NMST:一种基于线段树的持久性内存管理优化方法

侯泽毅[1] 万 虎[1] 徐远超[1,2]
(首都师范大学信息工程学院 北京 100048)[1]
(中国科学院计算技术研究所计算机体系结构国家重点实验室 北京 100190)[2]

摘 要 新型非易失存储介质(Non-Volatile Memory,NVM)的出现引发了编程模型的革新。现有的基于函数库的编程模型为存储系统提供的ACID特性解决了数据一致性问题,但是在分配持久性内存时,延迟较大,不能很好地满足应用程序对动态内存分配速度的要求。针对现有函数库编程模型中存在持久化内存管理和分配低效的问题,以目前最具代表性的函数库编程模型NVML为基础,提出了一种基于线段树的持久性内存管理分配优化方法NMST;另外,针对线段树在持久性内存分配过程中维护连续空间时开销较大的问题,提出构造多粒度叶子结点的线段树的方法。实验结果表明,相比于NVML原始方法,NMST方法在分配持久性内存时使延迟降低了36.9%,而优化后的NMST方法在分配持久性内存时使延迟降低了43.6%。实验结果也证明,性能提升的大小与调用NVML函数库的程序中实际持久性内存分配的次数及粒度紧密相关。

关键词 非易失存储介质,编程模型,线段树,持久性内存管理
中图法分类号 TP391 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2018.07.012

NMST:A Persistent Memory Management Optimization Approach Based on Segment Tree

HOU Ze-yi[1] WAN Hu[1] XU Yuan-chao[1,2]
(College of Information Engineering,Capital Normal University,Beijing 100048,China)[1]
(State Key Laboratory of Computer Architecture,Institute of Computing Technology,Chinese Academy of Sciences,Beijing 100190,China)[2]

Abstract The emergence of non-volatile memory (NVM) has led to the innovation of the programming model. The existing programming models based on function library provide ACID characteristics to solve the problem of data consistency for memory system. However, they introduce huge overhead when allocating persistent memory dynamically, thereby degrading the applications' performance. In this paper, an optimization approach based on segment tree was proposed to speedup persistent memory allocation and management. NMST targets NVM Library (NVML), namely a representative library programming model. Furthermore, an optimized NMST was proposed to mitigate the huge overhead of segment tree in maintaining continuous space by constructing segment tree with multi-granularity leaf nodes. The experimental results show that NMST reduces the latency by 36.9% compared with traditional methods when allocating persistent memory, and the optimized NMST reduces the latency by 43.6%. The results also demonstrate that performance improvement is closely related to the quantity and granularity of persistent memory allocation in programs.

Keywords Non-volatile memory, Programming model, Segment tree, Persistent memory management

近年来,随着半导体和新材料技术的发展,出现了很多新型的存储介质,如相变存储器[1](Phase-Change Memory,PCM)、自旋转移矩磁随机存取存储器[2](Spin-Transfer Torque Magnetic RAM,STT-RAM)、电阻式存储器[3](Resistive RAM,RRAM)等,将其统称为非易失存储(Non-Volatile Memory,NVM)。如今,新型非易失存储技术已进入实用阶段,如2015年英特尔和镁光联合发布了6TB容量的非易失内存3D XPoint[4]。这些存储介质具有功耗低、访问速度快、存储密度大、可按字节寻址、非易失等优点,可以满足大数据环境下对内存计算[5]的需求,能有效缓解现有内存扩展能力不足的问题。新型非易失存储介质对系统软件(如文件系统)、程序函数库(如持久性对象存储库)和应用程序(如数据库程序)的设计都产生了深远的影响。

持久性内存的出现引发了编程模型的革新,面向持久性内存的持久化编程模型能够直接利用非易失性存储介质的持久性,通过Load/Store接口快速访问存储系统中的数据结构,从而避免了先发起I/O请求再将数据格式转换为内存表示所带来的开销。因此,在使用NVM作为持久性存储的系

到稿日期:2017-07-29 修回日期:2017-10-22 本文受计算机体系结构国家重点实验室开放课题(CARCH201503),国家留学基金委和北京市成像技术高精尖创新中心资助。
侯泽毅(1993-),男,硕士生,主要研究方向为非易失存储技术;**万 虎**(1991-),男,博士生,主要研究方向为非易失存储技术;**徐远超**(1975-),男,博士,副教授,主要研究方向为计算机系统,E-mail:xuyuanchao@cnu.edu.cn(通信作者)。

统中，设计高效的内存持久化模型是提升系统性能的关键。

迄今为止，研究人员已提出了多种持久化编程模型[6]，包括原生持久化编程模型、文件系统持久化编程模型和函数库持久化编程模型。这些模型虽然为存储系统提供了 ACID 特性，解决了数据一致性问题，但是在分配持久性内存时存在较大的延迟。尤其在数据量爆炸式增长时期，应用程序对动态内存分配的速度具有极高的要求。如何在保证可用性的前提下提升持久性内存管理的效率，是持久性内存系统亟待解决的问题之一。

为了更好地提高持久化内存管理和分配的时间效率，本文以函数库编程模型中较有代表性的 NVML[7] 为基础，提出了一种基于线段树(Segment Tree)的高效 NVM 内存管理方法，简称为 NMST。本文的主要工作包括：

1)实测 NVML 函数库编程模型中分配持久性内存的开销，分析 NVML 中内存管理的性能瓶颈；

2)通过在 DRAM 上实现线段树辅助管理结构，减少了动态内存分配的开销；

3)进一步提出了线段树空闲空间管理优化方法，减少了线段树管理 NVM 空间分配时维护连续物理空间的开销；

4)通过充分的实验证明了，相比于 NVML 原有方案，NMST 方法使内存分配的延迟降低了 36.9%，而优化后的 NMST 方法使内存分配的延迟降低了 43.6%。

1　相关研究背景

1.1　持久性内存编程模型

为保证 NVM 上的数据具有一致性，研究人员提出了多种持久性内存编程模型。这些编程模型主要包括 3 类[6]：1)原生持久化编程模型(native persistence)，如面向 NVM 的持久性数据结构 CDDS[8] 和 NV-Tree[9] 等，它们以数据结构原生的形式将需要持久性存储的数据直接存储在 NVM 上；2)文件系统持久化编程模型(filesystem persistence)，如 BPFS[10]，SCMFS[11]，PMFS[12] 和 NOVA[13] 等，它们以非易失性文件系统的方式将底层的 NVM 组织起来后提供给上层应用程序使用；3)函数库持久化编程模型(library persistence)，如 Mnemosyne[14]，NV-Heaps[15]，Aerie[16] 和 NVML[7]，NVM 上的数据持久化和一致性更新工作由函数库完成。如图 1 所示，从左至右沿虚线分开依次为文件系统持久化编程模型、函数库持久化编程模型和原生持久化编程模型。

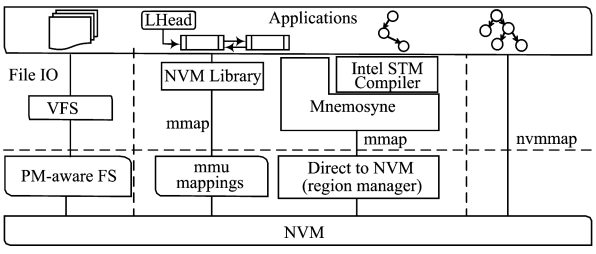

图 1　几种持久化内存编程模型

Fig. 1　Several persistent memory programming models

NVML 是 Intel 公司开发的持久化内存函数库，它作为非易失性存储介质上面一层的软件栈[17]，为上层应用非易失性存储提供了便捷的访问接口。特别地，为了支持应用程序更高效地完成持久性内存数据的一致性更新工作，NVML 提供持久化事务编程接口供用户程序访问持久性内存对象时使用。这为解决持久性内存系统的数据一致性问题提供了有效的方式。下一节将针对 NVML 这一具有代表性的函数库编程模型进行详细分析。

1.2　持久性内存分配

面向 NVM 的持久性内存编程模型通常以程序库的形式提供持久性内存数据的一致性更新功能，如 Mnemosyne[14]，NV-Heaps[15]，Aerie[16] 和 NVML[7]。这些函数库除了提供持久化事务编程接口外，通常还提供持久性内存分配器用于空间管理。在这些持久性内存分配器中，一次完整的持久性内存分配通常由预留持久性内存(reserve)、初始化持久性内存数据(initialize)、对外公开分配后的持久性内存地址(publish)等过程构成。分配的各个阶段被限制在事务内进行，以避免由于失效而引入持久性内存泄漏。nvm_malloc[18] 是一个独立的面向 NVM 的内存分配器，采用分步处理的分配方法，允许在事务外进行持久性内存预留和数据初始化过程，再以事务的形式完成真正的空间分配和发布过程，从而一定程度上减少了事务开销。Makalu[19] 是最新被提出的一个快速可恢复持久性内存分配器，仅将发布过程放在事务内进行。但是，一旦出现异常掉电或系统崩溃的情况，在事务外进行的持久性内存分配过程将导致内存泄漏，Makalu 通过垃圾回收机制来回收泄漏的持久性内存。以上内存分配器的不足之处在于：Mnemosyne 等内置持久性内存分配器使得事务被放大，引发 3～10 倍的写放大；区别于传统的编程范式，nvm_malloc 中一次持久性内存对象分配需要两次不同形式的接口调用，增加了程序员的负担；Makalu 有效地加速了内存分配，但恢复时进行离线垃圾回收的代价很大。持久性内存分配的效率紧紧依托于所采用的编程模型，到目前为止，只有 NVML 函数库编程模型相对比较成熟，因此本文主要针对 NVML 中的持久性内存分配管理进行优化。

2　NVML 内存管理的性能瓶颈分析

NVML 函数库编程模型在分配持久性内存时存在性能瓶颈，经过测试，其平均分配延迟长达 1000～2500 ns。本节首先介绍 NVM 上的物理布局以及 NVML 空闲空间的管理方式，在此基础上，再具体分析 NVML 内存分配的性能瓶颈。

2.1　NVM 上的物理布局

NVM 上的物理布局如图 2 所示，主要由以下几个部分组成：内存池头部(memory pool header)、内存池描述符(memory pool descriptor)、通道区域(lane)和持久性内存堆(persistent heap)。内存池头部有 4 kB 大小，用于记录内存池的相关信息，如 uuid、签名、版本号、创建时间和访问权限等；内存池描述符在 NVM 中的部分有 2 kB 大小，包含了描述当前内存布局的名称 layout、第一个 lane 的起始位置偏移和数目、heap 的起始位置偏移和大小、校验和等。lane 区域是特殊的日志区，从 8 kB 偏移开始，通常情况下其数目为 1024。最后剩下的全部物理空间即是持久性内存堆区域(堆的起始地址与 4 kB 页面大小对齐)。

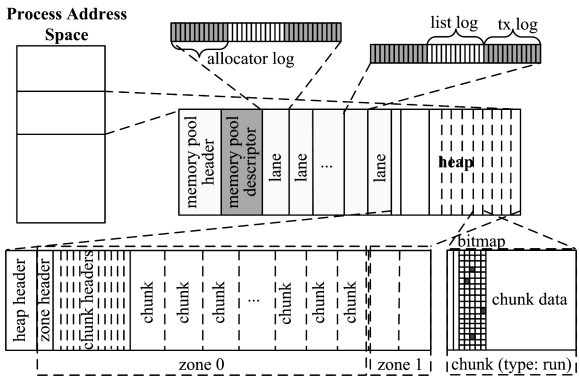

图 2 持久性内存分配器在 NVM 上的物理布局

Fig. 2 Physical layout of persistent memory allocation in NVM

持久性内存堆是真正用来存放用户数据的空间，包括记录持久性内存堆中相关信息的头部区域(heap header)，之后的空间按照 zone 等分，默认情况下，每个 zone 的大小最小是 768 kB，最大是 15 GB。zone 区域有 zone header，之后是统一存放所有 chunk 的 chunk header 区域，最后是包含最多 *MAX_CHUNK* 个 256 kB 大小的 chunk。每个 chunk_header 记录对应 chunk 的类型、标记，以及以 chunk 或者 block size 为单位的内存块数目。chunk 有两种类型：1)用于分配大块的内存空间，此时 chunk 无需细分；2)用于分配小于 256 kB 大小的数据空间，此时 chunk 会继续细分，称此 chunk 为 run。由于在分配小于 256 kB 大小的数据时，chunk header 用于记录细分信息的空间不够，因此每个用作 run 的 chunk 会被转换为用 chunk_run 类型来表示，并在头部添加位图(bit-map)信息用于管理 run 类型 chunk 内的更小的数据块的使用情况。

2.2 NVML 空闲空间的管理方式

由于 NVM 写延迟较高，通常在 DRAM 中建立辅助性数据结构来加速分配过程。NVML 中持久性内存分配器在易失性内存 DRAM 中使用辅助元数据来加速持久性内存的分配过程。持久性内存堆的物理空间按照 chunk 进行划分，用于大块内存的分配。当需要分配小块的内存时，就将 chunk 按照某个小粒度(run)进行划分。NVML 在 NVM 中使用位图的形式记录每个数据块的空闲状态，并根据位图信息在 DRAM 中创建对应的 crit-bit tree[20] 结构，使用该数据结构来管理持久性内存空闲空间。图 3 展示了持久性内存分配器在 DRAM 上的数据组织方式。crit-bit tree 又被称为位运算前缀树/字典树(bit-wise trie)，是一种有序树，其键是一串位元。crit-bit tree 结构在执行插入操作时主要对节点执行读操作，因此性能较好。NVML 中使用 crit-bit tree 来管理空闲空间时，每个叶子节点表示一段连续的空闲空间。

实际上，crit-bit tree 的每个叶子节点是一个 64 位的值，这 64 位按照从高位到低位依次保存了所要管理的空闲空间的大小索引、块偏移、所在 chunk 编号和所在 zone 编号。现代处理器能够保证 8 字节数据更新的原子性，为此，每个叶子节点所管理的空闲空间索引大小最大为 8，对应 8 个 chunk 或者 8 个 block。持久性内存分配器在首次构建或者重新启动后从已有的持久性堆重新构建易失性内存表示时，以 chunk 为粒度，整个堆空间中每 8 个 chunk 归为一个叶子节点来表示，并将叶子节点依次插入到树中，从而构建出整个持久性堆在内存中的表示。当需要分配小粒度的持久性内存时，从 crit-bit tree 中找到一个空闲的 chunk，将 chunk 转换为 run，并为 run 构建一棵 crit-bit tree 来表示。如图 3 所示，使用 crit-bit tree 管理 chunk 和 run 的方式相同，区别仅在于其叶子节点的部分域所表示的含义不同。

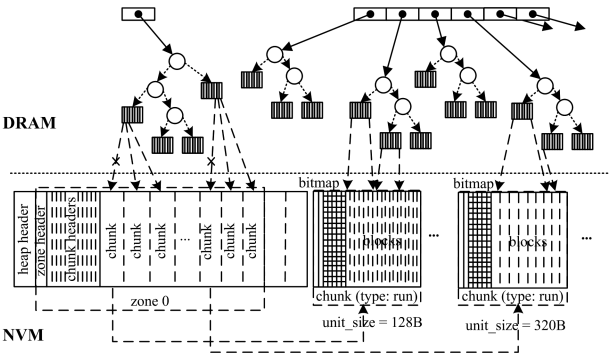

图 3 持久性内存分配器在 DRAM 上的数据组织

Fig. 3 Data organization of persistent memory allocation in DRAM

2.3 NVML 内存分配的性能瓶颈

由于 crit-bit tree 具有有序性，因此可以按照最佳适应方法(best-fit)来分配空闲块。在 DRAM 内 crit-bit tree 辅助管理的 NVM 上分配一段连续空间的过程包括两步：1)从树上找到大小正好或者大于所需块数的最小块的节点；2)删除这个节点。如果获取到的空间大于所需要分配的空间大小，那么还需要把剩下的空间再作为一个节点重新插入到 crit-bit tree 中。例如，当需要获取 3 个块大小的空间时，则首先找到目前树上的最小空间(node *A*，8 个 block 大小)，然后删除 node *A* 这个节点，再插入一个 5 个块大小的节点(node *B*)到树上；如果下次还需要分配 3 个块大小的空间，那么类似地删除 node *B* 节点，再插入一个 2 个块大小的节点(node *C*)到树中。

NVML 对空闲空间的管理分两个过程：1)在 NVM 上使用 bitmap 来管理持久性内存堆空间；2)在 DRAM 上使用 crit-bit tree 结构辅助管理持久性内存堆空间(目的是加速持久性内存分配的过程)。整个分配过程中，维护持久化内存分配中空闲空间一致性的开销占比大概为 20%～25%。同时，通过实际测试发现，现在 NVML 中获取一段连续空闲空间的过程基本都需要一次删除节点操作和一次插入节点操作(即获取到大小正好的节点的情况极少)。树(crit-bit tree)上节点的插入和删除操作由于都需要动态分配内存，会导致整个内存分配过程的开销很大。实验测试表明，在 DRAM 的 crit-bit tree 中分配所需空间这一过程的开销，占总分配开销的 35%～45%。经过分析，确定开销主要在于，在 crit-bit tree 上需要进行频繁的小内存分配，每次对树的插入和删除操作都需要做动态内存分配，并且需要加互斥锁。实验测试结果显示，这将导致持久性内存分配的延迟显著增加，成为影响持久性内存分配性能的瓶颈。

3 基于线段树的 NVML 内存管理

本节首先引入线段树(segment tree)数据结构的基本概念和原理，然后介绍构造和更新线段树来管理 NVM 空间的

方法，最后分析在 NVML 中使用 segment-tree 代替 crit-bit tree 来管理 NVM 空间以及分配持久性内存时的优势。

3.1　线段树的引入及基本原理

线段树的引入主要源于两个操作：区间查询和区间修改。假设现在维护一段[0，n]的区间，则包含下面两种操作，且一共执行 k 次。

1）区间查询：查询一段区间的数值和。

2）区间修改：修改一段区间的数值。

如果每次操作都遍历区间，那么时间复杂度将为 O(kn)，在 k 和 n 较大时是极为低效的。线段树就是利用区间拆分的思想来解决这类连续区间动态查询和修改问题的一种数据结构。

线段树是一种二叉搜索树，每个节点最多有两棵子树的树结构，子树通常被称作“左子树”(left subtree)和“右子树”(right subtree)。对于线段树中的每一个非叶子节点[a，b]，它的左子树表示的区间为[a，(a+b)/2]，右子树表示的区间为[(a+b)/2+1，b]。因此，线段树是平衡二叉树，最后的子节点数目为 N，即整个线段区间的长度。线段树的每个节点存储了一个区间(线段)，使用线段树可以快速查找某一个节点在若干条线段中出现的次数，时间复杂度为 O(log$N$)。

图 4 给出一个[1，6]区间的线段树结构，每个节点存储一个区间(这里的存储区间并不是指存储这个区间中所有的元素，而是只需要存储区间的左右端点即可)，所有叶子节点表示的是单位区间(即左右端点相等的区间)，所有非叶子节点(内部节点)都有左右两棵子树。对于所有非叶子节点，它表示的区间为[l，r]，那么令 mid 为(l+r)/2 的下整，则它的左子树表示的区间为[l，mid]，右子树表示的区间为[mid+1，r]。基于这种结构，叶子节点保存一个对应原始数组下标的值。由于树是一个递归结构，两个子节点的区间并，正好是父节点的区间，因此可以通过自底向上的计算在每个节点都计算出当前区间的最大值。需要注意的是，由于线段树是一棵平衡树，因此其高度为 log(n)。

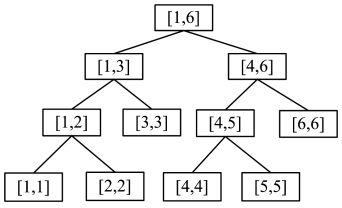

图 4　用于 NVM 空间管理的线段树

Fig. 4　Segment tree for NVM space management

3.2　构造 NVM 空间管理线段树

系统启动时，需要完成 NVM 内存管理模块的初始化，初始化工作包括在 DRAM 中构建辅助管理结构(即线段树结构)。构建线段树的步骤为：1）划分 NVM 空间；2）建立相应的 NVM 空间管理线段树。线段树叶子节点保存的单位为内存块，一般来说，内存块的大小划分受限于多种因素，包括操作系统内核架构、进程管理、存储管理等。在 NMST 方法的设计中，线段树叶子节点用来保存对应的一段 NVM 区间的信息，包括 data，lson 和 rson，它们分别代表该节点管理 NVM 空间的数据域、左儿子指针和右儿子指针。

NVM 空间管理线段树的构造是一个二分递归的过程。其总体思路是从区间[1，n](其中 n 为 NVM 线段树所要管理的空间的块数目)开始，以二分形式进行拆分，将左半区间分配给左子树，右半区间分配给右子树，然后继续递归构造左右子树；当拆分到单位区间时(即遍历到了线段树的叶子节点，此时节点的左地址等于右地址，叶子节点管理一个内存块)，执行回溯操作；回溯时，对于任何一个非叶子节点，需要根据两棵子树的情况进行统计，并计算当前节点的数据域。

NVM 空间管理线段树的构造过程如算法 1 所示，构造线段树的调用如下：SEGTREE_BUILD(1，1，r)。其中，p 代表节点编号，l 代表左地址，r 代表右地址。构造 NVM 内存管理线段树的过程中，步骤 1 代表初始化第 p 个节点的数据域，并根据实际情况实现 reset 函数；步骤 4 代表递归构造左子树；步骤 5 代表递归构造右子树；步骤 6 代表进行回溯操作，利用左右子树的信息来更新当前节点。

算法 1　SEGTREE_BUILD(p，l，r)

```
1. Nodes[p]. Reset(p,l,r);
2. IF (l<r)
3. mid=(l+r)≫l;
4. SEGTREE_BUILD (p≪1,l,mid);
5. SEGTREE_BUILD (p≪1|1,mid+1,r);
6. Nodes[p]. UPDATEFROMSON();
```

建立线段树的内存开销与线段树中节点的最大编号 max_num 相关，线段树存放的结构体数组空间也取决于最大的下标 max_num−1。由于线段树的父节点区间平均分割到左右子树，因此线段树是完全二叉树，对于包含 N 个叶子节点的完全二叉树，它一定有 N−1 个非叶节点，共 2N−1 个节点，因此存储线段树需要的空间复杂度为 O(N)。而 crit-bit tree 是前缀树的 bitwise 优化版本，一般来说空间复杂度不会超过节点数目与每个节点长度的积，即 O(N * i)。因此，相比于原来的 crit-bit tree，线段树不会产生更大的存储开销。

3.3　更新 NVM 空间管理线段树

当需要进行内存分配时，就需要更新 NVM 线段树中的管理结构。当需要分配|y−x|大小的内存空间时，就需要更新 NVM 线段树在[x，y]区间的值，更新时所做的具体工作视具体情况而定，可以是将[x，y]区间的值都变成 $value$(即覆盖操作)，也可以是将[x，y]区间的值都加上 $value$(即累加操作)。更新 NVM 空间管理线段树时采用二分的方法，即将[1，n]区间不断拆分成多个子区间[1，r]，当更新区间[x，y]完全覆盖被拆分的区间[1，r]时，则更新[1，r]区间内节点的数据域。

另外，NMST 采取延迟更新的方法对 NVM 空间管理线段树进行更新操作。延迟更新是指：在对 NVM 空间的某个区间进行修改时，并不直接修改该节点在线段树上的所有后代节点，而是只在该区间对应的线段树节点上进行标记，称为 lazy 标记。直到进行实际分配 NVM 的操作，必须要处理该节点的后代节点时，才向下修改后代节点的信息。延迟更新

的思想可以有效提高修改节点信息的效率。

NVM 空间管理线段树的更新过程如算法 2 所示。具体过程的解释如下：步骤 1 代表区间$[l,r]$和区间$[x,y]$无交集，函数直接返回；步骤 3 代表区间$[x,y]$完全覆盖$[l,r]$；步骤 4 代表更新第 p 个节点的数据域；步骤 6 代表进行延迟更新操作，记录 lazy 标记；步骤 8 代表递归更新左子树；步骤 9 代表递归更新右子树；步骤 10 代表进行回溯操作，利用左右子树的信息来更新当前节点。

算法 2 SEGTREE_INSERT(p,l,r,x,y,val)

```
1. IF(!is_intersect(l,r,x,y) )
2. return;
3. IF(is_contain(l,r,x,y) )
4. Nodes[p]. UPDATEBYVALUE(val);
5. return;
6. Nodes[p]. GIVELAZYTOSON();
7. mid=(l+r)≫1;
8. SEGTREE_INSERT(p≪1,l,mid,x,y,val);
9. SEGTREE_INSERT(p≪1|1,mid+1,r,x,y,val);
10. Nodes[p]. UPDATEFROMSON();
```

3.4 NMST 的内存分配及其优势

在 NVML 函数库编程模型中，构造线段树来管理一段 NVM 空间之后，便可进行持久性内存分配。在 DRAM 中使用线段树作为辅助管理结构，进行持久性内存空间分配的过程为：首先，判断 NVM 线段树所管理的内存空间是否满足要申请分配的空间需求，若申请的内存区间长度大于根节点表示的最大空间，则将该申请挂起，否则在空闲内存区间查找满足要求的节点；然后，修改线段树中与内存区间相关的节点信息，并且标志该区间的状态为已用。

在 NVML 函数库编程模型使用的 crit-bit tree 管理结构下，分配持久化内存空间时总是需要进行一次插入节点操作和一次删除节点操作。实际的应用程序执行过程中存在大量的上述两种操作，而这两种操作都需要动态分配内存和添加互斥锁等操作，从而导致了很大的延迟。当采用线段树结构代替 DRAM 中的 crit-bit tree 结构来管理 NVM 空间时，就构造出了一棵 NVM 线段树。采取线段树结构代替原有的分配管理结构，能降低内存分配的延迟，这主要是由于线段树的静态特性，使得在分配非最小单位整数倍的内存空间时无需进行多余空间的回插操作，即减少了 NVM 分配时回插操作以及数据结构计算的开销。

4 NMST 线段树管理的优化

在实现了基本的 NMST 线段树之后，我们测试了在 NVML 中当持久性内存分配的大小小于 128 kB 和小于 8 kB 时，分配某个大小(64 字节的倍数)的持久性内存所需要的块数目(最大是 8 个块)情况。统计结果如表 1 所列，从中可以看出，在 NVM 内存分配的大小小于 8 kB 时，有 75 种情况都可以由某种粒度的块的“1 块”来满足分配要求；有 24 种情况需要分配某种粒度的“2 块”才能满足要求。由此可以得出结论：在持久性内存分配中，尤其是小粒度的内存块分配时，绝大多数情况下只需要某些粒度的“1 块”即可满足分配要求。

表 1 不同大小的持久性内存分配块数目需求

Table 1 Number of blocks required for different sizes of persistent memory allocation

分配块数	分配大小<8 kB	分配大小<128 kB
1	75	546
2	24	296
3	2	286
4	10	298
5	0	276
6	6	278
7	0	272
8	11	155

本文中的 NMST 管理方法是以一个 block 作为 NVM 线段树叶子节点单位的，而基于线段树的管理结构在维护连续的空闲空间时存在较大的开销，因此 NMST 的性能仍然有待提升。其原因在于：那些不能以某种粒度“1 块”方式满足的持久性内存分配过程会导致线段树进行连续空间的维护操作，进而带来性能损失。如果在每次分配持久性内存时都能够用某种粒度的“1 块”方式解决，则可以避免上述开销。

然而，NVM 线段树是一棵静态树，在其构造阶段就已经把整块 NVM 空闲空间管理起来了，构造好的 NVM 线段树每次分配持久性内存的最小粒度是固定的。因此，解决上述问题的关键在于：如何让静态的 NVM 线段树具有一定的动态特性，即能够将不同大小的分配需求均转换成一次 NVM 线段树更新操作。

鉴于此，本文提出了优化的 NMST 方法。优化后的 NMST 方法采取构造多种叶子节点粒度的 NVM 线段树的方法来解决这一问题，即在管理 NVM 空间时，每棵 NVM 线段树管理一个 chunk，而这些 chunk 又按照不同粒度划分，如 1 个 block、2 个 block 等，一直到 8 个 block，线段树的叶子节点就以这些粒度来组织。这样，在 NVML 的初始化阶段，就已构造好这些不同叶子粒度的线段树。在需要分配持久性内存时，直接按照能满足分配要求的最小粒度来选择要分配的 chunk(即确定了本次持久性内存分配是在哪一种叶子节点粒度的 NVM 线段树管理空间上进行的)，就可以在尽可能减少内存分配碎片的基础上，避免 NVM 线段树管理结构维护连续空闲空间的操作，进一步减小了持久性内存分配的延迟。

5 实验评估

5.1 实验平台及测试程序

本文的实验验证平台采用的硬件环境是 Dell PoweEdgeT430 服务器，配置为：Intel C610 芯片组，Intel(R) Xeon(R) CPU E3-1220 系列处理器，4 个处理器核，内存容量为 32 GB，硬盘为 1TB 机械硬盘。软件环境为：Ubuntu 14.04 server 版，NVML v1.2 版本，valgrind 软件套装中的 callgrind 程序性能监测统计工具，以及可视化程序性能结果文件查看工具 FlameGraph。另外，由于只是模拟分配 NVM 空间时 NVML 中 DRAM 上辅助管理空间的数据结构优化，实验测试的性能提升来源于数据结构的分配，而对于 NVM 内存分配之后的读写操作不加考虑，因此无需对 NVM 的访问延迟进行模拟。

为了评估持久性内存分配的性能，首先测试了 NVML 中 obj_oom 文件目录下的一系列测试程序，它们是 NVML 自带的标准测试程序，用来进行持久性内存分配操作。具体做法是，按照指定的空闲空间和分配大小不断地进行持久性内存分配操作，直至设定的空闲空间被分配完毕为止。为了减小实验误差，提高实验结果的可靠性，将持久性内存空闲空间设置得较大，然后测试以不同的分配粒度分配完这块持久性内存空间所需要的时间，最终以进行大量持久性内存分配的平均延迟大小来衡量函数库编程模型的性能。

另外，为了更加真实地反映本文提出的 NMST 方法和优化的 NMST 方法对持久性函数库编程模型的优化效果，需要进一步测试在实际的程序负载运行时由于编程模型的优化所带来的程序整体性能的提升。因此，我们还选择了另外一些位于 pmembench 目录下的基准测试程序，包括：btree，rbtree 和 hashmap_TX，它们均调用 NVML 函数库编程模型来实现对 NVM 空间的使用，执行过程中均涉及到持久性内存的分配。通过测试这些测试程序执行各自操作时的吞吐率来衡量 NMST 方法的效果。

本文在实验过程中利用了程序性能分析工具 callgrind 和可视化结果查看工具 FlameGraph 来直观地反映评测结果；另外，上文分析过程中一些性能瓶颈的分析等结论，也是通过这两种工具监测统计之后直观呈现出来的。

5.2　实验结果及分析

首先，在评估 obj_oom 文件下一系列程序的性能时，对多次持久性内存分配的情况进行了测试。为了保证实验结果的可靠性，持久性内存分配的总次数的量级达到 $10^5 \sim 10^8$，每次分配的持久性内存块大小分别为 128 kB，256 kB，384 kB 和 448 kB，取大量测试结果的平均值来进行对比分析。结果如图 5 所示，图中横坐标表示每次分配的持久性内存块的粒度，纵坐标为每次分配持久性内存的平均延迟。空白条柱代表传统的 NVML 内存管理方法，阴影斜纹条柱代表采用 NMST 持久性内存管理方法。

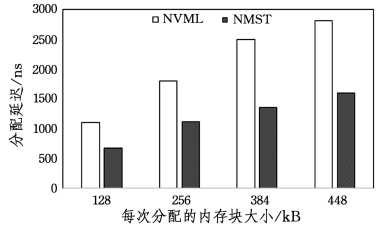

图 5　NMST 内存分配延迟

Fig. 5　Memory allocation delay of NMST

从图 5 可以看出，从左往右，随着每次分配的持久性内存块大小递增，分配每块的延迟也有所增加，这是由于每次分配的内存粒度越大，需要更新的管理结构叶子节点就越多，延迟也就相应地越长。实验以不同的持久性内存块粒度来分配 NVM 空闲空间，对于不同粒度，相比于传统的 NVML 方法，使用 NMST 方法都能带来性能提升，平均来说，内存分配的延迟降低了 36.9%。

采用优化的 NMST 管理方法之后，由于省去了 NVM 线段树维护连续空间的一些开销，持久性内存分配的性能得到了进一步的提升。具体测试结果如图 6 所示，其中 OP_NMST 代表优化后的 NMST 方法。平均来说，优化的 NMST 方法相比于基本的 NMST 方法，内存分配延迟降低了 7%～13%；相比于传统的 NVML 内存管理方法，延迟降低了 43.6%。另外，从图 6 中还可以看出，针对不同粒度的持久性内存块分配，优化的 NMST 方法均能达到较好的优化效果。相比于 NVML 原始的内存管理方法，NMST 的平均性能优化范围为 33.6%～38.9%，而优化后的 NMST 的性能优化范围为 40.2%～46%。

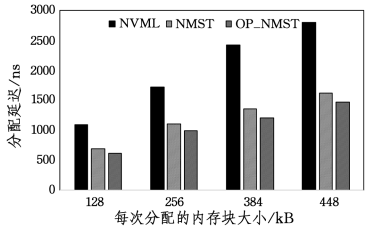

图 6　优化的 NMST 内存分配延迟

Fig. 6　Memory allocation delay of optimized NMST

对 pmembench 目录下的基准测试程序的测试结果如图 7 所示，其中横坐标代表 3 种基准测试程序，纵坐标表示程序执行各自操作的归一化的吞吐率。可以看出，实际应用程序执行时，相比于原来的 NVML 函数库方法，采用 NMST 方法使吞吐率平均提升了 13.8%，而优化的 NMST 方法的程序吞吐率平均提升了 16.4%。

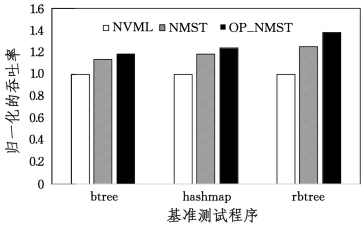

图 7　基准测试程序的吞吐率

Fig. 7　Throughput of benchmark programs

性能的提升来源于这些基准测试程序在执行的过程均需执行持久性内存分配操作，而且每次分配的持久性内存大小不定，因此 NMST 方法和优化的 NMST 方法提升了这些程序的吞吐率。性能的提升与程序中的持久性内存分配的次数以及粒度相关。

结束语　本文提出了一种基于线段树的持久性内存管理分配优化方法 NMST，其通过优化函数库编程模型中的空闲空间组织方式，可以实现快速的持久性内存分配；考虑到线段树在维护连续性空间时开销较大的问题，进一步提出了优化的 NMST 方法，其通过构造多种粒度的叶子节点的线段树来避免上述问题。

实验结果表明，相比于传统的 NVML 内存管理方法，NMST 方法在分配持久性内存时使延迟降低了 36.9%，而优化后的 NMST 方法在分配持久性内存时使延迟降低了 43.6%。在评估的 NVML 库自带的真实测试程序中，NMST

（下转第 115 页）

analytical chemistry[J]. Microchemical Journal, 2016, 124: 45-54.

[12] ABIDIN H Z, DIN N M, JALIL Y E. Multi-Objective Optimization (MOO) Approach for Sensor Node Placement in WSN[C]// 2013 7th International Conference on Signal Processing and Communication Systems (ICSPCS). IEEE, 2013: 1-5.

[13] ROSEN K H. Discrete Mathematics and its Applications[M]. NewYork: McGraw-Hill, 2012.

[14] FEARNLEY J, SAVANI R. The complexity of the simplex method[C]// Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing. ACM, 2015: 201-208.

[15] CARNIELLI W, MARIANO H L, MATULOVIC M. Reconciling first-order logic to algebra[J]. CLE e-Prints, 2015, 15(1): 1-26.

[16] CARNIELLI W, MATULOVIC M. The method of polynomial ring calculus and its potentialities[J]. Theoretical Computer Science, 2015, 606: 42-56.

[17] ZHANG H, HOU J C. Maintaining sensing coverage and connectivity in large sensor networks[J]. Ad Hoc & Sensor Wireless Networks, 2005, 1(1/2): 89-124.

(上接第 83 页)

以及优化的 NMST 方法与传统的 NVML 方法相比,性能提升了 10%～20%,具体提升程度与程序中的持久性内存分配次数及分配粒度有关。

参 考 文 献

[1] WONG H S P, RAOUX S, KIM S B, et al. Phase Change Memory [J]. Proceedings of the IEEE, 2010, 98(12): 2201-2227.

[2] APALKOV D, KHVALKOVSKIY A, WATTS S, et al. Spin-transfer torque magnetic random access memory (STT-MRAM) [J]. ACM Journal on Emerging Technologies in Computing Systems, 2013, 9(2): 13.

[3] LEE C B, KIM C J, LEE D S. Resistive random access memory: US, US 20090302315 A1 [P]. 2009.

[4] Intel and Micron. Intel and micron produce breakthrough memory technology[EB/OL]. (2015-07-28)[2017-02-01]. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology.

[5] JIANG T, ZHANG Q, HOU R, et al. Understanding the behavior of in-memory computing workloads[C]// IEEE International Symposium on Workload Characterization. IEEE, 2014: 22-30.

[6] NALLI S, HARIA S, HILL M D, et al. An analysis of persistent memory use with WHISPER[C]// Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2017: 135-148.

[7] Intel. NVML: Non-volatile memory library[EB/OL]. [2017-02-01]. https://github.com/pmem/nvml.

[8] VENKATARAMAN S, TOLIA N, RANGANATHAN P, et al. Consistent and durable data structures for non-volatile byte-addressable memory[C]// Proceedings of the 9th USENIX Conference on File and Storage Technologies. 2011: 61-75.

[9] YANG J, WEI Q, CHEN C, et al. Nv-tree: reducing consistency cost for nvm-based single level systems[C]// Proceedings of the 13th USENIX Conference on File and Storage Technologies. 2015: 167-181.

[10] CONDIT J, NIGHTINGALE E B, FROST C, et al. Better i/o through byte-addressable, persistent memory[C]// Proceedings of the 22nd ACM Symposium on Operating Systems Principles. ACM, 2009: 133-146.

[11] WU X, QIU S, NARASIMHA R A L. Scmfs: A file system for storage class memory and its extensions[J]. ACM Transactions on Storage, 2013, 9(3): 7.

[12] DULLOOR S R, KUMAR S, KESHAVAMURTHY A, et al. System software for persistent memory[C]// Proceedings of the Ninth European Conference on Computer Systems. ACM, 2014: 15.

[13] XU J, SWANSON S. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories[C]// Proceedings of the 14th USENIX Conference on File and Storage Technologies. 2016: 323-338.

[14] VOLOS H, TACK A J, SWIFT M M. Mnemosyne: Lightweight persistent memory[C]// ACM SIGARCH Computer Architecture News. ACM, 2011, 39(1): 91-104.

[15] COBURN J, CAULFIELD A M, AKEL A, et al. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories[J]. ACM Sigplan Notices, 2011, 46(3): 105-118.

[16] VOLOS H, NALLI S, PANNEERSELVAM S, et al. Aerie: flexible file-system interfaces to storage-class memory[C]// Proceedings of the Ninth European Conference on Computer Systems. ACM, 2014: 14.

[17] GAO L S, IYER B. Analyzing Complementarities Using Software Stacks for Software Industry Acquisitions [J]. Journal of Management Information Systems, 2006, 23(2): 119-147.

[18] SCHWALB D, BERNING T, FAUST M, et al. nvm_malloc: Memory allocation for nvram[C]// Accelerating Data Management Systems Using Modern Processor and Storage Architectures Workshop. In conjunction with VLDB, 2015.

[19] BHANDARI, KUMUD, DHRUVA R, et al. Makalu: Fast recoverable allocation of non-volatile memory[C]// Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM, 2016.

[20] HANANDEH F, AISMADI I, KWATHA M M. Evaluating alternative structures for prefix trees[C]// Proceedings of the World Congress on Engineering and Computer Science. 2014: 109-114.