基于标签化RISC-V架构的进程共享资源管理系统 ——操作系统课程设计 最终报告

清华大学 电子工程系 尤予阳

目录

- 基于标签化RISC-V架构的进程共享资源管理系统 ——操作系统课程设计 最终报告
  - 目录
  - 背景和已有工作
  - 设计目标
  - 设计方案
    - 硬件
    - 固件：lrv-rust-bl
    - 操作系统：标签化 uCore-SMP
      - 内存布局
      - 时钟周期
      - 页表项属性
      - 硬盘外设
      - 标签系统
      - 与内核共同调试和 debug
        - 调度器初值
        - 子进程回收内存泄露
        - syscall 中的悬垂指针
    - 应用程序
      - sort
      - prime
      - jammer
      - monitor
      - dsid_demo
  - 使用方法
  - 成果展示
  - 后续优化方向
  - 致谢
  - 参考

背景和已有工作

为了提高使用效率，计算机系统中的很多资源往往由多个使用者共享，例如 CPU 、最后一级缓存（LLC）、内存、网络等。其中一些资源由操作系统进行管理，如 CPU 时间由任务调度器进行分配，内存空间通过页映射、地址空间等方法进行管理；另一些资源则对操作系统和应用“透明”，而由底层硬件的控制器进行管理，如 LLC 、内存总线带宽等。这种透明的设计简化了硬件抽象，降低了软件设计复杂度，但在出现资源争用时，也导致软件性能出现难以预见和管理的波动，从而降低应用程序的服务质量和用户体验。

目前已有若干工作旨在降低资源争用的影响。Linux 内核中的 cgroups 机制能够以进程组为单位，统计和限制其使用的内存空间、 CPU 占用率、硬盘和网络吞吐率等[1]；英特尔的资源调配技术能够在其数据中心中实现缓存和内存监控及分配[2]；Arm 在 Armv8-A 中引入了内存资源分区与监控（MPAM）扩展[3]。

中科院计算所包云岗老师带领的团体提出了标签化 RISC-V 架构[4]，通过标签（DiffServ ID，DSID）使得上层的应用能够向硬件传递资源需求、服务质量、安全性等语义。标签跟随资源请求在系统中传播，操作系统可以配置底层硬件控制器对于不同标签的请求的响应方式，从而实现对于非托管共享资源的调度，能够在出现资源争用时保证高优先级的应用服务质量。

清华大学计算机系的陶天骅同学在本次课程设计中，实现了 uCore-SMP[5] 系统，在 uCore 系统上增加了多核处理器支持，并添加了许多内核功能和特性，且整体代码较为精简，模块化程度高，便于移植适配和修改。本课程设计将主要在标签化 RISC-V 架构和 uCore-SMP 系统的基础上进行。

设计目标

本设计将 uCore-SMP 系统适配到标签化 RISC-V 架构的 FPGA 版本上，并基于二者在内核中实现调度进程所使用的 LLC 和内存带宽资源的功能，提供相应的接口和演示程序，对资源调度的效果进行展示。

设计方案

本设计的整体架构如下图所示。

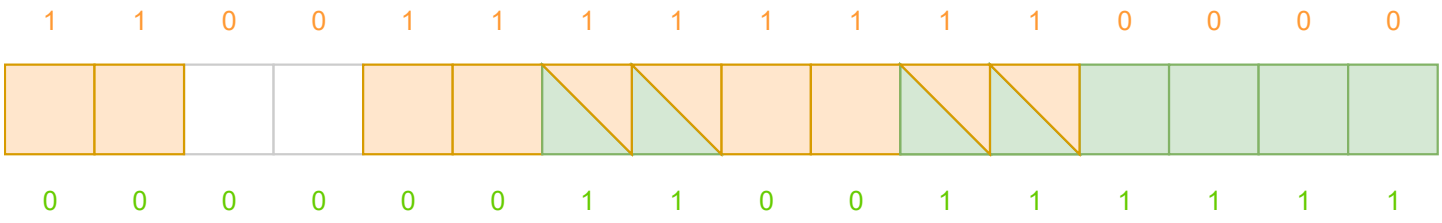
应用程序	dsid_demo, sort, jammer, prime, monitor
ABI	dsid syscall
操作系统	Labeled uCore-SMP
SBI	RustSBI
固件 / 启动器	lrv-rust-bl
硬件	Labeled RISC-V, ZCU102 FPGA

硬件

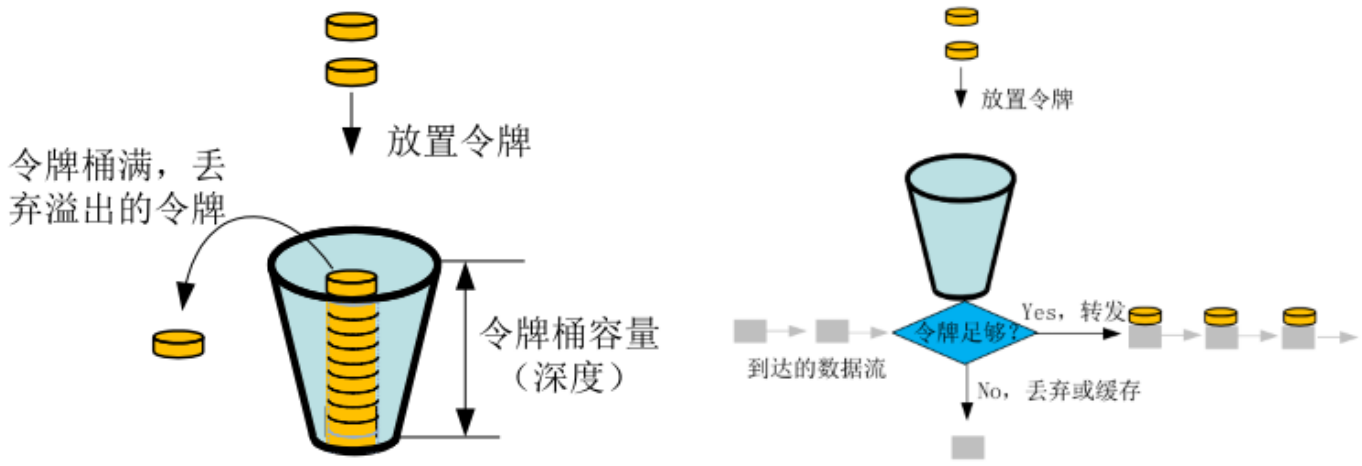
设计使用标签化 RISC-V 架构的 FPGA 版本[6]和赛灵思公司的 ZCU102 开发板[7]，搭载一颗 Zynq® UltraScale+™ XCZU9EG-2FFVB1156E MPSoC 。该芯片的处理系统（PS）部分具有四个 Arm A53 核心，运行 Linux 系统，主要功能为烧写 FPGA 比特流，辅助其复位和启动，并转发串口数据。可编程逻辑（PL）部分实现为四个 Rocket Core[8]，ISA 为 RV64IMAC，时钟频率 100MHz，具有 2MB 共享 L2 缓存和 2GB DRAM 。

标签化 RISC-V 架构在每个核心上添加了一个 CSR 记录当前核心上正在运行的程序的标签，并使用该标签标记程序的资源请求。该寄存器编号为 0x9C0 ，访问权限为 S 态可读写。标签控制平面以 MMIO 形式映射到 0x20000 地址，可以向其中写入标签的缓存和内存控制参数，并读取标签对应的 L1 到 L2 的流量。

L2 缓存被划分为 16 块，使用位掩码对标签对应进程可用的缓存区域进行控制，若某位为 1 ，则表示该进程可以使用该块缓存；反之则不能。若某块缓存在多个标签的掩码中均为 1 ，则该块缓存由这些标签对应的进程共享。示意图如下：



内存带宽使用令牌桶算法进行控制。每个标签有一个令牌桶，控制器每隔 `freq` 个时钟周期会向桶中放入 `inc` 个令牌，桶的容量为 `size` ，当令牌数量达到桶容量时，令牌数量不再增加。一个令牌对应 64 比特的数据量，进程每次发起一次内存读写请求，会消耗标签令牌桶中相应数据量的令牌，当桶中没有令牌时，请求会被挂起。当进程持续访问内存时，其最大内存带宽近似为 $inc \cdot \frac{clock}{freq} \cdot 64\,\mathrm{bps}$ 。令牌桶算法示意图[9]如下：



固件：lrv-rust-bl

该部分基于 RustSBI 的 QEMU 参考实现[10]进行开发，使用 Rust 语言编写。PL 所用串口外设为 UART Lite[11]，与 QEMU 中所用的 NS16550A 不同，需要重新实现其初始化和收发代码。

该 RISC-V 核心不支持非对齐 load/store 指令，此类指令会触发相应的非对齐异常，但同时内核和应用程序中又无法完全避免这些指令，为了实现二者的兼容，固件在相应的异常处理例程中解码触发异常的指令，根据其选择的寄存器和非对齐地址偏移量，使用两次对齐的 load/store 进行模拟。需要注意的是，该平台同时支持压缩指令（C）扩展，因而指令长度可能为 2 或 4 字节，指令本身可能处于非对齐地址上，且 load 和 store 各有三种（常规指令、基于通用寄存器中地址的压缩指令、基于栈指针寄存器中地址的压缩指令），指令格式不尽相同，解码过程较为繁琐。

该启动器代码仓库位于 https://github.com/Gallium70/lrv-rust-bl ，对该启动器更详细的介绍见代码仓库中相应的文档。

操作系统：标签化 uCore-SMP

uCore-SMP 在设计时目标平台为 QEMU ，将其移植到 FPGA 平台主要需要解决如下问题：内存布局、时钟周期、页表项属性和硬盘外设。移植完成后，还需添加对标签系统的支持。代码仓库位于 https://github.com/TianhuaTao/uCore-SMP/tree/label-riscv 。

内存布局

QEMU 中内存的起始地址为 0x80000000，而 FPGA 平台起始地址为 0x100000000，相应的内核启动地址也要调整到 0x100200000，这两个值需要在 `memory_layout.h` 和 `kernelld.py` 中进行修改。QEMU 中配置总内存空间为 128MB，而尽管 FPGA 平台有 2GB DRAM，由于内存带宽有限，uCore-SMP 在启动时又会将所有内存地址遍历并初始化。为了减少操作系统的启动等待时间，综合考虑内核和用户程序所需的内存空间，最终将可分配的内存限制在 32MB 。

时钟周期

QEMU 中 CPU 运行频率（即 `cycle` 寄存器的增加频率）为 3GHz，而实时时钟（RTC）频率（即 `time` 寄存器的增加频率）为 12.5MHz； FPGA 平台上 CPU 频率为 100MHz，RTC 频率为 10MHz，需要在 `timer.h` 中修改。

页表项属性

根据 RISC-V 规范中虚拟内存章节[12]的描述，页表项中有“已访问（A）”和“已修改（D）”两个标志位，对这两个标志位，规范允许两种处理方式：

> - *When a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, a page-fault exception is raised.*
> - *When a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, the implementation sets the corresponding bit(s) in the PTE. The PTE update must be atomic with respect to other accesses to the PTE, and must atomically check that the PTE is valid and grants sufficient permissions. The PTE update must be exact (i.e., not speculative), and observed in program order by the local hart. Furthermore, the PTE update must appear in the global memory order no later than the explicit memory access, or any subsequent explicit memory access to that virtual page by the local hart. The ordering on loads and stores provided by FENCE instructions and the acquire/release bits on atomic instructions also orders the PTE updates associated with those loads and stores as observed by remote harts.*

即直接触发页错误异常，或由硬件实现直接对页表项置位。FPGA 平台实现为前一种，而 QEMU 为后一种。这两个标志位主要应用于虚拟存储系统中的页面置换，而 uCore-SMP 中对此并无支持，简单起见，可修改 `virtual.c` ，在分配新的页表项并映射物理页时，直接将 A 和 D 置位。

硬盘外设

QEMU 中通过 virtio 提供了虚拟块存储设备，FPGA 平台中没有。在此情况下，为了仍然能够使用文件系统，陶天骅同学提供了虚拟磁盘接口和 ramdisk 功能，通过宏开关在编译期进行切换。系统初始化时， ramdisk 将在内存中开辟一块区域，并将其按照 nfs 格式进行初始化。

标签系统

在内核初始化时，将标签系统控制平面基址以恒等映射挂载到 0x20000 地址处，权限为内核可读写。读取和写入参数的方式为基址+偏移，偏移量单位为 32bit ，即需要将地址强制转换为指向 32 位无符号整数的指针类型。具体各参数的偏移量和写入方法见 `dsid.c/h` 。

目前该系统支持至多 8 个不同的进程标签。进程标签存储于进程控制块中，在程序上下文切换时，内核将标签写入相应的 CSR 。内核的进程标签硬编码为 0 。

内核提供三个和标签相关的系统调用：

```
int set_dsid(int pid, uint32 dsid);
int set_dsid_param(uint32 dsid, uint32 freq, uint32 size, uint32 inc, uint32 mask);
uint32 get_l2_traffic(uint32 dsid);
```

- `set_dsid` 将 `pid` 对应进程的标签设为 `dsid` ，若进程不存在，则返回 -1 ，否则返回 0 ；
- `set_dsid_param` 设置 `dsid` 标签相应的令牌桶参数和缓存掩码，参数含义见“硬件”章节。只有非零的参数会被写入控制平面。
- `get_l2_traffic` 读取 `dsid` 标签对应的 L1 和 L2 缓存之间的数据流量，单位为 64bit 。

与内核共同调试和 debug

标签系统开发过程中，主要查出内核中以下几个缺陷。

调度器初值

uCore-SMP 使用 stride 算法进行调度，在初始实现中，fork 产生的子进程的 stride 会置为 0 。在使用标签系统监测负载程序的缓存性能时，设置父进程每隔一秒输出一次信息。第一批负载表现正常，但启动第二波负载时，会出现直到负载运行结束父进程才会输出的情况。分析后发现，由于子进程 stride 为 0 ，而运行第二波负载时，父进程已有一定的 stride ，此时调度器会持续优先调度子进程，导致父进程陷入“饥饿”。解决方法为，将子进程的 stride 置为和父进程相同。

子进程回收内存泄露

初始实现中，若父进程没有使用 wait/waitpid 等待子进程结束，则子进程占用的内存和文件等资源在进程结束后不会被回收，当大量 fork 时会耗尽系统内存。为此，陶天骅同学在内核中加入了 reparent 机制，当进程退出时会检查其子进程，若子进程已退出则回收，否则将其父进程置为 NULL ； 若退出的进程父进程为 NULL ，则立即回收自身占用的资源，这样解决了这一内存泄露问题。

syscall 中的悬垂指针

系统调用进入内核中 syscall 函数时，会保存进程的 trapframe 地址，在具体系统调用执行完成后，使用该地址中的 a0 寄存器保存系统调用的返回值。在运行标签负载时发现，当用户程序运行到 0x1070 地

址（用户态下的虚拟地址）时会触发非法指令异常，stval 寄存器显示相应的指令值为 0，但查看相应程序的反汇编发现此处并非 0。起初怀疑是固件中非对齐 load/store 模拟存在问题，导致加载用户程序时出现数据异常；尽管发现固件确实存在问题，但解决该问题后依然会触发非法指令异常。

后来使用内核中 pushtrace 工具在各处记录 0x1070 对应内存地址的数值发现，当执行 execv 系统调用时，会回收原有的 trapframe 内存页，该页可能被后续用于加载用户程序的代码段，程序分配的新 trapframe 的 a0 和 a1 寄存器会分别用于存储 `argc` 和 `argv`；但 execv 执行完成返回 syscall 函数中时（其返回值总为 0），仍会使用旧的 trapframe 地址，而 a0 寄存器对应变量在 trapframe 中相应的偏移量即为 0x70，向其中写入 0 会破坏用户程序代码，导致错误。解决方法为，当系统调用为 execv 时，在 syscall 函数中不写入 `trapframe->a0`。

应用程序

为了使用标签系统，测量和展示标签系统的控制效果，设计了以下几个用户态应用程序：sort、jammer、prime、monitor、dsid_demo。

sort

对 501202 个 32 位无符号整数，使用基数排序法进行排序，基为 65536，数据使用随机数生成器和固定的种子生成，并使用 sharedmem 作为动态申请内存的方式。每个进程会将自己的 pid 加入共享内存名字中，以确保多个负载不会申请到同一块共享内存。程序将自身的运行时间作为返回值，单位是毫秒。

这是一个内存密集型负载，消耗约 4.5MB 内存空间和 8MB/s 的内存带宽。

prime

计算 2 到 52021 之间质数的个数。使用朴素算法，对于每个数 x，枚举 2 到 x/2 之间的数，判断是否能够整除 x，若能，则 x 不是质数。这是一个计算密集型负载，可以用于验证多核正常运行，且运行时间几乎不受内存带宽限制的影响。

jammer

这是一个干扰负载，在一个有 524288 个 32 位无符号整数的数组上随机进行读写，消耗内存带宽。申请内存的方式与 sort 程序相同，消耗约 2MB 内存空间和 8MB/s 的内存带宽。

monitor

一个监测系统资源使用状况的程序，监测的资源包括 CPU 使用率、可用内存空间、各进程的运行时间、状态和内存占用、各标签的 L1 和 L2 之间的流量。每隔一秒输出一次，运行十秒后退出。其中

CPU 、内存和进程使用设备文件抽象接口，标签流量使用 `get_l2_traffic` 系统调用。运行界面如下：

```
--------------------------------------------------------------------------
 uCore-SMP Resource Monitor                              Time: 4 s
--------------------------------------------------------------------------

CPU
[||||||||||||||||||||||||||||||                    ]  Core  0    59%
[||||||||||||||||||||||||||||||||||||               ]  Core  1    73%
[||||||||||||||||||||||||||||||||||||||||||||||||    ]  Core  2    87%
[||||||||||||||||||||||||||||                      ]  Core  3    57%

Memory
Total :   32 MB        Free  :   17 MB
[|||||||||||||||||||||||                           ] 45%

Process | pid | ppid | dsid | heap | mem | cpu time | state
shell     1     -1     0      0      12288 172753     SLEEPING
dsid_demo 8     1      0      0      12288 140        SLEEPING
sort      9     8      1      0      12288 2953       RUNNING
jammer    10    8      2      0      8192  2900       RUNNING
monitor   11    8      0      0      12288 2339       RUNNING


dsid | L1-L2 traffic (KB/s)
0      6491
1      4496
2      4383
```

dsid_demo

本程序中内置了若干组令牌桶和缓存掩码参数用于分配给指定的负载，并执行 `monitor` 程序监测系统资源的使用情况。

使用方法为 `dsid_demo app1 param1 [app2 param2 [app3 param3]]` ，最多同时启动 3 个负载。例如， `dsid_demo sort 2 prime 1` 将为 `sort` 进程分配 2 号参数组，为 `prime` 进程分配 1 号参数组。由于可用的标签数量有限，同时为了方便监测起见，标签并不与参数组一一对应，而是为每个负载分配不同的标签， `appX` 的标签编号即为 X 。在上例中， `sort` 进程标签为 1 ，而 `prime` 进程标签为 2 。

各组参数具体的设计，是为了实现以下四种测试场景：

- 不调控
- 内存带宽 5M：5M，缓存 768K：768K
- 内存带宽 7M：2M，缓存 1280K：256K
- 内存带宽 8M：1M，缓存 1280K：256K

内置参数组如下（其中 size 参数值均为 0x800）：

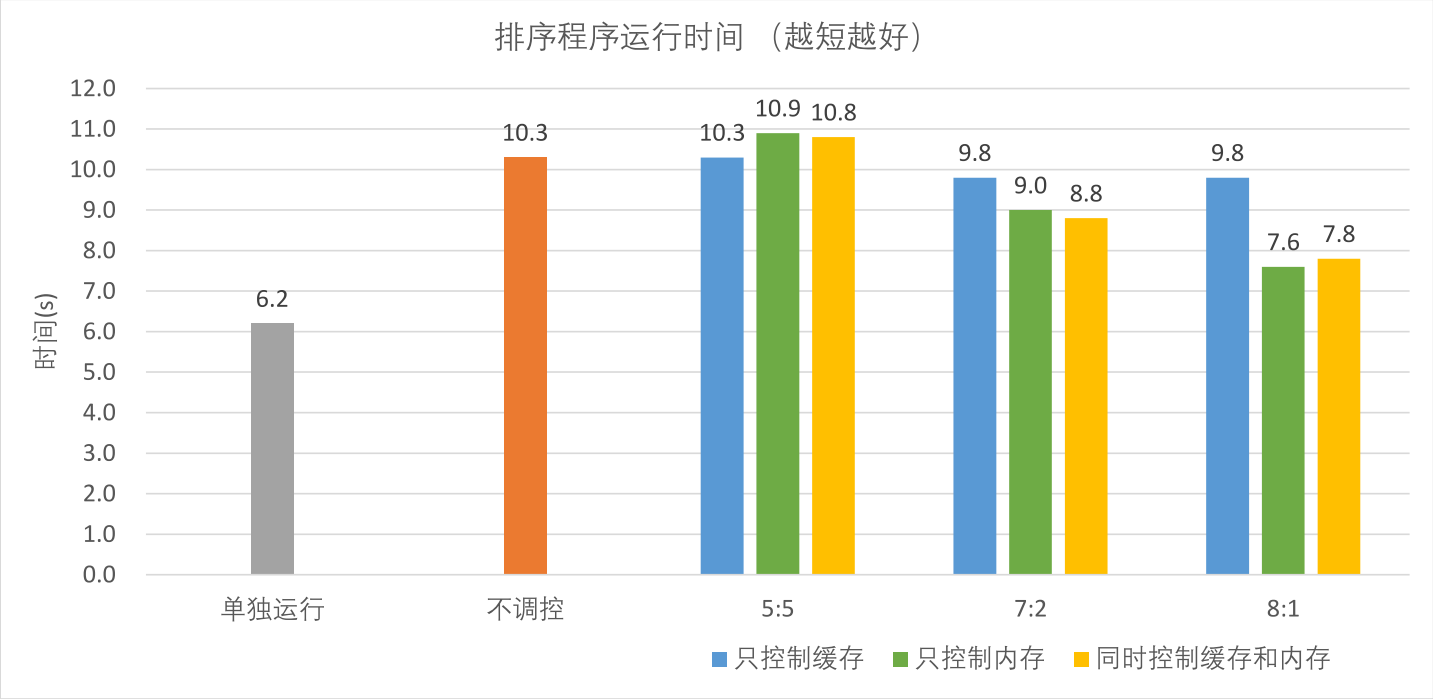
编号	freq	inc	cache mask	备注
0	10000	40	0x000F	用于内核进程和监测程序，内存带宽约 3MB/s，缓存 512K，与其他参数组隔离
1	100	100	0xFFF0	内存带宽不限，缓存 1536K
2	7800	80	0xFFC0	内存带宽约 8MB/s，缓存 1280K
3	7800	10	0x0030	内存带宽约 1MB/s，缓存 256K
4	7800	80	0xFFF0	内存带宽约 8MB/s，缓存 1536K
5	7800	10	0xFFF0	内存带宽约 1MB/s，缓存 1536K
6	100	100	0xFFC0	内存带宽不限，缓存 1280K
7	100	100	0x0030	内存带宽不限，缓存 256K
8	9300	60	0x03F0	内存带宽约 5MB/s，缓存 768K
9	9300	60	0xFC00	内存带宽约 5MB/s，缓存 768K
10	9300	60	0xFFF0	内存带宽约 5MB/s，缓存 1536K
11	100	100	0x03F0	内存带宽不限，缓存 768K
12	100	100	0xFC00	内存带宽不限，缓存 768K
13	7800	70	0xFFC0	内存带宽约 7MB/s，缓存 1280K
14	7800	20	0x0030	内存带宽约 2MB/s，缓存 256K
15	7800	70	0xFFF0	内存带宽约 7MB/s，缓存 1536K
16	7800	20	0xFFF0	内存带宽约 2MB/s，缓存 1536K

使用方法

1. 根据标签化 RISC-V 代码仓库中的文档[4:1]配置 FPGA 平台，笔者在该过程中遇到的若干问题及解决方案见《在 FPGA 上复现标签化 RISC-V 架构——踩坑指南》[13]；
2. 根据 lrv-rust-bl 代码仓库[14]中的文档，编译得到 lrv-rust-bl.bin；
3. 根据标签化 uCore-SMP 代码仓库[15]中的文档，编译得到 ucore-smp.bin；
4. 将 lrv-rust-bl.bin 和 ucore-smp.bin 复制到 PS 部分，按照 labeled-RISC-V-boot 仓库[16]中的文档，复位启动 RISC-V 部分，进入 uCore-SMP 。

成果展示

FPGA 平台内存带宽约 12MB/s，测试场景如 “dsid_demo”章节 中所述。测试结果如下：



可见在存在干扰情况下，给排序程序分配内存带宽越多，其性能越接近不受干扰时的情形，显示出内存带宽调控的效果；当内存带宽 7:2 分配时，控制缓存性能较优，而 8:1 分配时，不控制缓存缓存性能较优，这可能是因为缓存的分配比例介于二者之间，8:1 分配时，若不控制缓存，排序程序对缓存的利用率可以高于 7:2 时。

后续优化方向

1. 自动和闭环控制：将应用的服务质量需求描述自动转换为标签系统的控制参数，并不断监测调节，提高灵活性、准确性和鲁棒性；
2. 延迟控制：增加对应用服务延迟的测量和监测机制，展示标签系统控制延迟波动、提高用户体验的功能；
3. 适配更多资源和场景：如图形计算的算力和存储、分布式系统中跨结点的标签传递、嵌入式平台中的实时性保证；
4. 更多启动器功能：添加 Remote Fence、Hart 状态管理等 SBI 扩展，为操作系统提供更完善的支持。

致谢

感谢清华大学计算机系的向勇、陈渝老师在整个课程设计过程中的指导，建议笔者在计算所工作的基础上开展本设计并与陶天骅同学合作，提供硬件平台，并联系到张传奇博士提供帮助。

感谢中科院计算所的张传奇博士，从开发板上配置 SD 卡启动的拨码开关，到各类开发环境问题，再到标签系统的诸多细节，向笔者提供了“保姆级”的帮助，极大缩短了笔者在复现标签化 RISC-V 架构过程中花费的时间。

感谢清华大学计算机系的陶天骅同学，在笔者进入内核设计阶段时，他开发的 uCore-SMP 已经较为完善，让笔者基本能够开箱即用；为了解决 FPGA 平台没有 QEMU virtio 设备的问题，他连夜开发了 ramdisk 功能支持，确保文件系统能够继续正常运行；此外，还为笔者在内核中调试提供了诸多帮助。

感谢华中科大的蒋周奇、车春池同学，对启动器的架构和非对齐 load/store 模拟等功能提供的建议和帮助。

参考

1. cgroups(7) — Linux manual page ↩
2. 英特尔® 资源调配技术 (英特尔® RDT) ↩
3. Arm Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A ↩
4. Labeled RISC-V: A New Perspective on Software-Defined Architecture ↩ ↩
5. uCore-SMP GitHub 仓库 ↩
6. labeled-RISC-V GitHub 仓库 ↩
7. Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit ↩
8. Rocket Chip Generator ↩
9. 令牌桶工作原理 ↩
10. RustSBI GitHub 仓库 ↩
11. AXI UART Lite v2.0 LogiCORE IP Product Guide ↩
12. RISC-V Page-Based Virtual-Memory Systems ↩
13. 在 FPGA 上复现标签化 RISC-V 架构——踩坑指南 ↩
14. Labeled RISC-V Rust Bootloader GitHub 仓库 ↩
15. 标签化 uCore-SMP GitHub 仓库 ↩
16. labeled-RISC-V-boot GitHub 仓库 ↩