conference on High Performance Embedded Architectures and Compilation (HiPEAC), pages 47–67, November 2005. DOI: 10.1007/11587514\_5 76 |
| 1. M. Van Biesbrouck, L. Eeckhout, and B. Calder. Considering all starting points for simultaneous multithreading simulation. In Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 143–153, March 2006. 78 |
| 1. M. Van Biesbrouck, L. Eeckhout, and B. Calder. Representative multiprogram workloads for multithreaded processor simulation. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), pages 193–203, October 2007. DOI: 10.1109/IISWC.2007.4362195 78 |
| 1. M. Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 45–56, March 2004. DOI: 10.1109/ISPASS.2004.1291355 78, 91 |
| 1. T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Simulation sampling with livepoints. In Proceedings of the Annual International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 2–12, March 2006. 73, 76 |
| 1. T.F.Wenisch,R.E.Wunderlich,M.Ferdman,A.Ailamaki,B.Falsafi,and J.C.Hoe. SimFlex: Statistical sampling of computer system simulation. IEEE Micro, 26(4):18–31, July 2006. DOI: 10.1109/MM.2006.79 7, 55, 66, 78, 79, 95 |
| 1. E. Witchell and M. Rosenblum. Embra: Fast and flexible machine simulation. In Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 68–79, June 1996. 51, 54, 71 |
| 1. D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In Proceedings of the 1991 SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 79–89, May 1991. DOI: 10.1145/107971.107981 75 |
| 1. R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In Proceedings of the Annual International Symposium on Computer Architecture (ISCA), pages 84–95, June 2003. DOI: 10.1145/859618.859629 64, 66, 67, 74, 77, 93 |
| 1. R. E. Wunderlich,T. F. Wenisch, B. Falsafi, and J. C. Hoe. Statistical sampling of microarchitecture simulation. ACM Transactions on Modeling and Computer Simulation, 16(3):197–224, July 2006. DOI: 10.1145/1147224.1147225 64, 66, 67, 74, 77, 93 |
| 1. J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), pages 266–277, February 2005. 70 |
| 1. J. J. Yi, D. J. Lilja, and D. M. Hawkins. A statistically rigorous approach for improving simulation methodology. In Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA), pages 281–291, February 2003. DOI: 10.1109/HPCA.2003.1183546 27, 34 |
| 1. J. J. Yi, R. Sendag, L. Eeckhout, A. Joshi, D. J. Lilja, and L. K. John. Evaluating benchmark subsetting approaches. In Proceedings of the 2006 IEEE International Symposium on Workload Characterization (IISWC),pages 93–104,October 2006.DOI: 10.1109/IISWC.2006.302733 29 |
| 1. J. J. Yi, H. Vandierendonck, L. Eeckhout, and D. J. Lilja. The exigency of benchmark and compiler drift: Designing tomorrow’s processors with yesterday’s tools. In Proceedings of the 20th ACM International Conference on Supercomputing (ICS), pages 75–86, June 2006. DOI: 10.1145/1183401.1183414 17 |
| 1. M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 23–34, April 2007. 55, 61 |