基于 GEM5 的计算机系统搭建及性能改善方法研究

王明君

（计算机科学与技术系，西安交通大学） 计算机 74 班 2173320641

摘 要 本文基于 GEM5 模拟器，实现了 x86 架构和 ARM 架构下的多 CPU，多级 Cache 的计算机系统的搭建。在 ARM 架构下，完成了对 Cortex-A9 处理器，Cortex-A15 处理器的仿真模拟。总结了系统调用模式和全系统模式下的同种计算机系统的性能差异。比较了 x86 和 ARM 架构的计算机的性能差异。通过分析模拟器输出的数据文件，总结了计算机性能的改善方法。在计算机系统中增加 CPU 的数量、适量增加 Cache 可以有效提高计算机执行程序的效率，节省时间；使用超标量技术和增加运算逻辑单元可以有效提升计算机性能。

关键词 GEM5;计算机性能;处理器;ARM;x86

Research on Computer System Construction and Performance Improvement Method Based on GEM5

Wang Mingjun

(Department of Computer Science and Technology, Xi'an Jiaotong University) ClassNo：74 Student ID：2173320641

Abstract Based on the GEM5 simulator, this paper implements the construction of a multi-CPU, multi-level cache computer system under x86 architecture and ARM architecture. Under the ARM architecture, the simulation of Cortex-A9 processor and Cortex-A15 processor is completed. The performance differences between the same computer system in the system call mode and the whole system mode are summarized. The performance differences between x86 and ARM architecture computers are compared. By analyzing the data files output by the simulator, the method for improving the performance of the computer is summarized. Increasing the number of CPUs and the appropriate amount of Cache in the computer system can effectively improve the efficiency of the computer's execution of the program and save time; using superscalar technology and adding arithmetic logic units can effectively improve computer performance.

Key words GEM5; computer performance; CPU; ARM; x86

1 引言

微处理器的指令集架构（Instruction Set Architecture）通常分为复杂指令集运算（Complex Instruction Set Computing，CISC），精简指令集运算（Reduced Instruction Set Computing，RISC），显式并行指令集运算（Explicitly Parallel Instruction Computing，EPIC）和超产指令集运算（Very Long Instruction Word）。GEM5 支持种类繁多的体系平台，这使得模拟器模块化程度较高且易于在不同 CPU 之间切换。本文只选取 CISC 中有代表性的 x86 架构，以及 RISC 中的 ARM 架构这两者进行研究。本文在 x86 和 ARM 架构下分别构建了多种结构的计算机系统，旨在研究和验证从结构上改善计算机性能的方法。

1 一文看懂arm架构和x86架构有什么区别 （https://zhuanlan.zhihu.com/p/95028674 2019-12-03）

1.1 X86架构计算机系统

x86 架构是重要地可变指令长度的 CISC，x86 目前被使用于大多数 PC 架构中，如笔记本、台式机、小型服务器。在 GEM5 中，x86 架构的表现主要是模拟了 64 位 x86 CPU，能够在 SMP 的配置模式下启动原始 Linux kernel。

1.2 ARM架构计算机系统

ARM 架构过去称作进阶精简指令集机器（AdvancedRISCMachine），它是一个 32 位精简指令集（RISC）处理器架构，其广泛地使用在许多嵌入式系统设计。由于节能的特点，ARM 处理器非常适用于移动通讯领域，符合其主要设计目标为低耗电的特性[1]。在 GEM5 中，ARM 架构的表现主要是模拟了 Cortex-A9，支持 Thumb，Thumb-2，VFPv3 和 NEON 指令集。

1.3 X86与ARM架构对比

X86 CPU 采用冯诺依曼体系结构，指令和数据存储在同一个存储器中，指令和数据只能采用相同宽度，并且总线在同一时刻只能传递一种内容，指令或者数据，速度受限。ARM CPU 采用哈佛体系结构，指令和数据采用不同的存储器，因此，指令和数据可以采用不同的数据宽度，并且总线操作较快。X86 CPU 是采用 CISC 指令集，拥有丰富的指令，每一代新的 CPU 为了保持对前期产品的兼容，会在保留原有指令的基础上扩展新的指令，导致指令系统越来越复杂，但是程序员有更多可选择的指令。ARM CPU 采用 RISC 指令集，只保留常用的指令，指令结构简单，程序员可选择的指令也少。

X86 架构和 ARM 架构的差异对比如表 1 所示[1]。

表 1 x86 架构和 ARM 架构在汇编上的差异

方面	x86	ARM
体系结构差异	冯诺依曼体系结构	冯诺依曼体系，哈佛体系结构
是否允许不对齐存储器地址访问	是	否
指令和数据的总线	相同	不同
程序与数据存储空间	不独立	相互独立
指令长度	不等长	等长
访问存储器的指令	所有指令	load/store 指令
指令结构	复杂	简单

1.4 计算机性能的提升方式

1.4.1 高速缓冲存储器 Cache

Cache 位于 CPU 和主存储器之间，其工作原理是保存 CPU 最常用数据；当 Cache 中保存着 CPU 要读写的数据时，CPU 直接访问 Cache。由于 Cache 的速度与 CPU 相当，CPU 就能在零等待状态下迅速地实现数据存取，大大提高了计算速率。在 Cache 的连接设计方面，可以采用组相联设计，提升相联度，降低存储器不同数据块争用同一 Cache line 的概率；在 Cache 数量上，Cache 又分为 L1Cache（一级缓存）和 L2Cache（二级缓存），L1Cache 主要是集成在 CPU 内部，而 L2Cache 集成在主板上或是 CPU 上，因此采用多级 cache，增加存储结构的中间层，可以减少缺失代价，在一定程度上提高数据存取性能。在本文中，进一步实现了拥有共享 L3Cache 的计算机系统，并研究了其对计算机系统性能的作用。

1.4.2 中央处理器 CPU

中央处理器（CPU，central processing unit）作为计算机系统的运算和控制核心，是信息处理、程序运行的最终执行单元。在一个计算机系统中增加 CPU 的数量，可以组成多 CPU 系统。这些处理器共享内存、I/O 通道、控制器和外部设备，并通过主板上的总线进行的通讯。实现多处理器的系统可以增加吞吐量，增加系统可靠性。并且对支持多线程的程序的执行效率可以有显著改善，因为每个线程可以分配给不同的处理器，使其进入并行运算的状态。

与多 CPU 系统类似的还有多核处理器系统，其是指一个 CPU 有多个核心处理器，处理器之间通过 CPU 内部总线进行通讯，共享使用内存，大规模减少了 cache 的数量和开销，其性能和开销比多 CPU 系统小[2]。在本文中，实现了多 CPU 系统，并将其与单 CPU 系统在 openMP[2]多线程编程框架下的计算性能进行测试和比较。

2 相关工作

Nathan Binkert 等人[2]提出的 GEM5 模拟器结构是 M5 和 GEMS 模拟器的结合，它支持大多数 ISA，

1 一文看懂arm架构和x86架构有什么区别 （https://zhuanlan.zhihu.com/p/95028674 2019-12-03）

2 CPU Basics: Multiple CPUs, Cores, and Hyper-Threading Explained （https://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/ 2018-10-12）

3 https://github.com/EWouters/Using-GEM5-for-Architecture-Exploration-in-Multi-Core-Processors 2018-4-30

4 1_ARM Cortex-A 系列处理器(A5、A7、A8、A9、A15)区别对比 （https://wenku.baidu.com/view/332a395487c24028905fc31b.html 2014-12-23）

并且有灵活的建模方式。GEM5 模拟器提供了种类繁多的组件和功能以实现其灵活性。图 2 展示了 GEM5 中不同的处理器和不同存储系统的在执行速度和正确性上的分布图。

GEM5 支持的 ISA 有 Alpha, ARM, MIPS, Power, SPARC, and x86。

GEM5 可以在两种模式下执行：系统调用模式（System-call Emulation，SE)和全系统模式（Full-System，FS）。在 SE 模式下，每当程序执行系统调用时，gem5 都会捕获并模拟该调用，通常会通过将其传递给主机操作系统。目前在 SE 模式下没有线程调度器，因此必须将线程静态映射到 CPU 核，使其与多线程应用程序一起使用受到限制。在 FS 模式下，GEM5 可以运行操作系统，并且支持中断，错误，优先级和 I/O 设备。FS 模式下的模拟正确性更高，可以执行的应用程序也更多样。

GEM5 支持四种 CPU 模型：AtomicSimple, TimingSimple, In-Order, 和 O3. AtomicSimple, TimingSimple 都属于无流水 CPU 模型，开销也较低。InOrder 模型是有序流水线 CPU，五级流水表现为：取值、译码、执行、访存、写回。O3 模型是无序流水线 CPU，七级流水表现为：取值、译码、重命名、发射、执行、写回、提交。

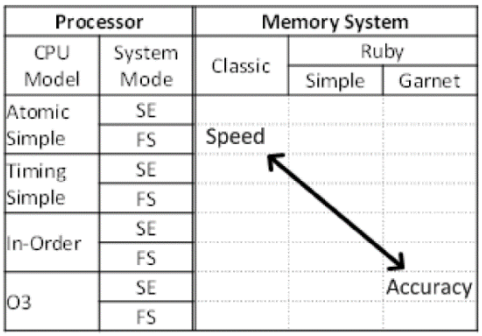

图 1 速度和正确性谱图

3 Gem5 指令集架构仿真与系统配置

3.1 X86架构

3.1.1 Syscall Emulation 模式下仿真

SE 模式下的计算机系统搭建主要使用 python 配置文件。Python 文件提供了初始化，配置和仿真控制。 GEM5 模拟器几乎在启动时立即开始执行 Python 代码；标准的 main（）函数是用 Python 编写的，所有命令行处理和启动代码都是用 Python 编写的。在此 python 文件中，可以指定 CPU 模型种类，cache 大小，cache 数量以及各种组件之间的连接。在 gem5 文件夹下的 config/example/se.py 是一个非常简便的系统配置文件，在命令行就可以对计算机系统的组件和参数进行配置。其他的配置命令可以通过输入”build/X86/gem5.opt configs/example/se.py －help”命令查询到。而在本文中，并未采用 se.py 作为计算机系统的搭建基准，而是在 configs/tutorial/文件夹下重新编写 python 配置文件。为了研究，此次实验总共搭建了 5 种 TimingSimpleCPU 的计算机系统：单 CPU 无 Cache，单 CPU 一级 Cache，单 CPU 二级 Cache，单 CPU 3 级 Cache，双 CPU 3 级 Cache（其中第三级作为两个 CPU 的共享 Cache）；分别存储于文件“simple.py” “one_level.py” “two_level.py” “three_level.py” “mtc.py”中. 此外，文件夹中也留存了以上计算机系统使用 AtomicSimpleCPU 和 O3CPU 的 python 文件。

四种计算机系统的逻辑结构示意图如图 2，图 3，图 4，图 5，图 6 所示。各组件配置参数如表 2 所示。

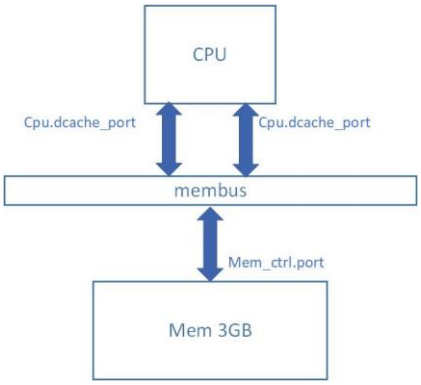

图 2 单 CPU 无 Cache 逻辑结构图

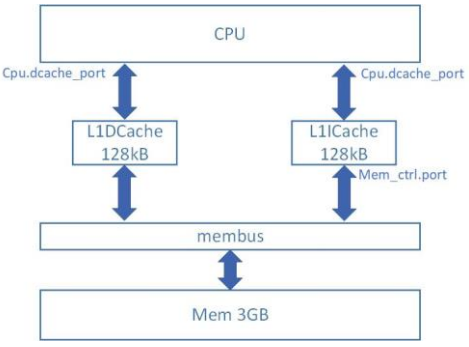

图 3 单 CPU 1 级 Cache 逻辑结构图

1 cortex （https://baike.baidu.com/item/cortex/4476976?fr=aladdin）

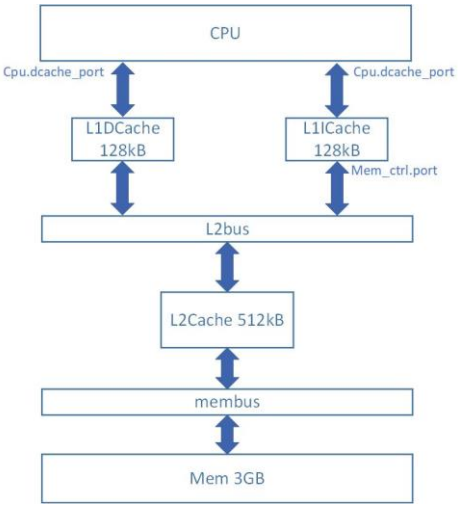

图 4 单 CPU 2 级 Cache 逻辑结构图

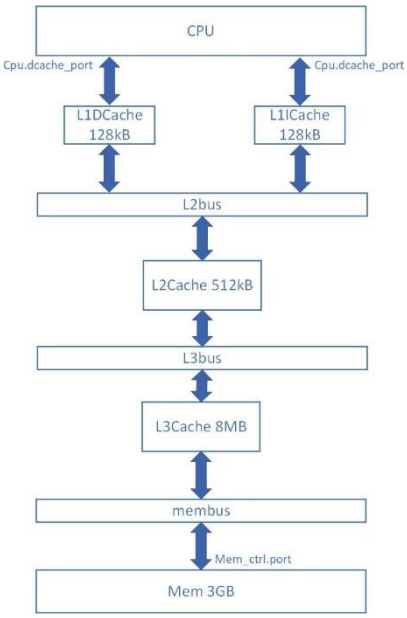

图 5 单 CPU 3 级 Cache 逻辑结构图

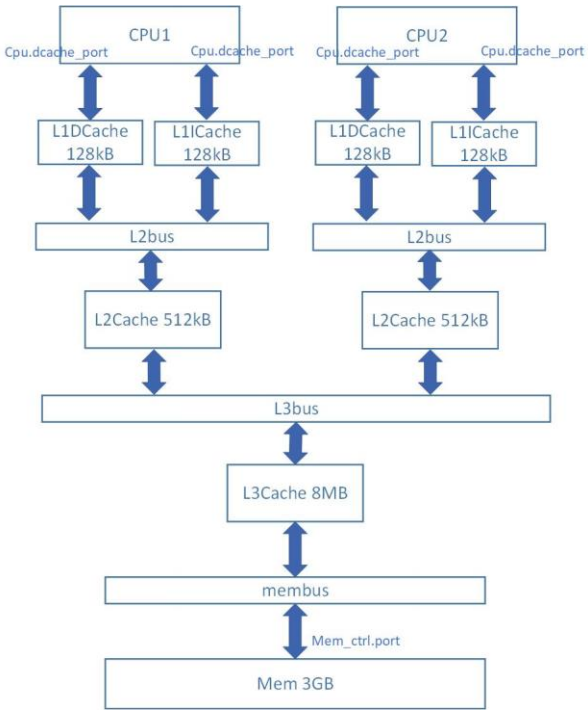

图 6 双 CPU 3 级 Cache 逻辑结构图

表 2 x86 下各组件配置参数

参数	值
CPU 模型	AtomicSimpleCPU, TimingSimpleCPU, O3CPU
时钟频率	3GHz
L1DCache 大小	128kB
L1ICache 大小	128kB
L1Cache 组相连	4
L2Cache 大小	512kB
L2Cache 组相连	8
L3Cache 大小	8MB
L3Cache 组相连	8
Mem 范围	3GB

3.1.2 Full System 模式下仿真

FS 模式需要用到 linux 系统的镜像文件。与 SE 模式类似，config/example/fs.py 文件提供了一个非常简便的在 FS 模式下的系统配置文件。而在本文中，并未采用 fs.py 作为计算机系统的搭建基准，而是在 configs/full_system/文件夹下重新编写 python 配置文件。为了研究，本次实验构建了一个双 CPU，3 层 Cache 的计算机系统。存放于运行脚本 run.py

1 一文看懂arm架构和x86架构有什么区别 （https://zhuanlan.zhihu.com/p/95028674 2019-12-03）

2 CPU Basics: Multiple CPUs, Cores, and Hyper-Threading Explained （https://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/ 2018-10-12）

3 https://github.com/EWouters/Using-GEM5-for-Architecture-Exploration-in-Multi-Core-Processors 2018-4-30

4 1_ARM Cortex-A 系列处理器(A5、A7、A8、A9、A15)区别对比 （https://wenku.baidu.com/view/332a395487c24028905fc31b.html 2014-12-23）

和系统配置 system.py 中。CPU 的类型实现了 AtomicSimpleCPU，TimingSimpleCPU 和 O3CPU 三种不同的 CPU 模型。与 SE 模式下搭建过程有所不同，FS 模式中用到了 x86.py 文件，该文件初始化了大部分 x86 特定的系统参数，包括 I/O、多处理器支持和 BIOS。因为实验中使用了双 CPU 的结构，所以 x86.py 中的 BIOS 设置需要进行调整，主要的代码调整是在高级可编程中断控制器（Advanced Programmable Interrupt Controller ，APIC）和 CPU 入口分配这两处，具体的调整措施将在实验验证部分详细说明。其中，APIC 是为了适应 multiple processor（MP，多处理器）环境而引入的机制，整个 APIC 体系可以分为两大部分：Local APIC 和 I/O APIC。Local APIC 是整个 APIC 体系的核心，它在处理器的内部；而 I/O APIC 是芯片组的一部分，位于南桥芯片上。每个 local APIC 有自己的 local APIC ID，这个 ID 决定了了逻辑处理器在系统总线上的地址，可以用于处理器间的消息接收和发送，也可用于外部中断消息的接收。I/O APIC 能适用于多处理器环境上。I/O APIC 可以发送中断消息到指向的逻辑处理器上。FS 模式下的计算机系统的逻辑结构示意图如图 7 所示。

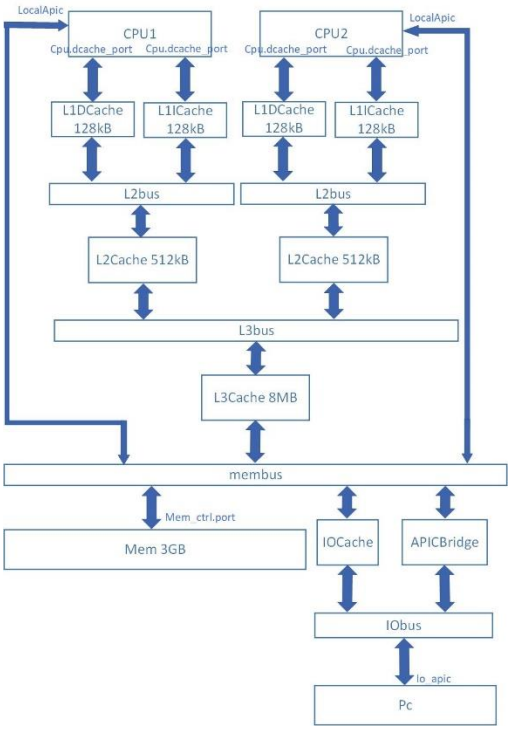

图 7 FS 模式下双 CPU 3 级 Cache 逻辑结构图

3.2 ARM架构

与 x86 架构类似，在 gem5 文件夹下的 config/example/se.py 是一个非常简便的系统配置文件，在命令行就可以对计算机系统的组件和参数进行配置。因为 se.py 文件内涉及到了 ISA 类型的条件判断，所以这个配置文件适用于多种 ISA。但考虑到实验的多样性，在本文中并未采用 se.py 作为计算机系统的搭建基准，而是在 configs/armconfig/ 和 configs/tutorial/文件夹下重新编写 python 配置文件。本次实验中在 ARM 架构下搭建了四种计算机系统：双 CPU 3 层 Cache 的计算机系统、Cortex-A9 的单 CPU 系统、Cortex-A15 的单 CPU 系统、2 个 Cortex-A9 和 2 个 Cortex-A15 的四核 CPU 系统。其中，四核 CPU 系统的逻辑结构图如图 7 所示。Cortex 作为 ARM 架构下的一类处理器，基于 ARMv7 架构。ARM Cortex-A 系列应用型处理器应用于智能手机、数字电视、企业网络和服务器。高性能的 Cortex-A15、可伸缩的 Cortex-A9 处理器均共享同一架构，因此具有完全的应用兼容性，支持传统的 ARM、Thumb 指令集和新增的高性能紧凑型 Thumb-2 指令集 1。

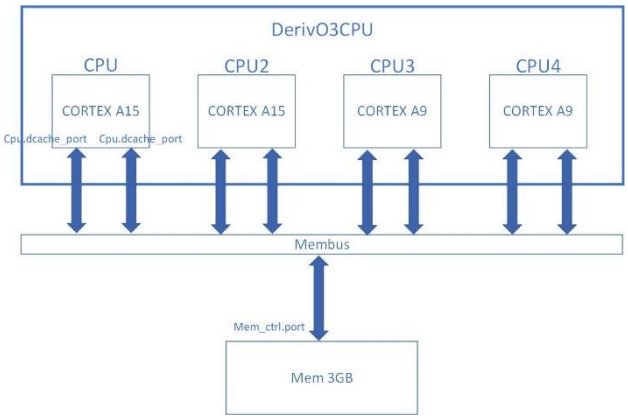

图 8 ARM 架构下 4 核系统逻辑结构图

3.2.1 Cortex-A9

ARM Cortex-A9 处理器隶属于 Cortex-A 系列，指令双发射，基于先进的推测型八级流水线。因为 Cortex-A9 属于乱序 CPU，所以在本文中采用 O3CPU 作为 CPU 初始模型，在具体的配置中还需要进一步修改 CPU 的参数以大致符合 Cortex-A9 的规格。关于配置文件中需要设置的 CPU 参数如表 3 和表 4 所示，其中的拾取（fetch）宽度、解码（decode）宽度、分发（dispatch）宽度、发射（issue）宽度是流水线的参数，用到了 2 路超标量（superscalar）技术[4]：其核心是指在 CPU 中有两条流水线，并且每时钟周期内可以完成一条以上的指令。这里所用的超标量指令流技术一般用在超标

1 cortex （https://baike.baidu.com/item/cortex/4476976?fr=aladdin）

量流水线的前端，即取指段和译码段。算术逻辑单元（ALU）、乘法（MUL）、除法（DIV）、读写端口（RdWrPort）是功能单元（Functional Unit）中的参数[3]。

3.2.2 Cortex-A15

ARM Cortex-A15 处理器隶属于 Cortex-A 系列，指令三发射，支持 15 级以上的流水线[4]。因为 Cortex-A15 属于乱序 CPU，所以在本文中采用 O3CPU 作为 CPU 初始模型，在具体的配置中还需要进一步修改 CPU 的参数以符合 Cortex-A15 的规格。关于配置文件中需要设置的 CPU 参数如表 3 所示，用到了 3 路超标量技术（superscalar）。

表 3 Cortex-A9 和 Cortex-A15 参数配置

参数		Cortex-A9	Cortex-A15
superscalar frontend	decodeWidth	2	3
	fetchWidth	2	3
	issueWidth	2	3
	dispatchWidth	2	3
Functional Unit	IntALU	1	2
	IntMultDiv	1	1
	RdWrPort	1	1

表 4 ARM 下各组件配置参数

参数	值
CPU 模型	O3CPU
时钟频率	2GHz
L1DCache 大小	32kB
L1ICache 大小	32kB
L1Cache 组相连	4
L2Cache 大小	256kB
L2Cache 组相连	4
Mem 范围	3GB

4 实验验证

4.1 测试程序

实验中使用了三种测试程序：BP 算法[5]，使用 openMP 并行编程框架的圆周率算法，多线程实现数的递加。

BP 算法是一种被用于神经网络中的参数学习算法，其学习过程由信号的正向传播(求损失)与误差的反向传播(误差回传)两个过程组成。由于在训练数据的过程中需要进行大量的计算和循环，因此本文将其作为在单 CPU 中的运行的测试程序，以验证 cache 级数和 cpu 类型对计算机性能的影响。

圆周率算法的基本思路是使用下列公式：

$$arctan(1) = \frac{\pi}{4}$$

$$\left(\arctan(x)\right)' = \frac{1}{1+x^2}$$

在求解$arctan(1)$时使用矩形法求解：

$$\int_a^b f(x)\mathrm{d}x = y_0 \Delta x + y_1 \Delta x + \cdots + y_{n-1} \Delta x$$

$$\Delta x = \frac{b-a}{n}$$

$$y = f(x)$$

$$y_i = f\left(a + i \cdot \frac{b-a}{n}\right) \quad i = 0,1,2,\ldots,n$$

$$取 a = 0, b = 1.$$

因为求解过程中包括了大量的迭代，且可以并行执行，故测试程序中加入了 openMP 框架：

```
    omp_set_num_threads(NUM_THREADS);  //设置线程
 #pragma omp parallel private(i)
//并行域开始，每个线程都会执行该代码
{
    double x;
    int id;
    id = omp_get_thread_num();
    for (i=id, sum[id]=0.0;i< num_steps; i=i+NUM_THREADS){
        x = (i+0.5)*step;
        sum[id] += 4.0/(1.0+x*x);
    }
}
```

引入 openMP 后，可执行程序可以分配到不同的 CPU 上并行执行，因此本文将其作为在多 CPU 中的运行的测试程序以验证增加 CPU 数量对计算机性能的影响。源文件中需要导入新的头文件：“#include <omp.h>”。在编译文件时，命令行输入的命令格式为：“gcc -fopenmp -o 文件名 文件名.c”

1 一文看懂arm架构和x86架构有什么区别 （https://zhuanlan.zhihu.com/p/95028674 2019-12-03）

2 CPU Basics: Multiple CPUs, Cores, and Hyper-Threading Explained （https://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/ 2018-10-12）

3 https://github.com/EWouters/Using-GEM5-for-Architecture-Exploration-in-Multi-Core-Processors 2018-4-30

4 1_ARM Cortex-A 系列处理器(A5、A7、A8、A9、A15)区别对比 （https://wenku.baidu.com/view/332a395487c24028905fc31b.html 2014-12-23）

多线程程序将共享同一进程中的全部系统资源，如虚拟地址空间，文件描述符和信号处理等等。但同一进程中的多个线程有各自的调用栈(call stack)，自己的寄存器环境（register context)，自己的线程本地存储(thread-local storage)。在 C 语言中，用 pthread_create 函数创建线程，pthread_exit 函数结束线程，pthread_join 函数指定线程等待。由于 SE 模式下没有线程调度器，使其与多线程应用程序一起使用受到限制，因此该测试程序用于比较 SE 模式和 FS 模式下计算机系统执行任务的区别。源文件中需要导入新的头文件：“#include <pthread.h>”。在编译文件时，命令行输入的命令格式为：“gcc 文件名.c -o 文件名 -lpthread”

4.2 X86架构

4.2.1 多级 cache 的创建与测试

创建多级 cache 需要首先修改 config 文件夹下的 caches.py 配置文件，添加待引入的 Cache 的类定义，如 L2Cache、L3Cache 等，这些类都继承自 GEM5 定义好的 SimObject 类 Cache 组件。若要增加 L3 总线，需要在文件/src/mem/XBar.py 中定义 L3XBar。 实验建立了 4 种 cache 层次，分析输出的 stats.txt 文件发现，增加了一级 cache 后对 CPU 的执行效率提高了 67.92 倍，但是在增加了二级 cache 和三级 cache 之后，CPU 的执行效率并没有增长，反而降低了 0.068%。而理论上采用多级 cache，增加存储结构的中间层，可以减少缺失代价，在一定程度上提高数据存取性能。为了验证实验结果的可靠性，又分别使用 AtomicSimpleCPU 和 O3CPU 对这四种计算机结构进行测试，得到的结果如图 9 和表 5 所示。经对比发现，当 cache 层次增加至 L2Cache 时，O3CPU 和 AtomicSimpleCPU 下的计算机指令速率都程上升趋势，但当增加了 L3Cache 之后，执行速率又有下降，O3CPU 的三层 Cache 结构相比二层 Cache 结构下降了 1.27%；AtomicSimpleCPU 的三层 Cache 结构相比二层 Cache 结构下降了 2.45%；TimingSimpleCPU 的三层 Cache 结构相比二层 Cache 结构上涨了 0.47%。而三种结构的 cache 的缺失率在 AtomicSimpleCPU 和 TimingSimpleCPU 下都成不变趋势，在 O3CPU 下随着 cache 层次增多而下降，如图 10 所示。对于这种与理论不相符的情况，可能的解释是实验是在 SE 模式下执行，SE 模式对计算机系统的模拟有时会偏离真实情况。另一种猜测，导致 cache miss 的可能性有多种，其中可能包括程序中有冗余代码，cache 替换算法的问题，使得即使增加了 cache 级数也不能保证命中率的增加。

同时，通过对比三种 CPU 之间的区别，可以发现 AtomicSimpleCPU 的速度明显大于其他两种 CPU，O3CPU 的速度则是最慢。实验证实了在图 1 中显示的三种 CPU 之间的速度关系。通过分析 stats 输出文件，也发现 AtomicSimpleCPU 并没有 Miss Status and Handling Register（MSHR）有关的数据，也没有各级 cache 的确实延迟数据。MSHR 可以看作一个有一定大小的队列，先入先出，用来存放数据请求 miss 的事件。对于有多层的 cache，CPU 对 L1 发出数据请求，如果命中了，则将数据送入 CPU。如果 L1 中没有数据（发生了 cache 缺失），将数据请求加入到 MSHR 中，排队向下一级 cache 进行请求数据。如果下一级 cache 将数据发送回来了，则 MSHR 就会将队列中对应的请求删除。这些带有 mshr 的延迟和未命中率提供了有关缓存未命中时 CPU 必须停顿多长时间的度量。而 AtomicSimpleCPU 是最简单的模型，一个 cycle 完成一条指令的执行，memory 模型比较理想化，访存操作为原子性操作，适用于快速功能模拟，因此可能缺少了 MSHR 参数。

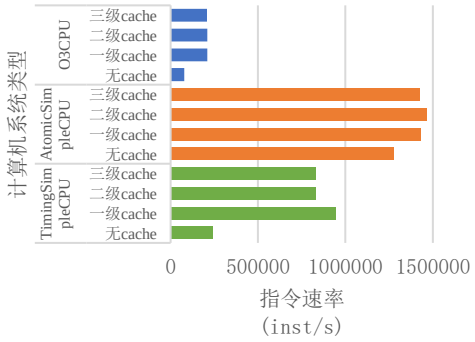

图 9 不同计算机系统下的指令速率关系

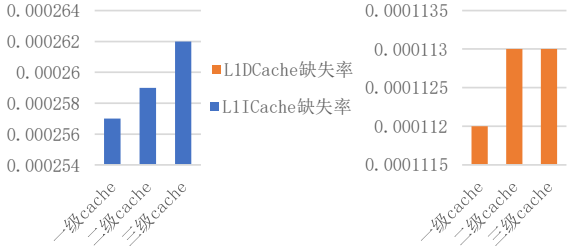

图 10 03CPU 模型在多级 cache 下的 L1Cache 缺失率

1 cortex （https://baike.baidu.com/item/cortex/4476976?fr=aladdin）

表 5 三种 CPU 下 cache 级数与计算机性能的关系

	TimingSimpleCPU				AtomicSimpleCPU				O3CPU			
	无cache	一级cache	二级cache	三级cache	无cache	一级cache	二级cache	三级cache	无cache	一级cache	二级cache	三级cache
指令速率(inst/s)	242350	945266	828059	831912	1279056	1429779	1463439	1427594	78458	209441	211407	208725
运行时间(s)	4.780095	0.076922	0.076974	0.077029	0.035995	0.035995	0.035995	0.035995	0.286029	0.008767	0.008685	0.00869
L1Icache 缺失率		0.000014	0.000014	0.000014		0.000014	0.000014	0.000014		0.000257	0.000259	0.000262
L1Dcache 缺失率		0.000093	0.000093	0.000093		0.000093	0.000093	0.000093		0.000112	0.000113	0.000113
L2cache 缺失率			0.983329	0.983329			0.983329	0.983329			0.981868	0.981431
L3cache 缺失率				1				1				1
L1Icache 缺失延迟		48900.83745	62726.43064	76326.71745						50598.3059	64011	78708.90463
L1Dcache 缺失延迟		49369.12342	62269.82911	75999.0538						50763.2219	62922.9665	77071.77954
L2cache 缺失延迟			62326.10678	69589.75354							63306.04	77936.46812
L3cache 缺失延迟				55603.75354								63950.46812

4.2.2 多 cpu 的创建与测试

SE 模式下，在配置文件夹下创建多 CPU 的系统配置文件，核心操作是声明两个 CPU，并分别创建一个线程与其相对应。加入的程序代码段如图 11 和图 12 所示。通过运行此结构并查看 config.ini 文件可以验证结构的正确性，如图 13 所示。

```
system.cpu1 = AtomicSimpleCPU()
system.cpu1.createThreads()
system.cpu2 = AtomicSimpleCPU()
system.cpu2.createThreads()
```

图 11 CPU 声明部分

```
multiprocesses = []

process = Process()

process.cmd = ['tests/test-progs/srccode/multithread']
multiprocesses.append(process)

system.cpu1.workload = multiprocesses[0]
system.cpu2.workload = multiprocesses[0]
```

图 12 CPU 工作任务分配

```
[system.cpu1]
type=AtomicSimpleCPU
children=dcache dtb icache int
branchPred=Null
checker=Null
clk_domain=system.clk_domain
cpu_id=-1
default_p_state=UNDEFINED
[system.cpu2]
type=AtomicSimpleCPU
children=dcache dtb icache interr
branchPred=Null
checker=Null
clk_domain=system.clk_domain
cpu_id=-1
default_p_state=UNDEFINED
do_checkpoint_insts=true
```

图 13 双 CPU 的 config.ini 文件

为了验证增加 CPU 数量可以有效提高计算机性能，本文使用对比实验，将同一段程序以并行运行的形式和以串行运行的形式放入双 CPU 系统，分析导出的 stats.txt 文件，现将结果总结如表 6 所示。计算速率比较如图 14 所示。分析数据发现，双 CPU 系统中，在两个 CPU 共同工作进行计算时，比较较只有一个 CPU 工作时，其耗时（sim_seconds）缩短了 45.96%。且由表格中 CPU1 和 CPU2 提交的指令数和周期数可以表明，在双 CPU 系统中确实由 2 个 CPU 同时运作来执行程序，使得程序执行的耗时变短，计算变快。然而值得注意的是，2 个 CPU 一起工作的性能（host_inst_rate）比只有 1 个 CPU 单独工作时的性能降低了 13.70%。造成这种现象的原因应当是由于在两个 CPU 一起工作时，其原因可能是每一个 CPU 都需要有较为独立的电路支持，有自己的 Cache，而他们之间通过总线进行通信。在这样的架构上运行一个并行程序，每一个线程被分配到一个独立的 CPU 上，线程间的所有协作都要走总线，而共享的数据更是有可能要在好几个 Cache 里同时存在，提升了总线开销，降低了计算机的性能。也可以发现在双 CPU 系统中使用 1 个 CPU 的计算机性能比单 CPU 系统中使用 1 个 CPU 降低 6.56%，其原因也应当是计算机架构复杂程度的提升导致性能的轻微下降，说明了如果仅仅增加 CPU 数量但不将计算任务多线程地分配给这些 CPU，那么在这种架构下计算机的性能会因计算机复杂化的结构而降低。

1 一文看懂arm架构和x86架构有什么区别 （https://zhuanlan.zhihu.com/p/95028674 2019-12-03）

2 CPU Basics: Multiple CPUs, Cores, and Hyper-Threading Explained （https://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/ 2018-10-12）

3 https://github.com/EWouters/Using-GEM5-for-Architecture-Exploration-in-Multi-Core-Processors 2018-4-30

4 1_ARM Cortex-A 系列处理器(A5、A7、A8、A9、A15)区别对比 （https://wenku.baidu.com/view/332a395487c24028905fc31b.html 2014-12-23）

表 6 双 CPU 与单 CPU 性能表现

	双 CPU 系统		单 CPU 系统
	双 CPU 运作	单 CPU 运作	单 CPU 运作
sim_seconds 模拟时长（s）	0.001324	0.002448	0.002448
sim_ticks 模拟时钟数	1.32E+09	2448244638	2448244638
host_inst_rate 指令速率(inst/s)	1009502	1169714	1251856
host_op_rate 操作速率(op/s)	1722748	1994576	2134657
sim_insts 执行指令数	3465332	3444780	3444780
host_mem_usage 内存使用(bytes)	3370632	3365344	3357152
cpu1.Branches CPU1 的分支数	154020	251877	251877
cpu1.committedInsts CPU1 提交的指令数	1854764	3444780	3444780
cpu1.numCycles CPU1 周期数	3975492	7352127	7352127
cpu2.Branches CPU2 的分支数	102701	0	
cpu2.committedInsts CPU2 提交的指令数	1610568	0	
cpu2.numCycles CPU2 周期数	3975411	0	

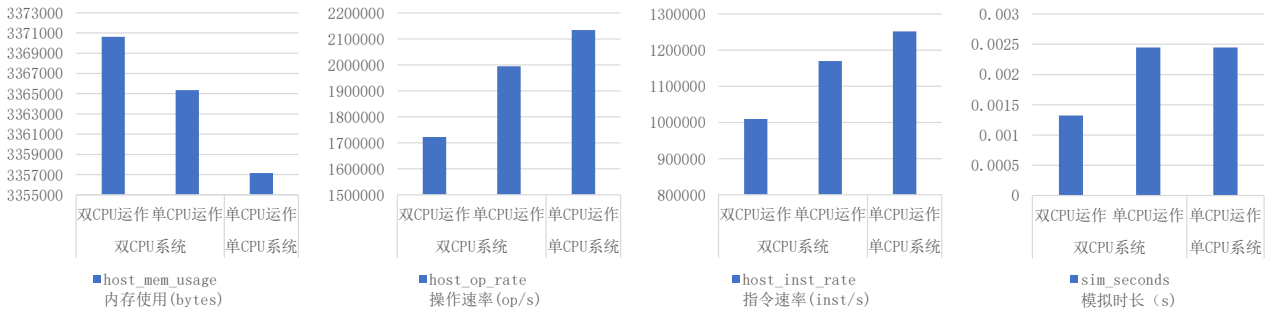

图 14 不同 CPU 数量计算机系统和不同数量 CPU 运作下的系统速率比较

FS 模式下的多核创建需要先修改 x86.py 配置，具体的修改部分见图 15 和图 16 。即 cpu 的增加需要扩张 apicbridge 的地址范围，local_apic 和 io_apic 的配置。而执行脚本 run.py 和系统配置文件 system.py 的编写可以参照 SE 模式下的多 CPU 配置过程。在命令行输入“build/X86/gem5.opt --outdir=m5out/bpnnrst2 configs/full_system/run.py --script=tests/test-progs/runscript/test.tcS”，以 run.py 作为脚本开启 FS 模式后，进入系统，执行 BP 算法的程序，分析输出的 config.ini 文件，如图 17 所示，可以看到除了与 SE 模式下类似的 CPU1 和 CPU2 配置之外，还增加了 APIC 桥、IO APIC、LOCAL APIC 组件，作为全系统仿真的补充部分，系统的类型也由 SE 模式下的 System 变为了 LinuxX86System。

```
system.apicbridge.ranges = [AddrRange(interrupts_address_space_base,
                                      interrupts_address_space_base +
                                      cpus * APIC_range_size
                                      - 1)]
```

图 15 apicbridge 地址范围配置

```
for i in range(cpus):
    bp = X86IntelMPProcessor(
            local_apic_id = i,
            local_apic_version = 0x14,
            enable = True,
            bootstrap = (i ==0))
    base_entries.append(bp)
io_apic = X86IntelMPIOAPIC(
        id = cpus,
        version = 0x11,
        enable = True,
        address = 0xfec00000)
```

图 16 Local apic 和 io apic 的配置

1 cortex （https://baike.baidu.com/item/cortex/4476976?fr=aladdin）

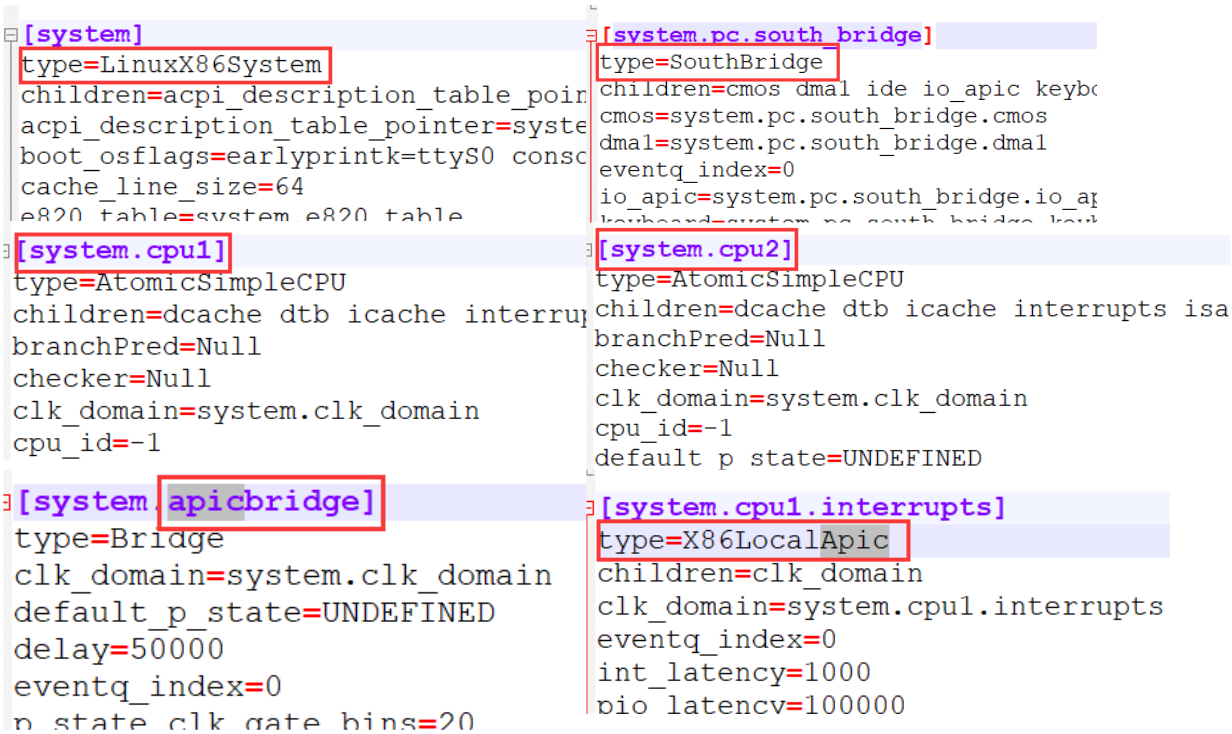

图 17 FS 下多 CPU 系统的输出文件 config.ini

4.2.3 SE 模式与 FS 模式处理多线程能力测试

在 SE 模式和 FS 模式中分别运行多线程程序 test/test-progs/srccode/multithread，来检验两种模式下对于线程处理的能力。

SE 模式下，在命令行输入：“build/X86/gem5.opt --outdir=m5out/semultithread configs/tutorial/mtc.py”，执行结果如图 18 所示。结果表明在 SE 模式下即使使用两个 CPU 的计算机系统，仍然无法成功创建线程 2。

FS 模式下，运行 run.py 脚本，进入系统，执行结果如图 18 所示。结果表明线程 2 成功被创建，且可以调用两个线程对同一个数进行叠加。证明了 SE 对于多线程程序的局限性，体现了 FS 模式模拟具有更高的准确性和可靠性。因为在完整系统模式下，GEM5 模拟从 CPU 到 I / O 设备的所有硬件，这使 GEM5 无需修改即可执行二进制文件。比较两种模式的输出文件 stats.txt（表 7），发现 FS 模式下的指令速率比 SE 模式高 62.68%，操作速率比 SE 模式高 49.81%。然而 FS 模式所花费模拟的时长远大于 SE 模式，所执行的指令数量也远大于 SE 模式，这也导致了在 FS 模式下执行程序需要等待大量时间，实验效率较低。

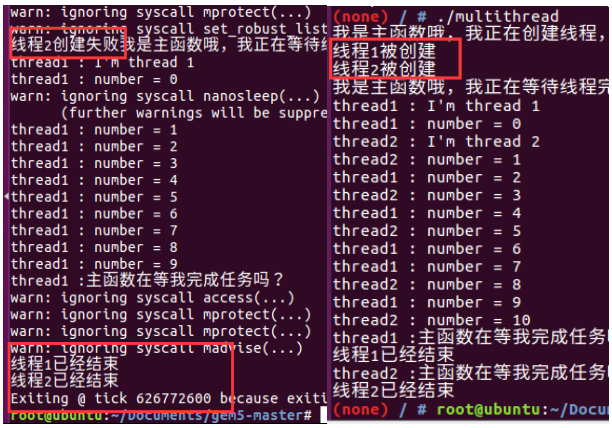

图 18 SE 模式下多线程程序运行结果（左）和 FS 模式下多线程程序运行结果（右）

1 一文看懂arm架构和x86架构有什么区别 （https://zhuanlan.zhihu.com/p/95028674 2019-12-03）

2 CPU Basics: Multiple CPUs, Cores, and Hyper-Threading Explained （https://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/ 2018-10-12）

3 https://github.com/EWouters/Using-GEM5-for-Architecture-Exploration-in-Multi-Core-Processors 2018-4-30

4 1_ARM Cortex-A 系列处理器(A5、A7、A8、A9、A15)区别对比 （https://wenku.baidu.com/view/332a395487c24028905fc31b.html 2014-12-23）

表 7 SE 与 FS 模式下计算机性能比较

	SE	FS
host_inst_rate 指令速率(inst/s)	482338	784655
host_op_rate 操作速率(op/s)	933930	1399114
sim_seconds 模拟时长（s）	0.00063	20.338552
sim_insts 执行指令数	245720	213307133

4.3 ARM架构

4.3.1 测试文件交叉编译

交叉编译的目的是在一种平台上编译出能运行在体系结构不同的另一种平台上的程。x86 与 ARM 的指令集彼此不兼容，所以在编译时要使用针对 ARM 平台的 arm-linux-gcc 编译器，以此编译出能够在 ARM 上运行的程序。编译的命令为“arm-linux-gcc 文件名.c -o 文件名 -static”。在此需注意，如果不使用“-static”，在执行程序时会显示“Failed to open file /lib/ld-linux.so.3.”的动态编译库缺失问题，所以应当在编译时指明为静态编译。

4.3.2 CORTEX A9 型处理器和 CORTEX A15 型处理器的创建及测试

CORTEX A9 处理器搭建如图 19 和图 20 所示，CPU 的类型是 O3CPU，需要设定 2 路超标量和功能单元。

```
cpu = system.cpu0
cpu.decodeWidth = 2
cpu.fetchWidth = 2
cpu.issueWidth = 2
cpu.dispatchWidth = 2
```

图 19 设定 CORTEX-A9 超标量

```
class IntALU(FUDesc):
    opList = [ OpDesc(opClass='IntAlu') ]
    count = 1

class IntMultDiv(FUDesc):
    opList = [ OpDesc(opClass='IntMult', opLat=3),
               OpDesc(opClass='IntDiv', opLat=20) ]
    count = 1

class RdWrPort(FUDesc):
    opList = [ OpDesc(opClass='MemRead'), OpDesc
(opClass='MemWrite') ]
    count = 1

# Attach the Functional units
cpu.fuPool = FUPool(FUList=[IntALU(), IntMultDiv(),
RdWrPort()])
```

图 20 设定 CORTEX-A9 功能单元

同理搭建 CORTEX-A15 处理器，CPU 类型是 O3CPU，需要设定 3 路超标量和功能单元。如图 21 和图 22 所示。

```
cpu = system.cpu0
cpu.decodeWidth = 3
cpu.fetchWidth = 3
cpu.issueWidth = 3
cpu.dispatchWidth = 3
```

图 21 设定 CORTEX-A15 超标量

```
class IntALU(FUDesc):
    opList = [ OpDesc(opClass='IntAlu') ]
    count = 2

class IntMultDiv(FUDesc):
    opList = [ OpDesc(opClass='IntMult', opLat=3),
               OpDesc(opClass='IntDiv', opLat=20) ]
    count = 1

class RdWrPort(FUDesc):
    opList = [ OpDesc(opClass='MemRead'), OpDesc
(opClass='MemWrite') ]
    count = 1

# Attach the Functional units
cpu.fuPool = FUPool(FUList=[IntALU(), IntMultDiv(),
RdWrPort()])
```

图 22 设定 CORTEX-A15 功能单元

用这两种 CPU 分别组成二级 Cache 计算机系统，并分别测试同一段程序，其输出文件 stats.txt 如表 8 所示。相应的柱形图如图 23 所示。在计算机指令速率（host_inst_rate）方面，CORTEX-A15 比 CORTEX-A9 高了 21.77%。在指令平均时钟周期数（CPI）方面，CORTEX-A15 比 CORTEX-A9 小了 39.45%。在总能耗（totalEnergy）方面，CORTEX-A15 比 CORTEX-A9 小了 40.29%。因此可以说明 CORTEX-A15 比 CORTEX-A9 速度更快，能耗更小，综合性能更高。

表 8 A9 型和 A15 型 CPU 运行程序能力对比

	CORTEX-A9	CORTEX-A15
sim_seconds 模拟时长（s）	0.042945	0.025639
host_inst_rate 指令速率(inst/s)	259048	315444
host_op_rate 操作速率(op/s)	273223	332705
cpi_total 指令平均时钟周期数	0.978933	0.584439
totalEnergy(mJ) 总能耗	19982432205	11931183510

1 cortex （https://baike.baidu.com/item/cortex/4476976?fr=aladdin）

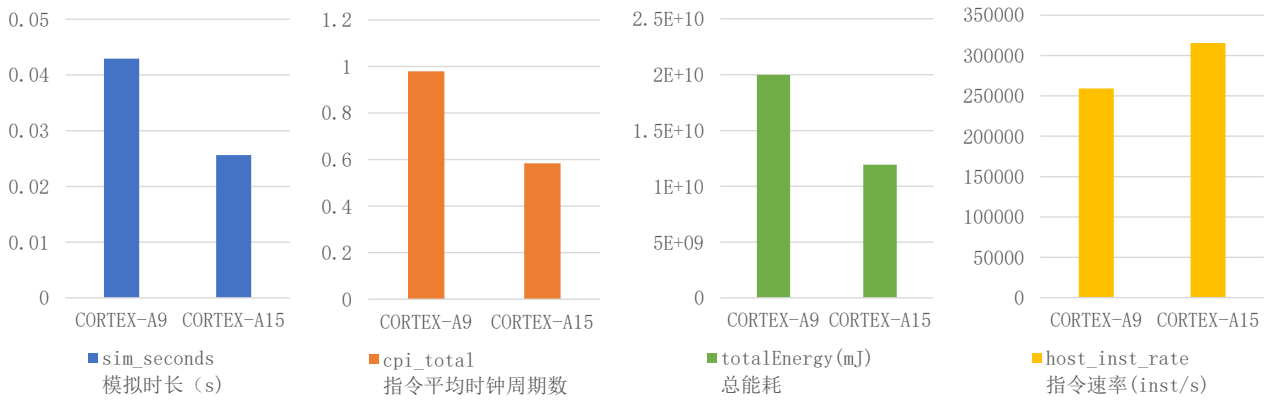

图 23 A9 型和 A15 型 CPU 运行程序能力对比

4.4 ARM架构与X86架构对比测试

选用双 03CPU 三级 cache 的计算机架构作为实验系统，分别运行四种测试程序，其中 BP 算法过程复杂，指令数最多，运算量也最大。圆周率迭代算法、斐波那契数列和 helloworld 的运算量依次递减。整理输出文件 stats.txt 中的数据，总结如表 9 和图 24 所示。可以发现，ARM 架构下的计算机运行速度普遍大于 x86 架构下的计算机运行速度。然而，当程序中计算量较大时，ARM 架构的指令速率就没有优势了。并且，从能耗的角度来看，ARM 架构计算机执行 BP 算法和圆周率迭代算法的功耗远大于 x86 架构的计算机。只有在执行计算量较小的两个测试程序时，ARM 架构的功耗才小于 x86 架构。出现这种现象的原因可能与 ARM 采用精简指令集,X86 采用复杂指令集有关。虽然前者每条功能简单,单条指令耗电低；而后者每条指令复杂,单条指令耗电高，但是如果完成同一个计算量大的功能,精简指令集需要指令较多,而复杂指令集需要指令少,可能导致 ARM 架构的能耗反而大于 x86 架构的能耗。

表 9 x86 和 ARM 架构在不同测试程序中的表现

	BP 算法		求圆周率迭代算法		斐波那契数列		helloworld	
	x86	ARM	x86	ARM	x86	ARM	x86	ARM
sim_seconds 模拟时长（s）	0.00869	0.074075	0.001029	0.012582	0.000195	0.000023	0.00002	1.6E-05
host_inst_rate 指令速率 (inst/s)	217334	211745	148994	342473	71554	134094	51248	63784
host_op_rate 操作速率(op/s)	299075	234134	278473	361213	134363	150840	92413	73488
cpi_total 指令平均时钟周期数	0.415877	0.601639	1.266207	0.430635	3.612919	2.490744	10.44686	10.4855
totalEnergy 总能耗	4.078E+09	3.4495E+10	504289470	5856907380	115693665	13260885	12484500	9832200

1 一文看懂arm架构和x86架构有什么区别 （https://zhuanlan.zhihu.com/p/95028674 2019-12-03）

2 CPU Basics: Multiple CPUs, Cores, and Hyper-Threading Explained （https://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/ 2018-10-12）

3 https://github.com/EWouters/Using-GEM5-for-Architecture-Exploration-in-Multi-Core-Processors 2018-4-30

4 1_ARM Cortex-A 系列处理器(A5、A7、A8、A9、A15)区别对比 （https://wenku.baidu.com/view/332a395487c24028905fc31b.html 2014-12-23）

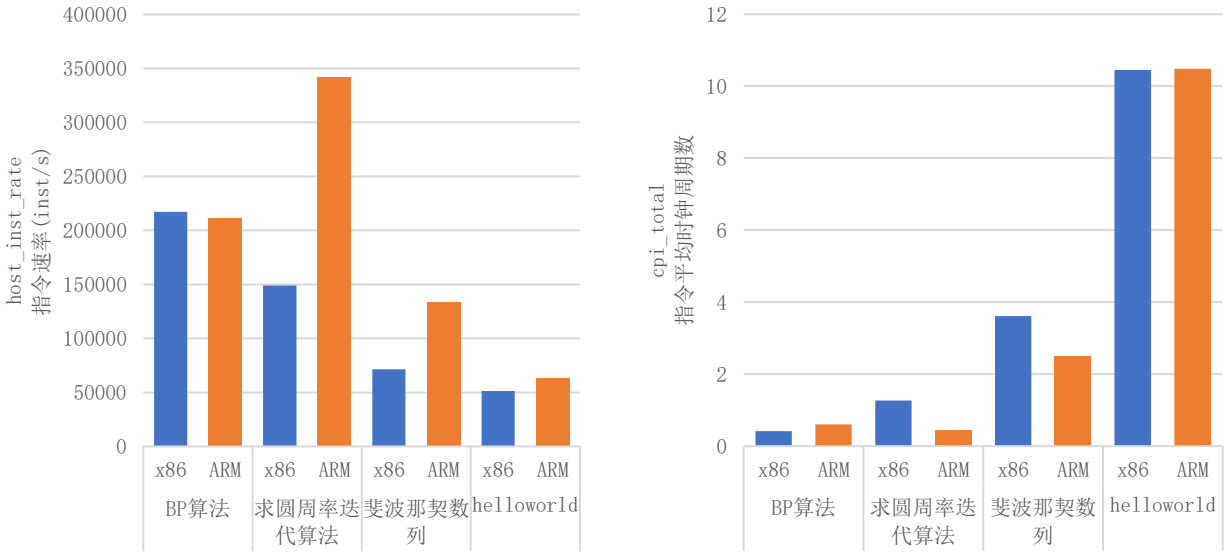

图 24 x86 和 ARM 架构在不同测试程序中的指令速率（左）和 CPI（右）

5 结论

本文搭建了 x86 架构和 ARM 架构下的计算机系统，实现了多 CPU 系统，多级 Cache 系统，Cortex-A9 处理器，Cortex-A15 处理器的仿真模拟。比较了 x86 和 ARM 架构的计算机的性能差异。

通过对比实验和对输出文件 stats.txt 的分析，发现了增加 L1Cache 对于计算机性能的提升有显著作用，使其运算速率提高了 67.92 倍。而在该层次上添加 L2Cache 和 L3Cache 则只对性能起到了微小提升甚至反而降低的效果。而增加 CPU 可以对并行化程序运行的耗时缩短 45.96%。因此，如果不考虑其他的硬件开销，在计算机系统中增加 CPU 的数量或者适量增加 Cache 可以有效提高计算机执行程序的效率，节省时间。

GEM5 作为模拟器，其提供的 SE 模式和 FS 模式在启动、耗时、运算速率、准确性、线程支持度方面各有优劣。FS 模式模拟了 I/O、操作系统、中断控制等组件，模拟的计算机系统更接近真实的计算机水平。因此在强调结果准确性和性能完备性的标准下，应该选择 FS 模式作为模拟环境，但若追求快速模拟，可以选择 SE 模式。实验中搭建的 Cortex-A9 处理器，Cortex-A15 处理器作为 ARM 架构下常用的处理器类型，拥有多路超标量，其执行效率相比一般的指令单发射更高，而拥有两个整数算术逻辑单元的 Cortex-A15 比 Cortex-A9 性能表现更好。因此，使用超标量技术和增加运算逻辑单元可以有效提升计算机性能。

X86 和 ARM 在指令集上的差异影响了两者的性能在不同的执行环境下的表现。在本文中，用乱序 O3CPU 作为中央处理器，发现 x86 和 ARM 在执行不同计算量的程序中所表现的性能差异规律性难以把握：在计算量小的执行程序中，大多数衡量指标显示 ARM 架构的指令速率和功耗优于 ARM 架构；然而在计算量大的执行程序中，大多数衡量指标显示 x86 架构的性能和功耗反而优于 ARM 架构。

参 考 文 献

[1]. GuoYi, LiuWenzhi . Comparison of differences in assembly language : from X86 to ARM. Chinese Journal of Computers, 2019, 22(9): 39-42 (in Chinese)
(过怡，刘文芝. 汇编语言差异比较——从X86 到ARM. 江苏科技信息, 2019, 22(9): 39-42)

[2]. Dagum, Leonardo, and Ramesh Menon. ”OpenMP: an industry standard API for shared-memory programming.” Computational Science & Engineering, IEEE 5.1 (1998): 46-55

[3]. N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish,

1 cortex （https://baike.baidu.com/item/cortex/4476976?fr=aladdin）

M. D. Hill and D. A. Wood, "The gem5 simulator," in ACM SIGARCH, 2011

[4]. S. Palacharla, N. P. Jouppi and J. E. Smith, "Complexity-Effective Superscalar Processors," Conference Proceedings. The 24th Annual International Symposium on Computer Architecture, Denver, Colorado, USA, 1997, pp. 206-218, doi: 10.1145/264107.264201.

[5]. Rumelhart, D.E. (1), R.J. (1) Williams, and G.E. (2) Hinton. 2020. "Learning Representations by Back-Propagating Errors." Nature 323 (6088): 533–36. Accessed July 3. doi:10.1038/323533a0.

1 一文看懂arm架构和x86架构有什么区别 （https://zhuanlan.zhihu.com/p/95028674 2019-12-03）

2 CPU Basics: Multiple CPUs, Cores, and Hyper-Threading Explained （https://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/ 2018-10-12）

3 https://github.com/EWouters/Using-GEM5-for-Architecture-Exploration-in-Multi-Core-Processors 2018-4-30

4 1_ARM Cortex-A 系列处理器(A5、A7、A8、A9、A15)区别对比 （https://wenku.baidu.com/view/332a395487c24028905fc31b.html 2014-12-23）