软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(5):2362−2380 [doi: 10.13328/j.cnki.jos.007192] [CSTR: 32375.14.jos.007192]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

SPARC 架构下低时延微内核进程间通信设计*

苏浩然, 李文泰, 古金宇, 臧斌宇, 陈海波, 管海兵

(上海交通大学 软件学院, 上海 200240)

通信作者: 古金宇, E-mail: gujinyu@sjtu.edu.cn

摘　要: 微内核系统将系统服务迁移到用户态运行, 因其架构隔离性而具有高可靠性的优势, 这一优势与航天领域的需求相契合. SPARC 架构的处理器被广泛应用于航天飞船、卫星载荷以及星球车的控制设备上, 而该架构的寄存器窗口机制会影响微内核进程间通信 (inter-process communication, IPC) 的性能, 其核间中断 (inter-processor interrupt, IPI) 也会严重影响跨核 IPC 的效率. IPC 作为微内核系统的关键机制, 对微内核上应用程序的整体性能十分重要. 基于对 SPARC 寄存器窗口机制的观察, 重新设计实现寄存器组机制, 由系统内核对寄存器窗口进行分配和管理, 并藉此实现 SPARC 架构上的 BankedIPC. 同时, 在多核场景下, 针对 SPARC 上 IPI 性能较差的问题, 设计实现 FlexIPC 以优化跨核 IPC 的性能. 使用这些方法对自研微内核 ChCore 上已经实现的通用的同步 IPC 进行优化. 测试表明, 优化后 SPARC 上微内核的 IPC 平均性能提升至原来的 2 倍, 应用性能提升最高可达 15%.

关键词: 进程间通信; 微内核; SPARC 架构; 性能调优

中图法分类号: TP316

中文引用格式: 苏浩然, 李文泰, 古金宇, 臧斌宇, 陈海波, 管海兵. SPARC架构下低时延微内核进程间通信设计. 软件学报, 2025, 36(5): 2362–2380. http://www.jos.org.cn/1000-9825/7192.htm

英文引用格式: Su HR, Li WT, Gu JY, Zang BY, Chen HB, Guan HB. Low-latency Microkernel IPC Design for SPARC Architecture. Ruan Jian Xue Bao/Journal of Software, 2025, 36(5): 2362–2380 (in Chinese). http://www.jos.org.cn/1000-9825/7192.htm

Low-latency Microkernel IPC Design for SPARC Architecture

SU Hao-Ran, LI Wen-Tai, GU Jin-Yu, ZANG Bin-Yu, CHEN Hai-Bo, GUAN Hai-Bing

(School of Software, Shanghai Jiao Tong University, Shanghai 200240, China)

Abstract: Microkernels migrate system services to user mode. Thanks to the isolated framework, microkernels are superior in high reliability, which meets the needs of the aerospace field. SPARC processors are widely applied on the control equipment of spacecraft, satellite payloads, and planetary vehicles. The register window mechanism of SPARC will affect the performance of inter-process communication (IPC) on microkernels. Besides, its inter-processor interrupt (IPI) also seriously affects the efficiency of cross-core IPC. As a key mechanism, IPC is vital to the overall performance of applications on microkernels. Through observing the register window mechanism, this study redesigns and implements the register bank mechanism, where the register window is allocated and managed by the kernel. Thus BankedIPC on SPARC is implemented. At the same time, as IPI underperforms on SPARC, FlexIPC is designed to optimize the performance of cross-core IPC. These approaches are employed to optimize the general synchronous IPC implemented on a self-developed microkernel ChCore. According to the test, the average IPC performance of microkernels on the optimized SPARC is about two times better with the application performance up to 15%.

Key words: inter-process communication (IPC); microkernel; SPARC; performance optimization

在传统的宏内核架构下, 与操作系统功能相关的所有实现都被统一放置在内核特权级下运行, 因此宏内核架构常常会带来可靠性和安全性问题, 比如当内核中的一个模块发生错误或者遭到攻击时, 就可能导致整个系统崩

* 基金项目: 国家自然科学基金青年基金 (62202292); 上海市科技创新行动计划 (22511101102)
收稿时间: 2023-04-03; 修改时间: 2024-01-18; 采用时间: 2024-03-28; jos 在线出版时间: 2024-06-14
CNKI 网络首发时间: 2024-06-17

溃或被攻击者完全掌控. 微内核操作系统将大部分系统服务和功能迁移到内核外部, 作为用户态应用来提供服务, 普通的用户态应用可以通过进程间通信使用这些系统服务, 内核中则仅保留内存管理、线程调度、进程间通信等最基本的功能, 以此提供了较强的可靠性与易演化性, 并在安全关键的场景中得到了成功应用[1−10].

微内核提供的可靠性与航天等对可靠性要求很高的领域需求相契合, 因此在航天领域的设备控制系统上部署微内核系统有助于为其系统整体的可靠性提供保障. 航天领域的控制系统广泛使用 SPARC 架构处理器. 欧洲航空局为航天设备开发了 ERC32[11]处理器, 后续又转为 LEON 系列处理器[12,13], 具有防辐射、高性能、低成本等特性. 中国在航天设备上也大规模采用了自研的 SPARC 架构的 BM3803, BM3823 等处理器[14]以及 S698 系列处理器[15]. 目前, SPARC 架构的处理器在诸如航天飞船、卫星载荷、星球车以及空间站等航空航天设备上被使用, 占据重要地位[14,16−18]. 然而 SPARC 架构的一些架构特性却严重影响了微内核上 IPC (进程间通信) 的性能.

在微内核架构上, 各种系统服务和驱动都作为用户态进程运行, 因此应用程序必须通过 IPC 才能与这些系统服务进程通信, 进而能够调用这些系统服务. 此外, 系统服务之间可能存在依赖关系, 某个系统服务可能需要向其他系统服务发送 IPC 来实现其功能. 比如应用程序调用文件系统接口, 可能需要向文件系统服务发送 IPC, 而文件系统服务可能又需要向硬盘相关的系统服务发送 IPC, 在这一场景下, 微内核需要发生 4 次用户态与内核态之间的切换, 而宏内核则仅需 1 次即可完成. 许多研究表明, IPC 带来的额外开销是微内核系统性能劣于宏内核的重要因素之一[19−25].

许多工作[24−35]都曾专门对微内核 IPC 进行过优化, 其中一些工作利用了具体架构的特性, 但很少有工作关注 SPARC 架构上的 IPC. 在 SPARC 架构上运行微内核时, 其 IPC 性能往往较差, 这是由 SPARC 架构本身的寄存器窗口特性导致的. SPARC 架构的所有通用整型寄存器以窗口形式组织, 每个窗口包含 24 个寄存器可供使用, 相邻窗口间有 8 个寄存器是重叠的, 因此函数调用可以通过滑动至下一个窗口来高效完成, 但其寄存器数量与窗口数量成正比, 即使只有 8 个寄存器窗口也意味着有 136 个通用寄存器. 其大量的通用寄存器严重增加了微内核下用户态进程进出内核以及线程切换时保存/恢复上下文开销[36], 使得 IPC 性能较差. 我们观察到在现代处理器的高性能背景下, SPARC 架构的寄存器窗口设计并不能对程序性能带来明显提升, 寄存器窗口设计所针对的优化目标——函数调用也并不再是程序的性能瓶颈. 受文献 [37] 的寄存器组 (Register bank) 设计的启发, 我们提出了 BankedIPC, 其核心在于设计实现了在软件层面对 SPARC 架构寄存器窗口进行管理和分配的方法, 由微内核为每个线程分配一个包含若干个寄存器窗口的寄存器组, 并允许多个线程的上下文同时位于寄存器中, 以大幅提升上下文切换性能进而实现对 IPC 的性能优化. 对于需要传输的数据较少的 IPC, 可以直接使用寄存器传递这些数据以避免访存开销, 从而提升性能. 然而在正常运行的过程中, SPARC 架构并没有像文献 [34] 中提到的 x86 架构里那样的空闲寄存器用以存放这些 IPC 数据, 因此无法采用这一优化方法. 而在使用上述寄存器组的设计后, 相邻寄存器组之间会产生一些冗余的空闲寄存器, 因此这些寄存器就能够被用来传递 IPC 数据. 我们在采用寄存器组设计的同时, 使用这些空闲寄存器传递 IPC 数据, 充分利用了 SPARC 架构提供的大量寄存器, 进一步提升了 SPARC 上微内核在处理需要传输的数据较少的 IPC 时的性能.

测试发现, 在多核 SPARC 处理器上, 运行在不同 CPU 核心间的应用在进行跨核 IPC 时性能往往较差, 因此本工作还针对跨核 IPC 进行了专门优化. 在多核处理器上, 一些应用并不会占用全部处理器核心资源, 因此我们从文献 [38] 中得到启发, 提出了 FlexIPC, 令 IPC 的客户端线程和服务端线程运行在不同的处理器核心上, 并通过轮询共享内存中特定标志位的方式转移控制流, 这样就可以避免传统 IPC 进出内核带来的较大性能开销, 也能够防止 SPARC 架构下性能较差的 IPI (核间中断) 影响跨核 IPC 的性能.

自研微内核 ChCore 上实现了通用的同步 IPC 机制, 与主流的微内核实现相比, ChCore 上的 IPC 具有较好的性能[35]. 本工作将 ChCore 移植到了 SPARC 架构上, 并在 ChCore 原有的 IPC 机制的基础上, 实现了前文所提的对 IPC 的优化. 本工作在航天领域中使用的真实硬件上测试了在使用各类优化策略前后 ChCore 上 IPC 的性能以及调用用户态系统服务的性能, 验证了这些优化方法的有效性. 本文的主要贡献如下.

1) 提出了内核对 SPARC 架构寄存器窗口进行高效分配管理的设计方法, 设计了软件层面的寄存器组机制, 优化了 SPARC 架构下上下文切换的性能;

2) 基于内核的寄存器组设计, 提出了 BankedIPC, 实现快速上下文切换的同时, 使用空闲寄存器传递 IPC 数据, 将寄存器 IPC 的思想应用在 SPARC 架构上;

3) 针对 SPARC 多核处理器, 提出了 FlexIPC 以优化跨核 IPC 的性能;

4) 在航天领域使用的真实硬件 S698PM 上对 ChCore 上优化前后的 IPC 性能进行了测试, 测试表明优化后 IPC 时延降低至 50%.

1 背景知识

1.1 SPARC 架构及其寄存器窗口机制

SPARC 架构在提出时是一种灵活度较高且具有可扩展性的设计并取得了较大成功. 一方面, Sun Microsystem 公司在其工作站和服务器环境上广泛采用了 SPARC 架构的处理器; 另一方面, 欧洲航天局也基于 SPARC 架构开发 ERC32 处理器[11], 用于嵌入式航空航天设备中. 进入 21 世纪后, LEON 系列的 SPARC 架构处理器在欧洲航天设备及国际空间站上被广泛应用, 成为航天领域内的主流处理器. 中国在发展航天事业的过程中, 也在航天设备上采用了 SPARC 架构的处理器[39], 并自主研发了 BM3803, BM3823, BM3883 系列[14]以及 S698 系列[15]SPARC 架构处理器, 目前也逐步应用到了航天设备上.

SPARC 架构的可扩展性的一个例子就是其寄存器窗口机制. 寄存器窗口机制是 SPARC 架构的特色之一, 在这种设计下, SPARC 处理器的寄存器分布如图 1 所示. SPARC 是一种 RISC 架构, 拥有大量通用寄存器, 但是在应用程序运行时, 并不是所有寄存器都是对其可见的. 在应用程序运行的任一时刻, 其只能访问一个特定的寄存器窗口中的 24 个寄存器以及 8 个不属于任何寄存器窗口的全局寄存器, 共计 32 个通用寄存器. 如图 1 所示, 每个寄存器窗口的 24 个寄存器又可以划分为 8 个 in 寄存器, 8 个 local 寄存器以及 8 个 out 寄存器. 此外, SPARC 架构还提供了一个被称为 CWP 的系统状态寄存器用于标识当前使用的是哪个寄存器窗口. 这种设计方案的目的是提升应用程序进行函数调用的性能, 应用程序在进行函数调用时仅需执行一条指令滑动到下一个窗口, 而无需对之前使用的寄存器进行保存, 此外从图 1 也可以看到, 相邻的寄存器窗口之间有 8 个寄存器是共享的, 即当前窗口的 out 寄存器与下一个窗口的 in 寄存器对应了相同的寄存器, 这样就可以方便地进行参数和返回值的传递: 调用者函数需要将参数存放在其 out 寄存器中, 执行函数调用指令后, 对于被调用函数来说, 传入的参数就在 in 寄存器中了, 返回值的传递同理. 在函数调用完成后, 仅需再执行一条指令即可返回之前的窗口, 由于调用者函数曾经使用的寄存器本来就在这个窗口中, 因此就无需再对其使用的寄存器进行恢复. 此外值得注意的是, SPARC 上的寄存器窗口成环状排列, 即最后一个寄存器窗口的 out 寄存器与第 1 个寄存器窗口的 in 寄存器相同, 各寄存器窗口首位相接成环, 因此本文后续的示意图中使用环来代表 SPARC 上的寄存器. SPARC 架构的寄存器窗口设计能够有效提升函数调用和返回的性能, 避免了在函数调用时保存寄存器的开销. SPARC 架构处理器上寄存器窗口的数量是由硬件制造商设计决定的, 这是 SPARC 架构可扩展性的体现之一. 为了达到更好的函数调用性能, 制造商可以选择至多实现 32 个寄存器窗口, 但寄存器窗口数量的增多也会导致硬件复杂性显著提升.

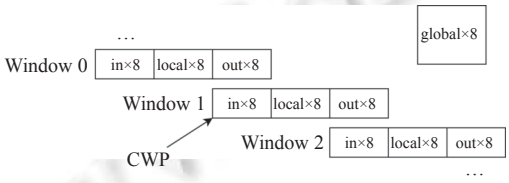


图 1 SPARC 架构的寄存器窗口布局

1.2 微内核与进程间通信

微内核是一种经典的操作系统设计, 它将大部分操作系统服务和功能迁移到内核外部, 作为用户态应用来提供服务, 如文件系统、网络协议栈和设备驱动等. 这与早期宏内核架构中, 将与操作系统功能相关的所有函数都统

一放置在单独的内核代码中并在内核特权级下运行的设计有所不同. 在宏内核架构下, 与操作系统功能相关的所有函数都被统一放置在单独的内核代码中, 在内核特权级下运行. 因此宏内核架构常常会带来可靠性和安全性问题, 比如当内核中的一个功能出错误或者遭到攻击时, 就可能导致整个系统崩溃或被攻击者完全掌控. 而微内核中仅保留内存管理、线程调度、进程间通信等最基本的功能, 普通的用户态应用可以通过内核提供的进程间通信功能使用用户态的系统服务, 以此提供较强的可靠性与易演化性.

微内核的可靠性和安全性也伴随着性能上的问题. 由于微内核中各种关键的系统服务和驱动都被视为一个用户进程, 因此其他应用程序在需要使用这些系统服务时, 就不得不通过进程间通信 (IPC) 来进行. 而进程间通信会导致进出内核以及上下文切换, 这两者都会带来额外开销, 而宏内核架构下仅需进出内核即可完成应用程序的请求. 此外, 微内核中应用程序的一个请求可能会涉及多个系统服务, 而这些系统服务很可能是作为不同的进程运行的, 因此在完成这一请求的过程中就可能发生多次 IPC, 带来明显的性能下降. 因此 IPC 的性能始终是微内核性能的瓶颈之一. 这一现象自初代微内核 Mach[40]问世时就被发现, 以 L4[34]为代表的第 2 代微内核对 IPC 进行了优化, 大幅提升了 IPC 性能, 并以此为基础衍生出了诸如 Pistachio, Fiasco 等微内核, 庞大的 L4 微内核家族在学术和工业界被广泛应用[41]. seL4 微内核[33]为 IPC 引入了能力 (Capability) 机制, 它使用能力表示用户程序对系统资源的使用权限, 由内核负责管理, 并可通过 IPC 在不同进程间传输这一权限, 从而可以防止用户程序滥用系统服务, 进一步提升了微内核的安全可靠性.

在微内核几十年的发展过程中, 有许多工作[24−35]都针对微内核的 IPC 性能进行了优化. 近年来, 一些工作着眼于利用硬件特性进一步提升 IPC 性能. 比如 SkyBridge[25]利用了 Intel 处理器的 vmfunc 特性, UnderBridge[35]则利用了 Intel 处理器的 MPK 特性. 然而这些优化工作只能在特定的处理器上使用, 对于目前在日常中比较少见但在航天领域广泛使用的 SPARC 架构, 则鲜有相关研究.

2 研究动机

2.1 SPARC 架构寄存器窗口的开销

在 SPARC 架构提出之时, 函数调用的性能开销确实是应用程序性能的瓶颈之一, 因此寄存器窗口的设计能够有效提升 SPARC 架构下寄存器的性能. 但是随着时代的发展, 函数调用时访存带来的性能开销逐渐下降, 应用程序的主要性能开销之一在于进出内核. 进出内核时大多数情况下需要对应用程序的上下文进行保存, 而对于 SPARC 架构而言, 其所有寄存器窗口都需要被保存. 由于 SPARC 架构寄存器数量太多 (如果有 8 个寄存器窗口, 就有共计 136 个通用寄存器), 并且遍历时还需要对窗口进行滑动, 于是保存、恢复用户程序上下文的开销就被进一步放大了. 我们在自研微内核 ChCore 上测试了 SPARC 架构上单次 IPC 的时延分布, 并排除了 IPC 服务端处理请求的耗时, 结果如表 1 所示, 可见 IPC 中保存与恢复上下文的耗时就占了 IPC 总时延的 50% 以上, 对 IPC 性能影响十分严重. 对于微内核而言, 由于 IPC 的广泛使用, 进出内核发生得更为频繁, 进一步限制了应用程序的性能.

表 1 SPARC 上 IPC 时延分布

操作	占比 (%)
上下文保存与恢复	57.0
进程切换与IPC处理	37.2
其他	5.8

我们对比了在 SPARC 架构上, 应用程序全程仅使用一个寄存器窗口以及使用全部寄存器窗口的性能. 为了使应用程序全程仅使用一个寄存器窗口, 在对其进行编译时开启了编译器的-mflat 选项, 这一选项使得编译器像其他架构那样在函数调用时使用栈保存寄存器, 并且不会生成滑动窗口的指令. 我们首先在 Linux 上测试了 nbench 中各程序的性能, 结果如图 2 所示, nbench 会运行一系列有代表性的算法测试程序并测量其吞吐量, 大多数应用程序在仅使用一个寄存器窗口时, 性能下降不是很明显或几乎没有, 都在 2% 以内, 对于某些函数调用频繁的应用, 性能下降也均在 10% 以内.

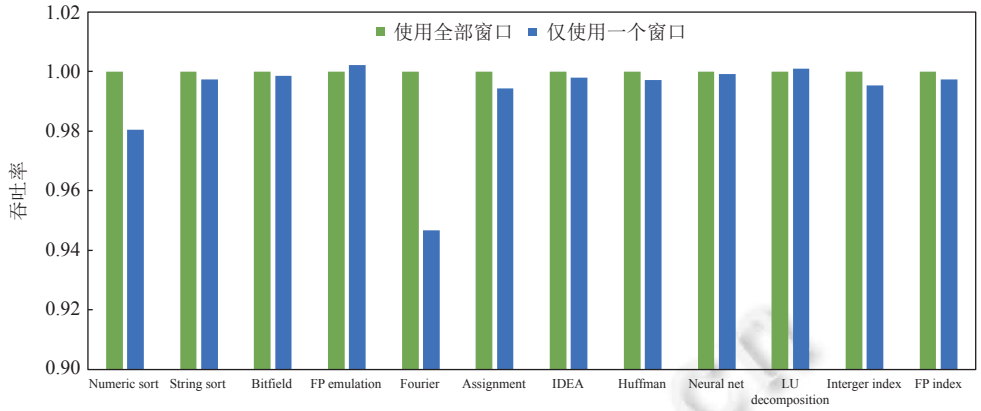


图 2 nbench 测试结果

我们认为这一性能下降相比之后可进行的优化带来的收益来说是可以接受的. 在现代设备上, 与寄存器窗口对函数调用带来的优化相比, 进出内核带来的性能损失更加难以接受, 对于频繁使用 IPC 的微内核系统而言更是如此. 但是令应用程序仅使用一个寄存器窗口显然浪费了 SPARC 提供的大量寄存器, 因此我们可以针对寄存器窗口的特性对内核进行重新设计与优化, 使内核掌握对全部寄存器窗口的分配和管理, 从而尽最大可能发挥寄存器窗口机制提供的价值. SPARC V8 架构规范[42]就曾经提出减少线程对寄存器窗口的使用以提升上下文切换性能的设计思想, 之后的 V9 规范[43]进一步实现了利用寄存器窗口的快速上下文切换, 而在文献 [44] 中提出可以利用这一点来改进 IPC 性能. 文献 [37] 则是在架构设计层面提出了寄存器组 (Register bank) 的设计以提升微内核系统性能, 而 SPARC 架构的寄存器窗口很适合采取这种寄存器组的设计. 受这一寄存器组的设计的启发, 我们提出了在软件层面对 SPARC 架构寄存器窗口进行管理和分配的方法, 由微内核为每个线程分配一个包含若干个寄存器窗口的寄存器组, 并允许多个线程的上下文同时位于寄存器中, 以大幅提升上下文切换性能进而实现对 IPC 的性能优化.

2.2 使用寄存器传递 IPC 数据

在微内核架构的系统中, 系统服务作为用户态应用程序运行, 普通应用程序请求系统服务需要通过 IPC 来完成, 因此微内核中 IPC 的使用有可能十分频繁. 自研微内核 ChCore 采取了共享内存的方式传递 IPC 数据, 即为 IPC 客户端和服务端分别映射一部分共享内存, 二者通过对共享内存进行读写以交换数据. 然而在多数场景下, IPC 可能无需传递大量数据, 仅需传递少量参数. 文献 [34] 指出, 平均来说 50%–80% 的 IPC 仅需传递 8 个字节以内的数据 (不包括发送者 ID), L4[34]就利用这一观察对这些传递数据量较小的 IPC 进行了高度优化, 实现了 IPC 的快速路径, 其核心设计在于直接使用一些空闲的寄存器来传递 IPC 数据, 并且测试表明这能够将 IPC 时延降低至原来的 48%. 我们在 S698PM 开发板上分别测试了使用内存和寄存器传递少量 IPC 参数时的性能, 结果如图 3 所示, 由于在使用寄存器传递参数时, 不必判断参数数量, 可以简单地按照最大数量传递, 所以随着 IPC 负载的增加, 寄存器传参的性能基本不会变化; 而使用内存传递参数时, 性能则会随着负载的增加而逐渐下降, 此外, 即使在负载为 0 时, 使用寄存器传参的性能也不会明显差于使用内存, 因为拷贝寄存器的开销相对于整个 IPC 过程来说是可以忽略不计的. 但在某些架构下, 往往并没有较多空闲寄存器, 这使得 L4 的寄存器传参 IPC 应用场景受限, 如果用户程序需要传递的数据量略大, 就无法利用这一 IPC 快速路径了. SPARC 架构下通用寄存器数量很多, 但传统的 SPARC 架构上的通用操作系统一般不会对寄存器窗口进行手动管理, 应用程序会使用全部的寄存器窗口, 这样也就导致几乎没有空闲寄存器可用, 难以使用寄存器传递 IPC 数据. 如果微内核能够对 SPARC 的寄存器窗口进行主动管理, 那么就可以充分利用其寄存器数量的优势, 从而可以专门预留一部分寄存器用于 IPC 参数传递, 尽可能实现零拷贝 IPC, 既能够为 IPC 寄存器数据传递提供更大的数据量支持, 也可以避免对其他应用程序的性能产生较大影响.

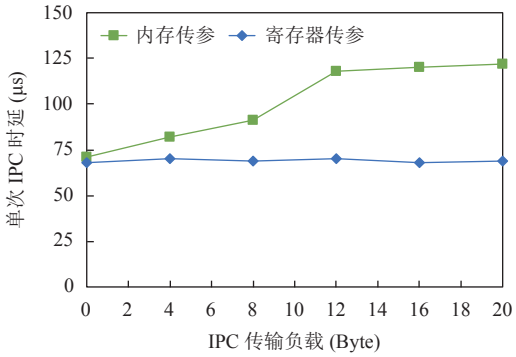


图 3　IPC 使用寄存器和内存传参的性能比较

2.3　跨核 IPC

目前具有多核心的处理器被广泛使用, 这要求操作系统针对多核场景进行特殊设计. 但一部分应用程序仅会使用处理器的单个核心, 不会或者不需要使用其他处理器核心. 在运行这类应用程序时, 为了充分发挥多核 CPU 的优势, 可以将操作系统运行在空闲的 CPU 核心上. 我们可以将这种思想应用在微内核操作系统上, 将用户态的不同系统服务进程运行在空闲核心上, 系统服务和应用程序同时运行在不同的核上, 有助于降低应用程序请求系统服务时的响应时延, 这样普通的用户应用程序就需要发起跨核 IPC 来请求各种系统服务. 一般而言, 为了保证微内核系统的确定性和可靠性, 跨核 IPC 请求发生时, 操作系统需要触发核间中断 (IPI) 来通知被请求的 CPU 核心. 如果不使用 IPI 进行通知, 那么服务端进程响应该 IPC 请求的时延便缺乏了确定性保证, 这也与将系统服务运行在其他核上的目的相悖. 但 SPARC 架构并没有对 IPI 性能进行充分的优化, 如表 2 所示, 在 S698PM 上对 SPARC 上 IPI 的性能测试表明, 其单次 IPI 的平均时延为 34 μs, 与在单核上完成一次普通 IPC 的耗时相当, 因此跨核 IPC 的性能开销将至少增加一倍, 这会严重影响多核处理器上微内核的性能, 甚至导致使用多核运行程序的性能反而不如单核的情况. 为了避免 IPI 的性能影响跨核 IPC 的性能, 我们可以尝试绕过 IPI 来进行跨核 IPC. FlexSC[36]针对多核场景, 提出了“无异常 (exception-less)”的系统调用, 即用户态线程不需要陷入内核, 而是和内核协同通过轮询共享内存的方式, 使运行在其他核心上的内核线程及时处理系统调用请求. 我们借鉴了这一设计思想, 利用 IPC 客户端和服务端的协同设计, 避免 IPC 请求陷入内核以及触发 IPI, 在不需要陷入内核的条件下实现核间的低时延 IPC, 这样既能利用多核 CPU 带来的优势, 又可以避免影响微内核系统本身的确定性和可靠性.

表 2　IPI 与 IPC 性能比较

操作	时延 (μs)
单核IPC	55
IPI	34
跨核IPC	108

3　IPC 优化方法

本节介绍本工作对 SPARC 架构上的微内核 IPC 进行优化的方法. 基于在第 2 节中提到的几个研究动机, 我们提出了两种方案优化 SPARC 架构上的微内核 IPC 性能. 首先, 本工作利用了 SPARC 架构独特的寄存器窗口机制, 重新设计了内核对寄存器窗口的管理和分配算法, 主要针对在同一个处理核心上进行 IPC 的场景, 提出了 BankedIPC. 此外, 我们基于用户态轮询的设计思想, 设计实现了 FlexIPC 以优化跨核 IPC 的性能.

3.1　BankedIPC

当 IPC 客户端线程发起 IPC 请求后, 为了尽可能快地处理此 IPC 请求, 需要直接切换到对应的 IPC 服务端线

程. 在自研微内核 ChCore 上, 如果 IPC 服务端线程此时没有运行在其他处理器核心上, 那么内核就会将服务端线程调度到当前客户端线程的核心上运行. IPC 服务端线程处理完请求后, 也需要再立即切换回原来的客户端线程. 这样整个 IPC 流程就涉及 2 次上下文切换, 每次上下文切换操作都需要保存和恢复一个完整的线程上下文. 而 SPARC 架构上寄存器众多, 如果一个线程使用了全部的寄存器窗口, 那么它需要保存和恢复的寄存器可达 136 个, 而保存和恢复这些寄存器则会为 IPC 带来不可忽视的性能开销. 本工作设计实现了 BankedIPC, 通过重新设计内核管理分配寄存器窗口的策略, 减少了每次上下文切换的耗时, 优化了 IPC 性能. 下面将首先介绍内核管理寄存器窗口的 Register bank 策略, 这是 BankedIPC 的基础, 之后将进一步说明 BankedIPC 如何利用 Register bank 策略实现快速上下文切换以及寄存器 IPC.

3.1.1 寄存器组

在第 2.1 节中提到, 现代 SPARC 处理器上, 与使用全部寄存器窗口相比, 仅使用一个单独的寄存器窗口并不会对应用程序的性能产生较大影响. 因此, 我们可以限制每个应用程序能够使用的寄存器窗口数量, 使其仅能够使用一部分寄存器窗口. 我们为每个线程能够使用的寄存器分配一个寄存器组 (Register bank), 当该线程正在运行时, 它的寄存器组与处理器中真实的通用寄存器相对应, 一般为一个寄存器窗口 (出于性能考虑也可增加其寄存器窗口的数量); 而当该线程因为被抢占等种种原因换出到内存中后, 其寄存器组与内存中的“虚拟寄存器”相对应, 即该线程曾经使用的寄存器的值按照其寄存器组的结构被保存在内存中, 这与普通的上下文保存/恢复相似. 寄存器组的整体设计如图 4 所示.

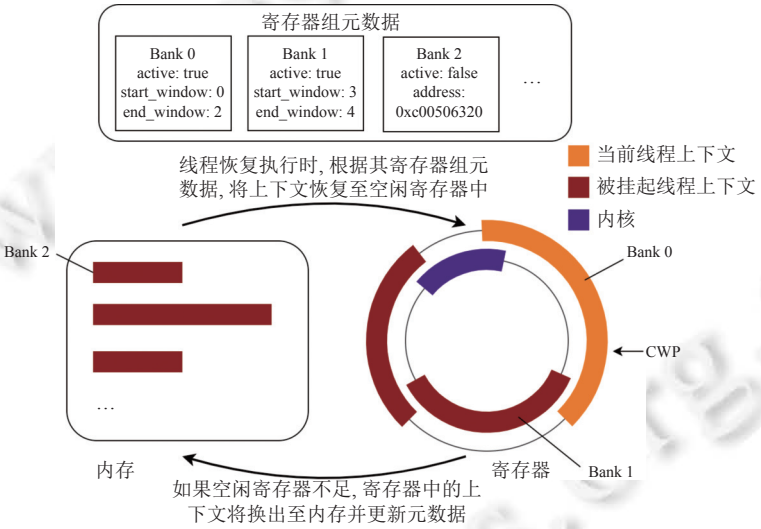


图 4 寄存器组设计概览

当操作系统内核需要将某个之前被换出到内存的线程恢复运行时, 会首先检查目前处理器上是否有足够多的满足该线程寄存器组要求的空闲寄存器. 如果有足够的空闲寄存器, 内核会将该线程保存在内存中的上下文恢复到这些空闲寄存器中, 并记录该线程的寄存器组和这些真实寄存器的映射关系, 之后该线程就可以在这些寄存器上恢复执行. 如果没有足够多空闲的寄存器, 则需要将一些正在运行的线程的上下文换出到内存中, 以获取足够的空闲空间. 以图 4 为例, 当需要将图中 Bank 2 对应的线程恢复执行时, 发现已经没有可用寄存器, 此时需要将某个寄存器组 (比如 Bank 1) 换出到内存. 在这种设计方案下, 将一个线程的寄存器组换出到内存的过程是懒惰的, 即只有在处理器可用的空闲寄存器数量不足时才会发生上下文的保存和恢复, 而不会在某个线程被挂起时立即发生. 这种设计方案的优势在于, 当被暂时挂起的线程短时间内将要恢复执行时, 不必重新切换上下文, 因为它的上下文之前就已经位于寄存器中了, 并且没有被立即换出到内存. 自研微内核 ChCore 上的 IPC 实现与这种场景一

致, 因此这样的设计能够有效提升 IPC 等场景下的性能.

由于每个进程不需要占用全部的寄存器窗口, 因此我们可以同时将多个线程的寄存器组映射在真实寄存器中, 即寄存器中同时保存了多个线程的上下文. 那么在需要将某个线程的上下文换出到内存中时, 就可能涉及在多个寄存器组之间进行选择. 我们认为具体的换出策略与寄存器组机制的设计是正交的, 可以选择简单的先进先出策略, 也可以利用每个进程的调度信息判断是否需要将某个进程挂起并将该进程的寄存器组换出到内存中, 这与所使用的处理器的寄存器窗口总数以及系统的具体目标和需求有关, 此处不进行详细讨论.

下面将具体说明使用寄存器组机制后的上下文切换流程. 如算法 1 所示, 当内核决定切换到某个线程时, 需要调用代码中的函数来激活该线程的寄存器组. 每个线程的寄存器组数据结构 (struct reg_bank) 中, active 字段表示其当前是否已经处于激活状态, 即已经位于某个处理器核心的某个寄存器窗口中; cpu_id 字段则表示如果其已经被激活, 其上下文所在的处理器核心编号. 如果该寄存器组已处于激活状态并且位于当前处理器核心的某个寄存器窗口中, 则激活函数可直接返回; 否则说明该线程的上下文被存储在其他处理器核心的某个寄存器窗口上, 因此需要发送 IPI 来通知该处理器核心, 先将它的上下文保存到内存中, 然后再在当前处理器核心上继续完成激活. 之后, 通过 reg_bank_assign 函数在当前处理器核心上选择空闲的寄存器窗口作为目标线程的寄存器组. 注意到每个处理核心都维护了对应的寄存器组元数据 (struct reg_bank_meta), 其 activated_reg_bank 字段记录了当前处理器核心上的窗口占用情况, 因此可以从中得知是否有寄存器窗口空闲, 在没有任何窗口空闲时, 默认采取先进先出策略驱除一个寄存器组以释放被占用的窗口. 分配完成后, 被分配的寄存器窗口被记录到 reg_bank 结构体的 start_window 字段. 之后则将目标线程的上下文换入到被选中的窗口中, 此时如果元数据中记录了被选中的窗口正在作为某个其他线程的寄存器组, 则说明这个线程被驱逐, 需要将其上下文换出到内存中, 上下文的换入/换出由伪代码中的 reg_bank_swap 函数代表, 在实际代码中这一部分主要由汇编代码实现. 最后则需要更新相关数据结构并设置系统寄存器的值. 下面将介绍执行对寄存器组进行交换 (即执行 reg_bank_swap 函数) 的实现方法.

算法 1. 激活寄存器组.

```
1 void reg_bank_activate(struct reg_bank *bank)
2 {
3       struct reg_bank_meta *meta = reg_bank_meta[current_cpu_id];
4       struct reg_bank *bank_out;
5       if (bank->active == true) {
6          if (bank->cpu_id != current_cpu_id) {
7             send_ipi(bank->cpu_id, REG_BANK_REVOKE, bank);
8          } else {
9             return;
10         }
11      }
12      reg_bank_assign_window(meta, bank); // Set bank->start_window.
13      bank_out = meta->activated_reg_bank[bank->start_window];
14      if (bank_out != NULL) {
15         // Swap is implemented with assembly code.
16         reg_bank_swap(bank->start_window, bank->context, bank_out->context);
17         list_del(&bank_out->reg_bank_node);
18         bank_out->active = false;
19      } else {
```

```
20          reg_bank_swap(bank->start_window, bank->context, NULL);
21      }
22      list_append(&bank_in->reg_bank_node, &meta->activated_bank_queue);
23      bank_in->cpuid = current_cpu_id;
24      bank_in->active = true;
25      meta->activated_reg_bank[bank_in->start_window] = bank_in;
26      update_regs(bank); // Update PSR and WIM (system registers).
27 }
```

SPARC 架构下, 即使内核也无法以全局索引来访问所有通用寄存器, 只能访问当前寄存器窗口内的寄存器和全局寄存器, 而当前的寄存器窗口由处理器状态寄存器 PSR 中的 CWP 位决定. 此外, SPARC 处理器还是用 WIM 寄存器标记了当前无效的寄存器窗口, 如果 CWP 指示的当前窗口与 WIM 标记的无效窗口一致, 那么就会触发 window_overflow 或 window_underflow 这两种系统陷阱 (trap). 图 4 中, CWP 指向了当前线程寄存器组中的某个窗口, 当该线程运行时, WIM 中与该线程寄存器组对应的窗口被标记为有效, 其余窗口无效. 为了实现寄存器组机制, 我们需要内核能够对所有寄存器窗口进行管理和分配, 因此在内核需要控制寄存器窗口时, 需要将 WIM 寄存器清空, 使内核暂时获得对所有寄存器窗口的控制权限, 此时内核就能够通过修改 PSR 中的 CWP 来访问任意寄存器窗口, 在内核需要做的操作完成后, 会将 CWP 和 WIM 的值恢复, 以跳转回内核原来所处的寄存器窗口, 继续完成其他操作. 比如, 在需要恢复某个线程的上下文时, 内核会找到可以容纳该线程上下文的寄存器窗口, 然后跳转到该窗口并将占据该窗口的线程上下文 (如果有的话) 保存到内存中, 之后再将需要恢复的线程上下文写入寄存器中. 如果这一过程涉及多个窗口的保存或恢复, 则内核还需要相应地在寄存器窗口间进行滑动, 由于 WIM 已被清零, 所以这种滑动不会导致任何陷阱发生, 能够顺利完成.

为了保证微内核下不同进程之间的隔离性和可靠性, 在实现上述寄存器组机制时还需要注意以下问题. 首先, 在交换寄存器组时必须将寄存器窗口中所有的寄存器全部覆写, 以保证换入的线程无法访问被换出线程存留在寄存器中的数据. 此外, 在退出内核进入用户态时, 需要对 WIM 寄存器进行修改, 禁止应用程序访问其寄存器组之外的其他寄存器窗口, 以防止某个线程读取其他同时位于寄存器中的属于其他进程的上下文. 当用户态应用程序试图使用 SAVE 或 RESTORE 指令滑动到其寄存器组之外的寄存器窗口时, 由于这些寄存器窗口在 WIM 中被标记为无效, 因此会触发 window_overflow 或 window_underflow 陷阱, 由内核进行进一步处理, 从而能够阻止该线程的越界访问.

出于性能考虑, 我们为系统的内核单独分配了一个寄存器组, 包含一个寄存器窗口, 并且将该寄存器组固定在真实寄存器中, 如图 4 所示. 这样应用程序在进入内核后, 就可以直接跳转到内核对应的寄存器组中, 从而不需要再为内核分配额外的空闲寄存器窗口, 也不需要保存当前线程的上下文. 但这样的设计会减少可用的空闲寄存器窗口的数量, 进而导致可以同时在寄存器中存在的线程数量减少, 因此可能会使得对线程上下文的交换更加频繁. 但微内核系统下, 用户态应用程序经常需要进出内核, 我们认为对进出内核时上下文的切换进行专门优化是值得的.

实现寄存器组机制时的主要挑战是, 各寄存器组使用的寄存器窗口是不能直接相邻的, 这是因为 SPARC 架构中相邻的寄存器窗口之间有 8 个寄存器是共享的, 如果寄存器组所使用的寄存器窗口相邻的话, 那么不同线程间就会出现共享的寄存器, 这会导致线程之间相互影响, 也难以保证隔离性. 另一个重要原因是, 当应用程序触发陷阱进入内核时, 会自动滑动至下一个寄存器窗口用于进行陷阱处理, 如果寄存器组直接相邻, 那么在陷阱处理时就有可能进入其他线程正在使用的寄存器窗口中. 因此, 每个寄存器组都需要包含一个额外的“冗余”寄存器窗口用于与其他寄存器组进行分隔, 如图 5 所示. 如果一个寄存器组包含 3 个寄存器窗口, 那么其能够完全使用的寄存器只有前两个寄存器窗口中的所有寄存器, 第 3 个寄存器窗口将作为“冗余”窗口. 这样看似会导致一整个寄存器窗口的浪费, 但实际上仅会浪费 8 个寄存器. 这是因为冗余窗口中的 8 个 in 寄存器与上一个寄存器窗口共享, 被

其所属的寄存器组使用, 8 个 out 寄存器与下一个窗口共享, 被相邻的寄存器组使用, 只有 8 个 local 寄存器是不会被利用到的. 但是这 8 个 local 寄存器也并不是完全被浪费了, 它们还可以用于进行陷阱处理以及在第 3.1.3 节中提到的用于暂存 IPC 数据.

图 5 冗余寄存器窗口

3.1.2 快速上下文切换

基于第 3.1.1 节中提到的寄存器组机制, 我们可以实现非常高效的上下文切换, 最大限度地减少 SPARC 架构下上下文切换时对大量寄存器的保存与恢复开销. 由于我们在寄存器组的设计中采取了懒惰的上下文切换策略, 映射到真实寄存器中的寄存器组一般总是达到处理器的寄存器组容纳上限的, 这就可以保证有尽可能多的线程上下文位于寄存器中. 如果上下文切换的源线程和目标线程的上下文同时位于寄存器中 (但位于不同的寄存器窗口), 那么显然在进行此类切换时就不需要保存/恢复它们的上下文, 仅需切换当前可见的寄存器窗口即可, 这可以通过修改处理器状态寄存器 PSR 中的 CWP 位实现, 而对 PSR 的修改仅需几个时钟周期即可完成, 这与通过访存来对线程的上下文进行保存和恢复相比是微不足道的.

快速上下文切换对于微内核 IPC 的优化效果更为明显. 能够使用快速上下文切换的关键前提是, 上下文切换的源线程和目标线程对应的寄存器组已经同时被映射在了真实寄存器中, 在一般情况下的上下文切换中这一限制条件不一定能够满足. 对于微内核 IPC 这一场景来说, 一般而言两个进程之间的 IPC 不会仅进行一次, 而同样的 IPC 重复发生时, IPC 两端的线程很有可能在之前的通讯过程中已经被换入到了真实寄存器中, 这样再次进行 IPC 时就可以受益于快速上下文切换带来的优化.

需要注意的是, 即使使用快速上下文切换也并不能避免所有保存与恢复上下文的访存操作, 因为一些被所有线程使用的寄存器 (包括 8 个全局通用寄存器以及各种处理器状态寄存器) 仍然需要在进出内核时使用内存中预留的空间进行保存和恢复, 因为这些寄存器被所有线程同时使用, 无法像一般的通用寄存器那样利用 SPARC 的寄存器窗口机制进行隔离. 此外, 在实现过程中我们发现一些有特殊用途寄存器, 比如用于保存返回值的寄存器, 也是很有必要保存在内存中的, 因为内核很有可能经常需要对这类寄存器的值进行修改以向用户态传递信息, 或者读取它们的值以获取用户态数据 (比如进行回溯或者获取系统调用号等). 虽然在快速上下文切换时, 仍然需要使用内存保存一部分上下文, 但与之前每次上下文切换都需要保存全部寄存器窗口的全部寄存器相比, 采取寄存器组机制后仍然需要使用内存保存的寄存器数量已经大幅减少, 其优化幅度这对于寄存器窗口数量越多的处理器越明显.

3.1.3 寄存器 IPC

使用寄存器传递 IPC 信息的思想早已有之. 对于消息长度较短的 IPC, L4 内核[34]将其需要传递的数据放入寄存器中直接传递到另一端, 从而避免了对数据传递的内存拷贝. 然而, L4 只能使用一些不被占用的寄存器来传递信息, 但是在大多数架构上, 这种寄存器数量很少, 这导致寄存器 IPC 只能适用于消息长度很短的 IPC 请求, 使用场景受限.

然而 SPARC 架构的通用寄存器数量非常多, 寄存器组机制的设计也能够为我们提供比较大量的空闲寄存器用以传递 IPC 信息. 在第 3.1.1 节中提到, 为了保证各线程间的隔离, 以及为了能够正常处理程序执行时遇到的陷阱, 每个寄存器组都会额外占用一个冗余的寄存器窗口, 该冗余窗口的主要用途一般在于陷阱处理. 另外由于我们

为内核预留了一个寄存器组 (包括两个寄存器窗口) 用于减少进出内核的开销, 那么这个冗余的陷阱处理窗口所需要完成的任务就非常简单, 仅需保存必要的寄存器 (如全局寄存器等) 的值然后切换到内核寄存器组即可. 这一过程并不需要占用大量的寄存器, 甚至经过优化, 可以完全避免对这一陷阱处理窗口内 in 寄存器的影响, 仅使用 8 个 local 寄存器即可完成. 于是陷阱处理窗口内的 8 个 in 寄存器就可以用来保存 IPC 传递的信息. 这样, 对于消息长度较短的 IPC, 其信息可以通过参数的方式使用寄存器进行传递, 这一过程中可以使用当前寄存器组的陷阱处理窗口内的 in 寄存器来暂时保存这些参数. 具体流程如图 6 所示, 当 IPC 发起线程发起 IPC 调用时, 它可以将需要传递的信息作为 IPC 调用函数的参数, 根据 SPARC 函数调用惯例, 这些参数会被保存在当前寄存器窗口的 out 寄存器中. IPC 调用会导致陷阱, 陷入内核后处理器会自动滑动至下一个相邻窗口即陷阱处理窗口, 该窗口内的 in 寄存器保存了传递的参数. 之后将使用空闲的 local 寄存器完成必要寄存器的保存并跳转到内核窗口, 然后切换至被 IPC 调用的线程. 接下来内核将切换到之前 IPC 调用线程的陷阱处理窗口, 将暂存在 in 寄存器中的 IPC 信息转移到全局寄存器中, 然后再切换到 IPC 被调用线程的陷阱处理窗口, 将全局寄存器中保存的信息转移到被调用线程的 in 寄存器中, 最后返回用户态执行被调用线程 IPC 处理函数. 返回用户态时, 窗口会自动滑动, 传递的 IPC 信息会位于当前窗口 out 寄存器中, 根据 SPARC 调用惯例, 对于 IPC 处理函数来说, 传递的 IPC 信息已经位于该函数的参数中了, 从而实现了 IPC 信息的寄存器传递. 这种方案既能够有效利用寄存器组设计中不得不预留出的陷阱处理窗口, 又能够在避免影响程序正常运行的情况下扩展寄存器 IPC 能够利用的寄存器数量, 从而优化短 IPC 的性能.

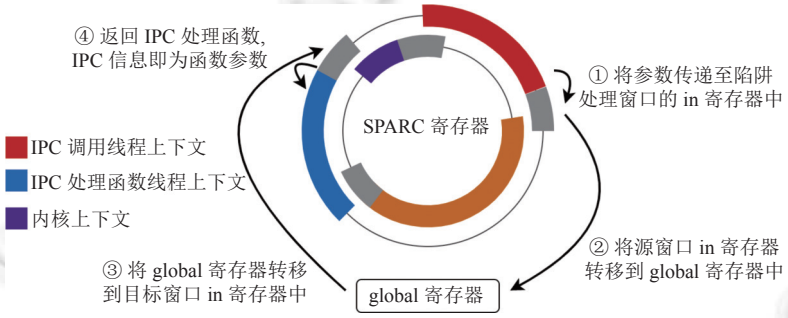


图 6 寄存器 IPC 流程

3.2 FlexIPC

在多核处理器上, 一些应用程序可能无法完全利用多个核心的资源, 此时就可以将一些系统服务部署到单独的核心上运行, 而应用程序在向这样的系统服务发起请求时, 就会导致跨核 IPC. 许多微内核都采用了基于线程迁移的 IPC 实现方案, 这种实现方式可以确保 IPC 请求能够被及时响应. 然而对于跨核 IPC 来说, 跨核线程迁移需要 IPI 提供支持, 此时 IPI 就成为跨核 IPC 的性能瓶颈之一. 我们在 S698PM 上的测试表明, SPARC 架构上 IPI 性能并不理想, 制约了跨核 IPC 的性能, 因此我们设计实现了 FlexIPC 来规避 IPI 带来的较高开销.

FlexIPC 基于轮询共享内存的方式实现 IPC 服务端和客户端的控制流转移. 发起 IPC 的客户端线程需要在 IPC 两端进程的共享内存中设置相关标志, 而服务端线程则会对共享内存中的对应区域进行轮询, 当发现客户端的消息已经准备好时, 就立即开始对 IPC 消息进行处理. FlexIPC 在自研微内核 ChCore 上的具体实现方法如算法 2 所示. 在 FlexIPC 调用发起时, 会将共享内存中的 start 标志位置为 1, 之后发起线程会一直轮询 finish 标志位的内存直至其变为 1. 而 IPC 服务端线程则会轮询其与所有客户端线程的共享内存, 在发现 start 标志位为 1 后就可以开始对此 IPC 消息进行处理, 处理完毕后可以在共享内存中写入返回值并将 finish 标志位置为 1, 这样 IPC 发起线程就可以得知 IPC 已经完成. 每个客户端线程在建立 FlexIPC 连接前, 首先需要发起一个正常的 IPC 调用, 目的是让服务端线程得知自己的存在以及共享内存的地址, 这样其之后发起的 FlexIPC 请求才能在服务端轮询的过程中被发现并处理. 对于跨核 IPC 来说, FlexIPC 请求会比较快地被其他核心上正在运行的服务端 IPC 处理线程发现,

其间并不需要打断任何进程的运行, 也不需要使用 IPI 等机制通知服务端进程, 因此能够规避发起 IPI 进行线程迁移带来的额外开销. 此外, 每次 FlexIPC 的过程中并不会涉及与 IPC 相关的系统调用, 一切均在用户态完成, 这也避免了进入内核带来的额外开销.

算法 2. FlexIPC 的实现.

```
1 long flex_ipc_call(ipc_msg_t *ipc_msg)
2 {
3       ipc_msg->flex_ipc.flex_start = true;
4       while (ipc_msg->flex_ipc.flex_finish != true) {
5             // spin loop
6       }
7       ipc_msg->flex_ipc.flex_finish = false;
8       return ipc_msg ->flex_ipc.ret;
9 }
10 void flex_ipc_handler()
11 {
12       long ret;
13       for client in clients {
14             ipc_msg = (ipc_msg_t *)(client->server_shm_addr);
15             if (ipc_msg->flex_ipc.flex_start == true) {
16                ipc_msg->flex_ipc.flex_start = false;
17                ret = do_something(ipc_msg);
18                ipc_msg->flex_ipc.ret = ret;
19                ipc_msg->flex_ipc.flex_finish = true;
20             }
21       }
22 }
```

4 测试与评估

4.1 测试环境

我们进行了一系列测试以检验我们对 SPARC 架构上微内核 IPC 优化的效果. 测试使用的硬件平台是珠海欧比特公司自研的抗辐照型高性能处理器 SOC S698PM, 该处理器遵循 SPARC V8 标准, 内部集成了 4 个相同的处理器核心, 此外还具有板载 8 MB SRAM 和 256 MB DDR2 内存. S698PM 硬件平台是在航天领域的星载计算机中被实际使用的, 因此在该硬件上的测试结果对于实际应用而言更具可参考性. 由于本工作是针对 SPARC 架构上的微内核 IPC 进行优化, 而目前支持 SPARC 架构的微内核系统比较少, 所以我们首先将自研微内核 ChCore 移植到了 SPARC 架构上. ChCore 在其他架构上实现了通用的同步 IPC 机制, 我们针对 SPARC 架构的特性, 对 ChCore 上原有的同步 IPC 机制进行了一系列优化工作, 所以我们的测试均在自研微内核 ChCore 上进行, 主要目的是对比优化前后的微内核 IPC 的性能, 以及分析各优化方法对性能的提升效果.

值得注意的是, 由于绝大多数内核上的 IPC 都无法避免需要通信的进程间的上下文切换, 即使能够避免, 也一般是采用了 IPI 等手段通知其他 CPU 核心上正在运行的进程来传递信息, 这就意味着一般情况下, 其他微内核在实现 IPC 的过程中很难同时避免上下文切换和 IPI, 而这两者分别是本文提出的 BankedIPC 以及 FlexIPC 的主要

优化目标. 而 SPARC 架构的寄存器数量较多以及 IPI 性能较差这两个特性是与架构设计相关的, 因此虽然我们目前仅在 ChCore 上进行了测试, 但本文优化 IPC 的方法在其他 SPARC 架构的微内核上也是可行的.

4.2 微基准测试

在本小节中, 我们希望通过微基准测试回答以下问题.

• RQ1: BankedIPC 能对 ChCore 上的单核 IPC 性能带来多少提升?

• RQ2: BankedIPC 提升 IPC 性能的原理是什么?

• RQ3: 内核采用的寄存器组机制能否对非 IPC 场景带来优化效果? 是否存在局限性?

• RQ4: FlexIPC 能对 ChCore 上的跨核 IPC 性能带来多少提升?

(1) RQ1: BankedIPC 能对 ChCore 上的单核 IPC 性能带来多少提升?

为了说明 BankedIPC 对 ChCore 上 IPC 的优化效果, 我们对优化前后单次 IPC 的平均时延进行了测试. 我们创建了两个测试进程作为 IPC 客户端和服务端, 客户端则创建多个线程同时向服务端发送一定数量的并发 IPC 请求, 对于每个被测试的 IPC 并发数量, 我们运行了 3 次上述测试并取平均值, 得到每次 IPC 的平均时延. 我们测试了两种不同类型的 IPC, 一种是零数据 IPC, 即客户端不通过 IPC 传输任何数据, 服务端接收到 IPC 请求后也不会进行任何处理, 而是直接返回, 测试这种 IPC 可以使我们重点关注到 IPC 控制流转移的性能; 另一种测试类型是传输少量数据的 IPC, 客户端会传输大小为 20 B 的数据, 服务端会读取这些数据并进行简单处理, 并将处理结果返回, 这一数据量足以容纳于寄存器中, 进行此测试的目的是比较使用寄存器传递数据和共享内存传递数据的性能. 我们在 ChCore 上分别测试了未优化 (Base) 和采用 BankedIPC 优化后的两种 IPC 在上述两种场景下的性能. 为了避免跨核 IPC 带来的额外影响, 本测试中每个 IPC 客户端线程和服务端线程都位于同一个处理器核心上. 测试结果如图 7 所示.

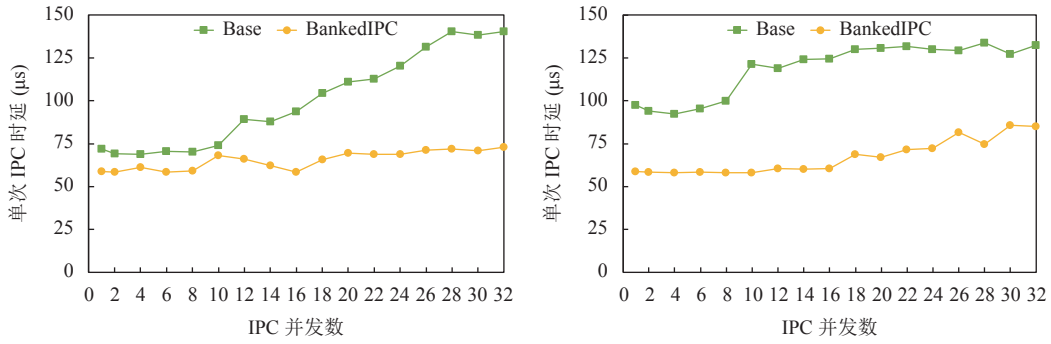


图 7 BankedIPC 时延测试结果

从测试结果可以看到, 采用寄存器组设计后, 每个线程仅使用一个寄存器窗口, 且寄存器的保存和恢复是懒惰的, 因此线程上下文切换的开销明显下降, 从而显著提升了 IPC 性能. 通过图 7(a) 可以看到, 当 IPC 不传递任何数据时, BankedIPC 的性能在 IPC 并发数量较少时略优于 Base, 随着并发数量的增加, Base 上线程切换带来的上下文保存/恢复开销急剧增大, 导致其性能明显下降; 而 BankedIPC 采用了寄存器组设计后的, 上下文切换时需要做的访存操作减少, 即使并发数很多时也没有出现特别严重的性能下降. 相比之下, 如果仅关注控制流转移的性能, BankedIPC 的平均 IPC 时延下降至原来的约 66%. 通过图 7(b) 可以看出使用寄存器传递数据相比使用内存传递数据带来的性能提升. 使用共享内存传递数据时, 需要进行一定的访存操作, 期间还可能触发页错误从而导致进一步的额外开销. 而使用寄存器传递少量数据则可以避免这些问题, 在传递 20 B 的数据时, 完全使用寄存器传递数据的 BankedIPC 的平均时延下降至原来的 53%.

(2) RQ2: BankedIPC 提升 IPC 性能的原理是什么?

我们对采用了寄存器组机制的 BankedIPC 的时延分布进行了拆解分析, 并与优化前的时延分布进行对比, 结

果如图 8 所示. 可以看到, BankedIPC 中上下文切换带来的开销占比明显下降, 这是因为在使用寄存器组机制后, 需要保存的寄存器窗口数量大幅减少. 此外, 当 IPC 客户端线程和服务端线程都处于同一个处理器核心上的“激活”状态时, 在这两个线程上下文之间进行切换时仅需保存和恢复几个全局寄存器. 注意到相比之下, BankedIPC 的进程切换的开销占比明显增多, 这是因为内核能够使用的寄存器窗口数量也减少了, 性能会有一定程度的下降, 但这部分性能损失低于寄存器组机制带来的性能提升, 本文后面的测试中我们还会进一步研究这一性能损失. 图 8 中“其他”主要是在用户态系统库中的开销, 这部分开销的绝对值在优化前后基本没有变化, 但由于优化后总时延下降, 所以其占比自然有所提升.

(3) RQ3: 内核采用的寄存器组机制能否对非 IPC 场景带来优化效果? 是否存在局限性?

BankedIPC 的核心设计之一在于内核采用寄存器组的机制管理和分配寄存器窗口, 寄存器组机制有助于提升线程上下文切换的性能, 而这也是 BankedIPC 能够显著提升 IPC 性能的关键原因之一. 但上下文切换不仅是 SPARC 架构 IPC 性能的瓶颈, 一些不使用大量 IPC 的应用, 其性能可能也受到 SPARC 架构原本低效的上下文切换的影响. 因此, 我们使用一个简单的多线程测试程序来测试寄存器组机制能否对一些不使用大量 IPC 的应用带来性能提升. 测试应用会创建多个线程以并发地对共享数据进行简单的处理, 每个线程完成一部分处理工作后都会调用 yield 让出执行权以等待其他线程, 每个线程被平均地绑定到各个处理器核心上. 测试结果如图 9 所示.

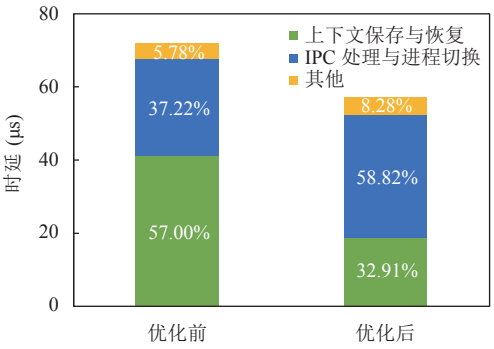


图 8 IPC 时延分布比较

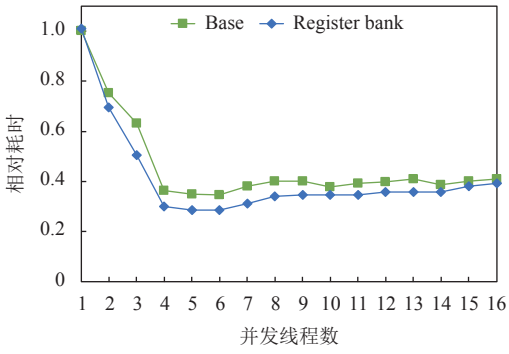


图 9 上下文切换测试结果

图 9 中, Base 表示内核未开启寄存器组, Register bank 表示内核开启寄存器组, 图中的运行时延是相对于 Base 在线程数为 1 时的运行耗时的比值, 运行耗时越少表示性能越好. 当线程数目较少时, 各个线程运行在各自的处理器核心上, 不会发生太多上下文切换, 因此寄存器组带来的优化效果不明显. 随着线程数量的提升, 每个处理器核心都被分配了多个线程, 这些线程之间需要频繁的上下文切换, 在寄存器组机制下, 一个核上的多个线程的上下文可以在寄存器中共存, 占据不同的寄存器窗口, 在上下文切换时仅需保存/恢复少量全局寄存器. 所以此时对于测试程序而言, 寄存器组机制能够带来明显的性能提升, 其中在线程数为 7 时性能提升最为明显, 达到了 23%. 随着线程数量的继续增加, 每个核上运行的线程超出了寄存器窗口中可供容纳的寄存器组的最大值, 因此在内核调度新的线程时, 目标线程的上下文很有可能并不位于寄存器中, 仍然需要从内存中换入并将寄存器窗口中已有的某个寄存器组换出到内存, 导致此时的优化效果降低. 这与 IPC 的场景不同, 因为在 IPC 的场景中, 客户端线程和服务端线程之间进行连续的 IPC 通信时, 两个线程的上下文大多数情况下都会位于寄存器中. 通过测试可以发现, 即使对于不使用大量 IPC 的多线程应用, 内核的寄存器组机制仍然可以加速其线程间上下文切换的效率从而提升应用整体性能, 当应用程序的线程数目适当, 每个处理器核心都有充足的寄存器窗口容纳各线程的上下文时, 这种性能提升更加明显.

虽然寄存器组机制有助于提升上下文切换的性能, 但由于开启寄存器组后限制了应用程序能够使用的寄存器窗口, 因此其函数调用性能会受到影响. 具体来说, 如果应用程序能够使用多个寄存器窗口, 那么在进行函数调用和返回时, 仅需执行 SAVE/RESTORE 指令即可切换到新的寄存器窗口, 从而自由地使用新窗口内的寄存器; 而启

用寄存器组机制后, 则必须限制应用只能使用一个寄存器窗口, 这样每次函数调用时就必须将“被调用者保存(callee-saved)”的寄存器保存到栈上, 从而带来额外的访存开销, 函数返回时同理. 在第 2.1 节中, 我们测试了 nbench 中的应用在仅使用一个寄存器窗口时的性能损失情况, 测试结果如图 2 所示, 这些应用程序为计算密集型, 基本不需要上下文切换, 其中大多数应用程序中性能下降都在 2% 以内. 为了进一步明确函数调用带来的额外开销有多少, 我们设计了简单的测试程序专门测试函数调用带来的额外开销, 测试程序中使用了简单的递归函数来模拟调用链深度. 测试结果如图 10 所示. 正常情况下, 仅使用一个寄存器窗口时, 平均每次函数调用会带来约 6 个时钟周期的额外开销 (在运行测试的硬件平台上相当于大约 30 ns). 因此对于上下文切换 (包括线程切换、系统调用、IPC 等) 次数较少且频繁发生函数调用的应用来说, 寄存器组机制的优化效果可能并不理想. 然而当函数调用深度较高时, 使用多个寄存器窗口的开销反而超过了仅使用一个寄存器窗口, 这是因为当所有寄存器窗口都被使用后, 如果再执行 SAVE 指令, 就会触发 trap, 需要将之前最早被使用过的寄存器窗口清空并保存到栈上, 从而获得新的可供使用的寄存器窗口; 而对于仅使用一个寄存器窗口的应用来说, 不论其函数调用链有多深, 每次函数调用的开销都是相似的. 所以对于函数调用层次较深的应用, 寄存器组带来的额外开销可以得到平衡.

(4) RQ4: FlexIPC 能对 ChCore 上的跨核 IPC 性能带来多少提升?

我们接下来测试了 FlexIPC 对跨核 IPC 的优化作用. 此次测试仍然采取前述的零数据 IPC 以关注控制流转移的性能开销. 为了使用 FlexIPC, 测试程序将 IPC 服务端线程绑定在了单独的一个处理器核心上, 其余客户端线程运行在其他核心上; 未优化的跨核 IPC 使用线程迁移实现, 其客户端线程平均分布在所有核心上. 测试结果如图 11 所示. 可以看到, 即使当 IPC 并发数量很少时, 由于跨核 IPI 的性能问题, 基于线程迁移的 IPC 性能就远差于单核的情况, 但 FlexIPC 可以非常有效地改善这一情况, 并发数较少时 FlexIPC 能够使跨核 IPC 的时延下降 86%. 随着并发数量的增多, 由于原始版本的 IPC 性能瓶颈受限于 IPI 而并非并发数, 所以其性能下降并不严重, 此外由于服务端只有一个线程在进行处理, 客户端线程增加时 FlexIPC 的性能也在逐步下降, 即使如此, 在 32 个 IPC 并发时, FlexIPC 的时延仍然只有原来的 49%.

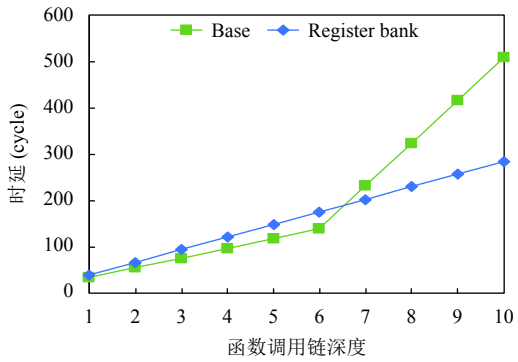


图 10 寄存器组对函数调用的性能影响

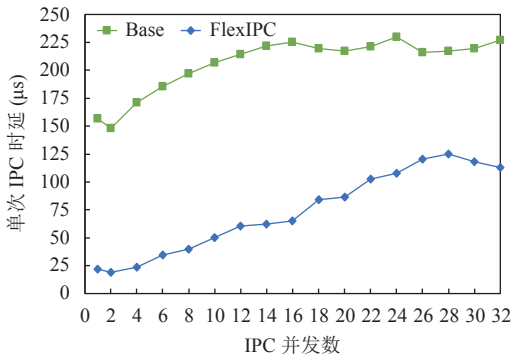


图 11 FlexIPC 时延测试结果

BankedIPC 和 FlexIPC 采用不同的思路对 IPC 性能进行优化. BankedIPC 的核心是使用内核的寄存器组机制减少上下文切换开销, 而 FlexIPC 则是使用用户态轮询的思想避免绕过内核, 直接避免上下文切换的开销. 整体来说, 由于 FlexIPC 必须独占一个处理器核心的资源, 且绕过了内核, 所以其性能提升相较 BankedIPC 来说更加明显, 尤其是在线程数量较少时, FlexIPC 可降低 86% 的时延而 BankedIPC 此时仅能降低约 20% 的时延. 但是在 IPC 请求不密集的时候, FlexIPC 服务端的轮询会导致处理器资源的浪费, 也会影响系统上其他应用程序的性能; 而 BankedIPC 则更加灵活, 对资源的利用率更高, 也对用户更加透明. 因此, FlexIPC 适用于对性能追求更高且需要持续占用资源提供 IPC 服务的应用程序; 而 BankedIPC 则更加通用, 适用于需要按需处理 IPC 请求的场景. 虽然 BankedIPC 主要针对单核场景, 但是也可在跨核 IPC 的场景下单独使用 BankedIPC, 不过这种方案的性能提升并不明显, 原因是传统跨核 IPC 的性能瓶颈主要在于 IPI 而非上下文切换, 且跨核情况下无法利用到 BankedIPC

的寄存器传参机制.

4.3 系统服务测试

微内核上的应用程序大量使用 IPC 的场景一般情况下主要是用于调用系统服务, 比如文件系统、网络服务等, 应用程序在调用文件系统接口时都会向用户态运行的文件系统服务发送 IPC 请求, 于是我们使用在 ChCore 上实现的内存文件系统 tmpfs, 测试了应用程序文件操作的性能. 我们测试了测试程序调用一些经典的文件系统接口时的性能, 比较了优化前 (Base) 以及采用 BankedIPC 方法优化后 (Optimized) 的性能, 以吞吐率作为衡量指标, 并且将测试结果以优化前的性能作为标准进行了规范化处理, 测试结果如图 12 所示, 可见对 IPC 的优化多数情况下能够提升应用在使用系统服务时的性能. 对于一些参数量较小的文件系统接口, 比如 lseek 和 ftruncate 操作, 由于能够将参数全部容纳于寄存器中, 性能提升比较明显, 达到了 15%; 很多文件系统接口都需要传递文件路径作为参数, 由于文件路径长度不确定, 默认使用内存传递, 这样的文件操作性能提升并不明显. 对于一些处理起来比较复杂的文件操作如 symlink, 性能反而出现了下降, 这是因为采用单一寄存器窗口后, 函数调用开销增大, 如果出现较频繁的函数调用则会性能产生影响, 但通过合理的编程设计, 我们认为这一现象应当是并不常见的.

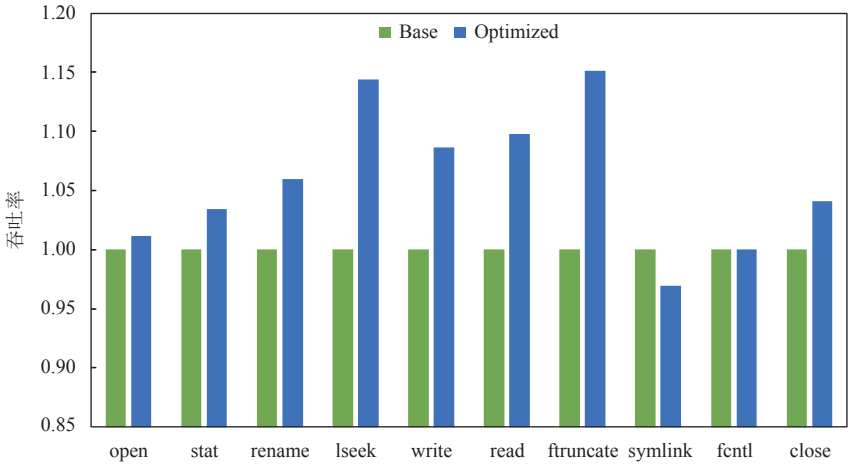


图 12 文件系统服务测试结果

5 总 结

本文提出了对 SPARC 架构上微内核 IPC 的几种优化方法. 为了减小 SPARC 上保存大量寄存器上下文带来的额外开销, 本文设计实现了寄存器组机制, 通过内核对寄存器窗口的分配管理, 限制每个线程对全部寄存器窗口的使用. 本文基于寄存器组机制实现了 BankedIPC, 通过减少上下文切换的高昂开销提升微内核 IPC 性能. 此外, BankedIPC 还可以利用“冗余”的寄存器传递 IPC 参数, 充分利用 SPARC 上丰富的寄存器, 同时实现小数据 IPC 的零拷贝. 对于多核场景, 本文提出了 FlexIPC, 通过轮询共享内存的方式完成 IPC, 避免陷入内核以及 IPI 的发生, 大幅优化了跨核 IPC 的性能. 在自研微内核 ChCore 上的测试表明, 寄存器 IPC 可以将单次小数据 IPC 时延降低至优化前的 53%, 而 FlexIPC 则能够将跨核 IPC 时延降低至优化前的 49%, 而对于使用文件系统服务的应用, 在某些测试场景下性能提升最高可达到 15%.

References:

[1] David FM, Chan EM, Carlyle JC, Campbell RH. CuriOS: Improving reliability through operating system structure. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. San Diego: USENIX Association, 2008. 59–72.

[2] Isaac OA, Okokpujie K, Akinwumi H, Juwe J, Otunuya H, Alagbe O. An overview of microkernel based operating systems. IOP Conf. Series: Materials Science and Engineering, 2021, 1107: 012052. [doi: 10.1088/1757-899X/1107/1/012052]

[3] Marty M, de Kruijf M, Adriaens J, Alfeld C, Bauer S, Contavalli C, Dalton M, Dukkipati N, Evans WC, Gribble S, Kidd N, Kononov R, Kumar G, Mauer C, Musick E, Olson L, Rubow E, Ryan M, Springborn K, Turner P, Valancius V, Wang X, Vahdat A. Snap: A microkernel approach to host networking. In: Proc. of the 27th ACM Symp. on Operating Systems Principles. Huntsville: Association for Computing Machinery, 2019. 399–413. [doi: 10.1145/3341301.3359657]

[4] Bijlani A, Ramachandran U. Extension framework for file systems in user space. In: Proc. of the 2019 USENIX Conf. on USENIX Annual Technical Conf. Renton: USENIX Association, 2019. 121–134.

[5] Ford B, Hibler M, Lepreau J, Tullmann P, Back G, Clawson S. Microkernels meet recursive virtual machines. In: Proc. of the 2nd USENIX Symp. on Operating Systems Design and Implementation. Seattle: Association for Computing Machinery, 1996. 137–151. [doi: 10.1145/238721.238769]

[6] Wu FN, Dong MK, Mo GQ, Chen HB. TreeSLS: A whole-system persistent microkernel with tree-structured state checkpoint on NVM. In: Proc. of the 29th Symp. on Operating Systems Principles. Koblenz: Association for Computing Machinery, 2023. 1–16. [doi: 10.1145/3600006.3613160]

[7] Levin R, Cohen E, Corwin W, Pollack F, Wulf W. Policy/mechanism separation in Hydra. In: Proc. of the 5th ACM Symp. on Operating Systems Principles. Austin: Association for Computing Machinery, 1975. 132–140. [doi: 10.1145/800213.806531]

[8] Shapiro JS, Smith JM, Farber DJ. EROS: A fast capability system. In: Proc. of the 17th ACM Symp. on Operating Systems Principles. Charleston: Association for Computing Machinery, 1999. 170–185. [doi: 10.1145/319151.319163]

[9] Hildebrand D. An architectural overview of QNX. In: Proc. of the 1992 Workshop on Micro-kernels and Other Kernel Architectures. Seattle: USENIX Association, 1992. 113–126.

[10] Reynolds F. An architectural overview of alpha: A real-time, distributed kernel. In: Proc. of the 1992 Workshop on Micro-kernels and Other Kernel Architectures. Seattle: USENIX Association, 1992. 127–146.

[11] Gaisler J. Concurrent error-detection and modular fault-tolerance in a 32-bit processing core for embedded space flight applications. In: Proc. of the 24th IEEE Int'l Symp. on Fault-tolerant Computing. Austin: IEEE Computer Society, 1994. 128–130. [doi: 10.1109/FTCS.1994.315650]

[12] Gaisler J. A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. In: Proc. of the 2002 Int'l Conf. on Dependable Systems and Networks. Washington: IEEE Computer Society, 2002. 409–415. [doi: 10.1109/DSN.2002.1028926]

[13] Själander M, Habinc S, Gaisler J. LEON4: Fourth generation of the leon processor. In: Proc. of Data Systems in Aerospace. Istanbul, 2009. https://www.sjalander.com/research/pdf/sjalander-dasia2009.pdf

[14] Li Z, Li WX, Jin T. Transplantation and application of operation system based on BM3803. Radar Science and Technology, 2016, 14(3): 311–316, 323 (in Chinese with English abstract). [doi: 10.3969/j.issn.1672-2337.2016.03.015]

[15] Jiang XH, Li FH, Qi B. S698 SoC and applications on SPARC. Microcontrollers & Embedded Systems, 2007, 7(8): 84–85 (in Chinese). [doi: 10.3969/j.issn.1009-623X.2007.08.028]

[16] Andersson J, Hjorth M, Johansson F, Habinc S. LEON processor devices for space missions: First 20 years of LEON in space. In: Proc. of the 6th Int'l Conf. on Space Mission Challenges for Information Technology (SMC-IT). Madrid: IEEE Computer Society, 2017. 136–141. [doi: 10.1109/SMC-IT.2017.31]

[17] Tong JG, Anderson IDL, Khalid MAS. Soft-core processors for embedded systems. In: Proc. of the 2006 Int'l Conf. on Microelectronics. Dhahran: IEEE Computer Society, 2006. 170–173. [doi: 10.1109/ICM.2006.373294]

[18] Guzmán D, Prieto M, Sánchez S, Almena J, Rodríguez O, Meziat D. Improving the LEON spacecraft computer processor for real-time performance analysis. Journal of Spacecraft and Rockets, 2011, 48(4): 671–678. [doi: 10.2514/1.50209]

[19] Engler DR, Kaashoek MF, O'Toole J. Exokernel: An operating system architecture for application-level resource management. In: Proc. of the 15th ACM Symp. on Operating Systems Principles. Copper Mountain: Association for Computing Machinery, 1995. 251–266. [doi: 10.1145/224056.224076]

[20] Li CP, Ding C, Shen K. Quantifying the cost of context switch. In: Proc. of the 2007 Workshop on Experimental Computer Science. San Diego: Association for Computing Machinery, 2007. [doi: 10.1145/1281700.1281702]

[21] Liedtke J. A persistent system in real use-experiences of the first 13 years. In: Proc. of the 3rd Int'l Workshop on Object Orientation in Operating Systems. Asheville: IEEE Computer Society, 1993. 2–11. [doi: 10.1109/IWOOOS.1993.324932]

[22] Tsafrir D. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In: Proc. of the 2007 Workshop on Experimental Computer Science. San Diego: Association for Computing Machinery, 2007. [doi: 10.1145/1281700.1281704]

[23] Gu JY, Li H, Li WT, Xia YB, Chen HB. EPK: Scalable and efficient memory protection keys. In: Proc. of the 2022 USENIX Annual Technical Conf. Carlsbad: USENIX Association, 2022. 609–624.

[24] Du D, Hua ZC, Xia YB, Zang BY, Chen HB. XPC: Architectural support for secure and efficient cross process call. In: Proc. of the 46th Int'l Symp. on Computer Architecture. Phoenix: Association for Computing Machinery, 2019. 671–684. [doi: 10.1145/3307650.3322218]

[25] Mi ZY, Li DJ, Yang ZH, Wang XR, Chen HB. SkyBridge: Fast and secure inter-process communication for microkernels. In: Proc. of the 14th EuroSys Conf. Dresden: Association for Computing Machinery, 2019. 9. [doi: 10.1145/3302424.3303946]

[26] Courtaud C, Brandenburg BB. G(IP)^{2}C: Temporally isolated multiprocessor real-time IPC with server-to-server invocations. In: Proc. of the 29th IEEE Real-time and Embedded Technology and Applications Symp. (RTAS). San Antonio: IEEE Computer Society, 2023. 276–288. [doi: 10.1109/RTAS58335.2023.00029]

[27] Bershad B, Anderson T, Lazowska E, Levy H. Lightweight remote procedure call. In: Proc. of the 12th ACM Symp. on Operating Systems Principles. Litchfield Park: Association for Computing Machinery, 1989. 102–113. [doi: 10.1145/74850.74861]

[28] Xia YB, Du D, Hua ZC, Zang BY, Chen HB, Guan HB. Boosting inter-process communication with architectural support. ACM Trans. on Computer Systems, 2021, 39(1–4): 6. [doi: 10.1145/3532861]

[29] Ford B, Lepreau J. Evolving mach 3.0 to a migrating thread model. In: Proc. of the 1994 USENIX Winter Technical Conf. San Francisco: USENIX Association, 1994. 97–114.

[30] Levy HM. Capability-based Computer Systems. Bedford: Digital Press, 1984. 1–250. [doi: 10.1016/C2013-0-01290-X]

[31] Vilanova L, Jorda M, Navarro N, Etsion Y, Valero M. Direct inter-process communication (dIPC): Repurposing the CODOMs architecture to accelerate IPC. In: Proc. of the 12th European Conf. on Computer Systems. Belgrade: Association for Computing Machinery, 2017. 16–31. [doi: 10.1145/3064176.3064197]

[32] Watson RNM, Norton RM, Woodruff J, Moore SW, Neumann PG, Anderson J, Chisnall D, Davis B, Laurie B, Roe M, Dave NH, Gudka K, Joannou A, Markettos AT, Maste E, Murdoch SJ, Rothwell C, Son SD, Vadera M. Fast protection-domain crossing in the CHERI capability-system architecture. IEEE Micro, 2016, 36(5): 38–49. [doi: 10.1109/MM.2016.84]

[33] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S. seL4: Formal verification of an OS kernel. In: Proc. of the 22nd ACM SIGOPS Symp. on Operating Systems Principles. Big Sky: Association for Computing Machinery, 2009. 207–220. [doi: 10.1145/1629575.1629596]

[34] Liedtke J. Improving IPC by kernel design. In: Proc. of the 14th ACM Symp. on Operating Systems Principles. Asheville: Association for Computing Machinery, 1994. 175–188. [doi: 10.1145/168619.168633]

[35] Gu JY, Wu XY, Li WT, Liu N, Mi ZY, Xia YB, Chen HB. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In: Proc. of the 2020 USENIX Conf. on USENIX Annual Technical Conf. Berkeley: USENIX Association, 2020. 27.

[36] Liedtke J. Lazy context switching algorithms for sparc-like processors. 1993. https://os.itec.kit.edu/downloads/publ_1993_liedtke_lazy-context-switching.pdf

[37] Ottlik S. Reducing overhead in microkernel based multiserver operating systems through register banks [MS. Thesis]. Karlsruhe: Karlsruhe Institute of Technology, 2010.

[38] Soares L, Stumm M. FlexSC: Flexible system call scheduling with exception-less system calls. In: Proc. of the 9th USENIX Conf. on Operating Systems Design and Implementation. Vancouver: USENIX Association, 2010. 33–46.

[39] Ma Z, Qiao L, Yang MF, Li SF. Verification of operating system exception management for SPARC processor architecture. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1631–1646 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6241.htm [doi: 10.13328/j.cnki.jos.006241]

[40] Black D, Golub DB, Julin DP, Rashid RF, Draves RP, Dean RW, Forin A, Barrera J, Tokuda H, Malan GR, Bohman D. Microkernel operating system architecture and Mach. Journal of Information Processing, 1991, 14(4): 442−453.

[41] Elphinstone K, Heiser G. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In: Proc. of the 24th ACM Symp. on Operating Systems Principles. Farminton: Association for Computing Machinery, 2013. 133–150. [doi: 10.1145/2517349.2522720]

[42] SPARC Int'l, Inc. The SPARC Architecture Manual: Version 8. Englewood Cliffs: Prentice-Hall, 1992. 202–204.

[43] Weaver DL. The SPARC Architecture Manual: Version 9. Englewood Cliffs: Prentice-Hall, 1994. xviii–xix.

[44] Hsieh WC, Kaashoek MF, Weihl WW. The persistent relevance of IPC performance: New techniques for reducing the IPC penalty. In: Proc. of the 4th IEEE Workshop on Workstation Operating Systems. WWOS-III. Napa: IEEE Computer Society, 1993. 186–190. [doi: 10.1109/WWOS.1993.348151]

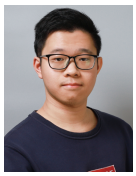
附中文参考文献:

[14] 李钊, 李文新, 金田. 基于 BM3803 的操作系统移植和应用研究. 雷达科学与技术, 2016, 14(3): 311–316, 323. [doi: 10.3969/

j.issn.1672-2337.2016.03.015]

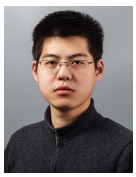
[15] 蒋晓华, 李付海, 祁波. SPARC 体系的 S698 系列 SoC 及其应用. 单片机与嵌入式系统应用, 2007, 7(8): 84–85. [doi: 10.3969/j.issn.1009-623X.2007.08.028]

[39] 马智, 乔磊, 杨孟飞, 李少峰. 面向 SPARC 处理器架构的操作系统异常管理验证. 软件学报, 2021, 32(6): 1631–1646. http://www.jos.org.cn/1000-9825/6241.htm [doi: 10.13328/j.cnki.jos.006241]



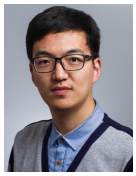
苏浩然(2000－), 男, 硕士生, 主要研究领域为操作系统, 系统安全.

臧斌宇(1965－), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为操作系统, 计算机体系结构.



李文泰(1996－), 男, 博士生, 主要研究领域为操作系统, 系统可靠性.

陈海波(1982－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为操作系统, 并行与分布式系统, 虚拟化, 系统安全.



古金宇(1994－), 男, 博士, 助理研究员, CCF 专业会员, 主要研究领域为操作系统, 系统安全.

管海兵(1971－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为虚拟化, 云计算, 分布式计算.