软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(6):2012−2029 [doi: 10.13328/j.cnki.jos.006566]

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

面向 SGX2 代新型可信执行环境的内存优化系统*

李明煜, 夏虞斌, 陈海波

(上海交通大学 软件学院 并行与分布式系统研究所, 上海 200240)

通信作者: 陈海波, E-mail: haibochen@sjtu.edu.cn

摘 要: 可信执行环境(trusted execution environment, TEE)是一种应用于隐私计算保护场景的体系结构方案, 能为涉及隐私相关的数据和代码提供机密性和完整性的保护, 近年来成为机器学习隐私保护、加密数据库、区块链安全等场景的研究热点. 主要讨论在新型可信硬件保护下的系统的性能问题: 首先对新型可信硬件(Intel SGX2 代)进行性能剖析, 发现在配置大安全内存的前提下, Intel SGX1 代旧有的换页开销不再成为主要矛盾. 配置大容量安全内存引起了两个新的问题: 首先, 普通内存的可用范围被压缩, 导致普通应用, 尤其是大数据应用的换页开销加剧; 其次, 安全内存通常处于未被用满阶段, 导致整体物理内存的利用率不高. 针对以上问题, 提出一种全新的轻量级代码迁移方案, 将普通应用的代码动态迁入安全内存中, 而数据保留在原地不动. 迁移后的代码可使用安全内存, 避免因磁盘换页导致的剧烈性能下降. 实验结果表明: 该方法可将普通应用因为磁盘换页导致的性能开销降低 73.2%–98.7%, 同时不影响安全应用的安全隔离和正常使用.

关键词: 机密计算; 可信执行环境; 系统安全; 性能优化

中图法分类号: TP311

中文引用格式: 李明煜, 夏虞斌, 陈海波. 面向 SGX2 代新型可信执行环境的内存优化系统. 软件学报, 2022, 33(6): 2012–2029. http://www.jos.org.cn/1000-9825/6566.htm

英文引用格式: Li MY, Xia YB, Chen HB. Memory Optimization System for SGXv2 Trusted Execution Environment. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 2012–2029 (in Chinese). http://www.jos.org.cn/1000-9825/6566.htm

Memory Optimization System for SGXv2 Trusted Execution Environment

LI Ming-Yu, XIA Yu-Bin, CHEN Hai-Bo

(Institute of Parallel and Distributed Systems, School of Software, Shanghai Jiaotong University, Shanghai 200240, China)

Abstract: Trusted execution environment (TEE) is an architectural solution for secure computing that requires confidentiality and integrity for private data and code. In recent years, TEE has become the research hotspot for machine learning privacy protection, encrypted database, blockchain security, etc. This study addresses the performance problem of the system under this new trusted hardware. The performance of the new trusted hardware, i.e., Intel SGX2, is analyzed. It is found that the paging overhead in SGX1 is no longer the main issue in SGX2 under the premise of configuring large secure memory. However, the setup of large secure memory leads to two new problems. First, the available range of normal memory is narrowed down, which increases the memory pressure of normal applications, especially big data applications. Second, secure memory is usually underutilized, resulting in low overall physical memory utilization. To solve the above issues, this study proposes a new lightweight code migration approach, which dynamically migrates the code of normal applications into secure memory, while leaving the data in place. The migrated code can use secure memory and avoid the drastic performance degradation caused by disk paging. Experimental results show that the proposed approach can reduce the runtime overhead of normal applications by 73.2% to 98.7% without affecting the isolation and the use of secure applications.

Key words: confidential computing; trusted execution environment; system security; performance optimization

* 基金项目: 国家杰出青年科学基金(61925206); 上海市“科技创新行动计划”(21511101502)
本文由“系统软件安全”专题特约编辑杨珉教授、张超副教授、宋富副教授、张源副教授推荐.
收稿时间: 2021-09-05; 修改时间: 2021-10-15; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

信息文明社会下, 数据保护和数据的安全计算愈发成为国家、企业乃至个人的重点关注对象. 2020 年 4 月, 《中共中央国务院关于构建更加完善的要素市场化配置体制机制的意见》指出, 数据也是一种生产要素, 和土地、劳动力、资本、技术等传统要素并列. 随着“数字中国”的建设, 国家陆续出台了多项法案来保证隐私安全. 2021 年 1 月开始实施的《中华人民共和国民法典》专门讨论了“隐私权和个人信息保护”的具体条例. 2021 年 9 月开始实施的《中华人民共和国数据安全法》也明确了如何规范数据处理活动并保证数据安全. 云计算平台在处理涉及隐私数据的计算时, 平台有义务保证隐私数据不被泄露和沉淀, 否则将追究相应责任.

对于数据的合规计算和隐私保护, 当前主流的研究方案被称为隐私计算[1]. 隐私计算主要分为机密计算、多方计算、差分隐私、联邦学习等技术手段. 其中, 机密计算是由硬件提供的可信执行环境(trusted execution environment, TEE)作为支持. 可信执行环境为敏感数据提供“可用不可见”的密态计算服务, 对计算任务本身没有特殊要求, 允许将现有应用程序加载到可信执行环境中保护执行, 具有较好的通用性和兼容性. 机密计算的主体是可信芯片, 包括了 Intel SGX[2,3]、AMD SEV[4]、Intel TDX[5]、ARM Realm[6]等. 在 RISC-V 方面, 国内有“蓬莱”[7]. 总体而言, 可信执行环境呈现出“百家争鸣”的态势.

本文关注目前工业界较为成熟的 Intel SGX, 其提供了物理隔离、内存加密、远程认证、密封存储等一系列安全特性(本文第 1 节). Intel SGX 目前已被广泛部署在阿里云、亚马逊云、谷歌云、微软 Azure、IBM 云等各大主流公有云基础设施之上[8], 用于机器学习隐私保护、加密数据库、区块链安全等热点场景. 现有部署的主要是 SGX1 代系统, 但是因为 SGX1 代服务器的安全内存极为有限(仅有 128 MB 或 258 MB 安全内存), 存在较大的性能问题, 性能开销甚至可达 6–11 倍[9,10].

本文面向新一代可信执行环境——SGX2 代云服务器, 于 2021 年 4 月发售. 本文是第一个对 SGX2 代性能进行完整量化研究的工作. 我们对 SGX2 代云服务器进行了完整的性能评测(第 2 节). 经过评测, 发现原先在 SGX1 代的主要矛盾发生了转移, 新的硬件特性同时带来了机遇和挑战. SGX2 代允许配置超大容量的加密内存, 原先第一代的安全内存有限的性能问题不复存在, 但与此同时, 却导致了操作系统和普通应用程序可用的普通内存被极大压缩的问题. 普通内存压缩的后果是普通应用和操作系统的内存压力增大, 发生换页事件的概率和次数会大幅度增加. 一旦操作系统受换页事件影响产生较大的停滞时间, 安全应用的性能也随之下降, 因为安全应用的调度归属于操作系统. 归根结底, 大容量的安全内存导致了物理内存的利用率下降.

针对这一问题, 本文提出了轻量级的代码迁移方案(第 3 节): 对于需要大量内存资源的普通应用, 在系统触发换页事件之前, 本文系统将普通应用的代码快速迁移入安全内存中. 迁移后的应用程序可以同时访问普通内存和安全内存, 从而提高物理内存的资源利用率, 有效避免了换页事件导致的系统性能影响. 我们使用内存密集型的负载对本系统进行了评测(第 4 节). 实验结果表明: 本文提出的轻量级解决方案可以将物理内存资源利用率提高 33.2%–58.6%, 迁移时间小于 10 ms, 同时可将应用程序的性能开销降低 73.2%–98.7%.

本文考虑新一代的可信执行环境安全系统——Intel SGX2 代服务器, 主要贡献包括:

(1) 首次量化分析了新一代的可信执行环境 Intel SGX2 代服务器的性能指标, 发现 SGX1 代上的普遍关注的安全内存受限问题不再成为主要矛盾;

(2) 指出 SGX2 代服务器的新的性能瓶颈: 配置大容量安全内存导致系统整体物理内存利用率下降, 同时, 操作系统和普通应用的换页概率大幅增加;

(3) 提出一种轻量化的代码迁移方案, 能够有效地避免系统中内存换页的开销和内存利用率不足的问题, 同时, 迁移开销与应用本身的数据大小无关;

(4) 开发了原型系统, 实验评估表明, 本文提出的新方案能够降低 73.2%–98.7%的性能开销, 证实了本文方案的有效性和实用性.

本文第 1 节介绍 SGX 的背景知识, 以及对已有的性能优化方案和相关工作进行梳理. 第 2 节对 Intel SGX2 代服务器进行性能评测, 对原有关注的性能问题进行剖析, 发现安全内存的性能问题已被新硬件很好地解决, 同时指出新的性能问题(即内存利用率和普通内存的换页问题). 第 3 节提出一种全新的轻量化代码迁移方案, 描述本文的观察、系统架构和工作流程. 第 4 节通过真实应用对本文方案进行实验评估, 阐明该方法的高效

性和实用性. 第 5 节介绍与本文有关的相关工作. 第 6 节对本文工作进行总结.

1 背景知识

1.1 Intel SGX的安全特性

作为一款成熟的可信执行环境方案, Intel SGX 允许进程在用户态创建一块“飞地(enclave)”. Enclave 运行在进程地址空间中, 能够对应用程序的关键隐私数据和对应处理逻辑提供强有力的保护支持. 具体而言, Intel SGX 提供了四大安全特性.

1. 物理隔离

Intel SGX 服务器要求在系统启动之初, 将物理内存进行显式划分. 划分的寄存器称为“处理器预留内存寄存器(processor reserved memory register, PRMR)”, 具体如图 1 所示. 一旦静态划分好, 在系统运行阶段便不再允许进行任何调整. 没有动态调整的可能, 便可有效防止在运行时被攻击的危险. 处理器预留内存, 即 PRM, 存放了 Enclave 的内存页(enclave page cache, EPC), 这些 EPC 负责存放 Enclave 的代码和数据. PRM 的主要特点在于: 运行在处理器上的任何非飞地(non-enclave)软件, 包括具有特权的操作系统内核、虚拟机监视器(hypervisor)、底层固件, 都无权查看 PRM 内的信息, 从而保证了软件层面的强制隔离. 拥有 Enclave 的进程本身, 其非飞地部分, 即普通内存部分, 也无法访问 Enclave. 这样可以保证程序自身的 bugs 不会影响 Enclave 的机密性和完整性, 适合保护诸如 SSL 的加解密处理逻辑, 防止诸如“心脏滴血(heartbleed)”[11]的漏洞.

2. 内存加密

只进行物理隔离是不够的, PRM 内保存的信息还可能通过物理暴力手段获得. 由于 SGX 安全特性的最初目的是为了保护云上的用户数据, 云服务器的所有者可以对 PRM 的内存数据进行获取. 例如, 将内存条换为具有持久性能力的非易失性内存(non-volatile memory, NVM), 进而通过停机等手段轻松获取 PRM 的内存快照. 操作系统还能通过控制设备的 DMA 来访问任意物理内存信息, 从而窃取 PRM 内的用户隐私. 为此, Intel SGX 处理器专门引入“内存加密引擎(memory encryption engine, MEE)”, 如图 1 所示. MEE 负责对 PRM 内的数据进行自动加解密, 数据只有在进入 CPU 核上的缓存才是明文, 否则, 在内存中始终以密文形式存在. 借助 MEE 的内存加密, Enclave 可以有效阻止针对物理内存的攻击.

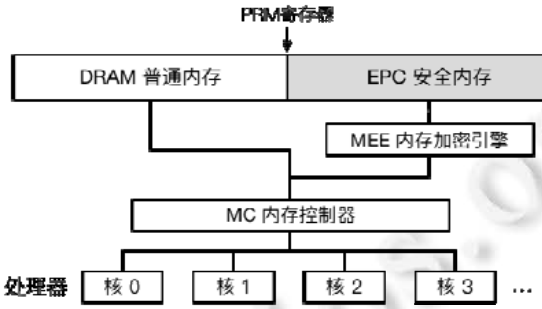

图 1 SGX 服务器上的物理内存划分架构

3. 远程认证

为了给用户提供数据并计算确实保护在 Enclave 中的凭证, Intel SGX 处理器提供了远程认证的能力. 处理器对于任何一个启动的 Enclave 实例, 都会在加载阶段计算其加载内容的哈希, 内容包括了所有代码段和全局数据段的所有状态, 随后生成由处理器密钥签名后的凭证. 用户首先检查远程认证的凭证是否由 SGX 处理器所签名, 随后根据凭证内的哈希判断 Enclave 实例状态的可信性. 借助远程认证, 管理员无法伪造一个恶意或非法的 Enclave 偷取用户信息.

4. 密封存储

由于 Enclave 只能工作在用户态, 一旦 Enclave 所在进程被杀死, 则状态全部丢失. 为此, SGX 处理器为

Enclave 提供了一种安全的持久化服务, 称为“密封机制(sealing)”, 即利用处理器内部的根密钥对 Enclave 数据进行加密. 加密后的数据可以存储到不可信操作系统管理的文件系统中, 随后, 如果同一个 Enclave 实例被再次启动, 则可以成功解密并读取“密封存储”后的持久化数据. 为了保证密封存储的安全性, 首先, 处理器的根密钥不可被任何软件所访问; 其次, 不同 Enclave 实例的密封密钥也不相同; 最后, 不同处理器上的同一 Enclave 二进制对应的密封密钥也不同, 从而保证了 Enclave 持久化数据的绝对安全.

1.2 SGX1代服务器的性能瓶颈

安全通常伴随着性能方面的牺牲, SGX 也不例外. 为了在体系结构层次提供如上的四大安全特性, Intel 处理器提供了一系列安全指令, 同时对 Enclave 的安全状态进行了严格检查. 整套模型还进行了形式化验证.

由于 SGX2 代服务器的发布时间较晚(2021 年 4 月 7 日才正式发布), 因此, 现有的对于 SGX 的性能优化已发表工作全部都是针对 SGX1 代机器的. 主要关注的性能问题有如下几个方面(如图 2 所示).

(1) 模式切换: 我们称具有 Enclave 保护区间的应用程序为安全应用. 因为安全应用的非 Enclave 部分无法直接访问其 Enclave 部分, 其隔离类似现有的“内核态/用户态”的划分, 因此要执行 Enclave 的代码和访问 Enclave 的数据需要进行显式的模式切换. 从 non-enclave 切换进 enclave 时(称为 ecall), 处理器会检查入口点(entry point)是否合法, 同时关闭处理器的调试能力, 防止任何可能的信息被盗取. 从 enclave 切换出 non-enclave 时(称为 ocall), 处理器需要将“转址旁路缓存”TLB 全部刷掉, 保证 EPC 安全内存不能被外部不可信部分所访问. 这些安全措施都导致较大的性能开销;

(2) 内存换页: 在 SGX1 代服务器上, EPC 安全内存的大小非常有限, 只有 128 MB, 少数服务器配置 256 MB 的安全内存大小. 对于现代程序而言, 如果在 EPC 中申请较大的堆内存加以使用, 则会引起大量的换页开销. SGX 最早的工作是如何将不经修改的二进制程序直接运行在 SGX Enclave, 利用了用户态的库操作系统(Library OS 或 LibOS)将二进制完全加载到 EPC 中[12]. 这种做法将整个程序完全跑在 Enclave 里, 其内存消耗远大于 128 MB 的可用物理安全内存. 一旦发生换页(swapping), 必须由内核参与协助, 首先包含了模式切换(必须从 enclave 切出到用户态, 再到内核态换页子系统); 其次, 换页的操作包含了 EPC 数据的重新加密. 由于安全应用在运行时经常出现缺页现象, 因此安全内存的换页事件几乎无时无刻都在发生, 开销非常巨大.

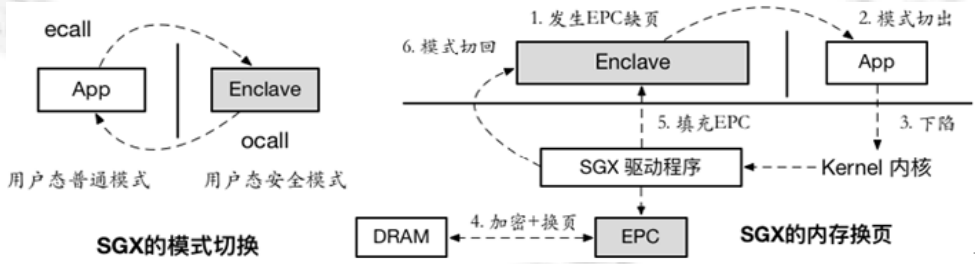

图 2 SGX Enclave 的模式切换和内存换页示意图

对于如上两大问题, 我们简要回顾已有的优化工作.

(1) 模式切换: 由于 SGX 安全无法执行系统调用, 因此必须借助模式切换请求内核服务. Haven[12]借助 LibOS 的设计将调度、内存管理等系统服务放入 Enclave 中, 以减少向外部的内核请求服务. Ryoan[13]在 Enclave 中实现内存文件系统(in-memory file system, IMFS), 进一步将频繁的文件操作在 Enclave 中直接完成. SCONE[14]和 HotCalls[15]则引入了异步调用的概念, 在 Enclave 和底层内核建立了共享缓冲队列, 利用多核的特点允许甲核在 Enclave 提出请求、乙核在内核态响应请求, 从而避免了单核情况下的频繁模式切换;

(2) 内存换页: SGX1 代 EPC 安全内存有限的问题饱受诟病. Eleos[10]率先发现 SGX 的“访存非对称性”特

点(如图 3 所示), 利用该特点, Eleos 实现了基于软件的用户态换页机制, 以普通内存为后端, 安全应用负责自身的数据加解密和自动换入换出, 全过程不发生模式切换和内核换页操作, 获得了 2–3 倍的性能提升. CoSMIX[16]则将这一软件换页的过程实现为编译器自动插桩的做法, 进而不必修改应用程序.

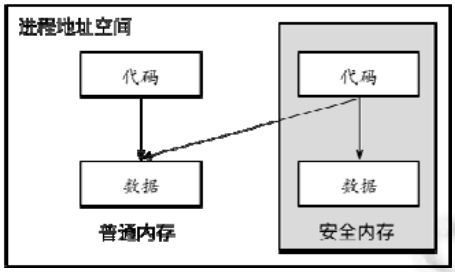

图 3 SGX 内存保护机制的“访存非对称性”模型

关于 SGX1 代机器上的一系列优化工作的分类和对比, 我们推荐读者阅读董春涛等人的相关综述[17].

1.3 SGX2代服务器的新特性

SGX2 代的规范自 2016 年发布[3], 直到 2021 年才正式发布支持 SGX2 代的云服务器[18]. 我们这里简要介绍 SGX2 代的新特性, 以便读者更好地理解本文后续用到的一些观察和技术点.

1) 动态更新

SGX1 代服务器不允许一个 Enclave 安全应用在运行时动态修改其内存权限、扩大其内存范围, 甚至不允许增加其最大支持的安全线程数量, 这些局限在 SGX2 代中都被合理地解决了. SGX2 代允许安全应用使用新指令, 按需地在特定区间内添加安全内存. 该特性在文档中被称为“动态安全内存管理(enclave dynamic memory management, EDMM)”. 这一特性有如下好处.

(1) 动态加载: 启动时不必一次性加载所有代码, 适用于现代软件工程中广泛使用的动态加载技术;

(2) 内存扩展: 安全应用往往无法知道自己运行时真正需要的内存页大小, 因此允许 SGX2 代安全应用动态扩展其内存页的做法可以很好地适配现代版堆分配器(heap allocator);

(3) 即时编译(just-in-time, JIT): SGX2 代允许动态修改页权限, 因此可以支持高级语言的即时编译技术, 将数据页变为代码页.

2) 大容量安全内存

SGX2 代的另一大改动是舍弃了原先的硬件哈希树设计, 不再对 EPC 安全内存进行完整性保护, 但引入了基于错误检查纠正内存(error checking and correcting memory, ECC memory)的校验机制和基于 Cache 的访问控制权限[18]. SGX2 代的安全保证见表 1. SGX2 代对软件层面的保护没有放松, 而是在硬件层面的妥协允许将加密内存的范围扩大至最大 1 TB 的范围, 这就有效解决了原先在 SGX1 代上的频繁换页的性能瓶颈.

表 1 SGX2 代的安全保证

保护措施	攻击方	SGX1 代云机器	SGX2 代云机器
保密性: 防止内容被窃听	软件	能	能
完整性: 防止内容被篡改	软件	能	能
防重放: 防止状态被回滚	软件	能	能
保密性: 防止内容被窃听	硬件	能	能
完整性: 防止内容被篡改	硬件	能	不能
防重放: 防止状态被回滚	硬件	能	不能

本文第 2 节将着重评测新一代可信执行环境——SGX2 代服务器的实际性能, 比较其在 SGX1 代上的性能瓶颈是否依旧存在.

2 SGX2 代服务器的性能分析

2.1 环境配置

根据我们的调研, 2021 年 7 月之前, 全球仅有阿里云提供了 Xeon3 服务器的公有云服务. Xeon3 服务器支持 SGX2 代安全特性, 允许配置大容量的安全内存. 我们通过阿里云得到一台分配有 EPC 安全内存的“安全增强通用型实例”虚拟机进行测试, 具体的 SGX2 代机器参数如下.

- 处理器: Intel Xeon Platium 8369B CPU @ 2.7 GHz;
- 缓存大小: 48 K L1d, 32 K L1i, 1280 K L2, 48 MB L3;
- 内存大小: 21 GB 的普通内存, 23 GB 的安全内存;
- 网络带宽基础: 2 Gb/s;
- 云盘带宽基础: 1.5 Gb/s.

测试环境为虚拟化环境, 虚拟机监视器为 KVM, 逻辑线程 vCPU 一共分配了 24 个. 客户机操作系统为 CentOS 7, Linux 内核版本为 4.19. 评测所使用的 Linux SGX SDK 和 SGX 驱动的版本都是 2.1.

作为对比, 本测评还包括了 SGX1 代的机器, 其机器参数如下.

- 处理器: Intel Core i7-7567U CPU @ 3.5 GHz;
- 缓存大小: 32 K L1d, 32 K L1i, 256 K L2, 4 MB L3;
- 内存大小: 16 GB 的普通内存, 128 MB 的安全内存.

由于我们不在 SGX1 代机器上进行网络和存储的测评, 因此不再列出网络和存储的配置.

据我们所知, 本文研究是第一个对 SGX2 代服务器进行完整性能评测并给出具体量化值的工作. 为了避免测量误差, 我们对微观评测(第 2.2 节–第 2.4 节)测量 5 000 次取平均值, 对宏观评测(第 2.5 节、第 2.6 节)测量 10 次取平均值. 微观评测不受虚拟化环境的影响, 宏观评测的内存访存延迟和应用程序性能则会受到虚拟化波动的影响, 但能反映出真实情况, 因为 SGX 的设计初衷就是为了保护公有云上的隐私数据.

2.2 微观指令评测

在本节中, 我们首先对 SGX2 代机器的指令微观开销进行评估, 测量使用的是处理器直接提供 RDTSC (read time-stamp counter)指令, 该指令可用于计量每条指令的微观开销, 即具体消耗的时钟周期. 作为对比, 我们将 SGX1 代的指令开销也列出来, 具体见表 2.

表 2 SGX1 代和 SGX2 代的微观指令开销(单位: 时钟周期)

指令	作用	SGX1 代服务器	SGX2 代服务器
ecreate	创建 enclave 实例	29 K	30 K
eadd	给 enclave 实例添加一页 EPC	10 K	5.3 K
eremove	给 enclave 实例删除一页 EPC	3.2 K	1.4 K
eextend	对 256 B 的 EPC 计算哈希	7.0 K	2.6 K
einit	完成对 enclave 实例的初始化	80 K	58 K
eenter	跳入 enclave 安全模式	26.8 K	9.2 K
eexit	跳出 enclave 安全模式	12.5 K	8.3 K
emodpr	动态修改 EPC 页面权限	无	3.5 K
emodt	动态修改 EPC 页面类型	无	3.5 K
eaug	运行时动态增加一页 EPC	无	8.3 K

由上可知, SGX2 代机器在微观层面上的性能要普遍好于 SGX1 代机器, 同时支持了更多指令和特性.

2.3 模式切换评测

由第 1.2 节可知, 模式切换(context switch)是 SGX 的一大开销来源. 由于跑在 SGX Enclave 无法直接执行系统调用或 I/O, 因此必须借助上下文切换回到普通用户态(user-mode, non-SGX)下. 本节对 SGX2 代机器的模式切换进行测试, 同样给出 SGX1 代的数据进行对比(见表 3).

表 3 SGX1 代和 SGX2 代的上下文切换开销

模式切换	作用	SGX1 代服务器(μs)	SGX2 代服务器(μs)
ecall	从 non-enclave 切换入 enclave	8	3
ocall	从 enclave 切换入 non-enclave	3	3

由表 3 可得, SGX2 代机器在模式切换上亦有提升, 其中, ecall 的开销下降了 62.5%.

2.4 访存开销评测

SGX 将物理内存静态划分为安全内存和普通内存. 二者在访存上的延迟并不相同, 原因是安全内存需要进行加密保护. 本节对安全内存的访问延迟进行了测试, 结果见表 4.

表 4 SGX2 代服务器的访存开销

访问数据量	普通内存访存开销	安全内存访存开销	安全/普通内存访存的开销百分比(%)
256 K	3 μs	3 μs	0.0
1 MB	15 μs	16 μs	6.6
16 MB	533 μs	564 μs	5.8
64 MB	3.9 ms	4.2 ms	7.7
256 MB	20.1 ms	21.5 ms	7.0

由评测结果可知, 安全内存的访问有一定访存开销, 相比于普通内存的访问开销在 10%以内.

2.5 运行时动态内存扩展时间评测

由于 SGX2 代允许在运行时动态扩展内存(这是 SGX1 代所不具备的), 这里评测 SGX2 代运行时申请内存的时间开销. SGX2 代运行时内存扩展使用 EAUG-EACCEPT 指令, 权限默认为可读可写. 由图 4 可知, SGX2 代动态内存扩展的时间随内存申请大小呈正相关趋势. 为了方便比较, 图 4 同样给出了普通内存的运行时动态内存扩展时间(使用 C 标准库的 malloc 接口), 包含了对目标内存大小的整体缺页中断时间.

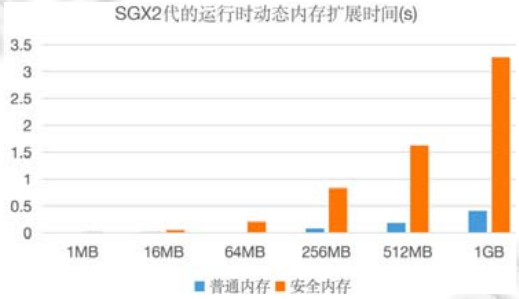

图 4 SGX2 代服务器的安全内存和普通内存的运行时动态扩展时间对比

图 4 测得的运行时动态内存分配时间同时适用于堆内存(heap)和栈内存(stack)的动态扩展. 通常, 影响应用程序运行时内存性能的主要原因是堆内存的使用, 栈内存的动态扩张速度远小于堆.

考虑到动态内存测试的完整性, SGX2 代还提供了安全线程数量的动态扩展. 其本质是让应用程序在操作系统的帮助下首先创建一个普通线程的“线程本地存储(thread local storage, TLS)”, 然后将 EPC 中的某一普通页(regular page)转换为 SGX 线程控制结构(thread control structure, TCS). 普通线程切换入 SGX 模式后, 使用新转换的 SGX TCS 即可完成安全线程的动态扩展. 经过微观评估测得, 普通线程和动态线程在 SGX2 代机器上的动态增加时延均在 120 μs 左右. 之所以安全线程的动态创建无明显开销, 原因是线程创建的多数工作在操作系统内核层面完成, 而 SGX 部分只负责一页 EPC 的页面类型转换(即 emodt 指令).

2.6 大内存应用程序评测

SGX1 代因为只支持 128 MB 或 256 MB 大小的安全内存而导致使用性能不佳. 我们这里运行若干大内存应用程序来评测 SGX2 代的大安全内存是否能够有效提升安全应用程序的使用性能.

首先, 我们分别在 SGX1 代和 SGX2 代的安全内存中运行 Tensorflow 机器学习训练程序, 分别是 CNN(卷积神经网络)、RNN(循环神经网络)、GAN(生成对抗网络)这 3 个程序. 测试结果如图 5 所示.

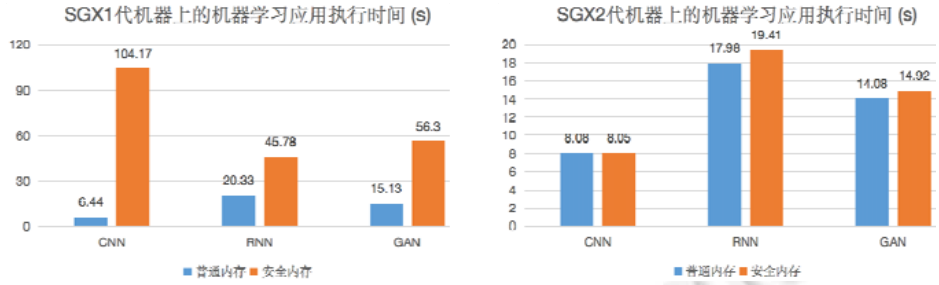

图 5　SGX1 代和 SGX2 代下的机器学习程序的执行时间对比

对应的内存使用情况见表 5.

表 5　用于测试的机器学习程序的内存使用情况

机器学习程序	内存占用大小(GB)	数据段(GB)	代码段(MB)
CNN	1.49	1.36	131
RNN	1.14	1.01	131
GAN	1.12	0.99	131

结果表明: 在 SGX1 代机器上, 基于安全内存与基于普通内存的 ML 程序执行时间相比, 开销在 1–16 倍不等; 而在 SGX2 代上, 执行开销则在 1%–27%之间. 由此可知: SGX2 代的大安全内存确实有效缓解了大内存应用程序的性能问题, 很大程度上提高了可信执行环境的实用性. 主要原因在于, SGX2 代的大安全内存有效避免了原 SGX1 代的频繁 EPC 换页问题. 由于换页需要内核参与, 因此存在大量的模式切换和数据重新加密与数据拷贝的处理器行为. 模式切换导致的 TLB 失效, 更是进一步恶化了大内存应用程序的执行效率.

2.7　评测总结

综合以上评测, 我们发现: SGX2 代服务器已将两大性能问题的主要矛盾解决掉了一半, 即频繁的安全内存换页问题, 主要是在硬件层面放弃哈希树, 从而允许配置大容量安全内存. 而模式切换的问题尽管仍然存在, 但可以由软件层面的异步调用来解决[14,15].

为了避免因为安全内存分配不足导致的性能开销, 运维人员倾向于将安全内存配置为较大值. 从公开的阿里云官网的“安全增强通用型实例”的规格说明来看, 安全内存在物理内存中的占比均默认配给为 50%[19]. 这确实能够解决安全应用的性能问题, 但却也引入了新的问题. 在 SGX2 代机器上, 性能方面的主要矛盾已经发生了转移.

2.8　问题描述

首先, 我们通过触发 SGX2 代机器上的 EPC 换页, 查看其换页开销是否如 SGX1 代那样昂贵, 用以说明配置大容量安全内存的必要性. 评测结果如图 6 所示.

图 6　SGX2 代的 EPC 换页开销评测(单位: s)

评测结果表明: SGX2 代的换页开销依然很大, 安全内存的换页事件对安全应用的时延影响有 3–7 倍. 因此, 管理员确实有必要配置大容量的安全内存.

值得注意的是: 一旦物理内存被显式划分为普通内存和安全内存后, 安全内存则不再能被操作系统和普通应用(我们称非安全应用为普通应用)所使用. SGX2 代机器上性能方面的主要矛盾发生了转移. 为了说明这一情况, 我们统计了 SGX2 代云机器在一天内安全内存和普通内存的利用率(出于保护客户数据的角度考虑, 我们没有公开这一数据集). 从内存利用率的统计中可以得出如下结论.

(1) DRAM 普通内存: 配置大容量 EPC 安全内存严重压缩了 DRAM 正常内存的可使用范围, 导致普通应用有更大概率因为内存不足(out-of-memory, OOM)而触发交换事件, 甚至被操作系统杀死;

(2) EPC 安全内存: 由于系统配置并预留(reserve)了大量 EPC 安全内存, 导致大量 EPC 资源大部分时期处于闲置状态, EPC 安全内存利用率不高.

在实际使用中, 如大数据分析(如 spark)、机器学习(如 tensorflow)、内存键值存储(如 redis)都可能消耗大量物理内存. 事实上, 普通内存的使用包括所有普通应用程序和操作系统内核, 因此极可能出现普通内存不够的情况.

综上, 我们认为: 出于对安全内存性能的考量, 有必要配置大容量安全内存, 但是大容量安全内存导致普通应用因内存不足带来的性能下降问题也应得到解决. 本文认为: 解决的首要任务是如何提高系统的整体物理内存利用率, 以尽可能地避免内存换页事件被触发.

3 系统设计——轻量级代码迁移方案

针对上文提出的普通内存可用范围小, 而安全内存利用率低的主要问题, 本节具体描述本系统的设计方案——一种轻量化的代码迁移方案.

3.1 观 察

在讨论本文提出的设计方案前, 我们首先介绍两点对本文场景下的具体观察. 这两点观察为本文的解决方案提供了有利机会.

(1) 安全内存内的代码段能够访问普通内存的数据段: SGX 提供的内存保护机制具有“访存非对称性”的特点(如图 3 所示), 即安全内存代码能够访问外部非安全内存的数据, 而非安全内存的代码不具有访问非安全内存数据的能力, 这一非对称性的特点提供了独特的优化机会. 先前的相关工作, 如 Eleos[10]和 CoSMIX[16], 为了在纯软件上拓展安全内存边界, 利用这种非对称性, 借助 Enclave 内部的软件级加密引擎将隐私数据加密写入非安全内存, 从而避免了 SGX1 代安全内存访存限制(128 MB 或 256 MB)引起的性能开销. 不同于 Eleos 和 CoSMIX 将数据从安全内存迁移出非安全内存(外向迁移), 本文则是将代码从非安全内存转移进安全内存(内向迁移), 而原先的非安全内存数据依旧能被访问. 这是因为 SGX Enclave 工作在用户态的进程中, 非安全内存和安全内存共享同一虚拟地址空间, 因此非安全内存的数据是内外代码均可访问的;

(2) 应用程序的代码段大小远小于数据段: 经测算, 本文在第 2.6 节测试的 3 个应用程序的实际代码段大小都远小于其数据段大小, 具体测算结果见表 5. 事实上, 现代软件经过工具链的优化, 如消除无用代码(dead code)、链接阶段优化(link-time optimization, LTO)等, 二进制可执行文件和共享库文件的大小均得了很好的裁剪. 例如源代码在千万行级别的 Linux, 其内核核心文件(vmlinuz)大小也不到 10 MB. 而由于现代内存大小的不断增长, 程序开发者在内存使用上(尤其是堆内存)不如以前那么束手束脚, 因此更倾向于大量开辟堆内存的使用. 加之现代软件优化大量使用基于缓存(caching/buffering)的优化思路, 将热数据缓存在内存中, 避免了频繁 I/O 导致的影响, 也进一步扩大了数据段的大小. 而本文讨论的应用场景, 更是因为大数据的使用或低时延的需求, 导致应用程序有大量数据内存的需求.

3.2 架构描述

本节描述本文系统的整体架构, 具体架构图如图 7 所示(其中, 橙色部分为本系统组件). 本文系统共分为两个组件.

(1) 迁移守护进程: 迁移守护进程是普通的用户态进程, 自身不使用安全内存. 迁移守护进程负责从操作系统内核获取普通内存和安全内存的使用量信息, 并定期地对内存用量大的普通应用程序进行排序, 在普通内存的使用量进入特定阈值时(如 95%)会被操作系统调度, 并进行代码迁移处理. 迁移守护进程在操作系统上属于特权进程(root process), 因此能够直接探测任意普通应用程序进程, 进入其地址空间对其执行代码迁移操作. 尽管迁移守护进程属于特权进程, 但无法控制安全应用, 因此不存在安全问题;

(2) 库操作系统(library OS 或 LibOS): 库操作系统 LibOS (本文接下来简称 LibOS)用于在安全内存区域管理 Enclave 飞地, 其自身驻留在安全内存中. 因为迁移守护进程无法负责迁移入 EPC 安全内存的程序代码, 因此需要有 LibOS 负责接管照应. LibOS 与其他安全应用处于不同的 Enclave 实例, 因此不存在迁移后的普通应用代码能够访问其他安全应用数据的机会, 从而保证不同应用间的安全隔离. LibOS 主要负责对迁入 EPC 的代码的“照看”工作, 包括提供系统调用的转发以及内存分配的分发等. 所谓内存分配的分发, 即 LibOS 会根据普通/安全内存的使用状况, 合理地选择可用的内存区块给应用程序(详见第 3.5 节). 被迁入 EPC 的普通应用代码能够同时访问普通内存和安全内存, 因此能够有效地解决物理内存利用率的问题.

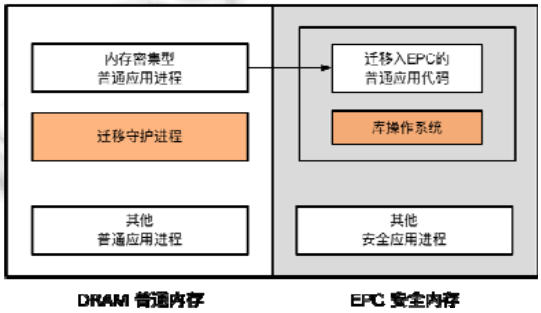

图 7 本文系统的整体架构

3.3 工作流程

本文提出了轻量级的代码迁移方案, 在应用程序面临普通内存逼近临界值时, 即马上要触发系统对交换分区的使用时, 自动唤醒本系统的代码迁移机制, 利用闲置的安全内存资源. 本系统的具体工作流程如下.

(1) 操作系统实时观察普通内存和安全内存的剩余可用量. 在普通内存即将超过使用阈值时, 判断安全内存是否存在可用空间: 若有, 则触发代码迁移; 若无, 则必须使用以磁盘为主的交换分区;

(2) 操作系统扫描每个普通应用程序的内存工作集(working set)大小, 找出普通内存使用大户, 我们称该普通应用为 NApp, 准备为 NApp 进行代码迁移工作;

(3) 操作系统调度本系统提供用户态迁移守护进程(daemon), 守护进程被唤醒后, 向待迁移的 NApp 发送 ctrl-z 信号, 将整个进程完全挂起;

(4) 守护进程以调试模式进驻(attach) NApp 进程的地址空间, 对其地址空间的内存布局加以扫描, 内存布局可以从内存文件系统的/proc/pid/maps 快速得到, 通过 r-xp 找出代码段, 代码段为主要迁移对象, 其余的 r--p 和 rw-p 为数据只读段和数据可读可写段, 本系统不对该部分进行迁移. 另外, 出于对堆栈信息的保护, 本系统将栈(stack)上的数据也进行转移, 并调整栈指针的位置, 保证栈上的返回地址不会被破坏掉;

(5) 找出待迁移的代码段对象和具体的范围后, 进驻 NApp 地址空间的守护进程首先将代码段复制到备

用区域中进行缓存, 随后逐段释放掉(free)原有的代码段, 并用安全内存 EPC 加以覆盖, 即用安全内存覆盖原有的普通内存. 注意, 这个覆盖并不是物理上的覆盖, 而是对代码段区域所对应的页表进行修改, 虚拟地址 VA 保持不变, 而物理地址 PA 从普通内存改为安全内存. 最后, 将代码从备用区域复制到安全内存 EPC 内. 整体代码的迁移流程如图 8 所示. 对于安全内存的初始化, 可以采用 SGX1 代的指令 EADD, 也可采用 SGX2 代的指令 EAUG+EACCEPT, 原因是 SGX2 代机器对两代指令均是兼容的. 与传统的安全应用加载过程相比, 本阶段具有如下不同.

- 软件部分: 无需解析 ELF 文件格式和重定向符号地址, 只作代码内容的纯拷贝, 故时延更低;
- 硬件部分: 无需生成远程验证(remote attestation, RA)的凭证, 故可跳过费时且繁琐的 EEXTEND 指令;

(6) 将基于普通内存的代码段就地替换为基于安全内存的代码段后, 守护进程需要额外加载一段代码进入 EPC 中, 即本系统提供的库操作系统(LibOS), 该 LibOS 负责 I/O 相关的系统调用的转发以及内存方面的管理. 守护进程随即跳入 LibOS 的入口点地址, 由 LibOS 配置迁移后普通应用的每个线程的线程本地存储(thread-local storage, TLS), 配置栈寄存器等信息, 随后通过 jmp 指令直接跳到第 1 步 ctrl-z 发送信号后的指令位置继续执行.

此时, 代码迁移已经全部完成. 迁移后的代码可以使用 EPC 安全内存页用于堆内存的申请.

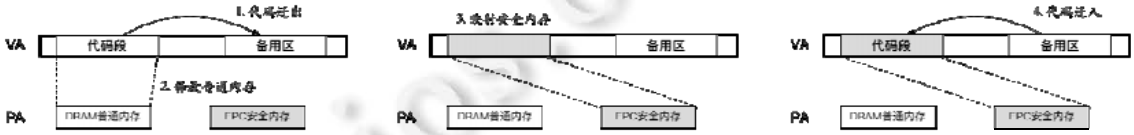

图 8 本文系统的代码迁移的基本流程

除了提供代码迁入的能力外, 还需提供代码迁出能力, 在普通内存恢复可用的情况下, 允许应用回归到原有使用模式. 具体流程为以上过程的反向过程, 详细步骤不再赘述, 这里仅讨论主要区别.

1) 对于步骤(3)而言, LibOS 需要追踪并获取被打断后的 Enclave 应用线程的上下文信息, 用于在迁出后恢复应用程序的栈地址和程序计数器信息. 对于该过程, LibOS 需要额外启用一个追踪线程, 负责获取所有应用线程的上下文信息, 如果复用其中已有的一个 Enclave 应用线程作为追踪线程的话, 会破坏原有应用线程的上下文;
2) 对于步骤(5)而言, 需要由 LibOS 主动将 EPC 代码复制到备用区域, 随后向操作系统申请释放该区域, 解除对 EPC 的映射, 同时应执行 EREMOVE 指令告诉硬件, 释放被本 Enclave 所占据的资源, 否则会造成 EPC 的内存泄露.

3.4 设计讨论

(1) 如何选择合适的迁移点?

本文的设计允许一个程序在任意时刻被中断并开始迁移, 其基本想法与 Popcorn Linux[20]类似. 对于多线程程序且存在互斥锁的情况, 迁移点的选择颇有讲究, 需要程序开发者提供一定的程序语义信息, 利用代码注解(annotation)的方法标注潜在可靠的迁移点. 这是因为以黑盒形式设计的任意时刻“盲目”迁移可能造成多线程间的死锁, 而迁移后的线程如果无法释放锁资源, 则必然导致迁移的失败.

(2) 如何迁移多线程的应用程序?

对于多线程程序而言, 可设立一个全局的“静止点(quiescence point)”. 唯有确定所有线程都到达该静止点且均处于挂起状态后, 才能实行代码迁移, 否则会导致安全问题(如 TOCTTOU 攻击). 多线程程序的迁移时间开销通常比单线程程序更高, 因为必须等到所有线程都到静止点后才开始迁移.

由于使用静止点的方案属于白盒方案, 本文对于如何在源代码中加入可靠的静止点而不影响原有应用逻辑、如何在多个静止点中选择合适的静止点等问题不作探索, 留作未来工作再加以研究和分析.

(3) 如何迁移多进程的应用程序?

对于多进程程序而言, 原理上依然可以使用本方法将同一普通应用的多个进程代码迁入到安全内存中. SGX LibOS 已经支持 *fork*(·)系统调用以及基于 SSL 的跨进程间通信(inter-process communication, IPC). 然而, 由于 SGX 体系结构的严格隔离特性, 每页 EPC 只归属于一个进程, 无法进行跨父子进程间的共享. 因此, 基于 *fork*(·)创建的多进程程序, 其父子进程的代码迁移必须使用多个 SGX Enclave 实例, 而不支持写时拷贝(copy on write, COW). 迁移后的应用程序, 每 *fork*(·)一个新进程都会消耗一个 Enclave 实例, 导致 EPC 的消耗将随着进程数的增加而增加. 这种消耗无疑是对 EPC 安全内存的严重浪费.

针对以上弊端, 本系统不建议对运行时要求使用 *fork*(·)的多进程应用程序进行迁移, 而是优先选择对多线程或多进程(但不 *fork*)的大数据应用程序进行迁移. 实际上, 大部分大数据应用程序都是使用多线程模型进行共享内存通信, 从而避免了频繁的进程间通信(IPC)和数据来回复制的开销.

(4) 为什么不在一开始就将应用程序代码迁入到 EPC 安全内存中?

一种激进的做法是在程序启动之初就将应用程序的代码段放到安全内存中, 而数据段先使用普通内存, 在普通内存消耗完毕后再使用安全内存. 我们认为该选择是一种“过度优化”, 本身并不合理. 原因如下.

1) 在真实云场景下, 操作系统无法预先判断哪个应用程序是内存大户. 本文选择的“按需”迁移的做法更为明智: 只对真正消耗大量内存的普通应用进行迁移, 避免了安全内存被过多占据;
2) 根据第 2.3 节和第 2.4 节的测试, 迁入安全内存的普通应用存在一定的开销成本, 在进行系统调用的模式切换、访存延迟上均有开销, 尽管不明显, 但仍不是零成本的. 提前将应用程序放入安全内存中, 则会导致在普通内存还未用满的情况下, 之前所有请求的延迟都会受到影响.

(5) 如何解决在安全内存不足的情况下, 普通应用的代码和数据的迁出问题?

本文系统提供的机制允许将普通应用的代码快速迁入 Enclave 中. 类似地, 本文系统同样可以实现代码的快速迁出. 然而, 在安全内存也同样不足的情况下, 普通应用驻留在安全内存的数据同样需要迁出, 而大量的数据迁出需要花费较长的时间. 本文分两类情况进行讨论.

1) 对于普通内存和安全内存均不足的情况, 此时操作系统(OS)的物理内存高度紧张, OS 可仿照现有 Out of Memory (OOM) Killer 机制, 选择杀死部分占有大量安全内存的普通应用, 快速腾出安全内存. 在无本文系统提供的内存优化支持条件下, 这部分普通应用本可能更早地被 OS 杀死, 本文系统推迟了 OOM Killer 作用的时间, 延长了大数据普通应用的服务时间;
2) 对于安全内存不足而普通内存恢复充足的情况, 迁入 EPC 的普通应用并不需要将占据安全内存的数据一次性迁出, 而可以选择“按需迁移”的做法, 保证应用的平滑运行. 相比于基于磁盘的按需换出, 本文基于内存拷贝的按需换出方案对应用的性能影响更小, 具体见后文第 4.4 节和图 11.

(6) 如何保证整体系统的隔离性和安全性?

由于普通应用和安全应用本身就运行在不同的进程内, 因此普通应用迁移后仍旧保持原有的地址空间隔离性. 此外, 迁入 Enclave 的普通应用所使用的 Enclave 实例和安全应用所使用的 Enclave 实例并不相同, 迁入后的普通应用代码不可能访问到安全应用的任何数据. 根据 SGX2 代的访问控制模型, 不同 Enclave 实例间的安全内存页 EPC 不存在共享的可能, 在硬件体系结构上直接保证了整体系统的隔离性和安全性.

3.5 分配策略

对于已经成功迁入安全内存的应用而言, 本系统提供的库操作系统 LibOS 必须提供一种合理的安全内存分配策略, 以实现较好的内存性能优化能力. 之所以这一能力属于 LibOS, 是因为分配策略的支持可以避免对应用程序的堆分配器(heap allocator)进行修改, 直接由本系统提供对底层资源可感知(low-level resource-aware)的分配策略.

在描述分配策略之前, 本文需解决 LibOS 是如何判断应用程序正在做内存分配的活动的: 对于使用 C 标准库的程序而言, 应用程序会调用 *malloc*(·)接口, 该接口由堆分配器提供, 常见的分配器有 dlmalloc、tcmalloc、jemalloc 等, 这些分配器在内存资源不足时会调用两个系统调用接口: *sbrk*(·)或 *mmap*(·). 而这两个

系统调用接口会使用特权指令 sysenter, 该特权指令会被 LibOS 捕获, 通过系统调用号判断这是一个虚拟地址空间扩大的请求, 于是 LibOS 便会介入, 根据当前安全内存和普通内存的使用情况合理地分配可用内存资源. 具体流程如图 9 所示.

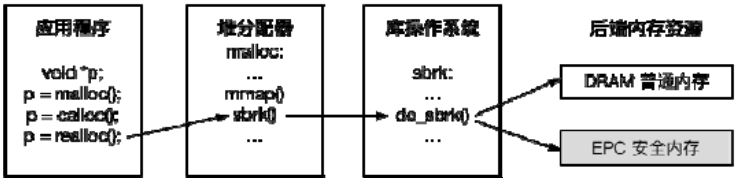

图 9 库操作系统的内存分配流程

接下来我们描述可行的分配策略.

- 优先级: 对于同时存在空闲的普通内存和安全内存的情况, LibOS 的分配器始终优先分配普通内存, 保证普通内存的使用率最高. 普通内存达到特定阈值时则分配安全内存;
- 提前分配: 由于申请新的 EPC 需要内核参与, 需要借助 EAUG 内核态指令来完成, 因此 LibOS 会提前分配更大块的内存页, 以降低 EPC 分配请求的缺页中断次数;
- 合并释放: 由于 EPC 的释放需要通知操作系统内核, 存在较大开销, 因此本文采用批处理(batch)的方法将多个 EPC 释放请求合并为 1 个, 从而避免了频繁的模式切换;
- 资源回收: 对于释放资源后又频繁分配 EPC 内存的应用程序, LibOS 可以将 EPC 资源驻留一段时间, 从而让分配操作回收上一次释放的资源. 该操作无需内核参与, 从而降低了开销.

4 实验评估

本节通过回答如下几个问题来说明本文内存系统的性能优化效果.

(1) 本系统代码迁移的开销时间是多少? 该评估说明了迁移导致的应用停滞时间, 应该越短越好;

(2) 使用本系统后, SGX2 代机器的物理内存资源利用率能有多大提升?

(3) 与现有的系统交换分区方案相比(如基于磁盘的数据交换分区方案), 本文提出的代码迁移方案对于应用程序的性能有多大提升?

4.1 环境配置

本节的实验继续选择在阿里云的“安全增强通用型”虚拟机上完成, 具体环境配置已经在第 2.1 节中给出说明, 这里不再赘述.

4.2 测试负载

我们选择如下两款内存密集型应用程序作为测试对象, 其内存使用量随着时间的推移逐渐增加.

- Redis (I/O 密集+内存密集): 基于内存的高性能 Key-Value Store (KVS)存储, 支持多种常用数据结构. Redis 在运行阶段会消耗大量内存用于缓存数据, 从而保证读写请求的低时延. 本测试中, 我们向 Redis 存储中连续写入 100 条值的长度为 100 MB 的键值数据, Redis 总共消耗内存 16.06 GB;
- Darknet YOLO (计算密集+内存密集): Darknet 是由 C++编写的开源深度学习框架, 其中, YOLO (you only look once)专门用于物体检测, 加载预先训练好的模型权重参数会消耗大量内存. 本测试中, 我们使用 yolov3.weights 模型对图片进行识别, 总共消耗 6.67 GB 内存.

4.3 测试方法

本节选择如下 3 组方案互为对照组.

1. 基于磁盘的交换分区方案: 这是 Linux 系统内核默认的交换方法, 一旦物理内存使用空间不足, 同时达到系统配置的阈值(/proc/sys/vm/swappiness), 此时内核将选择性地将非活跃的内存工作集换出

(swap out)到磁盘中. 由于本测试是在公有云上的虚拟机上完成的, 因此磁盘所使用的后端是云存储, 即分布式的 SSD 云盘. 目前, 分布式 SSD 是公有云的标准存储配置;

2. 基于压缩内存的分区方案: Linux Zswap[21]是基于磁盘的交换分区方案的增强版, 它有效地降低了传统的基于磁盘交换分区方案的 I/O 次数和对磁盘带宽的占用. 在物理内存空间不足的情况下, Zswap 将压缩一部分非活跃的内存页, 腾出更多物理内存空间用于容纳更多的工作集. 使用 Zswap 的交换系统本质上是使用 CPU 密集代替了 I/O 密集, 从而提升了系统的整体性能;
3. 基于本文提出的轻量级代码迁移方案: 在普通内存物理空间不足的情况下, 本文方案将普通应用的代码快速迁入到安全内存中, 从而避免了系统换页事件的触发, 最小化换页事件对系统和应用带来的影响.

本文将基于磁盘和基于压缩内存的分区大小均配置为 8 GB. 为了测试交换事件对系统和负载的影响, 我们需要在应用运行过程中触发系统的换页事件. 为此, 本文创建了一部分消耗内存的简单应用, 将系统的剩余可用普通内存调整, 见表 6 (剩余普通内存资源和应用实际内存需求之比控制在 1:4 左右).

表 6 应用程序内存消耗与可用内存情况

应用程序	应用程序需消耗内存大小(GB)	剩余可用普通内存大小(GB)	剩余可用安全内存大小(GB)
Redis	16.06	4	16
Darknet YOLO	6.67	2	16

4.4 实验结果

本小节的实验结果对应于回答本节开头提出的 3 个具体问题.

(1) 迁移时间

我们使用本文提出的轻量级代码迁移方案, 在普通应用发现普通内存即将不足时, 对普通应用的数据段进行迁移. 迁移时间见表 7. 实验结果表明, 迁移时间与代码段的大小呈现正相关. 由于应用程序的代码段远小于数据段大小, 通常只有 MB 级别, 迁移时间不超过 10 ms, 对于具有低时延要求的应用程序而言, 属于可接受的范围.

表 7 普通应用的迁移时间

应用程序	迁移时间(ms)	代码段大小(MB)	数据段大小(MB)
Redis	6.31	2.06	16 450.13
Darknet YOLO	4.93	1.36	6 926.90

(2) 内存资源利用率

如图 10 所示, 实验结果表明, 本文所采用的轻量级代码迁移方案可以很好地提高物理内存资源的利用率. 对于 Redis 应用程序而言, 内存资源利用率从 47.70%提升到了 76.14%; Darknet YOLO 的内存利用率从 47.27%提高到了 62.95%. 内存利用率的提高是由于普通应用使用了安全内存的原因所致, 进而避免了系统交换事件的发生.

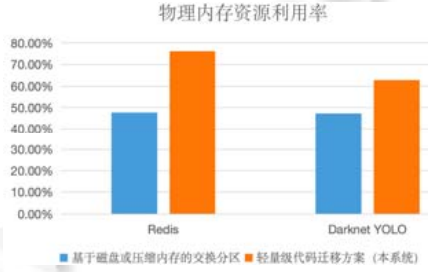

图 10 物理内存资源利用率

(3) 普通应用程序的性能影响

我们测试不同方案对应用程序的性能影响, 具体的性能数值展示如图 11 所示. 其中, 纵轴为单位时间(s);

“纯普通内存”表示剩余的普通内存可以充分容纳整个应用程序的工作集, 不会触发任何换页事件. 优化比的计算方法为

优化了多少时延÷优化前的时延开销×100%=

(优化后方案总时延–纯普通内存时延)÷(优化后方案总时延–优化前方案总时延)×100%.

如磁盘交换分区方案的时延开销为 56.13–12.28=43.85 s, 基于本文方案则将开销降低了 56.13–24.03=32.10 s, 优化比为 32.10÷43.85=73.20%.

图 11 普通应用的性能影响(单位: s)

从图中可以看出:

- 对于 Redis 应用程序而言, 本文方案比基于磁盘的交换分区方案优化了 73.20%; 本文方案比基于压缩内存的交换分区方案优化了 14.98%;
- 对于 Darknet YOLO 应用程序而言, 本文方案比基于磁盘的交换分区方案优化了 98.71%; 本文方案比基于压缩内存的交换分区方案优化了 98.85%.

4.5 结果分析

第 4.4 节第(1)部分、第(2)部分的实验结果可以充分说明本文方案对普通应用的挂起时间影响很小(即“迁移时间”); 同时, 还能提高物理内存资源利用率以避免换页操作. 本节接下来主要分析第 4.4 节第(3)部分的实验数据.

1. 对于 I/O 密集型+内存密集型的应用程序而言, 如内存键值存储, 我们发现, 基于压缩内存的方案和本文方案的效果都非常突出, 都能达到超过 50%的性能优化效果. 这是因为, I/O 密集型应用中的 CPU 利用率通常不高, 通过充分利用 CPU 时钟周期可以很好地避免磁盘 I/O 事件的触发. 我们进一步观察到: 如果将 Redis 应用一开始就迁入安全内存的话, 其整体性能延迟会达到 35.78 s, 远高于现在的 24.03 s 和压缩内存方案的 26.10 s. 这是因为, SGX Enclave 的代码无法进行 I/O 操作, 必须切换出安全模式, 因此, 频繁的 I/O 操作会导致应用程序的性能下降;
2. 对于计算密集型+内存密集型的应用程序而言, 如机器学习应用, 我们发现, 磁盘 I/O 对计算密集型应用带来的影响并没有像对 I/O 密集型应用那么显著: Darknet YOLO 的 39.90%相比于 Redis 的 357.08%, 后者的开销接近于 4 倍. 而基于压缩内存的交换方案甚至比基于磁盘的交换方案还要差, 这是因为, 应用程序本身就已经消耗处理器时钟周期了, 而压缩内存也要消耗处理器, 二者之间存在着处理器资源的严重竞争. 而本文方案只消耗很少的处理器周期(小于 10 ms)用于代码段的迁移, 其余基本不产生任何性能影响;
3. 针对不同的应用程序类型而言, 本文方案相对于基准(完全跑在普通内存上)有不同的性能影响. 对于 I/O 密集型应用而言, 开销主要来自于 I/O 发生时必要的模式切换, 本文方案在 Redis 上的开销接近于 100%(实际值为 95.68%). 而对于计算密集型应用而言, 本文系统的开销则几乎没有(0.5%), 该开销主要来自于安全内存的访存开销.

5 相关工作

除了本文行文中提到的相关工作之外, 本文的主要相关工作还包括了SGX应用程序的迁移和提升系统整体内存利用率的优化工作.

5.1 SGX应用程序的迁移

Gu 等人[22]发现: EPC 安全内存属于虚拟机监视器(hypervisor)无法看见的物理内存, 传统的基于虚拟机监视器的虚拟机迁移方案对于 EPC 不再适用. 对于这个问题, Gu 等人的方法是在 Enclave 内部埋入一个特定迁移辅助线程, 负责将 EPC 数据加密并拷出, 随后发往目标 Enclave 处恢复执行. 他们的核心贡献在于如何保证 Enclave 的迁移不能被 fork 攻击, 即只能有一个合法的目标 Enclave.

Alder 等人[23]则进一步发现: Gu 等人的工作只解决了 Enclave 在安全内存上的数据迁移问题, 对于包含状态的持久性数据, 如单向计数器和磁盘中的加密数据, 由于这些信息与 Enclave 所在的 CPU 绑定, 若直接迁移, 则会导致加密数据无法使用. 对于这一问题, Alder 等人将持久性数据也列入迁移对象, 并提出了纯软件的解决方案, 实现更为完备的迁移.

相比之下, 本文解决的问题是如何将一个普通应用迁入到安全内存中, 通过只迁移代码的方法保证最小化迁移时间开销, 实现了物理内存利用率的提高并避免了普通应用的磁盘换页开销.

5.2 系统内存利用率的优化

本文主要解决 SGX2 代机器因为安全内存配置较大导致的物理内存低利用率的问题, 这一问题非常类似于数据中心内部集群机器的内存利用率不平衡的问题. 该系列工作考虑到集群内不同机器的内存利用率分布不同, 利用率高的机器可以“借用”利用率低的机器的物理内存资源, 从而避免自身因为物理内存资源不足导致的交换成本.

InfiniSwap[24]通过 RDMA 技术将不同机器的内存连接起来, 允许内存密集型的应用程序直接访问远端机器的空闲物理内存资源. InfiniSwap 实现在块设备这一抽象上, 因此无需修改应用程序, 让内存密集型应用将自己的工作集扩展到数据中心的多台机器上.

Leap[25]则通过对大内存应用的访存模式进行剖析, 加入各种预取(prefetch)算法到 RDMA 网络的交换系统中, 避免因为使用远端内存导致的长尾延迟, 从而保证在集群内存高内存利用率的基础上缓存热点数据.

与以上工作相比, 本文工作则解决了单台机器因为物理内存资源划分导致的利用率低的问题: 安全内存利用率相对较低, 因此允许普通应用对安全内存加以利用. 与基于 RDMA 的工作相比, 本文的迁移方案需要 CPU 的参与, 而 RDMA 则可以直接由网卡进行数据搬运, 可大大节省 CPU 时钟周期. 但是本文提出的代码迁移方案只需要迁移非常小的代码段, 迁移时间不超过 10 ms, 因此只消耗了非常少的 CPU 时钟周期.

5.3 系统交换分区的优化

对于交换分区的优化, Linux 提供了 Zswap[21]机制, 即: 为了降低磁盘交换分区带来的 I/O 开销, 数据并不直接写到磁盘上, 而是将部分非活跃内存进行压缩. Zswap 和本文方法都可用于缓解普通内存不足的问题.

6 总结与展望

可信执行环境在隐私计算场景中扮演着重要角色. 本文面向新一代可信执行环境——Intel SGX2 代的服务器, 对其性能瓶颈进行了详细分析, 指出, 在新硬件特性下性能层面的主要矛盾已经发生了转移: 为了避免安全应用因为安全内存不足导致的严重性能开销, 服务器维护者倾向于将安全内存调整至较大值, 这一选择严重压缩了普通内存的可使用范围. 该做法导致了安全内存的利用率低下以及普通应用的内存紧张, 二者成为了新的主要矛盾. 本文使用大数据应用进行了详细测试并证明该问题的严重性.

针对 SGX2 代服务器遇到的新的性能挑战问题, 本文提出了一种轻量级的代码迁移方案. 对于需要大量内存的应用程序, 在临近发出内存换页时, 本文实现的库操作系统将其代码快速迁入到安全内存中. 被迁移

的普通应用程序可以使用安全内存作为新的内存资源, 同时, 仍可以访问原本驻留在普通内存上的原有数据. 本文在真实内存密集型应用程序上进行了实验评估, 实验结果表明, 本文方案的迁移时间不超过 10 ms, 同时能降低 73.2%–98.7%的应用程序性能开销, 对于计算密集型应用甚至能实现接近 0 (0.5%)的开销.

本文提出的方案对新兴的可信硬件系统具有通用性, 如 ARM 处理器上的 TrustZone、RISC-V 上使用 PMP, 二者均使用静态隔离方案来划分普通内存和安全内存, 存在安全内存利用率低和普通内存抖动的问题. 在未来工作中, 计划在下一步将我们的方案移植到不同平台上, 提供平台无关的可信硬件内存优化方案.

致谢 本文作者衷心感谢阿里云提供的 SGX2 代服务器设备, 以及阿里云基础软件 Java 团队的技术支持.

References:

[1] China Information Communications Institute. China Privacy Computing Industry Development Report, 2021 (in Chinese). http://www.caict.ac.cn/kxyj/qwfb/ztbg/202011/t20201110_361696.htm

[2] Hoekstra M, Lal R, Pappachan P, *et al*. Using innovative instructions to create trustworthy software solutions. HASP@ ISCA, 2013, 11(10.1145): 2487726–2488370.

[3] McKeen F, Alexandrovich I, Anati I, *et al*. Intel® software guard extensions (Intel® sgx) support for dynamic memory management inside an enclave. In: Proc. of the Hardware and Architectural Support for Security and Privacy 2016. 2016. 1–9.

[4] Kaplan D, Powell J, Woller T. AMD Memory Encryption. White Paper, 2021.

[5] https://www.intel.cn/content/www/cn/zh/developer/articles/technical/intel-trust-domain-extensions.html

[6] https://developer.arm.com/architectures/architecture-security-features/confidential-computing

[7] Feng E, Lu X, Du D, *et al*. Scalable memory protection in the PENGLAI enclave. In: Proc. of the 15th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2021). 2021. 275–294.

[8] https://github.com/ayeks/SGX-hardware

[9] Taassori M, Shafiee A, Balasubramonian R. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In: Proc. of the 23rd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. 2018. 665–678.

[10] Orenbach M, Lifshits P, Minkin M, *et al*. Eleos: ExitLess OS services for SGX enclaves. In: Proc. of the 12th European Conf. on Computer Systems. 2017. 238–253.

[11] Durumeric Z, Li F, Kasten J, *et al*. The matter of heartbleed. In: Proc. of the 2014 Conf. on Internet Measurement Conf. 2014. 475–488.

[12] Baumann A, Peinado M, Hunt G. Shielding applications from an untrusted cloud with haven. ACM Trans. on Computer Systems (TOCS), 2015, 33(3): 1–26.

[13] Hunt T, Zhu Z, Xu Y, *et al*. Ryoan: A distributed sandbox for untrusted computation on secret data. ACM Trans. on Computer Systems (TOCS), 2018, 35(4): 1–32.

[14] Arnautov S, Trach B, Gregor F, *et al*. SCONE: Secure Linux containers with Intel SGX. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2016). 2016. 689–703.

[15] Weisse O, Bertacco V, Austin T. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. ACM SIGARCH Computer Architecture News, 2017, 45(2): 81–93.

[16] Orenbach M, Michalevsky Y, Fetzer C, *et al*. CoSMIX: A compiler-based system for secure memory instrumentation and execution in enclaves. In: Proc. of the 2019 USENIX Annual Technical Conf. (USENIX ATC 2019). 2019. 555–570.

[17] Dong CT, Shen QN, Luo W, Wu PF, Wu ZH. Research progress of SGX application supporting techniques. Ruan Jian Xue Bao/ Journal of Software, 2021, 32(1): 137–166 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6095.htm [doi: 10.13328/j.cnki.jos.006095]

[18] https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf

[19] https://www.alibabacloud.com/help/zh/doc-detail/25378.htm

[20] Barbalace A, Sadini M, Ansary S, *et al*. Popcorn: Bridging the programmability gap in heterogeneous-ISA platforms. In: Proc. of the 10th European Conf. on Computer Systems. 2015. 1–16.

[21] Jennings S. Transparent memory compression in Linux. 2013. https://events.static.linuxfound.org/sites/events/files/slides/tmc_sjennings_linuxcon2013.pdf

[22] Gu J, Hua Z, Xia Y, *et al*. Secure live migration of SGX enclaves on untrusted cloud. In: Proc. of the 47th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN). IEEE, 2017. 225–236.

[23] Alder F, Kurnikov A, Paverd A, *et al*. Migrating SGX enclaves with persistent state. In: Proc. of the 48th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN). IEEE, 2018. 195–206.

[24] Gu J, Lee Y, Zhang Y, *et al*. Efficient memory disaggregation with infiniswap. In: Proc. of the 14th USENIX Symp. on Networked Systems Design and Implementation (NSDI 2017). 2017. 649–667.

[25] Al Maruf H, Chowdhury M. Effectively prefetching remote memory with leap. In: Proc. of the 2020 USENIX Annual Technical Conf. (USENIX ATC 2020). 2020. 843–857.

附中文参考文献:

[1] 中国信通院. 中国隐私计算产业发展报告. 2021. http://www.caict.ac.cn/kxyj/qwfb/ztbg/202011/t20201110_361696.htm

[17] 董春涛, 沈晴霓, 罗武, 吴鹏飞, 吴中海. SGX 应用支持技术研究进展. 软件学报, 2021, 32(1): 137−166. http://www.jos.org.cn/1000-9825/6095.htm [doi: 10.13328/j.cnki.jos.006095]

李明煜(1992－), 男, 博士生, 主要研究领域为系统安全, 隐私计算.

夏虞斌(1982－), 男, 博士, 副教授, 博士生导师, CCF 高级会员, 主要研究领域为操作系统, 系统虚拟化, 系统结构.

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, 系统安全, 系统结构.