中国科学技术大学

硕士学位论文

高通量众核处理器模拟器性能优化

作者姓名：　方 国 庆
学科专业：　计算机科学技术
导师姓名：　安虹 教授
完成时间：　二〇一六年六月

University of Science and Technology of China

A Dissertation for Master's Degree

Performance Optimization of Simulator for High-throughput Many-core Processor

Author's Name: Guoqing Fang

Speciality: Computer Science and Technology

Supervisor: Prof. Hong An

Finished Time: June, 2016

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文,是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____ 签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

□公开 □保密（____年）

作者签名：_____ 导师签名：_____

签字日期：_____ 签字日期：_____

摘 要

随着大数据应用日益普及，数据规模高速增长，对处理器性能提出巨大挑战。高通量众核处理器在运行速度、单位面积能耗以及扩展性方面的改进成为计算机体系结构领域的研究热点。由于芯片制造工艺复杂、成本高昂，软件模拟器作为体系结构研究的重要工具在研究的许多阶段中都发挥着非常重要的辅助作用。软件模拟器最重要的评价指标是模拟速度，尤其是在大规模众核结构模拟中，提高模拟速度至关重要。

本文对详细地分析了现有的模拟器加速技术，并将其归类为提高单条指令或指令块模拟速度的技术、减少被模拟指令数量的技术以及利用硬件平台的并行性和提高模拟算法的并发度的技术。在此基础上，本文提出以下两个方案进行对高通量众核处理器的模拟进行改进与优化。

（1）基于优化单条指令或指令块的思路，本文提出使用查找表方法对指令译码进行加速。由于宿主机上拥有相对充足的内存资源，故可以将模拟器运行过程中需要反复进行的一些计算预先完成并将结果以表的形式保留在内存中，以后遇到同样的计算时，只进行一次查表操作即可。本文对译码过程中遇到的 PopCount 问题、条件域检查以及踪迹缓存等方面使用查找表技术进行加速。查找表技术的引入既提升了指令译码的模拟速度，又使得编码实现更加简洁灵活。

（2）基于利用硬件模拟平台并行性和提高模拟算法并发度的思路，设计实现了从多个角度优化的并行离散事件模拟框架。首先，模拟框架采用随机映射的事件调度算法，显著提高了模拟过程中的负载均衡性；其次，框架采用 cycle-by-cycle 的时间推进算法消除了 CMB 同步算法中高额的同步开销，这得益于以红黑树结构对事件队列的管理；同时，框架基于单写者单读者模型实现未来事件队列的无锁化，避免了事件调度过程中的大量锁开销。最后，在模拟过程中，大数据负载中海量的离散数据访问请求表现为组件频繁的内存操作，本文选择内存池方案对其进行管理。内存池方案以少数几次大规模内存分配代替频繁的小规模内存申请和释放，回避了动态内存管理方案固有的操作延迟，在提升内存操作效率方面成效显著。

代表性的大数据应用包括单词计数（WordCount）、T 级数据排序（TeraSort）以及模式匹配（KMP）被用于本文加速方案的性能评估。实验结果表明查找表方案将 PopCount 问题解决速度提高 26.14 倍，并行离散事件模拟框架的优化将总体性能提升 3.94 倍。

关键词：高通量；众核模拟器；查找表；并行离散事件模拟；内存池

Abstract

With the increasing popularity of Big Data applications, the scale of data grows rapidly, which is posing substantial challenges to the performance of processors. Improvement of high-throughput many-core processors in aspects of processing speed, power consumption per unit area and scalability turns out to be the research hotspot in the field of computer architecture. As a result of the high complexity and cost of chip manufacturing, software simulator as a key tool in the research of computer architecture plays an important role in several stages of architecture research. Since simulation speed is a key indicator in evaluating software simulators, especially during simulation of large-scale many-core architecture, it is critical that simulation speed should be improved.

This paper analyses the existing simulation techniques in detail and categories them as techniques that speed up every single instruction or instruction block, techniques that reduce number of instructions being simulated and techniques that make use of parallelized computing resources as well as improve the concurrency of simulation algorithm. Based on that, this paper proposes following two schemes to retrofit and optimize the simulation of high-throughput many-core processors.

(1) Based on the idea of improving the simulation speed of single instruction or instruction block, this paper proposes the lookup table method to speed up instruction decoding. Since memory resource in host machine is relatively adequate, we can save as a table results of calculations that arise repeatedly. In this way, other appearances of the same calculation can be replaced by a single action of looking up the table. The PopCount problem, inspection of instruction condition field and trace cache that occurred during instruction decoding are speeded up with the lookup table method. Not only does the introduction of lookup table method improve the instruction simulation speed, but it also makes the code implementation more concise and flexible.

(2) In order to make use of parallelized computing resources and concurrency of simulation algorithm, Parallel Discrete Event Simulation(PDES) framework that optimized in several aspects are designed and then implemented; Firstly, in this framework, the random event mapping model is applied as a event scheduling algorithm, which improves the workload balance of working threads significantly; After that, the cycle-by-cycle time

advance algorithm was applied in the framework to eliminate high overhead of synchronization, which benefits from the introduction of red-black tree data structure to manage future event list; Meanwhile, in accordance with the single writer single reader model, this paper implements the future event list in a lock-free way, which avoids the heavy lock overhead during event scheduling; Finally, vast amounts of discrete data access requests in big data workloads show up as frequent memory operations from components during simulation, the memory pool scheme is introduced to manage them. With the memory pool scheme, massive number of small size memory allocation and deallocation operations are replaced with a very few number of large scale memory operations, Thanks to the reduction in number of memory operations, the efficiency of memory management improves significantly.

Typical Big Data applications including WordCount, TeraSort and KMP are used as benchmarks in evaluation of proposed accelerating schemes. Experiment results show that lookup table scheme speeds up as much as 26.14 times in solving the PopCount problem. The overall performance achieves a 3.94 times improvement because of the optimization in PDES framework.

Key Words: high-throughput; many-core simulator; lookup table; PDES; memory pool

目 录

插图目录

表格目录

第1章 绪论

1.1 研究背景和意义

1.1.1 大数据应用对体系结构设计方法提出新要求

搜索引擎、社交网络、多媒体分析、生物信息等大数据应用越来越广泛地出现在日常生活中，快速高效的处理海量数据至关重要。而早期计算机系统与云计算系统的主要设计目标是充分利用 CPU 的计算能力，对于大数据应用这种需要满足持续数据存取要求的应用场景考虑甚少。新兴大数据应用与传统计算机系统的负载存在本质性的区别，传统体系结构设计方法面临着多方面的挑战，迫切需要进行变革性的设计。

传统信息系统的组织是“数据围绕处理器流动”，在大数据应用场景下，数据量越来越大，移动数据所需要的时间开销也越来越大，进而不可被接受。而偏向“处理能力围绕数据流动”设计的优势也越来越明显。在新的设计下，除了以往单纯的数据加工，更重要的是关注对数据的“搬运”和对并发处理需求的满足

因此，“提高系统吞吐率和并发处理能力”正在代替“重视单任务的完成时间”而成为系统结构设计的出发点，高通量处理器的并发执行能力要提高到 10 亿级以上的规模才能够适应大数据应用需求。

如前所述，构建能够处理海量数据的、以数据为中心的计算机系统，最基本的要求是要消除不必要的数据流动，将更多的运行时间花费在数据处理上。将复杂的任务简化拆分、以大量弱核共同解决，而非以往用少量功能强大的通用核处理复杂任务。形象的说，新的设计思路要求以“蚂蚁搬大米”代替“大象搬木头”（李国杰，2013）。因此，不必要的数据存取、通信和计算是以数据为中心的系统结构必须要消除的。

高通量应用环境下，计算机系统结构将会朝什么样的方向发展？“计算机体系结构”概念最早由 IBM 研制 S/360 时提出。为了评估计算机系统的性能，当时一个重要的技术发明是区分了定点与浮点计算，将每秒浮点运算次数作为比较计算系统的一个重要指标。相应的，高性能计算机的优势主要体现在对科学计算中大量出现的浮点计算的加速上。而在高通量应用环境下，这个技术指标需要重新考虑，是否存在类似的技术突破？能否基于纷繁复杂的负载归纳出类似“定点与浮点”的基本操作，作为评估高通量计算系统的性能指标？

由于许多大数据应用主要是做数据的搬运，而不是数据加工，这导致大数据应用中包含的整数计算更多，因为数据移动一般不需要浮点计算。虽然商用处理器上已经可以达到非常高的理论浮点计算峰值，但是多数大数据处理过程中得到的实际结果与之相去甚远，单纯的采用传统 CPU 势必难以高效的运用于大数据处理。传统体系结构研究的一大核心问题是多级缓存的设计，但是通过对众多大数据应用的研究（Michael Ferdman et al, 2015）发现：在多数大数据应用中，一级缓存命中率较高，发挥着良好的掩盖延迟的作用，而二级和三级缓存的命中率很低，几乎不发挥作用。这说明在很多大数据应用中，数据的局部性很差，二级和三级缓存的设置是没有必要的。Michael Ferdman 等人的研究还发现，带宽利用率低是目前高通量计算最大的问题，而不是芯片内数据通道带宽低。因此适合进行大数据处理的计算机系统应当是以提高系统一段时间内的吞储量作为研究目标，并且重点关注实际带宽利用率的提升。

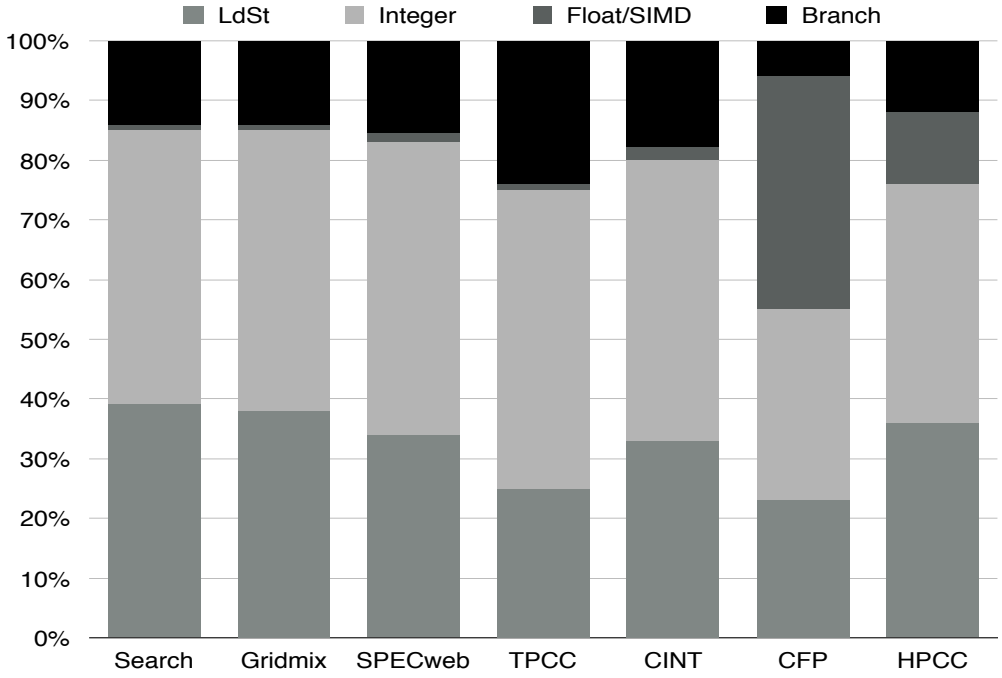

图 1.1 不同基准测试程序的指令特征（詹剑锋 等，2011）[41]

图 1.1 和图 1.2 给出了在相同 4 核英特尔 Xeon E5310@1.60GHz 平台上，搜索引擎类基准程序、Grid Mix、 SPEC Web 2005、TPCC、SPEC CPU 和 HPCC 应用在指令特征（包括 branch、float/SIMD、integer、load/store）和 CPI(cycle per instruction, 平均指令周期数)方面的差异。

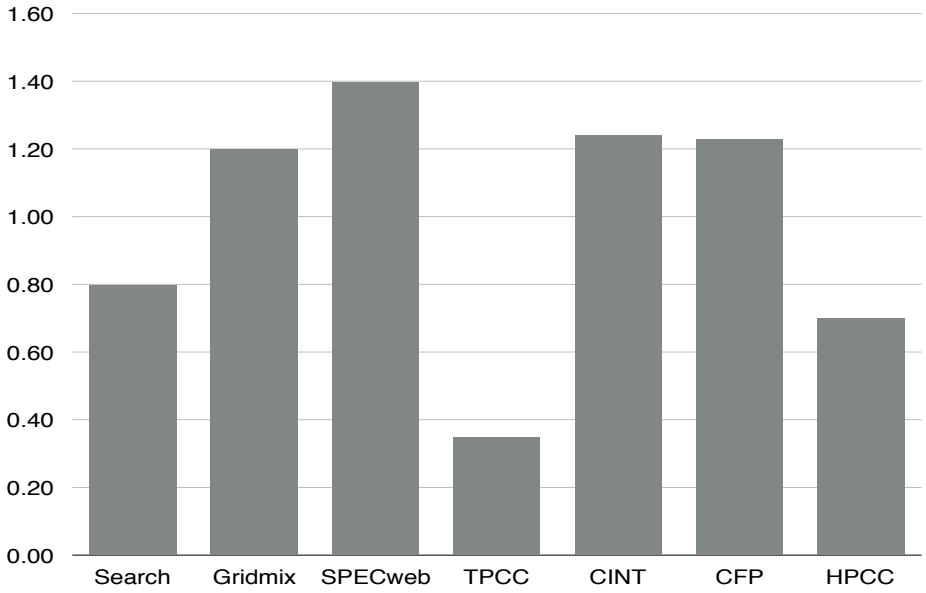

图 1.2 不同基准测试程序的 CPI（詹剑锋 等，2011）[42]

应用特征分析表明，大数据应用的负载特征与传统处理器体系结构设计存在明显的冲突，包括：

（1）大数据应用中非格式化的数据与处理器规则的数据格式不匹配；

（2)大数据应用中以数据为中心的特点与处理器设计中围绕计算资源的数据通路不匹配；

（3）特定数据处理需求与通用的结构设计不匹配；

（4）任务实时处理需求与现有处理器设计中的多级预测性特点不匹配。

针对大数据处理的需求进行 CPU 设计时应当将高吞吐、低延迟、易扩展、强实时等特性作为重点提升目标。

1.1.2 高通量系统与高性能系统的对比

谷歌、亚马逊、阿里巴巴等越来越多的企业通过互联网向大众提供计算资源、信息资源。不同于传统的高性能计算机，数据中心的计算机系统中存在着大量的应用请求和弱耦合性作业，它们是数据中心计算机系统的工作重点。这些请求之间相对独立，作业中的各个任务也鲜有通信。在服务这些请求或执行这些作业时，不可避免地，系统需要处理并分析海量的数据。

作为数据中心类的计算机系统，高通量计算机以高效处理高度并发且相对独立的负载为设计目标。

总结上述分析，我们发现，相比于传统高性能计算机，高通量计算机存在以下四个方面的显著特点：

- 高内聚、低耦合。由于全局通信和同步操作的普遍存在，传统并行应用存在高度的耦合性。而高通量计算系统为处理的大量独立分散的请求，为保证较高的可扩展性，多个服务莉程执行相同的代码，并且数据存在多个备份。由于每个服务实例的工作高度内聚，并且服务实例之间耦合度极低，因而具有良好的可扩展性。
- 以高通量为性能目标。高通量计算注重提高长时间跨度下单位时间内服务的请求数或提交的任务数。而传统大规模并行计算是以缩短单个应用的运行时间为关注的焦点。
- 系统复杂。以 MPI、OpenMP 为代表的传统科学计算系统具有高度优化的编程模型和以 intel vtune、Linux perf 为代表的的性能分析工具，极大的提高的程序设计和性能调优的效率。而以基于谷歌自己开发的 Map Reduce 和 Big Table 分布式文件系统的搜索引擎服务为代表的高通量计算应用系统则通常是有多个生产厂商或开源团队合作开发的结果，往往具有复杂的系统构造，并且缺乏统一的接口，可移植性差。
- 以多副本提高可靠性。传统的大规模并行计算解决可靠性问题的主要方法是的设置检查点，在任务出错的情况下通过回滚回到来正确状态。高通量计算应用则通过保存多个数据副本和任务的重新执行来消除可能出现的错误状态。在此类系统中，一个作业通常由多个可以彼此独立执行的任务组成，任务可以在失效时被可重新执行。

表 1.1 对两种计算系统的各方面差异进行的详细的比较。

表 1.1 高通量计算系统与高性能计算系统对比

对比项目	应用负载特征	并行性	性能成本	可靠性	性能目标
高通量计算机系统	网络、数据中心应用	硬件直接支持的多线程特性	影响企业竞争力和生存空间	数据多备份、单个服务实例的失败可通过重复运行的方式弥补	高通量：以单位时间段内系统提交任务总数为目标
	海量数据、高度并发				
	任务多样、负载变化显著				
	任务耦合低、数据局部性差				
高性能计算机系统	科学和工程计算	并行性提升成本高、难度大	性能优先、兼顾成本	系统相对脆弱，组成部分的错误都引起整体运行结果偏差	高速度：以尽快完成单个任务为目标
	类型单一、负载稳定但耗时				
	数据局部性好				

1.1.3 高通量系统对模拟器设计提出新要求

作为体系结构研究中的主要工具，高通量计算机系统的研究也先基于模拟器进行评估。顾名思义，与单核及多核模拟相比较，众核模拟最显著的变化是模拟的规模得到了巨大的提升。并且，由于新摩尔定律的存在，模拟的规模还在迅速的增长。为同时处理高度并发的请求，高通量众核处理器中需要配置大量的低耦合的、具有简单处理能力的核。以具有类似功能需求的 NVidia Kepler 架构的 GPU 为例，最多包含 15 个升级版本的流多处理器（SMX），共计 2880 个核（SP）。伴随着众核体系结构下核数量的增加，相应的功能部件数量也呈现出迅速的增长。在如此大的设计空间下，要获得一个综合表现突出的结构设计必定伴随着大量复杂的模拟，涉及到的细节越多，模拟复杂度越高。然而，承载模拟器的宿主机的性能却无法以同等的速度增长，近年来，处理器主频增长越来越慢以致趋于停滞。现有串行模拟器的模拟速度远无法满足高通量众核处理器模拟的性能需求，并行化的模拟成为必然选择。

虽然大规模并行模拟的研究已有诸多成果，相比于串行模拟，其模拟速度和规模都有了质的提高。但即使是典型的模拟系统如 MARSS（Avadh Patel et al, 2011）、MPI-Sim、LAPSE，其模拟规模也只停留在百个处理结点，并且无法进行时钟精确的性能模拟。面对高通量体系结构中千核万线程规模的模拟需求，现有模拟系统力有未逮。为快速地对高通量众核体系结构进行模拟，应当高效地处理以下两个方面：

（1）单条指令模拟速度的提升。作为执行驱动的模拟中最基本的模拟对象，指令的模拟速度对整个系统的推进具有重要意义。

（2）并行模拟框架的改进和优化。在体系结构并行模拟中，无法避免地涉及到从模拟对象到执行线程的映射和不同执行线程的同步。高效地解决这两个问题是提高模拟框架性能的关键。此外，并行模拟框架设计中昂贵的锁开销和大量内存操作开销也应当尽量避免。

1.2 BDSim并行模拟框架

1.2.1 并行离散事件模拟算法

离散事件模拟（Discrete Event Simulation, DES）是一种用于研究状态呈离散方式变化的系统的模拟方法（Wang, Jingjing et al, 2014）。当用于执行仿真模型的环境是多核或多处理器系统时，这种模拟方法称为并行离散事件模拟。这种方法已经广泛应用于包括计算机和通信系统、生物网络、军事战争游戏、在线游戏和经营管理在内的众多领域中的系统评估和分析。PDES 中非规则的、数据依赖的本质使其被

认为是一类难以使用超级计算机硬件的向量化技术提供性能收益的应用。日益增加的对模拟速度的需求形成了对串行模拟器的挑战。这种环境下，并行离散事件模拟以挖掘模拟模型中天然存在的并行处理能力，显著提升了模拟器的性能和容量，使得在更少的时间内研究更耗时、更细化的模型成为可能。

离散系统中涉及到的一些关键概念（Ostermann S et al, 2010）包括：

（1）事件（event），是指改变系统状态的即时操作。需要注意的是，由于事件的传递通过消息的收发进行，一个事件对应于一条消息，同时也对应一个模拟任务，因此，本文的描述中并不对三者进行明确的区分；

（2）未来事件队列（future event list, FEL），是一组按照发生时间排列的未来事件；

（3）时钟（clock），是代表当前模拟进度的变量；

（4）实体（entity），代表当前系统中任何需要明确表示的对象或组件；

（5）系统状态（system state），是指包含任何时刻为了描述系统所必需的所有信息的变量集合。

此外，一个完整离散事件模拟系统还需要：

（6）时间推进算法，用于确定事件的模拟顺序，并在没有外部事件发生时推进模拟时钟；

（7）事件调度算法，负责在模拟框架中完成由事件向工作线程的映射，并按照合理顺序处理事件。

图 1.3 在逻辑功能单元的粒度下对串行与并行二类离散事件模拟方式之间的区别与联系进行了展示。由图 1.3 易知，若干逻辑单元组成串行模拟模块，同时若干串行模块遵循时间轴并行工作则组成并行模拟系统（林健 等， 1998）。

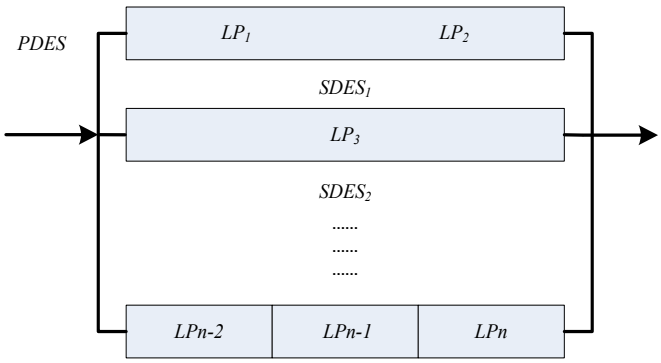

图 1.3 SDES 与 PDES 的关系（李文明 等，2015）[4]

1.2.2 BDSim 并行模拟框架

中国科学院计算技术研究所针对大数据应用开发出 BDSim 并行模拟框架（李文明 等，2015）[4-5]。在该框架的主体设计中主要包含组件、端口、框架服务单元等基本概念。下面对这些概念进行简要介绍。

组件（Component）：逻辑功能单元。基于指定的接口设计协议，模拟框架用户可以自行设计需要的模块，这些模拟以链接库的形式保存，在模拟框架配置阶段被动态载入。

端口（Port）：消息通信处理单元。端口是组件属性之一，组件为每个与其相连的其他组件保留一个端口，供消息收发使用。完成目标模型的拓扑结构配置后，系统为每个端口分配一个全局编号，消息生成时会将端口编号嵌入其中，供路由和寻址之用。

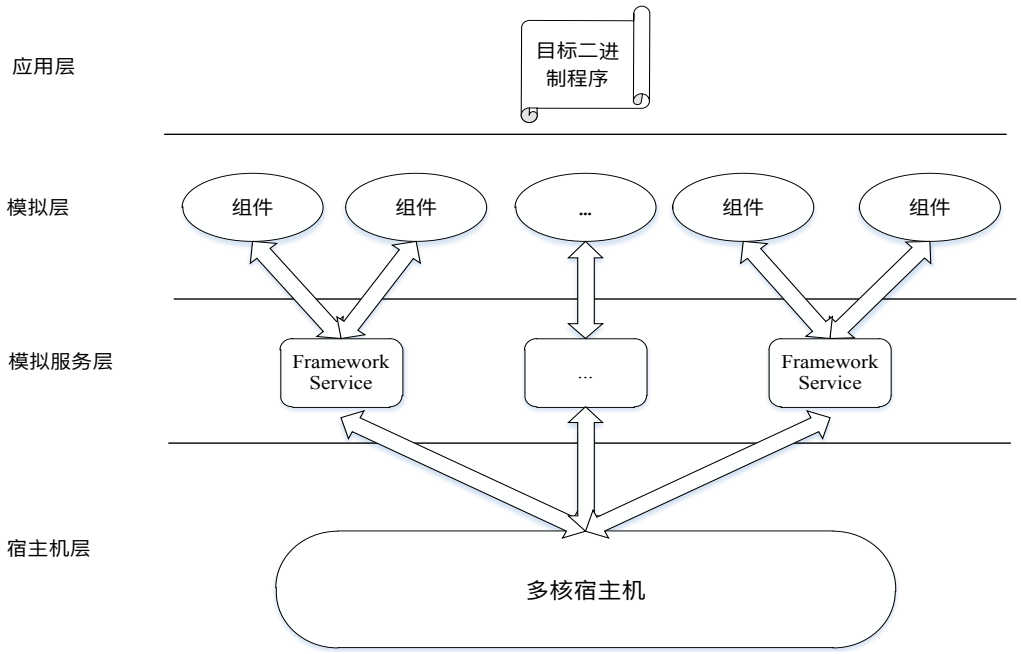

图 1.4 BDSim 框架设计（李文明 等，2015）

FS（Framework Service）：框架服务单元。框架服务单元在实现上基于宿主机上的线程，因而是框架并行化的基本单元。各个组件按照一定的方式将自己挂载到框架服务单元上。如图 1.4 所示。FS 完成对组件设计中必须处理的拓扑结构配置接口、通信细节处理、事件同步等重要问题的处理，是组件功能的实际执行者。借助于 FS，一方面用户只关注于组件的功能设计与实现，繁杂的并行通信和同步处理工作交由

FS 处理，大大提高模拟器使用者的生产效率；另一方面，FS 的引入充分利用了宿主机上高度并行化的计算资源，提高了系统的可扩展性，众核并行模拟成为现实。

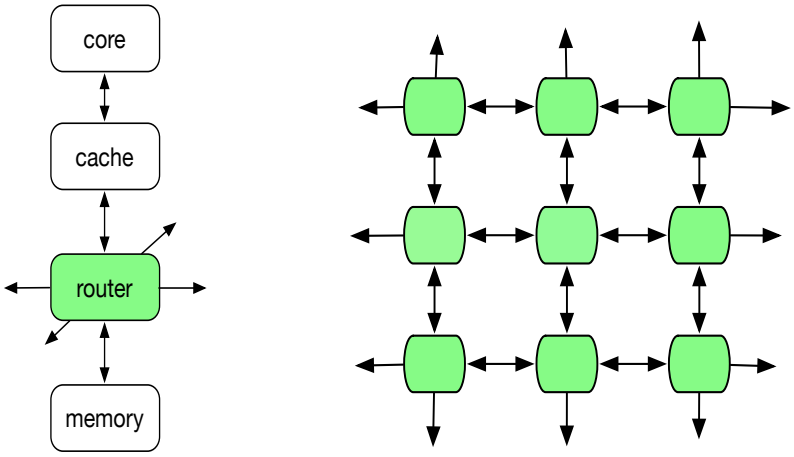

图 1.5 组件纵向连接及横向连接

根据并行模拟框架的设计，每个组件的端口被分配作特定的用途，组件之间的拓扑配置由其端口间互相连接而确定。本文中实验采用所示的拓扑配置。图 1.5 左侧展示了不同类型组件之间的连接结构：以 *router* 组件为核心，*router* 组件往上依次是 *cache* 组件和 *core* 组件，下侧是 *memory* 组件。*Router* 组件还剩余左右两个端口，用于与其他 *router* 组件相连，如图 1.5 右侧所示。*Router* 组件之间连接形成的片上网络的结构是可配置的，实验中使用的是 32x32 的 *mesh* 结构。*Core* 组件运行时采用 SMT＝8 的配置，共计 1024 个 *core* 组件，8192 个模拟线程，这使得被模拟的结构达到千核万线程的规模。

图 1.6 展示了 BDSim 的通信协议栈。BDSim 采用类似 TCP/IP 协议的分层结构。整体分为三层，每一层按照对应的消息协议，由 FS 完成层内消息的接收、处理和转发等工作，层次之间功能隔离、消息传递透明。其中 BDSim 层由同步层与通信层两个子层组成，维护组件之间同步，并完成消息通信的功能。

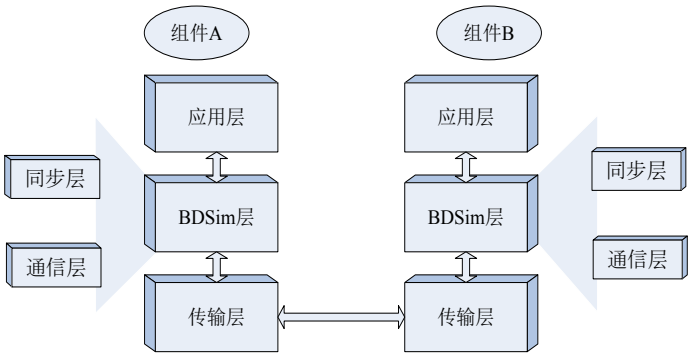

图 1.6 BDSim 通信协议栈（李文明 等，2015）[5]

组件间消息通信的方式为：组件接收来自相邻组件的消息，分拆消息提取本组件消息处理所需要的信息，并依据所依赖信息的不同决定不同的处理方案。例如，当一个 router 组件发现其接收到的消息是发送到与其相连的 memory 组件或 cache 组件时，那么它将会把打包消息分片，然后从端口 A 将消息发送到相连的 memory 组件或 cache 组件；否则，该 router 组件会进行路由选择，根据路由算法确定合适的下一跳后，选择合适的端口 B 将该消息或分片转发给相连的 router 组件。这些过程都是由 FS 完成，其中省略了对端口、组件编号映射过程的描述。

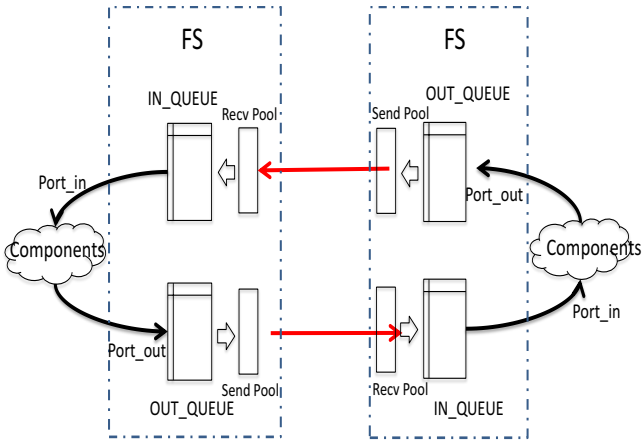

图 1.7 BDSim 通信结构

BDSim 并行模拟框架为所有 FS 分别配置私有的输入消息队列和输出消息队列，其中“输入”与“输出”概念的方向性是相对组件而言的。如所示，框架与上层模拟组件之间通过 Port_out 接口和 Port_in 接口进行消息通信。这两个接口像桥梁一样连接模拟框架与上层模拟器，承载消息通信的任务。在模拟器设计中，这两个接口由组件提供，不同的模拟组件其接口实现细节略有不同。一方面，组件通过 Port_out 发送消息到 FS 模块的 OUT_QUEUE。反过来，FS 在处理消息过程中，如果有新消息产生，将通过 Port_in 接口从 FS 的 IN_QUEUE 发送给模拟组件。如图 1.7 右侧所示。按照这种消息处理方法，不同组件之间的消息传输得以非常简便地实现。

1.3 BDSim 模拟框架的不足

1.3.1 BDSim 模拟框架并行性能表现

本文的实验是在 Quad-Core AMD Opteron 处理器上完成的，具体的支持模拟器运行的软硬件环境如表 1.2 所示。

表 1.2 宿主服务器配置

软件平台	
操作系统	CentOS Linux release 7.0.1406 (Core)
编译器	GCC 4.8.2
硬件平台	
CPU	4 块 Quad-Core AMD Opteron™ Processor 8347 HE
	每块 4 核，共 16 个物理线程
L1 Cache	私有，每个核有 64KB I-Cache 和 64KB D-Cache
Memory	62GB

为了更真实地对被模拟的高通量众核体系结构进行测试，本文以 wordcount、terasort、kmp 等典型高通量应用作为主要基准测试程序。针对其中某些性能指标，还选择不同规模的输入集分别进行测试。除此之外，为使得本文的研究成果更具说服力，还增加了典型高性能应用——LP（线性规划），以及简单的功能测试程序 test-math、test-add、test-sort。

统计 1 至 7 个工作线程的配置下线程等待时间如图 1.8 所示，横轴中从 1 增加到 7 的是工作线程数量。两条曲线分别统计了多线程与单线程配置下模拟器运行总时间的比值和每种线程数量配置下线程空等时间占总运行时间的比例。从图 1.8 中可以看到，对于使用多线程方式运行的模拟器，使用框架线程数增加并没有带来整体运行时间的降低。分析其运行时间的组成，可以发现，随着线程数的增加，线程在等待新消息到来所花费时间的比例逐渐增加，该比例从单个工作线程时的 30%提高到 7 个工作线程时的 80%以上，实际执行模拟任务的时间从 70%降低到 20%以下。

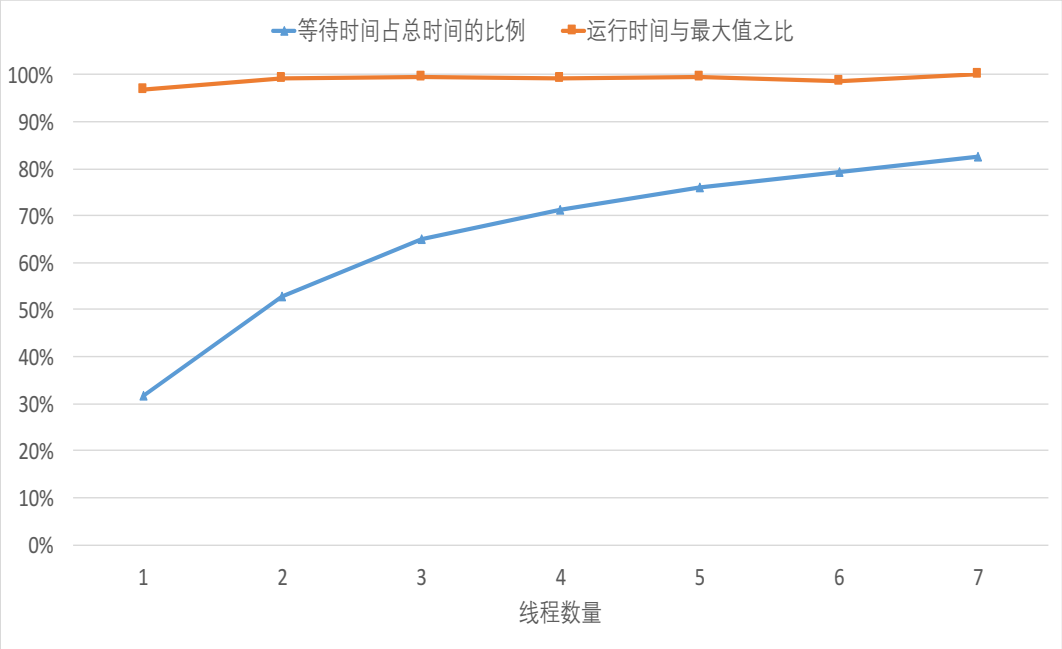

图 1.8 线程数量与运行时间及等待时间的关系

利用性能分析工具 perf 分析发现，模拟框架中最耗时部分实现的功能是：工作线程试图从自己的未来事件队列中获取待处理的消息，如果有新的待处理事件则获取之，否则继续检查该队列。这一功能耗费高达 55.02%的运行时间，说明工作线程不能被持续地喂饱，很可能出现了系统负载分配不均衡的问题。提高模拟器在利用宿主平台多线程资源情况下的性能，亦即提高模拟器的线程可扩展性，最重要的工作在于消除等待时间瓶颈、优化并行模拟框架中最耗时的事件调度开销。

1.3.2　锁操作开销

各个工作线程与主线程以事件队列作为通道进行通信：工作线程从事件队列中取出一组事件在本轮循环内进行处理，在处理结束后，带回新产生的事件。

BDSim 的未来事件队列设计中，调度线程与工作线程之间的通信借助于一个全局的消息队列，并使用一个互斥锁隔离各个线程的消息存取。大锁结构带来的问题是：在消息粒度较小而使得消息存取较频繁的情况下，多个线程将会频繁地申请该锁，存取消息的过程将变得严重串行化，从而成为事件调度功能的瓶颈。

1.3.3 负载均衡性

并行模拟框架中的“映射”包含两种情形：模拟任务到逻辑线程的分发以及逻辑线程到物理线程的匹配。高通量应用中消息或请求之间具有极低的耦合性，线程之间由于处理模拟任务而产生的通信可以忽略不计，本文简单地选择逻辑线程与物理线程之间进行一一对应的直接映射方式。因此，本文关注的映射专指从模拟任务到逻辑线程的事件分配过程。

框架调度的对象是系统中所有组件产生的消息。在高通量众核结构模拟中，这些组件包括 *core*、*memory*、*cache* 和 *mesh* 等，共计四类三千余个。BDSim 模拟框架中，组件按照其编号划分成若干组，每组组件的消息分发给一个工作线程处理。如图 1.9 所示，模拟层中连续若干个组件将消息统一分配给同一个框架服务单元（实现时对应于一个工作线程）

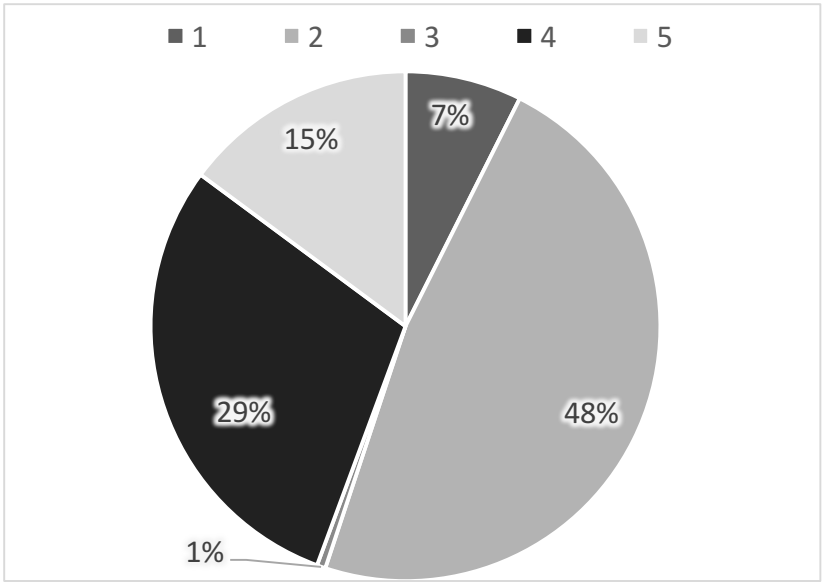

图 1.9 固定负载分配方案概率分布(*min_load* = 128)

采用固定映射策略的事件调度算法中，消息与工作线程的映射方式实现简单，且兼顾了组件大类之间的均衡性。但是由于模拟器面向的高通量应用具有任务多样、负载变化显著的特征，这种情况下，采用固定映射策略的模拟框架在实际中可能会

导致不同工作线程的工作量严重不均衡。组件的编号由其组件的类型及在拓扑中位置决定，同一类型的连续多个组件将被映射到同一个工作线程上。并且，不同类型的组件，其消息数量、延迟等属性特征差别较大。这两个原因很容易导致负载分配不均，多线程优势无法得到充分发挥。

更重要的是，BDSim 模拟框架使用的事件调度算法是依次为每个工作线程分配 *min_load* 数量的待处理事件，完成一个工作线程的任务分配之后才进行下一个工作线程的任务分配。由于 *min_load* 恒定不变，此种固定负载方式简洁明了，编码难度低。但是，同样由于 *min_load* 取值固定不变，没有考虑工作线程当前的负载压力，很容易造成工作线程之间负载失衡：在待分配事件总数有限的情况下，分配顺序靠前的工作线程优先被分配到 *min_load* 数量的事件，而分配顺序靠后的线程则很会被分配到非常少量的待处理事件，甚至一个事件都分配不到。*min_load* 取值为 128 的固定负载分配方案下，使用 5 个线程运行 kmp 测试算法的情况下，关注各个线程在每一轮事件分配中是否能够被分配到新的待处理事件。根据统计结果，按照百分比划分得到图 1.9。其中 5 个线程随机编号为 0～4。从图中各成分的比例可以发现，各个线程之间被分配到待处理事件的比例严重不均：1 号线程占用了 48%的事件分配次数，几乎是第二名的两倍，第三名的三倍，最不受该算法偏爱的 2 号线程仅得到不足 1%的分配机会。

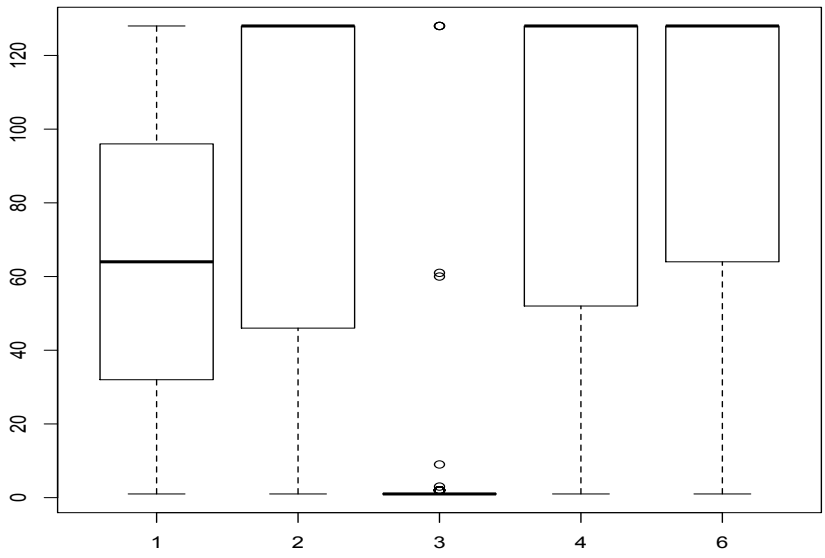

图 1.10 固定负载分配方案事件分布(*min_load* = 128)

此外，如图 1.10 所示，统计每次分配得到待处理事件的数量，使用箱线图统计。线程按照 1、2、3、4、6 随机编号。从图 1.10 可以看出，各个线程受该算法偏爱程度由强到弱排序依次为线程 6、线程 4、线程 2、线程 1 和线程 3，线程 3 偶尔能够分配到接近 min_load 数量，但是概率极低，多数情况下根本分配不到任何待处理事件。而线程 6 几乎总能分配到 *min_load* 数量的待处理事件。负载分配失衡由此可见一斑。

分配到充足待处理事件的工作线程能够持续工作下去，充分利用既有的计算资源；而分配到少量待处理事件的工作线程将很快完成分配的任务，进入空闲状态。由于负载分配失衡，引发对多线程计算能力的极大浪费。这种固定负载分配导致的负载失衡问题随着 *min_load* 取值的增加表现愈加显著。但是，在 *min_load* 取值较小时，每个线程很快的完成被分配的任务而进入请求新任务的状态，若待处理事件总量较大，则会导致频繁的事件调度，这部分开销不容忽视

1.3.4 CMB 同步算法

一个事件可以影响存储在 FEL 中的其他事件，导致它们被移除或者修改，这就导致模拟结果对事件处理顺序有至关重要的依赖性。并行离散事件模拟的推进要求简单方便的处理多个功能模块间事件发生的先后关系。按照推进策略激进程度的差异，将完全依据按时间戳大小次序对事件进行管理的 PDES 同步算法称为保守同步算法，否则称为乐观同步算法。为了在保证精确性的基础上实现并行模拟，本文中框架的实现采用保守的同步算法。

CMB 算法是经典保守同步算法，它在分布式模拟中有着广泛的应用。其根本原则是组件间依据单调非递减的次序处理事件，该算法为每一对连接管理独立的事件接收列表，并记录事件时间戳的最小值（李文明 等，2015）[7-8]。最小时间戳的逐步增大代表着系统模拟进度的持续推进。

基于 CMB 算法更新所有组件的最小时间戳是一个循环迭代的过程，容易引起死锁。在对象数目较多，且组件上下游关系形成若干长链或者长环的情况下，迭代的次数将快速增加，以至于成为了模拟框架的性能瓶颈。

1.3.5 动态内存管理

高通量计算机系统主要面向网络、数据中心类应用。这类应用中通常要求处理海量的用户数据，并发响应众多的应用请求。反映在模拟框架中就是大量旧消息的

释放和新消息的产生。如果将线程私有数据中消息处理过程展开，我们可以发现旧消息如何被解剖、销毁，以及新消息如何产生、传递。

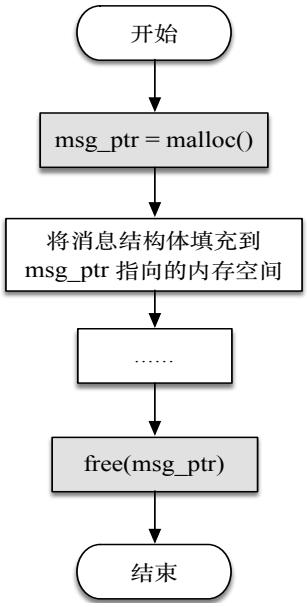

图 1.11 动态管理方案下的内存操作

为了完整地了解一个特定消息的生命周期，现在改从消息自身的角度观察。根据框架处理流程，一个消息在不同阶段中经历的操作概括如图 1.11 所示。

当产生一个新消息时，库函数 malloc()将被调用，为之分配必要的内存空间，用于存储消息中必要的时间戳、发送方、接收方、消息内容等信息；在接受方对该消息进行各项处理之后，库函数 free()将被调用来释放对应的存储空间。

然而，对内存的分配、追踪及释放操作的过程非常复杂，直接调用标准库中的内存处理函数又伴随着诸多弊端，包括：

（1）基于特定匹配算法的内存空闲块表操作，如内存块分割、空闲块合并等，这将引入额外的时间和空间开销。

（2）如本文遇到的频繁的内存申请、释放情况，很容易出现大量内存碎片，影响程序性能的提升。

（3）不正确的内存空间释放行为极易导致内存泄漏或引发段错误。

1.4 论文研究目标和主要工作

高通量众核体系结构千核万线程规模的模拟对模拟器的性能提出巨大挑战。BDSim 模拟框架的提出大大加速了千核万线程规模体系结构的模拟，但其在多线程

模式的配置下仍然存在诸多性能缺陷。本文致力于从指令译码和并行离散事件模拟细节的角度对基于 BDSim 框架的众核并行模拟器进行性能优化。

1.4.1 以查找表技术加速指令译码

指令或指令块是执行方式驱动的模拟器中最基本的模拟对象，基于提高单条指令或指令块执行速度的思路，本文采用查找表技术加速指令译码。查找表技术基于以空间换时间的策略，它充分利用模拟平台中相对充足的内存空间资源，将计算结果保存在内存中，从而减少花费在后续重复计算上的时间，以此获取更快的模拟速度。

本文中，查找表技术可以应用于 PopCount 问题、指令条件域判断、踪迹缓存等方面。

1.4.2 多角度优化的并行离散事件模拟框架

基于提高模拟器并发度的思路，本文依据并行离散事件模拟算法，实现了组件化的模拟框架。不同组件之间如果需要通信，则发送消息到模拟框架，由框架根据所属组件的类型，调用相应组件的处理算法完成对消息的处理或转发。基于利用模拟平台并行性的思路，对组件间海量消息的处理可以利用宿主平台的多核资源完成，每个核承担一部分消息的处理，整体执行时间将显著减少。

本文的研究工作在 BDSim 并行模拟框架上进行，针对 BDSim 并行模拟框架的不足，本文实现了如下的改进和优化：

（1）未来事件队列：从全局互斥锁到无锁化队列，消除未来事件队列中昂贵的锁开销;

（2）事件调度算法：从固定映射到随机映射，采取动态分配的策略保证多个工作线程间的负载均衡;

（3）时间推进算法：从 CMB 同步算法到 cycle-by-cycle 模型，使用红黑树管理全局事件队列以避免频繁的同步操作;

（4）内存操作：从动态内存管理到内存池方案，以对空闲链表的维护代替内存管理。

1.5 论文结构

本文共计六章，每一章的内容简要介绍如下：

第一章为绪论，首先介绍了论文选题背景及意义；然后对 BDSim 并行模拟框架的设计进行的简要介绍；特别指出 BDSim 模拟框架在多线程模式运行时存在的不足；最后对论文研究目标和主要工作进行说明。

第二章为相关工作，在对常用模拟器的研究基础上，将常用模拟器性能优化技术概括为三个主要思路：提高单条指令模拟速度、缩小待测程序规模以及并行化模拟。对每种加速思路给出其常用手段、典型应用及适用性分析。

第三章从 PopCount 问题、指令条件域检查和踪迹缓存三个角度介绍查找表技术在提升指令译码速度方面的应用，并且以 PopCount 问题为例，说明查找表技术的性能优势。

第四章从未来事件队列、时间推进算法、事件调度算法以及池化内存管理四个方面介绍对 BDSim 并行模拟框架的改进和性能优化。

第五章为全文总结，是对本文的工作回顾和概括，并阐述了本文工作的主要创新点和对未来研究工作的展望。

第2章 模拟器性能优化技术

传统串行模拟器无法满足高通量体系结构研究对大规模并行模拟的速度需求，但在以往模拟器研究中总结出的模拟器加速技术仍然具有重要的借鉴意义。本章通过对现有体系结构模拟器的研究，总结模拟器加速技术。本章将模拟器加速技术按照加速思路的差异划分为三类：提高单条指令或指令块的执行速度、减少模拟指令数量以及利用硬件平台并行性且提高模拟算法并发度。

2.1 提高单条指令或指令块的模拟速度

执行驱动（又作程序驱动，是指将为被模拟体系结构编译生成的二进制程序直接作为模拟器的输入）的模拟器，其执行的最基本对象是测试程序中一条条二进制指令。直观上看，如果可以提高每一条指令的执行速度，那么整个模拟器的运行时间也将会显著减少。这类加速技术包括各类软件调优及编译优化技术、直接执行（Miller J.E. et al，2010）、二进制翻译技术以及 FPGA 硬件加速等。

2.1.1 二进制翻译

SimOS、QEMU（Bellard Fabrice，2005）等模拟器使用指令二进制翻译和块链接的技术大大提高了模拟器的执行速度。二进制翻译是指将为待模拟系统结构生成的可执行程序转换为在宿主平台可执行程序的方法（李剑慧 等， 2007），通过在宿主机上运行转换后的指令达到跨指令系统兼容且加速模拟的效果。这种技术通过两种体系结构指令之间的转换，实现了以一条或多条宿主机上的指令模拟一条目标系统指令的功能。相比于使用穿线码技术的解释型模拟器，以 QEMU 为代表的引入二进制翻译的模拟器具有更高的执行速度。其实现方法如下：

（1）构建微操作集合。定义一个精简的微操作集合，要求能够完备地描述目标体系结构指令集中的每条指令。利用构建好的微操作集合，建立从被模拟指令到微操作集合一个子集的映射关系。

（2）生成动态代码产生器。将(1)中微操作集合内的每个微操作用 C 语言函数实现，编译产生中间目标文件。利用目标文件，得到动态代码产生器，其作用是拼接若干微操作，以实现对一条目标机器指令的模拟。

（3）提取目标代码。利用工具分析每个微操作编译产生的目标代码，抽取出其中实际完成微操作功能的部分，作适当的代码拷贝和连接。

（4）创建代码缓存块。利用代码缓存块存储微操作代码，存储的单位是基本块，之后的程序运行只需要代码缓存即可。

二进制翻译在提高模拟器执行速度方面成效卓著，但是由于使用了宿主机指令来模拟目标体系结构下的指令，踪迹信息的采集功能便难以实现。更重要的是该技术可移植性较差，因为每向一种新体系结构的宿主机添加支持，都需要重新实现从微操作到新体系结构二进制指令的缓存块构建。另外，软件模拟的 CPU 比硬件直接实现的 CPU 慢得多，相比在相应架构的机器直接运行，其速度一般有数量级的差异（佚名， 2015），因此二进制翻译的最大问题是代码转换造成的效率损失问题。

2.1.2 直接执行

Vmware、Simics 等模拟器使用直接执行的技术进行模拟。直接执行技术可以视为是二进制翻译技术在目标系统与宿主机指令集架构相同时的特例。由于具有相同的指令集架构，待运行的二进制程序可以直接运行于宿主机上，省去了许多不必要的翻译和函数调用，使得模拟速度大大加快。但是直接执行技术也存在一些固有的缺陷，如：

（1）模拟器运行于用户态，对于测试程序中的特权指令无法使用直接执行的方法，所以特权级别指令仍然需要进行模拟。

（2）直接执行状态有别于传统的模拟状态，这就引入了在直接执行状态与模拟状态之间的切换。频繁状态跳转引入的时间开销不可忽略。

（3）直接执行技术只在目标系统与宿主机指令集架构相同时可用，导致模拟器的可移植性很差。

2.1.3 FPGA 仿真

硬件执行速度通常远高于软件模拟的速度，在多核处理器研究给模拟器速度造成巨大压力的情况下，利用硬件加速模拟的技术便应运而生。现有的设计方案通过必要的综合和布局，即可使用现场可编程门阵列（Field Programmable Gate Array, FPGA）（Brown SD et al, 2012）烧录以及进一步试验。

FPGA 以其易于编程且具有较高运行速度的特点，被转用来搭建体系结构模拟器。无论是局部模拟，还是全系统模拟，FPGA 都有着突出的表现。FPGA 仿真既弥补完全硬件模拟的灵活性缺陷，又保留了硬件仿真所特有的速度优势，在处理器设计过程中得到越来越广泛的应用（维基百科，2016a）。加州大学伯克利分校的 RAMP 系统（Krasnov A et al，2007）、德克萨斯奥斯汀分校的 FAST 系统（Chiou Derek et

al, 2007）和普林斯顿大学开发的基于 Liberty 的 FPGA 系统（Penry DA et al, 2006）都是 FPGA 仿真技术的典型代表。

虽然基于 FPGA 的模拟系统实现了对模拟器的较好加速效果，但是该技术存在的一些缺陷也不容忽视，如：

（1）首先，借助于硬件进行加速的同时，不可避免的也引入了的硬件调试。由于系统调试十分复杂繁琐，每次参数调整都必须重新生成到 FPGA 映射的网表文件。复杂的调试过程是对开发者精力和耐心的巨大考验。

（2）其次，目前用作模拟的 FPGA 系统相比于目前主流通用处理器，其主频要慢至少 1 个数量级，其模拟的准确性相比于软件模拟并无优势。

（3）最后，FPGA 的低主频也使得该技术难以获得堪比于在主流商用处理器上的执行速度，导致该技术在进一步提升模拟速度方面力有未逮。

2.1.4 其他技术

通过提高单条指令或指令块加速模拟的含义比较宽泛，一般而言，如果模拟器程序执行速度提高了，指令模拟的速度也会有一定程度的提高，所以常用的软件调优及编译优化的手段也都可以被纳入该技术的范畴，如向量化、循环展开、提高程序局部性等。

2.2 减少模拟指令数

减少模拟指令（杨小溪 等， 2011）[6]的含义是仅对被模拟指令的一个子集实现性能模拟，且以其模拟数据推测整个程序的模拟数据。减少模拟指令数技术最关键的是如何高效地选择出具有足够代表性的部分进行时钟精确的性能模拟，既足够精简又充分反映完全模拟时所具有的特性。该技术常用的方法包括精简输入集、快速推进和采样。如 simplescalar（AUSTIN T et al, 2002）提供功能与时序两种模拟方式。在进行完全功能模拟后，利用 SimPoint（Hamerly G et al, 2005）提供的聚类功能，可以提供良好的采样模拟方案，既保持高精度又显著缩短模拟时间。

2.2.1 精简输入集技术

通常情况下，程序的模拟时间很大程度上由输入数据集的规模决定。常用测试程序集合中都带有标准的输入集。如果选择一个标准输入集作为测试的程序输入，执行时间短则数小时，长则几周，这对体系结构研究研究的效率造成严重的阻碍。

多数时候，研究者只是需要一个能够表现应用程序基本行为特征的输入集即可，完全不需要标准输入集那样完整而真实的模拟。

SPEC CPU 2000 测试程序集合包含三种不同的输入集：测试输入集、训练输入集和参考输入集。参考输入数据集是应用最广泛且能够更加细致全面的对系统进行测试的一个输入集。为了获取一个具有足够代表性精简参考输入集，首先对每个测试程序在使用参考输入集作为输入时的行为特征进行采集，分析其访存命中率、整型指令比例以及函数调用层次等多方面的规律；然后，在构造精简输入集的过程中要尽量保留这些关键行为特征。Osowski et al (2002)的实验表明，使用精简输入集作为测试程序输入时，在仅仅花费数百分之一的运行时间情况下，部分程序仍然保留着与参考输入数据集下相同的运行时特征。

无需对模拟器设计作任何调整，精简输入集技术就能够对被模拟程序的所有组成模块进行全面的运行。但是，分析提取测试程序的行为特征并构造足够精简的输入集的工作非常繁琐，其准确性也不稳定，导致该技术的应用有限。

2.2.2 截短模拟执行技术

由于应用程序运行过程中具有一定的时间局部性和空间局部性，因而可以假设程序运行过程中，任意执行区间的行为能在一定程度上代表程序的整体行为。基于这个假设，产生了截短模拟执行技术，缩短了模拟过程。使用该技术的模拟器中，只有很小一部分的指令会被模拟，其结果直接用于描述整体的特征。常用的方法有：

（1） Run Z。仅仅选择测试程序最开始的 Z 百万条指令进行模拟，由于测试程序的核心算法通常会在测试程序执行的中期或后期才出现，导致在这 Z 百万条指令中并没有对核心算法进行模拟。所以这种方法的准确性并不高，无法较好地代表整个测试程序的行为。

（2） Fast-Forward X + Run Z。相对于前一种方法，该方法在被模拟的 Z 百万条指令之前先进行 X 百万条指令的快速功能模拟，以保证在接下来的 Z 百万条指令能够体现被模拟程序主体部分的特征。这种方法克服了前一种方法的缺点，大大提高了截短模拟执行技术的代表性。但由于未进行充分地预热，因而常伴随着冷状态的缺陷：在对关键指令的性能模拟过程中，各个功能部件仍然处于初始化时的状态，而非正常性能模拟下所应有的状态。因此，后续指令的运行结果很容易出现一定的误差。

（3）Fast-Forward X + Warm-Up Y + Run Z。Warm-Up Y 步骤克服了第二种方法的不足。在将对核心算法部分使用性能模拟时，先使用一定数量的指令预热但不统

计功能数据，使得在模拟核心算法之前，所有部件处于完全性能模拟下所应有的状态。如此，Run Z 阶段得到的运行结果将会拥有更高的典型性。

Skadron et al (1999)的分析表明，尽管单纯模拟最开始阶段和未预热的缺陷在 Fast-Forward X + Warm-Up Y + Run Z 中得到了针对性的处理，但以上几种截短技术距真实模拟仍然有不可忽略的偏差，实验效果很难令人满意。

2.2.3 采样技术

采样技术利用一次完整功能模拟分析测试程序基本块子集的特点来推测完整应用的特点，再根据分析数据挑选适当数量有足够代表性的指令子集，以这部分指令子集的时序模拟结果推测完整执行情况下的时序模拟结果。采样技术以其较高的代表性和突出的加速效果而成为当前较为广泛采用的加速方案。根据样本指令子集筛选算法差异，采样技术主要包含代表元采样和周期采样两个分支。

（1）代表元采样技术：测试程序中难免会存在大量的循环或者递归调用，如果测试程序运行时间比较长，那么循环或者递归调用的数量往往也会更多（杨小溪 等，2011）[6]。众所周知，循环和递归调用是非常有规律的，循环的不同迭代或者递归的不同函数调用层次之间具有十分相似的行为特征。因此，以循环的某一次或多次迭代或递归的某一层或多层函数调用作为整个循环或递归的代表元，用代表元的模拟数据推断整个循环或递归的统计数据有较高的可信性。对于程序中所有类似的呈现一定规律的结构，分别取其代表元运行，并通过加权的方式综合所有代表元的统计结果，便可以预测完整运行测试程序时的统计结果。这一过程最关键的就是采用合适的算法选取真正具有代表性的代表元。最具代表性的代表元选取思路是：将可执行程序划分成基本块（Basic Block），以基本块为单位，统计执行频率，执行频率越高的基本块被认为具有更高的代表性。加州大学设计的 SimPoint 系统是该方法最具代表性的一个应用，它分析获取代表元的具体做法是在功能模拟过程中记录基本块的执行频率，再根据 K-Means 聚类方案选出最具代表性的若干个样本作为完全性能模拟时被执行的指令子集。同样基于局部代替整体的思路，代表元采样技术与截短模拟执行技术一样能取得非常显著（高达数十倍）性能提升，但是由于其选出的代表元真正概括了程序的整体行为特征，因此只造成很微小的性能偏差。

（2）周期采样技术：周期采样技术同样需要对测试程序进行划分，不同的是划分的结果是固定长度的程序段，并且不再需要寻找最具代表性的若干个分段。在每一个程序分段中使用截短模拟执行技术中的 Fast-Forward X + Warm-Up Y + Run Z 方法，值得说明的是这里的 X、Y、Z 的取值是动态调整的。假设现有程序段模拟结束

后的统计数据未能达到指定的精度，下一阶段的模拟过程中起预热作用的指令数量 Y 和性能模拟指令数量 Z 都将有一定程度的提高。由卡耐基梅隆大学设计研发的 Smarts（Wunderlich et al, 2004）系统是周期性采样技术的一个典型应用。与代表元采样相比，周期性采样因为需要对每个程序都进行三个阶段的模拟而更加耗时；同时，频繁的状态切换和参数统计（Yi, Joshua J. et al, 2005）也使其可用性不高。

2.3 利用硬件平台并行性和提高模拟算法并发度

形象地说，前面两种方法实现的功能分别是：更快的完成一件小事和减少事情的总量。而利用硬件平台并行性和提高模拟算法并发度的思路则是要雇佣更多工人在流水线上工作，在优化的制度下开发人口红利，集中力量办大事。那么这里就涉及到了两个方面："更多的工人"——并行化的硬件资源（Parallelism）和"流水线"——高度并发的模拟算法（Concurrency）。

2.3.1 并行化的硬件资源

在串行模拟速度受宿主机主频增速放缓而无法提升的情况下，高通量众核处理器在模拟规模和速度方面的需求都无法得到满足。随着制造工艺的改进及设计理念的变化，多核多线程的设计方案仍在持续提高系统的运算能力。近年来，多核 CPU 及多处理器系统凭借其突出的并行计算能力的优势迅速地占领了通用 CPU 和服务器的市场。借助于多核处理器强大的处理能力加速模拟成为必然选择。大规模使用 GPU 和 MIC 卡的加速方案所展现的巨大计算能力优势更让人叹为观止。传统上以 SimpleScalar 为代表的单核单线程方式运行的模拟器早已朝着以 Gem5 为代表的并行模拟器方向发展，充分利用现有系统中的多核多线程的优势加速模拟速度和精度已经被证明是必要且高效的。

利用并行技术对模拟器进行加速有很多实际案例，比如 GAS（吕慧伟 等，2013）模拟器在使用并行离散事件模拟方法对众核模拟进行加速后，保持同等模拟精度要求的同时，在 16 个核上获得 10.9 倍的性能提高。

2.3.2 高并发度的模拟算法

合理高效地利用并行化的硬件资源并非易事：手工并行化的方法虽然加速效果明显，缺失之繁琐；针对特定编程模型的方案虽然一定程度上实现了自动并行化，但是存在大量的移植性问题；另外，由于模拟器是一个细粒度程序，多线程或多进程间的同步与通信成为模拟器并行化过程中的最大障碍。

解决模拟器并行化中同步问题的一个最直接的方法是使用全局时钟，管理模块使用统一的系统时钟控制模拟进度，每个线程工作过程中如果对局部时钟有更新，便通知管理模块，在所有线程都更新时钟之后对全局时钟进行更新，并向所有线程广播全局时钟。缺点是每个线程更新时钟后便不能继续工作，直到系统时钟发出的广播消息被该线程接收到，由此产生的高延迟往往是不可忽略的。

针对全局时钟方案存在的高同步延迟问题，藤本等提出并行离散事件仿真（Richard M. Fujimoto, 1990）技术。该技术具有低耦合、易于移植的特点，通过局部的同步达到整个系统同步的效果，大大提高模拟器的并发度。藤本等提出如果系统中任何一个局部都已经达到同步状态，则整个系统也处于同步状态。如图 2.1 所示，假设与节点 m 相连的节点只有节点 i、节点 j 和节点 k。那么对节点 m 来说，只需要与节点 i、j、k 保持同步即可。这种去中心化的设计避免了集中控制的缺点，有较好的可扩展性。

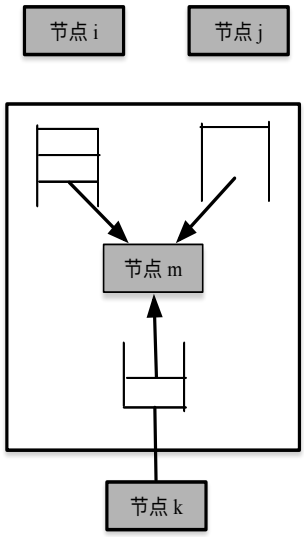

图 2.1 PDES 原理

节点 m 与每个相连节点都维护一个按时间戳排序的消息队列，为保持同步的状态，每次节点 m 从相连节点队列中取出带有最小时间戳的消息，进行分析并且按照时间戳保序原则向其他相邻节点发送消息。在处理过程中，如果某个消息队列为空，如节点 j 发往节点 m 的消息队列，那么按照“节点 m 是从节点 i、k 的队列中选择消息进行执行”，还是 “在确认不会受到节点 j 更小时间戳的消息之后才继续推进”区分出两种决策方式：乐观同步与保守同步。乐观同步采用“错误检测与回滚”的机制，大胆预测节点 m 不会收到来自节点 j 的更小时间戳的消息，便直接从节点 i、k 提取

事件，并按照既定规则进行分析。如果预测错误，则进行回滚，退回到之前一个同步的状态。保守同步方案则严格遵守事件因果关系，只有等到节点 j 发送过来一个消息或主动发送查询消息才会按照通用策略继续执行。

并行离散事件模拟算法专注于维护模拟单元之间的通信和同步关系，而这部分功能与被模拟单元的具体内容耦合性很低，于是可以将维护模拟单元之间通信与同步关系的并行离散事件模拟算法单独实现，并以应用程序接口(API, Application Programming Interfaces)的形式向其他模拟模块提供服务。这种模块化的设计将并行模拟器的部署、同步和通信等功能隔离开，模拟模块的修改不会影响并行模拟框架的正常工作。研究者无需关注框架的实现细节，具有非常好的设计灵活性。常见并行模拟框架有 USSF（Rao, Dhananjai M. et al, 2002）、HLA（Defense, U.S.D.o.，1998）及中国科学院研发的 BDSim 面向大数据应用的并行模拟框架。

2.1 本章总结

本章通过对现有模拟器加速技术的分析和总结，将模拟器加速技术归结为提高单条指令或指令块的执行速度、减少模拟指令数量以及利用硬件平台的并行性和提高模拟算法的并发度这三大类。

在“提高单条指令或指令块执行速度”一节中，详细分析了二进制翻译技术、直接执行技术、FPGA 仿真等方案的操作、优缺点、联系和典型应用。本文中查找表技术的使用正是基于提高单条指令模拟速度的思路进行的。

“减少模拟指令数量”一节说明了精简输入集、截短模拟执行和采样三种技术的基本原理和典型应用，并简要介绍了三种方案的加速效果和优缺点。在本文的初期工作中包含对代表元采样技术的应用，但是由于时间问题未能在最终的成果中展现。

“利用硬件平台的并行性和提高模拟算法并发度”一节介绍了从软硬件两个角度实现并行模拟的现实条件和理论支持。按照这一原则，本文设计和实现了从多个角度优化的并行模拟框架，显著提升了模拟器运行速度。

第3章 查找表技术加速指令译码

查找表技术是指利用数组或关联数组实现以简单的表查询操作替换复杂的运行时计算。由于省去了不必要的函数调用、分支转移及复杂算术运算，采用静态查找表的方案通常能取得显著的加速。

查找表技术的一个典型应用就是三角函数表（维基百科，2016b）。由于较高精度的三角函数值计算通常伴随着反复的循环或递归函数调用，这种耗时的计算在许多应用中是不可容忍的。规避这种复杂计算的方法是，在应用程序运行初期计算适当数量的三角函数值，譬如构建一个包含一定范围内所有整数角度正弦值的表，在应用程序以后的运行中，如果需要计算一个新的角度的正弦值，只需要从表中提取出临近整数角度的正弦值即可，从而避免使用数学公式进行一次完整的正弦值计算过程。

在模拟器实现中，本文利用查找表技术实现对指令译码过中的以下三个方面进行优化和加速；PopCount 算法的实现、指令条件域检查以及软件模拟踪迹缓存。

3.1 PopCount问题

ARMv6 指令集（ARM， 2016）中 Load/Store Multiple 指令（ARM Architecture Reference Manual， 2016）的格式如图 3.1 所示。

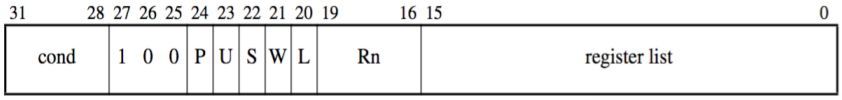

图 3.1 Load/Store Multiple 指令格式

其中 register list 域的每一个二进制位对应于 ARM 架构中的一个通用寄存器，bit0 指代 R0,bit15 指代 R15（即 PC 寄存器）。在指令执行过程中需要统计 register list 域中“1”的个数。求解二进制数中 1 的个数的问题，称作 PopCount 问题，又称作汉明距离问题。

3.1.1 遍历计数算法

进行任何优化之前，模拟器中使用算法 TraverseCount 计算给定 32 位整数 val 的二进制表示中“1”的个数。

算法 TraverseCount

（1）初始化计数变量 counter 为 0；

（2）如果 register list 不等于 0，执行（3），否则执行（7）；

（3）如果 register list 二进制表示的最低位是“1”，执行（4），否则执行（5）；

（4）计数变量 counter 增加 1；

（5）将 register list 逻辑右移 1 位；

（6）执行（2）；

（7）返回计数变量 counter 的取值。

通过循环和移位操作，算法 TraverseCount 从右往左遍历输入参数 val 的每一个二进制位，每个“1”的出现递增计数变量 counter。算法 TraverseCount 在深度流水的高性能计算机架构上将花费数以百计的时钟周期。问题在于该算法频繁地引入分支跳转和循环，严重损害执行效率。

针对如何高效地解决 PopCount 问题，人们已经广泛地进行了研究。某些处理器上设计了用来解决该问题的简单指令，并且有针对位向量对应并行化操作。而对于缺少这些特性的处理器，已知最佳的的解决方案是按照树状结构进行累加。

3.1.2 查找表技术解决 PopCount 问题

若允许较多的内存占用，可以得到比算法 TraverseCount 及树状累加算法更快的解决方案——查找表。思路是在内存中构造一个包含 256 个条目的表，各条目分别存储对应 8 位二进制数中“1”的数量。通过采用循环展开并截取 register 域的低 16 位的策略可以进一步加速该算法。采用以上策略优化后，查询 register 域中“1”个数如算法 LUTCount 所示。

算法 LUTCount

（1）构建包含 256 个条目的 PopCount 查找表；

（2）取出 register list 域中 0～7 位部分，存入变量 down；

（3）取出 register list 域中 8～15 位部分，存入变量 up；

（4）以 down 作为索引检索查找表 tab，得到 down 的二进制表示中“1”的个数位 tab[down]；

（5）以 up 作为索引检索查找表 tab，得到 up 的二进制表示中“1”的个数位 tab[up]；

（6）返回 register list 低 16 位的二进制表示中“1”的个数：tab[down]+tab[up]。

算法 LUTCount 不包含分支，仅有两次内存访问、几乎没有算术运算，大幅地提升了速度。

3.2 指令条件域检查

大部分 ARM 指令可以被“条件地”执行，如图 3.1 所示，每条指令包含一个从第 28 位至第 31 位共计 4 位的“条件域”(Condition Field)。表 3.1 描述了在给定指令操作码的情况下，如何依据程序状态寄存器 PSR 的 NZCV 标志判断该条指令是否能通过条件域判断而被执行。

表 3.1 条件列表

操作码	助记符	含义	NZCV 标志状态
0000	EQ	相等	Z 置位
0001	NE	不等	Z 清除
0010	CS/HS	进位 / 无符号数比较取值较大或相同	C 置位
0011	CC/LO	无进位 / 无符号数比较取值较小	C 清除
0100	MI	相减 / 负数	N 置位
0101	PL	相加 / 正数或零	N 清除
0110	VS	溢出	V 置位
0111	VC	未溢出	V 清除
1000	HI	无符号数比较取值较大	C 置位且 Z 清除
1001	LS	无符号数比较取值较小或相同	C 清除或 Z 置位
1010	GE	带符号数比较取值较大或相等	N 置位且 V 置位, 或 N 清除且 V 清除(N==V)
1011	LT	带符号数比较取值较小	N 置位且 C 清除, 或 N 清除且 V 置位(N!=V)
1100	GT	带符号数比较取值较大	Z 清除，且以下取其一：N 置位且 V 置位, 或 N 清除且 V 清除(Z==0, N==V)
1101	LE	带符号数比较取值较小或相等	Z 置位，或以下取其一： N 置位且 V 清除, 或 N 清除且 V 置位(Z==1 or N!=V)
1110	AL	无条件	-
1111	-	特殊条件	-

“条件地”执行意味着仅在当前状态寄存器（CPSR）中 NZCV 标志满足指定 cond 域要求时，该条指令才能够对编程模型状态、内存以及协处理器产生正常的影

响。如果这些标志位不满足这个条件域，那么，该条指令就被当作 NOP 处理，也就是说会像正常情况一样跳转到下一条指令进行执行，包含任何对中断和放弃预取的检查而不产生任何其他效果（ARM Architecture Reference Manual， 2016）。

3.2.1 原始算法

在指令译码阶段的最开始都会涉及到对条件域的检查，如算法 CalcJudge 所示，原有的解决方案是通过对 4 位条件域对应的 16 种取值使用 switch 选择后，按照表 3.1 的判断方法执行相应的计算，得到判断结果。

算法 CalcJudge

（1）从指令 inst 中抽取出条件域 cond；

（2）逐一判断 cond 取值与取值范围中每一项是否匹配，并参照表 3.1 执行相应的计算，结果存入 result 变量；

（3）返回 result，作为条件域判断的结果。

算法 CalcJudge 的繁琐耗时之处在于：

（1）switch 语句匹配过程：计算给定 switch 表达式的值，顺序检查与每个 case 常量的取值是否相同，直至找到匹配的 case 分支;

（2）读取 PSR 的每个状态标记位，并执行一些对应的逻辑运算;

（3） 函数调用开销。

3.2.2 查找表技术实现指令条件域检查

由于条件域 Cond 和程序状态寄存器 PSR 的高四位 NZVC 都只有有限的 16 中取值。因而可以采用查找表的方案加速指令条件域判断。查找表枚举出所有可能的 Cond 与 NZCV 配对的时的结果，如表 3.2 所示。表 3.2 的横轴罗列了所有条件域的取值，纵轴枚举状态寄存器中 NZCV 四个标志为的取值。单元格的取值是布尔型变量，取值为 1 表示通过条件判断，该条指令将会被执行，反之将跳过该指令。

算法 LUTJudge 描述如何借助于查找表，判断一条指令是否将被执行。

算法 LUTJudge

（1）构建包含所有 256 种 Cond 与 NZCV 匹配情况的二维查找表 tab；

（2）从程序状态寄存器 PSR 中取出 NZCV 域，放入变量 nzcv；

（3）从待判断指令 inst 中取出其条件域部分，放入变量 cond；

（4）以（nzcv，cond）为索引检索二维查找表 tab；

（5）返回检索结果 tab[nzcv][cond]作为判断结果。

算法 LUTJudge 中仅包含一次内存访问，不含任何函数调用、分支跳转和算术运算。

表 3.2 COND 域与 NZCV 匹配结果

		COND															
		EQ	NE	CS	CC	MI	PL	VS	VC	HI	LS	GE	LT	GT	LE	AL	NV
NZCV	0	0	1	0	1	0	1	0	1	0	1	1	0	1	0	1	0
	1	0	1	0	1	0	1	1	0	0	1	0	1	0	1	1	0
	2	0	1	1	0	0	1	0	1	1	0	1	0	1	0	1	0
	3	0	1	1	0	0	1	1	0	1	0	0	1	0	1	1	0
	4	1	0	0	1	0	1	0	1	0	1	1	0	0	1	1	0
	5	1	0	0	1	0	1	1	0	0	1	0	1	0	1	1	0
	6	1	0	1	0	0	1	0	1	0	1	1	0	0	1	1	0
	7	1	0	1	0	0	1	1	0	0	1	0	1	0	1	1	0
	8	0	1	0	1	1	1	0	1	0	1	0	1	0	1	1	0
	9	0	1	0	1	1	1	1	0	0	1	1	0	1	0	1	0
	10	0	1	1	0	1	1	0	1	1	0	0	1	0	1	1	0
	11	0	1	1	0	1	1	1	0	1	0	1	0	1	0	1	0
	12	1	0	0	1	1	1	0	1	0	1	0	1	0	1	1	0
	13	1	0	0	1	1	1	1	0	0	1	1	0	0	1	1	0
	14	1	0	1	0	1	1	0	1	0	1	0	1	0	1	1	0
	15	1	0	1	0	1	1	1	0	0	1	1	0	0	1	1	0

3.3 查找表技术模拟踪迹缓存

踪迹缓存（Trace Cache）是一个特别的指令缓存，它捕获动态指令序列。由于其每一行存储动态指令流的一个快照或者踪迹，这个结构被称作踪迹缓存。程序的踪迹是动态指令序列中包含至多 X 条指令并且以任意点起始的至多 Y 个基本块的序列。其中 X 指示踪迹缓存行大小，Y 代表给定分支预测器的吞吐量。一个踪迹完全是由一个起始地址和最多 Y-1 个描述后续路径的分支结果决定的（Rotenberg et al, 1996）。

模拟器中，可以添加一个软件模拟的踪迹缓存结构来存储指令块的译码结果。第一次遇到一个踪迹时，会在踪迹缓存中分配一行。当指令由 I-Cache 中取出时对应踪迹缓存行将被填充。如果程序执行过程中再次遇到同一个踪迹（起始地址相同且分支预测结果相同），踪迹缓存中相应的行就已经存在，它将被直接喂送到指令译码器中。否则，将按照正常渠道由 I-Cache 接收指令。

软件模拟的踪迹缓存结构本质上是一种运行时构建的动态查找表。踪迹缓存机制省去了大量重复的取指译码过程，加速模拟过程的推进。

3.4 查找表解决PopCount问题的实验验证

3.4.1 实验设计

PopCount 问题在模拟器的设计中仅出现在指令译码阶段内，是一个非常细粒度的处理过程，为避免时间统计程序对模拟过程的性能造成明显的干扰，本文使用独立的模拟程序将查找表方法与几种以计算为主的 PopCount 问题解决方案的加速效果进行对比。

实验中，计算 2 的 30 次方个随机数二进制形式中“1”的数量。首先，以查找表中入口数为自变量，观察模拟程序运行时间与查找表入口数的关系。其次，以 3.1.1 节 TraverseCount 算法为基准，对比多个算法的加速效果。

3.4.2 计算时间与查找表尺寸的关系

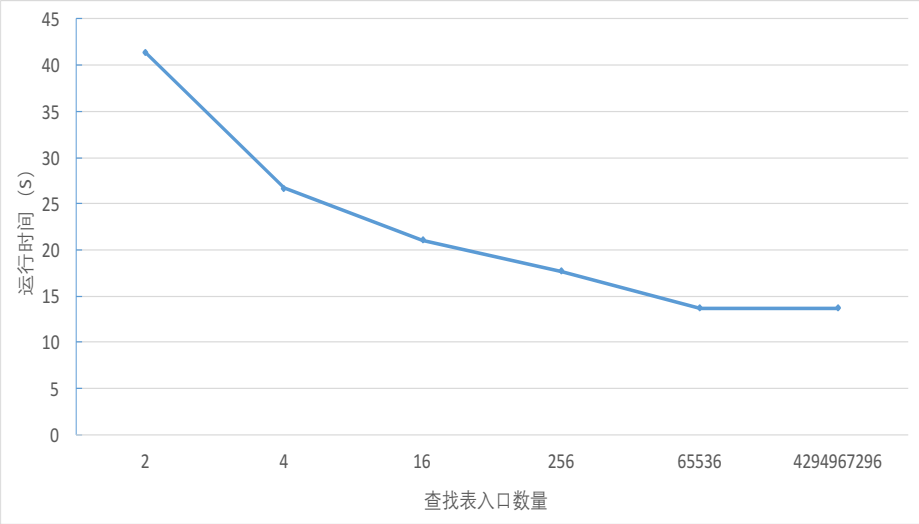

图 3.2 计算时间与查找表规模的关系

图 3.2 横坐标是查找表入口数量，须知查找表的入口数越多，存储查找表所需要的内存空间越大，但是计算给定整数的中“1”的数量需要进行的查表次数就越少。图 3.2 纵坐标是计算时间，以秒(s)计。从图 3.2 的折线变化可以看出，随着查找表入口数量增加，计算所花费的时间逐渐减少，当入口数量达到 65536 之后，即使继续增加也不能带来更快的运行速度，这是因为 65536 个入口的查找表已经可以完全将 16 位二进制整数完全包含进去，要统计$[0, 2^{16}-1]$区间内任何一个整数的二进制表示中“1”的个数，只需要访问一次查找表即可。

3.4.3 不同算法加速效果对比

图 3.3 中统计了 5 中不同的算法下，解决 PopCount 问题所需的计算时间。左侧纵坐标是每种方案进行查找所需要的 CPU 运行时间，右侧纵坐标是解决该问题的各种常用算法相对于原始 TraverseCount 算法(naive)的加速比，图中的折线连接了 5 种算法的加速比。值得注意的是，图中 Lower Bound 方案是指“不经过任何内存访问和数学计算，直接给出给定整型数字的二进制表示方式中“1”的个数”的方案，以此作为求解该问题的时间下界

可以看到，Fast 算法(利用位与运算特性实现)运行时间远小于 TraverseCount 算法，但是与 GCC 内建的 PopCount 函数相比则仍然是非常耗时的。而 LUTCount 算法的性能表现更加突出。最优的情况下 LUT 算法可以达到相对于 TraverseCount 算法 26.19 倍的加速效果，并且已经非常接近于求解该问题所能达到的时间下界

另一方面，构造查找表所需的工作量及占用的运行时内存空间与查找表入口数成反比，最优加速效果对应的内存空间占用达到 4GB。综合考虑加速效果和查找表占用的内存空间，本文以 256 个入口数作为系统运行时的配置，在该配置下，查找表方案达到 20.28 倍的加速，仅占用 256B 的内存空间。

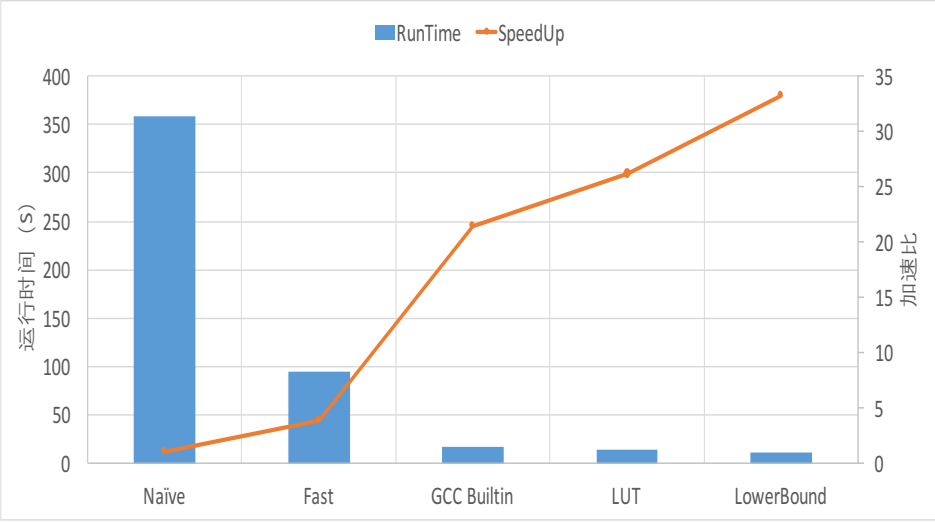

图 3.3 不同算法加速效果对比

3.4.4 实验总结

建立一个合适的查找表需要考虑以下两个方面的限制原则：可以使用的内存数量以及构建查找表的初始计算时间。前者是因为不能构建一个超过可用内存空间大

小的表格，尽管可以构建一个基于磁盘的查找表，但是这种以查找速度为代价的选择对于有一定实时性要求的应用并无明显优势。后者是因为如果构建查找表的时间开销不可接受，还不如退而求其次，直接使用完整数学计算或函数调用的方法。

需要注意的是，由于存储墙的存在，内存访问的速度低于处理器运算速度，导致在查找表所替换的计算量很小的情况下，加速效果并不明显。一方面，简单计算的速度要快于从内存读取数据的速度；另一方面，查找表增加了内存占用且加入到高速缓存竞争，规模过大的查找表将严重降低高速缓存命中率。这两方面原因共同限制了查找表方法加速效果的发挥。

3.5 本章总结

基于提高单条指令的模拟速度的思路，本章利用宿主机上充足的内存空间实现了查找表技术。本章首先介绍了查找表技术的定义和典型应用。在此基础上本章介绍了查找表技术的在模拟器指令译码过程模拟中的应用。

在 Load/Store Multiple 指令的模拟过程中涉及到 PopCount 问题，本章对此进行了详细的描述和常用解决算法的优缺点分析，最终选择查找表方案，并利用编译优化的技术结合实际应用场景进行优化。

指令条件域检查是每条指令的模拟过程中必须经过的步骤，本章在分析现有解决算法的不足后，给出了使用查找表技术解决该问题所需构建的完整查找表，使得指令条件域检查的过程变得简洁、高效。

软件模拟的踪迹缓存是查找表技术在指令译码过程中的另一个重要应用，与前两种方案不同的是，此处实现的动态变化的查找表，使用类似于数据缓存的查找替换机制。

最后，以 PopCount 问题为例验证查找表技术的性能优化效果。通过探索计算时间与查找表规模的关系确定最佳查找表入口数量，并以此最佳配置对比几种 PopCount 问题解法的性能差异。实验表明，查找表方案相对于模拟器中采用的算法达到最高 20.28 倍的性能收益。

第4章 多角度优化的并行离散事件模拟框架

本章在 BDSim 并行模拟框架基础上，针对模拟中存在的瓶颈从未来事件队列无锁化、事件调度算法、时间推进算法和内存管理四个方面对 BDSim 模拟框架进行性能优化。

4.1 无锁化设计的未来事件队列

无锁化的未来事件队列设计基于循环数组实现。循环数组是一个先入先出(FIFO)的数据结构，其设计如图 4.1 所示。在模拟框架中，循环数组充当管道的作用：一方面提供待处理事件给工作线程，另一方面存储处理过程中产生的新事件。头指针的下一个位置为第一个可读位置，尾指针对应前一次被写入的位置。从 head 往 tail 的顺时针递增方向存储了被写入但未读取的数据，如图 4.1 中的阴影部分表示；而空白部分表示未写入或者已读的数据。

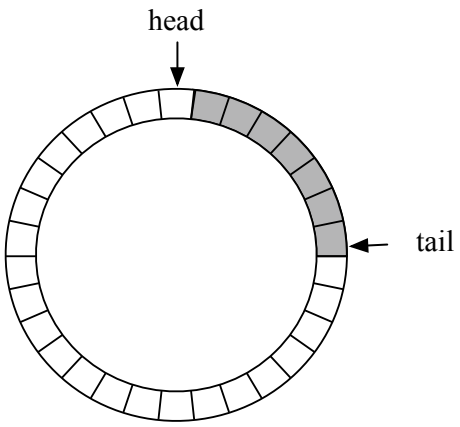

图 4.1 循环数组结构

单个读者和单个写者之间，借助于循环数组，不需要任何锁或者信号量机制，就可以实现对数据安全的访问。多线程环境下，调度线程与每个工作线程分别组成一个读者写者对。

无锁化事件队列依靠对几个边界变量的原子读和原子写操作完成。当一个要被访问的边界位置非法时，如试图读一个空位置或着向一个满队列写数据，那么等待，直到操作合法。

基于单一读者单一写者模型，消费者部分的工作过程如图 4.2 所示：

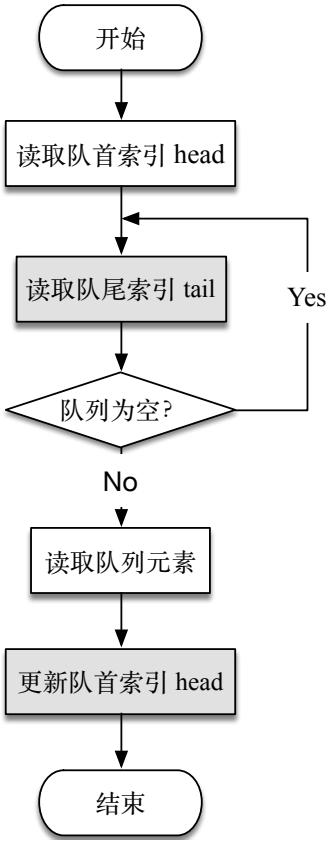

图 4.2 消费者工作流程

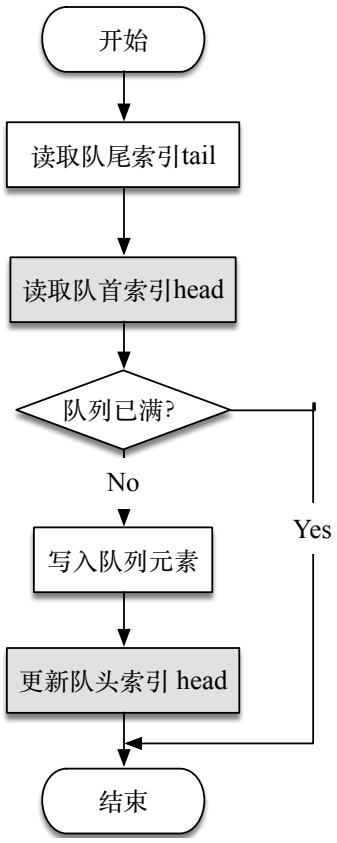

图 4.3 生产者工作流程

消费者线程读取队列的首尾索引，用于检查队列状态：若队列非空，则消费者从中读取队列的中存储的消息元素，并更新队首索引；否则继续等待至队列非空。为保证多核处理器下程序正确运行，流程图中阴影部分需要分别使用内存屏障（memory barrier）进行保护，以防止编译器从缓存中读取这些变量的值。类似的，生产者部分的实现过程如图 4.3 所示。

更进一步，为了降低开销，本文中循环数组存储的是数据指针，将原来大尺寸消息的拷贝和删除操作缩减为对单个指针的相应操作，如图 4.4 所示。这种实现方式消除了由于耗时的锁操作而引入的高昂开销，发掘出模拟框架固有的并行性。

```
typedef struct queue_type_desc{
    len_t len;
    VOLATILE int head __attribute__ ((aligned (CACHE_LINE_SZ)));
    VOLATILE int tail __attribute__ ((aligned (CACHE_LINE_SZ)));
    void** ele;
}que_t;
```

a) queue define

```
que_status enqueue(que_t *que,void* data)
{
    if((que->tail+1)%(que->len)==que->head) {
        return Q_OVERFLOW;
    }
    else{
    que->ele[que->tail]=data;
    barrier();
    que->tail = ( que->tail+1 )%(que->len);
      return Q_NORMAL;
    }
}
```

b) enqueue implementation

```
void* dequeue(que_t *que)
{
    void* data;
    if(que->head==queue->tail) {
        return NULL;
    }
    else{
        data = que->ele[que->head];
        que->head = (que->head+1)%que->len;
        return ele;
    }
}
```

c) dequeue implementation

图 4.4 无锁队列的结构定义及入队、出队操作

4.2 基于随机映射策略的事件调度算法

针对高通量应用中丰富而且多变的负载特征，为消除固定映射策略存在的不足，本文设计实现了采用随机映射策略的事件调度算法。其工作原理如图 4.5 所示

采用随机映射策略的事件调度算法中，在模拟层组件与框架服务单元之间添加一个消息分配的单元——Message Distributor，负责将模拟组件中产生的消息按照给定的算法分配到多个框架服务单元上。设每个工作线程在每轮分配过程中最多允许获得的未处理事件数量为 N，那么消息分配器的工作过程如下：

（1）遍历所有工作线程，统计其事件队列中当前剩余未处理的事件数量，设工作线程 *worker_i* 当前剩余事件数量为 l*eft_i*；

（2）统计未来事件队列中待分发的事件总数 *M*；

（3）按照随机选择的策略为每个工作线程分配 *new_i* 数量的未处理事件，使得 *left_i* + *new_i* 近似等于 *N*。

采用随机映射策略的事件调度算法中，消息分配器是一个独立的线程，统一处理组件接收到的消息，并按照工作线程的负载情况进行分发。由于不再按照拓扑结构分配消息，每个工作线程可能执行到任何一个组件的消息，每个组件的消息也可能被任何一个工作线程处理。由组件类型差异而导致的工作线程负载不均衡问题便不复存在。此外，带有消息分配器的随机映射机制的另一个优点是可以灵活地控制事件分配的粒度，本章对应的实验部分对此有详细的描述。

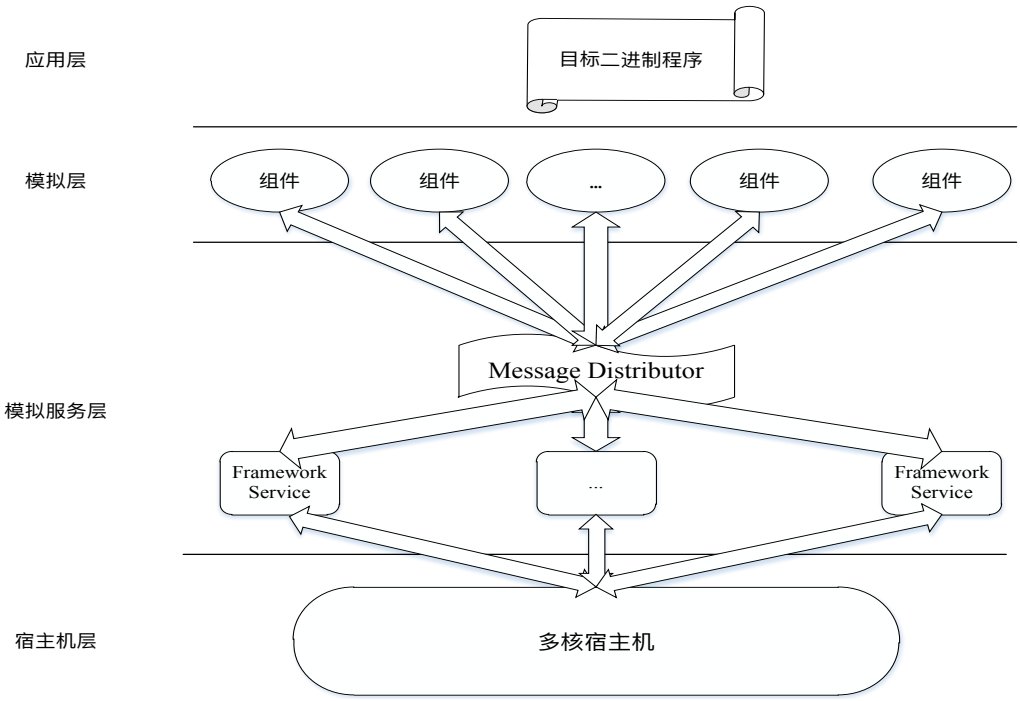

图 4.5 随机消息映射

消息分配器在实现中是一个单独的线程，用于获取消息请求并进行分发。如图 4.6 所示，增加消息分配器后的框架设计中，包含左侧调度线程部分和右侧工作线程部分。调度线程与工作线程的通信发生在图中数字 1、2 标识的虚线连接上。本章后续各节的优化都在此基础上进行。

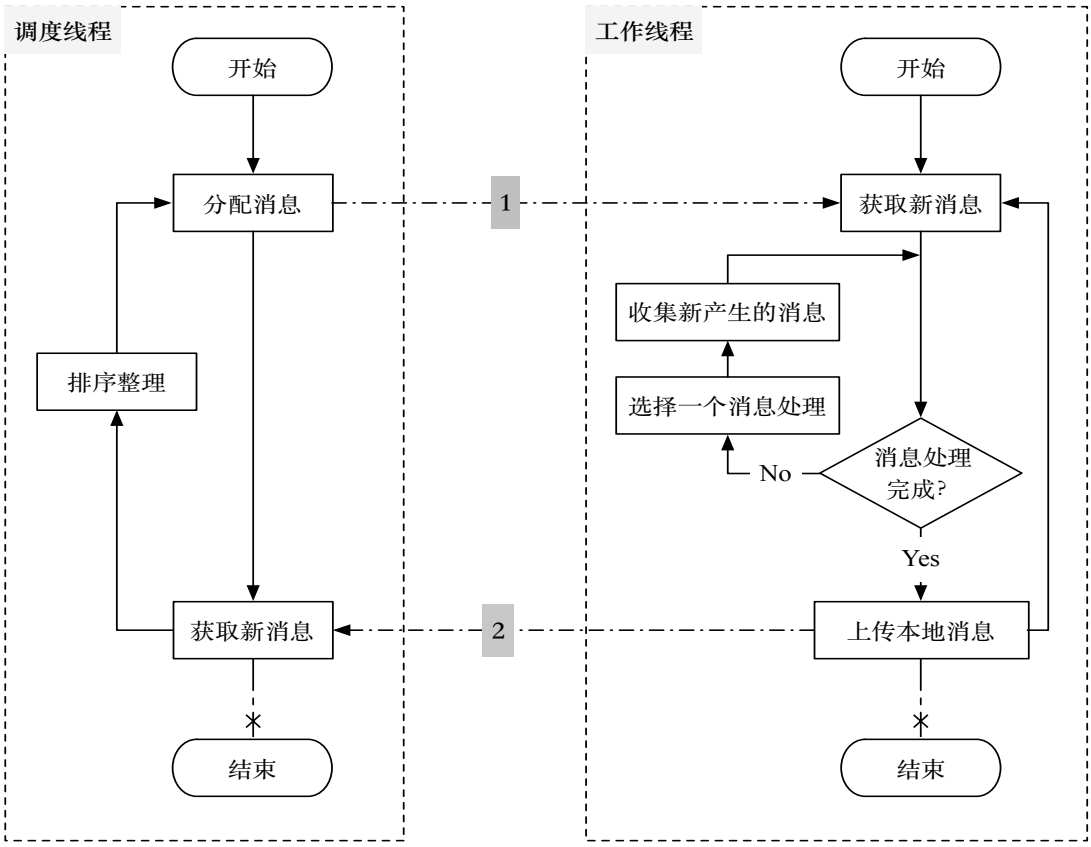

图 4.6 框架工作原理

4.3 基于cycle-by-cycle模型的时间推进算法

针对采用 CMB 算法进行同步时存在的不足，本文选择逐个 cycle 往前推进系统时间的策略，其基本思路是将所有未分配的消息按照时间戳排序，每轮消息分配仅处理最小时间戳的消息，具体的实现借助于红黑树结构。由于在插入、删除和查找操作方面突出的性能表现，红黑树在许多时间敏感的场合都有重要应用。本文采用红黑树管理消息链表：同一接收者、同一个时间戳的消息使用线性链表进行存储，不同的链表间使用红黑树管理。

图 4.7 展示了一个工作线程的私有事件队列与使用红黑树结构的全局事件队列之间的交互。Cycle-by-cycle 的实现算法描述如下：

（1）执行组件的初始化工作，某些组件在此时产生系统初始事件，纳入红黑树结构的管理之中，系统时间戳记为 0；

（2）获取红黑树中排序最前的事件链表，记录其时间戳为 T；

（3）系统时间更新到 T；

（4）将链表中的事件分配给工作线程处理（新产生的事件仍纳入红黑树管理之中）；

（5）重复步骤（2）。

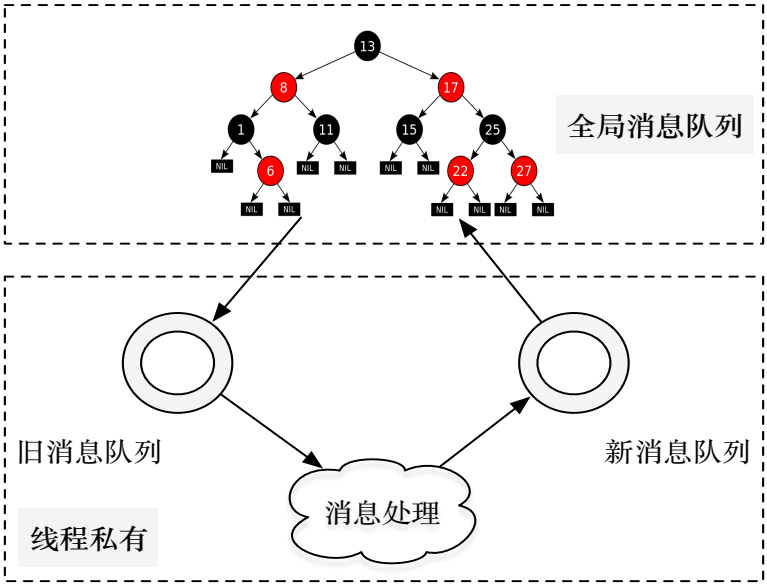

图 4.7 线程私有数据与全局消息队列的交互

按照时间戳递增的顺序管理消息队列，在主线程的统一调度下，工作线程逐个时间戳地执行被分配到的消息。主线程的工作只是简单的从消息队列中分发最小时间戳的消息给工作线程，并将返回的新消息纳入消息队列的管理之中。由于事件发生的先后顺序反应在了消息接收时间戳的大小关系上，而红黑树结构每次提供当前最小时间戳的消息链表，便省区了复杂的同步操作。因此，借助于红黑树管理消息链表，简单快捷地实现了系统时间戳的不断推进。

4.4 基于内存池的消息存储空间管理方案

动态内存管理方案如 C 语言中 malloc 或 C++中 new 操作受限于变化的块大小导致的碎片，当需要频繁操作时，在实际系统中性能表现不佳。一个更加高效的解决方案是预先分配一系列同等大小的内存块，称作内存池（Wikipedia, 2016）。

为消除标准库中动态内存管理方案的不足，本文采用内存池的方案加速消息内存空间的管理。

本文处理的消息格式高度相似，使得每次分配的内存空间尺寸一致，省却了用于记录块尺寸的元数据。为了方便地进行管理，新的方案中引入两个变量：内存池容量 *N* 和当前空闲内存块数量 *C*。在创建初始内存池时，*N* 和 *C* 相等。后续每次内存申请操作导致 *C* 减少 1，每个内存释放操作使得 *C* 增加 1。当需要进行必要的内存池容量扩充时，*N* 取值增加。为提高管理效率，内存池中空闲块实际采用线性链表的形式连接，如图 4.8 所示。图中 *new_msg* 指向的空闲块正在被从空闲块链表 *mem_pool* 中分配出来，用于消息结构体的填充。

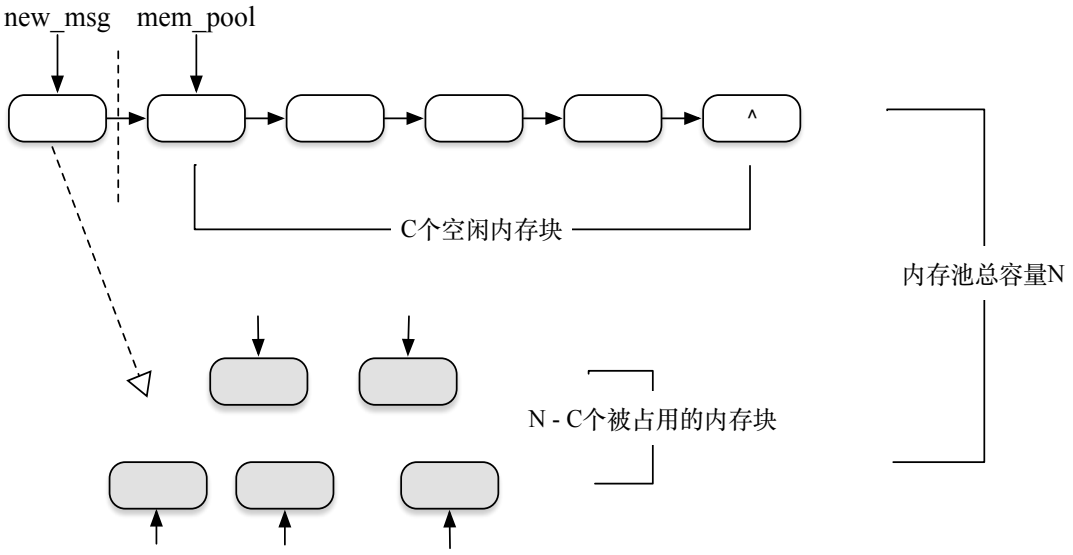

图 4.8 内存池管理方案示意图

在池化管理方案中，所有内存申请、释放实际都是向内存池发起操作请求，由它代为完成。图 4.9 左侧部分展示了实际内存申请操作的详细过程：内存池实际执行的操作仅仅是空闲块数量的统计、空闲块地址的返回，以及必要时的内存池容量扩充。

基于内存池的内存空间释放操作如图 4.9 右侧所示。此处的内存释放也并未实际向操作系统释放任何空闲内存，而是将暂时闲置的内存重新纳入内存池的管理中，以服务于后续的内存申请操作。内存的释放和申请的速度显著提升。使用内存池的优点包括：

（1）首先，内存池可以在固定的运行时间内分配内存。对于池中上千个对象的内存释放操作只需一个操作即可完成，而动态内存管理则需要为每个对象独立的释放内存。

（2）其次，区块尺寸固定的内存池不需要存储每次分配的元数据用于描述诸如区块大小等特征，这提供了显著的内存节约功能，尤其是在小尺寸内存分配的情况下。

（3）最后，内存池可以按照层次化的树状结构（Cline, Robert C， 1993）进行分组，非常适合某些特殊的编程结构，如循环和递归。

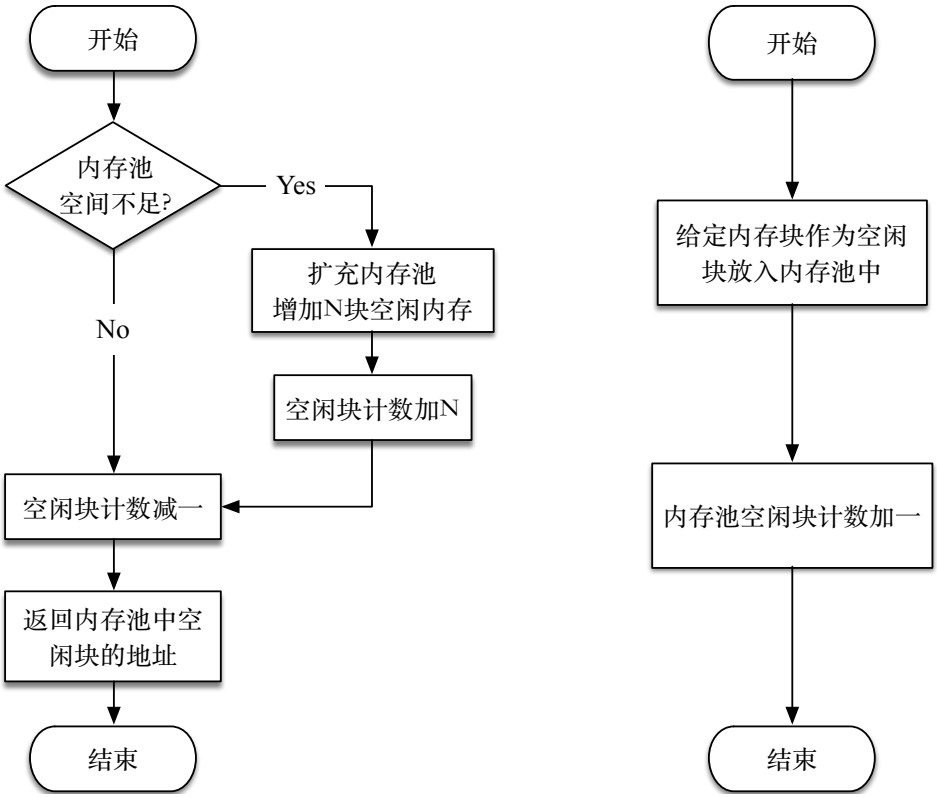

图 4.9 内存池方案中的申请与释放操作

4.5 实验验证

4.5.1 随机映射策略的负载均衡性

基于随机映射策略的一个改进的事件调度算法是在每次调度时兼顾线程当前剩余未完成的事件数量与未分配事件数量。调度的结果是尽量使得每个线程的未来事件队列中含有相同数量的待处理事件。称这个改进的事件调度算法为随机映射策略下的动态调度方案。

对多组*min_load*取值的固定负载调度方案与动态调度方案下各工作线程分配到事件的概率进行对比，如图 4.10 及图 4.11 所示。

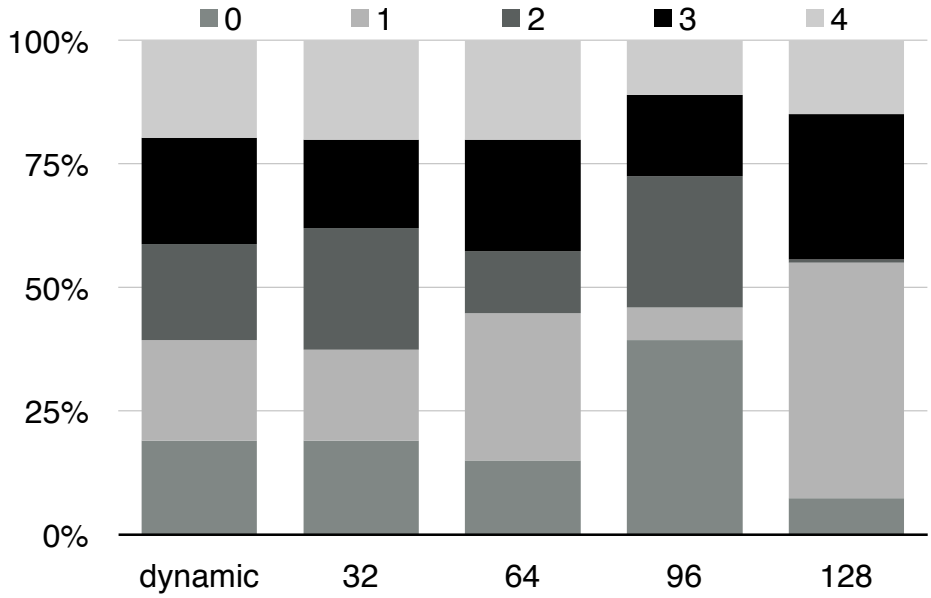

图 4.10 动态调度与固定负载分配方案对比

可以直观得看到，使用五个工作线程的配置时，各个线程在每轮事件调度中被分配到新事件的概率几乎相同，不存在静态分配方案中悬殊的差距。并且，观察图 4.12 可知，每轮分配中，各线程得到待处理事件的数量也基本一致。基于随机映射策略的动态调度方案达到几近完美的负载均衡性。

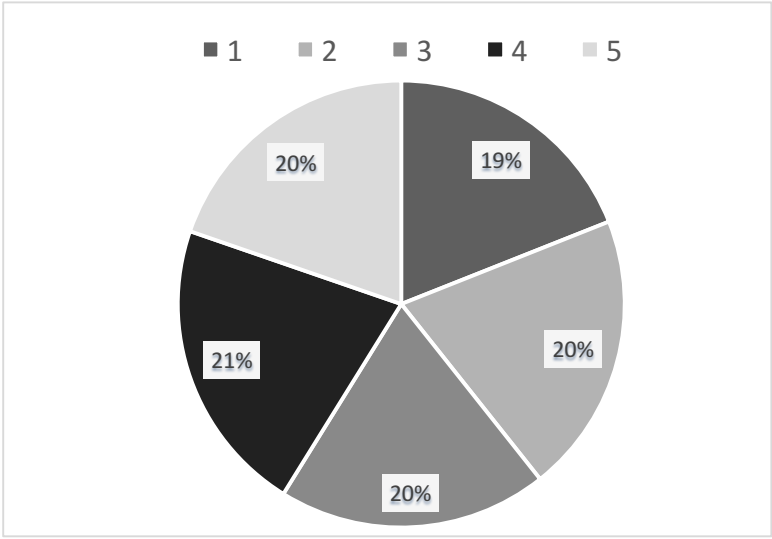

图 4.11 动态分配方案概率分布

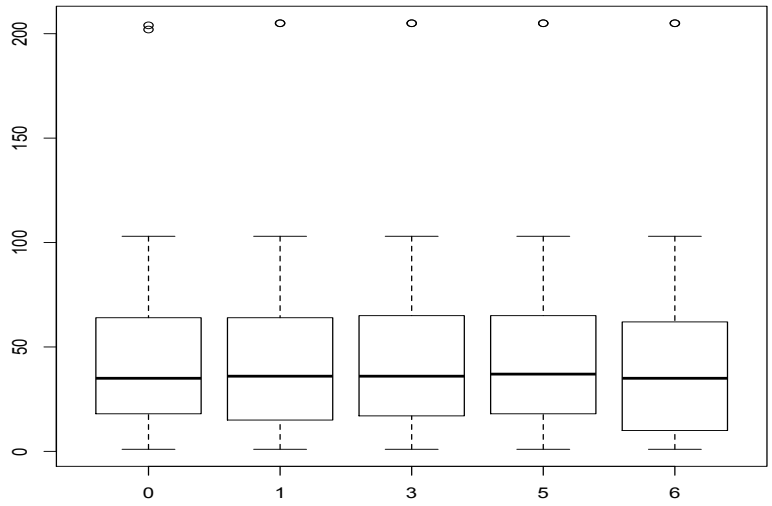

图 4.12 动态分配方案事件分布

4.5.2 内存池管理方案加速效果

内存池管理方案加速效果的实验验证选择了 test-math, tera-sort 以及 kmp 三组测试程序。并且，对于 kmp 程序选择了三个不同的输入数据集进行对比。

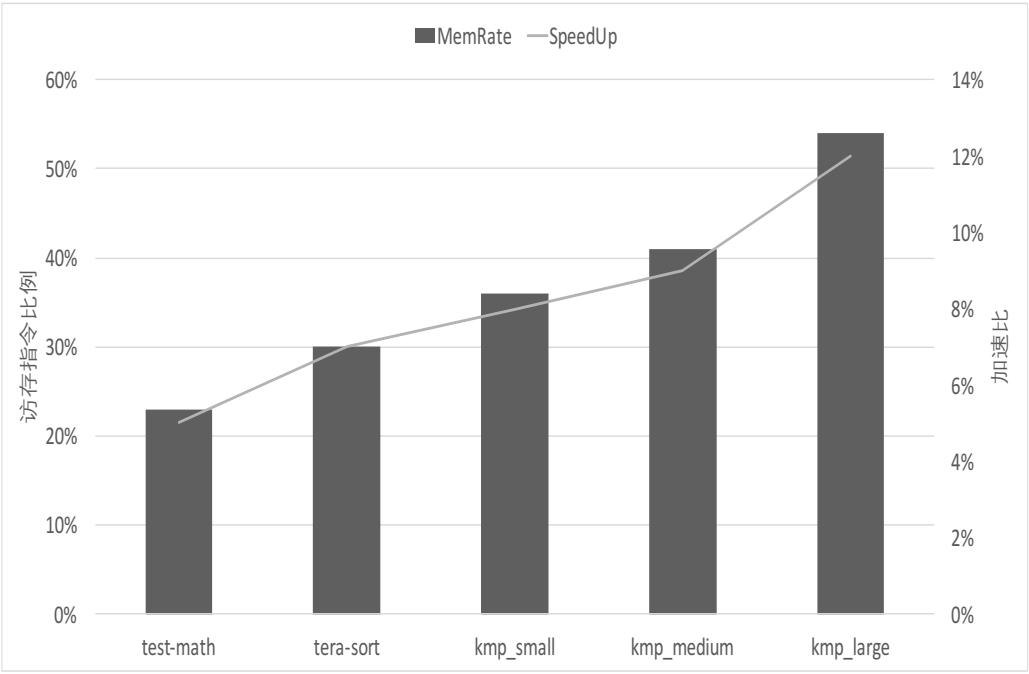

图 4.13 访存指令比例及加速比

实验的性能指标为加速增益，即采用内存池管理方案下的模拟速度相对于采用动态内存管理方案时模拟速度的提升比例。

统计三组测试程序及同一测试程序在不同规模数据集下运行时间，计算采用内存池方案的加速效果，如图 4.13 所示。

在指令总数一定的情况下，访存指令的比例越高，程序运行时产生的消息数量越多，加速效果越明显。访存指令的模拟过程中，模拟器调用框架提供的消息发送功能向其他组件发送新消息，其中伴随着旧消息的释放和新消息的生成。因此，内存池的加速效果取决于测试程序中访存指令的比例。测试程序 test-math 属于计算密集型应用，其加速效果显著低于访存指令比例较高的 tera-sort 与 kmp。并且，kmp 程序的输入集越大，访存指令比例越高，加速效果越显著，在大输入集的 kmp 程序中达到 12%以上的性能提升。

4.5.3 总体性能收益

在选择 1 个调度线程以及 1 至 7 个工作线程的配置下运行并行模拟框架，统计多组测试程序以及同一组测试程序在执行不同数量指令情况下程序的最大加速比，结果如图 4.14 所示。

图 4.14 的横坐标是被模拟程序的指令数量，纵坐标为最大加速比。在被模拟程序的输入集增大时，被模拟程序运行的指令总数也随之增加。本实验中，多数测试程序，如 kmp、tera-sort、wordcount 的主循环部分涉及大量的访存操作，并且被测试应用的数据集增大时，相对于计算指令，增加更多的是访存指令。由于只有对访存指令的模拟过程中会产生不同组件之间的消息传递，更多的访存指令意味着将使并行模拟框架中各个工作线程处于更加忙碌的状态。

图 4.14 中每个散点指示一个测试程序在指定的输入集下可以达到的最大加速比，并使用折线连接。观察可知，随着程序输入集的增大，同一个测试程序呈现出的加速效果也更明显。几乎每个测试程序都在测试的过程中呈现出最大 3 倍以上的实际加速效果或趋势。不同是，kmp 和 terasort 在很小输入集下即可达到非常显著的加速效果，而有的测试程序在非常大的输入集下才能达到同等的加速效果。综合来看，优化后的并行离散事件模拟框架在 7 个工作线程的配置下，可以达到最大 3.94 倍的性能提升，平均性能提升 3.17 倍。

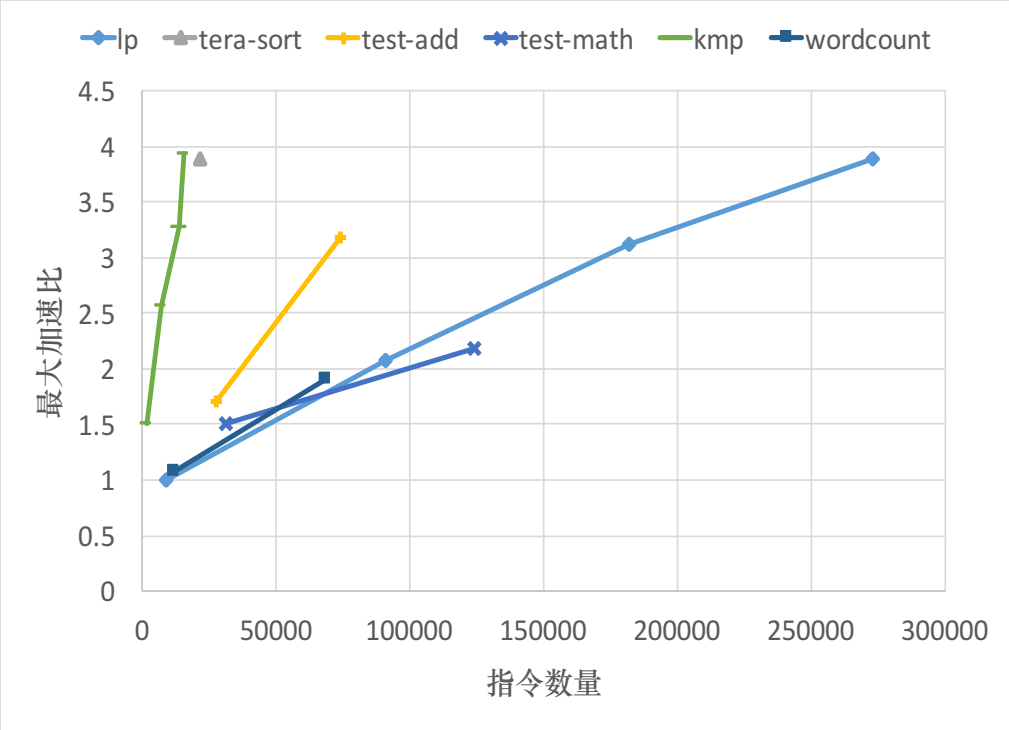

图 4.14 多线程加速效果

4.6 本章总结

基于利用硬件平台并行性和提高模拟算法的并发度的思路，本章设计并实现了从事件调度算法、未来事件队列、时间推进算法和内存池四个角度改进和优化的并行框架。

首先，针对事件同步、分配过程中昂贵的锁开销，本章实现了一个基于单一读者、单一写者模型的无锁事件队列，消除了因读取事件队列而引入的频繁的锁操作。

其次，本章中提出基于随机消息映射策略的事件调度算法，通过在模拟组件与框架服务单元之间添加一个消息分配器实现事件的集中调度。由于增强了对事件调度的控制能力，更容易实现一个负载均衡的动态调度方案。实验中，采用随机消息映射策略的动态事件调度方案达到了几近完美的负载均衡性，为提高多线程可扩展性奠定基础。

同时，为消除模拟框架时间推进算法中最小时间戳更新过程导致的性能瓶颈，本章提出采用红黑树结构管理事件，在此基础上可以实现逐个周期推进的时间推进算法。

最后，针对模拟过程中动态内存管理方案的性能缺陷，本章提出使用内存池技术管理消息存储空间，极大地缩减了复杂内存空间申请、释放操作的数量，节省了运行时间。

综合实验表明，改进的并行离散事件模拟框架在 7 个工作线程的配置下，可以达到最高 3.94 倍的性能提升，平均性能提升 3.17 倍。

第5章 论文总结

本文在分析和总结常用体系结构模拟器加速技术的基础上，从指令译码、事件队列无锁化、事件调度算法、时间推进算法及内存管理对基于并行离散事件模拟框架的众核并行模拟器进行加速。代表性的高通量应用程序测试实验表明加速效果显著。

5.1 研究工作总结

5.1.1 分析高通量处理器特征，总结常用模拟加速技术

本文首先分析大数据应用在数据格式、数据通路、处理需求以及实时性方面的特征，并且对高通量计算机系统与高性能计算机系统在负载特征、并行性、性能成本、可靠性以及性能目标等多方面进行对比。这些分析为模拟器加速技术的应用提供了重要依据。

学习和总结二进制翻译、穿线码、精简输入集、截短模拟执行、采样、FPGA 仿真等常用模拟器加速方法的原理、使用方法以及典型应用。根据对常用模拟器加速技术的了解将其分为三大类别：提高单条指令或单个指令块模拟速度、减少被模拟的指令总数以及提高模拟器的并发性和充分利用模拟平台的并行性。本文中使用的加速方案也是基于这些加速思路展开的。

5.1.2 千核万线程规模的模拟加速

本文致力于对高通量众核处理器的并行模拟进行加速，并发模拟的规模达到千核万线程级别。具体工作展开如下：

（1）采用查找表技术对模拟器指令译码进行加速。查找表技术被应用于 PopCount 问题求解、指令条件域判断和软件踪迹缓存。在简化代码的同时，取得了 26.14 倍的加速效果。

（2）从多个角度对 BDSim 并行模拟框架进行优化。优化的角度包括：引入随机映射的事件调度算法、实现无锁化的未来事件队列、采用 cycle-by-cycle 的时间推进算法，以及引入内存池方案优化内存管理。显著提高了工作线程间的负载均衡性，内存管理的时间开销降低 12.47%，并在使用 5 个工作线程进行模拟时即可达到接近 4 倍的性能提升。

5.2 未来工作展望

实验中发现，当工作线程数量达到 4 以后，继续增加工作线程数量获得的性能增益变缓。因此，下一步工作方向是进一步提高并行模拟框架的可扩展性，或者在可扩展性难以提升的情况下动态确定最优工作线程数。

如前所述，精简模拟过程中的指令数量被认为是在提高模拟器运行速度方面非常高效的。在本文工作完成的过程中也按照这一思路使用代表元采样技术对模拟过程进行加速，效果显著。然而，由于模拟器版本迭代频繁，该技术未能在最近模拟器版本中被继承下来。在最新版本模拟器中修复对该功能的支持将会是后期工作中非常值得期待的一步。

参考文献

李国杰. 2013. 大数据对计算机系统的挑战[R]. 长沙：CNCC 大数据论坛.

李文明, 叶笑春, 张洋, 宋风龙, 王达, 唐士斌, 谢向辉. 2015. BDSim: 面向大数据应用的组件化高可配并行模拟框架[J]. 计算机学报: 38(1)：1959-1975.

李剑慧, 马湘宁, 朱传琪. 2007. 动态二进制翻译与优化技术研究[J]. 计算机研究与发展: 44(1):161-168.

林健, 毛晶莹. 1998. 并行离散事件仿真 PDES 策略比较研究[J]. 系统工程理论与实践: 18(9):14-19.

杨小溪, 高晓彤, 张为华. 2011. 若干体系结构模拟器加速技术的分析与对比[J]. 计算机应用与软件: 28(8):5-8

吕慧伟, 程元, 白露, 陈明宇, 范东睿, 孙凝晖. 2013. 众核处理器和众核集群的并行模拟[J]. 计算机研究与发展: 50(5):1110-1117

詹剑辉, 王磊, 孙凝晖. 2011. 高通量计算机的性能评价[J]. 中国计算学会通讯:7(7):40-3.

维基百科. 查找表[EB/OL]. 2016. 维基百科, [2016-04-21]. https://zh.wikipedia.org/zh/查找表

维基百科. 现场可编程门阵列[EB/OL]. 2016. 维基百科, [2016-04-23]. https://zh.wikipedia.org/zh/现场可编程逻辑门阵列

佚名. 盘点龙芯指令系统二进制翻译技术发展战略[EB/OL]. 电子发烧友网, [2015-05-06]. http://www.elecfans.com/article/90/156/2015/0506370258.html

ARM Architecture Reference Manual [EB/OL]. ARM, [2016-2-8]. http://poincare.matf.bg.ac.rs/~milan/download/micro/arm.pdf

AUSTIN T, LARSON E, ERNST D. 2002. Simplescalar: An infrastructure for computer system modeling[J]. IEEE Transactions on Computers: 35(02): 59-67.

Avadh Patel, Furat Afram, Shunfei Chen, Kanad Ghose. 2011. MARSS: a full system simulator for multicore x86 CPUs[C]. Proceedings of the 48th Design Automation Conference: 1050-1055

Bellard, Fabrice. 2005. QEMU, a Fast and Portable Dynamic Translator[C]. USENIX Annual Technical Conference, FREENIX Track. Anaheim, CA, USA: 41-46.

Brown SD, Francis RJ, Rose J, Vranesic ZG. 2012. Field-programmable gate arrays[J]. Springer Science & Business Media: Vol 180.

Chiou Derek, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, Hari Angepat. 2007. FPGA-Accelerated Simulation Technologies (FAST): Fast,

Full-System, Cycle-Accurate Simulators[C]. Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. Chicago, USA: 249-261.

Cline, Robert C, and Daniel Garfinkel. 21 Sep. 1993. Method of managing memory allocation by association of memory blocks with a tree structure. U.S. Patent No. 5,247,634[P].

Defense, 1998. U.S.D.o., High Level Architecture Interface Specification Version 1.3 2[S].

Hamerly G,Perelman E,Lau J. 2005. SimPoint 3.0: Faster and more flexible program phase analysis[J]. Journal of Instruction Level Parallelism: 7(04).

Krasnov A, Schultz A, Wawrzynek J, Gibeling G, Droz PY. 2007. RAMP Blue: A message-passing manycore system in FPGAs.[C]. International Conference on Field Programmable Logic and Applications. IEEE: 54-61.

Michael Ferdman, Almutaz Adileh, Onur Kocberber, et al. Clearing the clouds: a study of emerging scale-out workloads on modern hardware[C]. Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems. London, UK, 2012:37-48.

Miller J.E., Kasture H., Kurian G., Gruenwald C., Beckmann N., Celio C., Eastep J., Agarwal A. 2010. Graphite: A distributed parallel simulator for multicores[C]. IEEE 16th International Symposium on High Performance Computer Architecture. Bangalore, India:1-12

Nikolaos Hardavellas, Stephen Somogyi, Thomas F. Wenisch, Roland E. Wunderlich, Shelley Chen, Jangwoo Kim, Babak Falsafi, James C. Hoe, Andreas Nowatzyk. 2004. SimFlex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture[J]. SIGMETRICS Performance Evaluation Review, 31(4): 31-34.

Osowski, AJ Klein, and David J. Lilja. 2002. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. Computer Architecture Letters 1.1: 7-7.

Ostermann S, Plankensteiner K, Prodan R, Fahringer T. 2010. Groudsim: An event-based simulation framework for computational grids and clouds[C]. In Euro-Par 2010 Parallel Processing Workshops: 305-313.

Penry DA, Fay D, Hodgdon D, Wells R, Schelle G, August DI, Connors D. 2006. Exploiting parallelism and structure to accelerate the simulation of chip multi-processors[C]. In High-Performance Computer Architecture, The Twelfth International Symposium:IEEE 29-40.

Richard M. Fujimoto. 1990. Parallel Discrete Event Simulation[C]. Communications of the ACM, 33(10):30-53.

Rao, Dhananjai M., and Philip A. Wilsey. 2002. An ultra-large-scale simulation framework[J]. Journal of Parallel and Distributed Computing: 1670-1693.

Rotenberg, Eric, Steve Bennett, and James E. Smith. 1996. Trace cache: a low latency approach to high bandwidth instruction fetching[C]. Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture: IEEE Computer Society 24-35.

Skadron, Kevin, et al. 1999. Branch prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques[C]. IEEE Transactions on Computers: 1260-1281.

Wang, Jingjing, Deepak Jagtap, N. Abu-Ghazaleh, et al. 2014. Parallel discrete event simulation for multi-core systems: Analysis and optimization[C]. IEEE Transactions on Parallel and Distributed Systems: 1574-1584.

Wikipedia. 2016. Memory Pool[EB/OL]. [2016-6-20]. https://en.wikipedia.org/wiki/Memory_pool.

Wunderlich RE, Wenisch TF, Falsafi B, Hoe JC. 2003. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In Computer Architecture, 2003. Proceedings. 30th Annual International Symposium: IEEE 84-95.

Yi, Joshua J., et al. 2005. Characterizing and comparing prevailing simulation techniques[C]. In 11th International Symposium on High-Performance Computer Architecture: IEEE 266-277.

致 谢

本文由我的导师安虹教授悉心指导完成，过去的研究生生活里安老师对我的科研倾注了大量心血，衷心地感谢安老师的付出。她交给我们进行科研创新的方法，知无不言；她传授我们丰富的生活经验，言无不尽；她引导我们健康、开朗地生活，不厌其烦；安老师以培养具有扎实工程基础、完备科研能力和强大心理素质的全方位人才的标准要求学生，她在计算机系统结构的发展历程和前瞻上的见解使我受益匪浅。和蔼可亲的安老师，以朋友间交流的方式与大家相处，让每一位实验室成员觉得温暖踏实；治学严谨的安老师对实验室参与的众多科学研究、项目合作和国际竞赛都投入大量的时间和精力，给同学们智力上的指导和精神上的支持。安老师对科学理想和现实生活之间平衡关系的深刻理解以及她的敬业精神永远值得我学习！

衷心感谢评审老师对本文认真细致地评审并给出中肯的评阅意见！临近毕业心态浮躁加之论文写作经验不足，未能保证应有的毕业论文质量。既给两位老师造成了糟糕的阅读体验，又使自己不能及时地进入到答辩环节，本人悔之不及。感谢相交多年的好朋友吴琳莉和吴义镇，在我论文修改工作停滞不前时给予我无私的支持和帮助，使我受益匪浅！

研究生一年级期间，作为实验室的新成员，感谢师兄师姐们给予的无私帮助和耐心指导！每周两次的讨论班上，陶醉得听师兄师姐们讲述工作进展和业内最新的科研成果实在是不可多得的享受，海博、王涛、程亦超他们提问题的艺术更加令人赞叹。除了忘我的科研，他们还让我学习到如何玩、教我做收益值高且半衰期长的事情。还能再跟汪朝辉学习健身吗？还可以跟小川、爱明、彭毅切磋球技吗？还有机会再与程亦超夜半三更偷偷摸摸地在无人的楼道里轮流拉那把 150 块钱的小提琴吗？

与林晗一起在计算所客座学习的一年是我见识空前开阔的一年。公司化的严格管理、相对频繁的人员变动以及帝都的繁华和阴霾都给我留下了深刻的印象。在这里不断有新人加入伴随毕业生和部分员工的离开，让我第一次感觉到人生相识不易。感谢这一年半里范东睿、王达、叶笑春等老师对科研工作的悉心指导；感谢李文明、张洋、朱亚涛师兄在项目工作中给予方方面面的关照；感谢雪夜里陪我一起游什刹海、吃烤红薯的李易、薛瑞、常成娟；感谢在 AMS 欣麓园公寓里多少次一起忘情言欢的兄弟姐妹们！

硕士生活的最后半年里，身边的一切都明显加快了步伐，催促着我离开这美丽的校园，伤感之情油然而生。所幸实验室还有一群欢快活泼的师弟师妹们，看着他们工作取得新的成果，常觉得后生可畏；听着他们闲暇时的互黑，总让人忍俊不禁。感谢你们，与你们的相处，虽然短暂，却无比开心！

这三年里，自始至终陪伴着我的是我的家人！对我帮助最大的是我的女朋友，相处五年来，每晚的电话联系总是能带走我一天的疲惫与烦躁。虽然无法像小时候一样被无微不至的照顾，父母的嘘寒问暖却是支持我前进的强大动力，感谢你们二十几年对我的关爱和辛勤付出！

最后，感谢自己！感谢自己二十年的努力，才能成为今天的自己，遇到这些人、经历这些事！

在读期间发表的学术论文与取得的其他研究成果

1. 发表论文

方国庆, 李文明, 余洋, 张洋, 叶笑春, 安虹, 高通量众核并行模拟加速技术研究，计算机工程.

2. 技术报告

方国庆, PDES 框架分析与性能调优, 2016.

3. 科研项目

[1] 国家核高基重大专项（课题号：2013ZX0102-8001-001-001）.

[2] 国家“863”计划项目“E 级超级计算机新型体系结构及关键技术路线研究”（课题号：2015AA01A301）.

[3] 国家 863 高技术研究发展计划项目（课题号：2012AA010901）.