持久性内存系统中
高效的数据一致性机制研究

（申请清华大学工学博士学位论文）

培 养 单 位：计 算 机 科 学 与 技 术 系
学　　科：计 算 机 科 学 与 技 术
研　究　生：任　晶　磊
指 导 教 师：郑　纬　民　教　授

二〇一五年十二月

Efficient Mechanisms for Supporting Crash Consistency in Persistent Memory Systems

Dissertation Submitted to
Tsinghua University
in partial fulfillment of the requirement
for the degree of
Doctor of Philosophy
in
Computer Science and Technology

by

Jinglei Ren

Dissertation Supervisor : Professor Weimin Zheng

December, 2015

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）根据《中华人民共和国学位条例暂行实施办法》，向国家图书馆报送可以公开的学位论文。

本人保证遵守上述规定。

（保密的论文在解密后应遵守此规定）

作者签名：_____ 导师签名：_____

日　　期：_____ 日　　期：_____

摘 要

新型非易失性存储介质，诸如闪存（flash）、相变内存（phase-change memory，PCM）、可变电阻式内存（ReRAM）等，可同时提供传统硬盘等外部存储器的数据持久化能力和接近动态随机访问内存（DRAM）等内部存储器的存取性能。非易失性内存介质及其软硬件系统共同构成持久性内存（persistent memory）系统，可以融合传统易失性内部存储和非易失性外部存储的优良特性，提升上层应用软件和系统整体的性能。

持久性内存系统使得内存数据在系统发生故障时依然得以保留。该特性在减少传统持久化机制带来的性能损耗方面作用显著，但于此同时，也使得发生系统故障时的数据一致性（crash consistency）问题尤为突出。而为保证故障时的数据一致性，往往需要对上层应用程序访问内存的接口加以限制。因此，数据一致性机制及其应用程序接口方式在持久性内存系统性能、易用性及两者间的平衡等方面扮演着非常重要的角色。

从应用层到系统层，再到硬件层，持久性内存提供接口的形式主要包括文件系统、事务性内存和硬件接口等。本论文研究了持久性内存系统在文件系统、事务性内存和软件透明三种主要数据存取接口方式下，如何设计和实现高效的故障时数据一致性保证机制的问题。主要创新点和研究成果包括：

- 文件系统接口下的多版本缓存事务技术。将原子性事务（atomic transaction）机制引入到操作系统页缓存中，解决由动态内存和非易失性内存构成的持久性内存系统的故障时数据一致性问题。将该技术应用于移动系统环境，在新的持久性内存系统假设下，改进现有手机文件系统设计，提出优化手机能耗和应用响应的新指标及相应的三组新算法。实际测试表明，该技术可使现有安卓平台上应用的响应时间和能耗分别下降 51.6% 和 35.8%。
- 事务性内存接口下的小缓冲区组技术。根据 NVM Express 接口和固态硬盘的新特性，为事务性内存设计了一种高效的一致性持久化机制，为持久性内存提供了一种新的实现方式。该系统采用的快照隔离技术可以使实时分析等只读负载不受持久化开销的影响；小缓冲区组（small buffer array）的设计，在保证故障时数据一致性的同时，可显著降低组提交（group commit）中提交者相互等待的时间，兼得理想的吞吐量和延迟。实验测评中，该设计在读写混合的负载下，可比最优的传统实现的性能提高 32.5%。
- 软件透明接口下的双模式检查点生成技术。提出支持对软件透明的故障时

数据一致性的混合持久性内存设计方法，通过双模式检查点生成技术高效地生成一致的可恢复的检查点。该方法同时在缓存块粒度和操作系统页粒度上产生检查点，可使软件执行与产生检查点的延迟重合，实现的停滞时间比页粒度的检查点生成机制减少了 86.2%；同时实现可行的存储空间占用，是块粒度检查点生成机制所需存储空间的 26%。

- 软件透明的数据一致性协议及其形式化证明。双模式检查点生成技术，对数据一致性保证提出了新的挑战。多个数据版本的隔离和维护，在程序执行和生成检查点过程重合的情况下变得尤为复杂。为此提出并利用状态机模型表达了故障时数据一致性协议；对代码级实现进行了符号抽象，利用不变式和数学归纳法对故障时数据一致性协议的正确性进行了形式化证明。

关键词：内存；非易失性；持久化；事务性内存；检查点

Abstract

Non-volatile memories (NVMs), such as flash, phase-change memory (PCM) and ReRAM, feature both the persistence capability of external storage and the high performance of internal memory. They promise persistent memory, an emerging tier in the memory and storage stack. Persistent memory has the potential to significantly increase the efficiency of the current system architecture.

Persistent memory systems ensure durability of memory data on system failures such as power outages and system crashes. This distinctive feature can save overheads of traditional data persistence mechanisms, but introduces a critical challenge: crash consistency of memory data. In order to guarantee such crash consistency, programs are typically constrained in accessing memory data by a certain form of software interface. The interface choice and its corresponding crash consistency mechanism largely determine the system performance and the ease of programming.

This dissertation presents my research on efficient mechanisms for supporting crash consistency in persistent memory systems, under three main forms of interfaces: file system, transactional memory, and the software-transparent interface. Main contributions of this dissertation include the following.

- For file systems, we introduce atomic transactions to the page cache of an operating system to ensure that memory data is flushed to persistent storage in a consistent manner. This technique is applied to smartphones whose DRAM and flash can be deemed as a persistent memory system. In order to optimize the energy efficiency and app responsiveness of smartphones, we design app/user-adaptive policies and algorithms to quantitatively trade off data staleness for energy efficiency/app responsiveness.
- For transactional memories, we propose a new buffering and group commit design, small buffer array, according to the characteristics of (potentially NVRAM-enhanced) NVM Express-attached SSD. It largely reduces the waiting latency in group commit, while saturating the bandwidth of the flash device. Moreover, we employ snapshot isolation to hide write latency from the critical path of read-only transactions, which brings significant performance improvement to real-time analytical workloads.

- With software-transparent interfaces, programs can safely access memory data using regular load/store instructions. Programmers do not need bother partitioning transient and persistent data or writing transactional code, but enjoy unmodified legacy code and better portability than using a particular transaction library. We establish the importance of this software-transparent approach and, to enable the approach, propose an efficient consistent dual-scheme checkpointing mechanism which synchronously checkpoints memory data at different granularities.
- Dual-scheme checkpointing brings a new challenge to the crash consistency guarantee: the isolation and maintenance of multiple versions of data are complicated by the overlap of program execution and checkpointing. We propose to use a state machine to model the consistency protocol, and formally prove the correctness of the protocol.

Key words: memory; persistence; non-volatile; transactional memory; checkpointing

第 5 章　双模式检查点生成技术

新兴的按字节编址的（byte-addressable）非易失性内存介质，诸如 STT-RAM[7,8]、PCM[11,12] 和 ReRAM[13] 等，预示着持久化内存系统的兴起。持久化内存可能发展成为内存和存储栈中一个新的中间层。持久化内存系统融合了传统内存系统（快速的 load/store 指令接口）和存储系统（数据持久化）的优良特性，模糊了两者之间的界限。它给应用带来的一个重要的益处是，可以通过 load/store 指令高效地直接地访问内存中的持久化数据，而不需要与存储设备进行页交换（page swapping）、在（反）序列化时更改数据格式以及触发臃肿的系统调用[14]。

持久化内存相对于传统的易失性内存系统引入了一个关键的要求，系统故障时的一致性保证[92,148,149]。这要求持久化内存系统，在掉电或系统失效等故障出现时，依然能够保证数据的一致性不受部分的或是乱序的 NVM 写的影响。我们以原子性地更新保存在 NVM 上的数据结构 A 和 B 为例，总会有对 A 或 B 的一个更改会先于另一个写入 NVM。如果系统在一个更改完成之后出现故障或掉电，两个数据结构可能处于不一致的中间状态，只包含部分更改。对于传统易失性内存，这不是一个问题，因为程序恢复后内存中的数据都会丢失。但对于持久化内存，这些不一致的数据会一直保留。所以，持久化内存系统需要保证保存到 NVM 的数据可以在系统重启后恢复到一个一致的状态，即维持数据的一致性。维持数据的一致性原本仅是对磁盘或闪存等存储系统的要求，但将数据持久化引入内存后，它即成为内存系统同样面临的一个挑战。

大多数之前的持久化内存设计依赖于程序员的人工努力来保证系统故障时的数据一致性[22,23,73,86,87,89]。应用的开发人员需要显式地使用特定的编程模型和软件接口来访问和操作内存中的持久化数据，以保护数据一致性不受部分写或是乱序写的影响。这种方式似乎给程序员提供了对数据持久化的完全控制，但要求程序员使用新接口管理持久化内存会带来多方面不良影响。首先，程序员必须用新的编程接口 API 来实现程序或者深度更改遗留代码，通常要显式地声明和划分持久的和临时的数据结构。其次，已经部署在各种系统中的使用遗留代码的应用无法利用持久性内存系统。第三，大多数之前的持久化内存设计需要使用事务性内存控制版本和对 NVM 写的顺序，而事务性内存的扩展性依然面临挑战[150–152]。第四，持久性内存应用的实现依赖特定的软件接口和运行时系统[22,23,86–88]，会带来跨系统的兼容性和移植性问题

本章工作的**目标**是为持久性内存系统设计高效的对软件透明的故障时数据一

致性保证机制。我们希望这样的设计可以使更多持久性内存的用例成为可能（如用于不修改的遗留程序或者非事务性的程序），并允许更多的程序员使用持久性内存系统而不必更改应用来保证数据一致性。

为此，我们提出了 ThyNVM，一种新的支持对软件透明的故障时数据一致性的基于 DRAM 和 NVM 混合架构的持久化内存系统。它允许基于事务的和非事务性的程序直接在持久化内存硬件上运行，而且获得完整的故障时数据一致性支持。ThyNVM 通过定期地在硬件辅助下生成检查点来保证系统在故障时可以恢复到一个一致的内存状态。然而，一般的检查点生成技术需要在将数据持久化到 NVM 的过程中令整个系统停顿。减少由于使用检查点生成技术导致的程序停顿是此类系统面临的关键挑战。

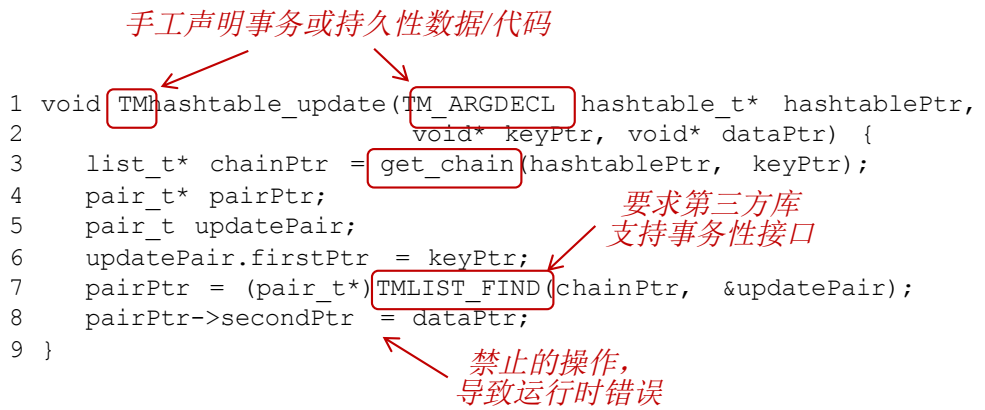
ThyNVM 采用两个方法减少检查点生成的开销。第一，它将检查点生成过程和程序执行过程重合。第二，为了高效地支持这种重合，它动态地决定数据生成检查点的粒度。该决定依据我们新的观察，即在应用停滞时间和元数据存储开销之间存在一个权衡。在较小的粒度上生成检查点，带来的停滞时间比较短，而对应元数据占用的空间却会非常大；在较大的粒度上生成检查点，可以降低元数据占用的空间，但是会导致较长的停滞时间。因此，单一的检查点生成技术（或者基于小粒度或者基于大粒度）难以达到最优的效果。为了解决这一问题，我们提出了*双模式检查点生成机制*，它可以同步地为分散的（低空间局部性）和集中的（高空间局部性）内存写分别基于 CPU 缓存块粒度和操作系统页粒度生成检查点。与使用日志[22,23] 或写时拷贝（copy-on-write，COW）[86,94] 的持久化内存设计相比，ThyNVM 显著减小了存储的损耗并增加了内存带宽的利用率。

综上，本章工作做出了如下几点贡献：（1）我们提出了一种新的持久化内存设计，支持对*软件透明*的故障时数据一致性保障。我们的设计允许基于事务的和无修改的遗留应用通过 load/store 指令接口使用持久化内存。（2）对于生成检查点的粒度，我们定位了*应用停滞时间和元数据空间开销之间的权衡关系*。在任意一个粒度上生成内存数据的检查点都不是最优的。（3）我们设计了一个新的*双模式检查点生成技术*。我们发现，低空间局部性的更改最好以细粒度生成检查点，而高空间局部性的更改最好以粗粒度生成检查点。我们的设计比页粒度的检查点生成机制减少了 86.2% 的停滞时间，而仅用块粒度的检查点生成机制所需硬件空间的 26%（位于内存控制器）。（4）我们在内存模拟器上使用我们经过形式化证明的一致性协议实现了双模式检查点生成机制。它可以在保证数据一致性的同时，通过将程序执行和检查点生成过程重合获得高性能。我们的解决方案可以达到不提供一致性支持的全 DRAM 系统性能的 95.1%。

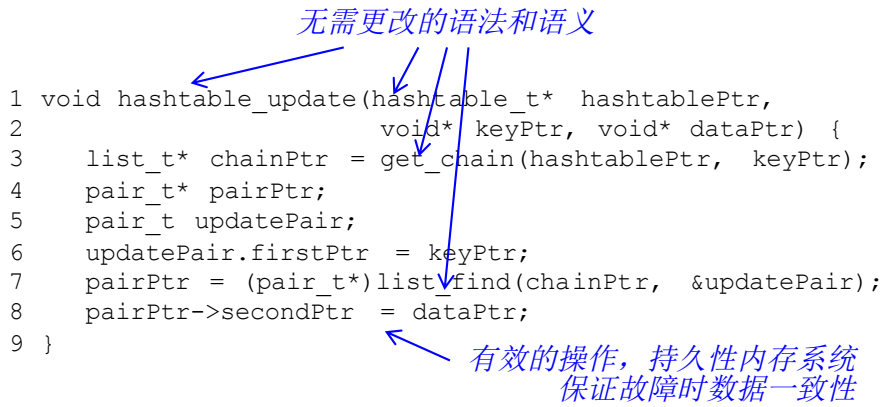
5.1　软件透明的一致性保证机制

ThyNVM 包含了两个设计选择：（1）对故障时数据一致性的支持是软件透明的，而不是依赖软件技术；（2）故障时数据一致性机制是基于检查点生成技术，而不是日志或写时拷贝。本节讨论采用这些技术选择的原因。

5.1.1　软件一致性保障技术的缺陷



(a)



(b)

图 5.1　实例代码：向持久化的哈希表中插入一个项。分别采用（a）传统的软件接口或（b）软件透明的 ThyNVM。

之前大多数持久化内存设计[22,23,86,89,94]要求应用开发者显式地定义持久性数据并使用特定库来访问数据——故障时数据一致性保障机制与软件的语义紧耦合。我们通过图5.1中的实例代码来展示这种持久化内存设计的不足。该图比较了一个更新持久性哈希表项的软件方法的两种实现（代码改编自 STAMP[153]）。图5.1（a）展示了使用类似于现有设计[22,94]的软件事务的实现；而图5.1（b）中的实例代码

采用了 ThyNVM 的对软件透明的故障时数据一致性保证机制。我们可以看到，在支持数据一致性时涉及软件支持，会有多方面的不足。

首先，手工划分临时性和持久性数据很麻烦而且容易出错。图5.1（a）展示了许多无法避免的标记。程序员需要对软件行为有深入的了解，并且小心地确定哪些数据结构需要持久化以及如何操控这些数据。例如，图5.1（a）的第 3 行从哈希表中读出一个链表。该行仅当这个哈希表实现固定数目的链表时才是正确的。否则，另一个并发线程可能正在重置这些链表而导致错误。为了防止这个问题出现，程序员需要使用事务，来保护对链表的访问（程序第 3 行）不受并发线程的影响。

其次，在一个统一的地址空间内，临时性和持久性数据之间的引用需要小心的管理。然而，对该问题的基于软件的解决方案并非最佳。例如，为了处理 NV-to-V 指针（非易失性指针指向易失性数据，图5.1（a）的第 8 行），NV-heaps[23] 会简单地抛出一个运行时错误以禁用此类指针。

第三，应用需要使用事务，通常是通过事务性内存接口（例如被 Mnemosyne[22] 使用的 TinySTM[154]），来更新持久化数据，如图5.1（a）所示。然而，不论基于硬件的还是基于软件的事务性内存，都存在多方面的可扩展性问题[150–152]。此外，开发人员必须使用新的 API 来实现或者重写各种代码库和程序，需要不可忽略的人工投入。不熟悉这些新 API 的程序员需要一个痛苦的学习过程。

最后，应用需要构建在特定的库之上，例如 libmnemosyne[22] 和 NV-heaps[23]。这会很大程度上降低持久化内存应用的移植性——在一个库上实现的应用，如果想采用其他的库，需要重新实现。编译器插装确实可以在一定程度上减轻程序员实现和移植代码的负担。然而，编译器会无差别地插装所有（或大部分）内存的读和写操作，带来可见的性能损耗。我们在 GCC libitm[155] 上实验了 STAMP 事务性标准测试[153]，结果插装本身最多可带来高达 8% 的性能损失。

我们采用了允许软件程序直接访问持久化内存而无需特别的支持故障时数据一致性的接口的使用模型。如图5.1（b）所示，我们可以不更改代码而实现数据结构的持久化。

5.1.2　日志及写时拷贝技术的缺陷

现有基于软件的持久化内存设计多采用日志[22,23] 或者写时拷贝技术[86,94] 维持数据一致性。基于日志的系统在日志中维护原始数据的额外副本，用以保存新的数据更新（重做日志）或者旧的数据值（撤销日志）。基于写时拷贝的系统在数据更新时会创建一个新的副本用于更新。然而，两种技术为支持数据一致性保障会引入较大的性能损耗。

日志可能消耗比原始数据大很多的 NVM 空间，因为每个日志项都是一个既包含数据又包含元数据（例如数据所在的地址）的元组，而且通常每次写操作都需要记录[22,23]①。此外，日志重放增加了故障后系统恢复时间，会抵消使用 NVM 而不是顺序访问的块设备带来的快速数据恢复的好处。

写时数据拷贝有两个缺点。首先，拷贝操作代价较大，会引发较长的停滞时间。其次，它不可避免地会复制未更改的数据，消耗额外的 NVM 带宽。这个缺陷在更新的地址很分散时尤为突出。

相比之下，生成检查点是一种更加灵活的方法，它定期地将易失性的数据及元数据写入 NVM 形成一致的内存数据的快照。我们采用检查点生成技术来克服日志和写时拷贝的弊端，但同时，降低检查点生成的开销尤为重要。为此，本章设计了新的检查点生成机制。

5.2　双模式检查点生成技术

5.2.1　粒度的权衡

检查点生成主要是获得数据工作副本的快照（工作副本即正在被活跃地更新的数据副本），并将其持久化到 NVM 中（即为检查点）。检查点生成主要有两个问题：（1）生成数据工作副本检查点的延迟；（2）为追踪数据的工作副本和检查点的位置而储存的元数据的开销。理想情况下，我们希望将两者都最小化。然而，它们之间存在复杂的权衡关系。在设计高效的检查点生成机制时，我们需要考虑这些权衡关系。

首先，元数据的开销决定于我们追踪数据工作副本和检查点的粒度。我们将该粒度称为检查点粒度。通过使用较大的检查点粒度（例如操作系统页粒度），我们只需要小量的元数据即可追踪大量数据的位置。②

第二，检查点生成延迟会受到数据工作副本的位置的影响。在一个混合内存系统中，工作副本可以保存在 DRAM 或者 NVM 中。我们分析了这两种选择.（1）DRAM：因为 DRAM 写入比 NVM 更快，将工作副本“缓存”在 DRAM 中可以提升应用的写操作的性能。然而，如果我们把工作副本缓存于 DRAM 中，该副本需要在生成检查点的过程中被写回到 NVM 中，导致较长的检查点生成延迟。（2）NVM：我们可以通过将工作副本保存在 NVM 中并在收到内存写请求时直接在 NVM 原地更新数据的方式来显著减小检查点生成的延迟。因为工作副本已经在

① 某些对于日志的改进或变形[105] 使用一个索引结构来合并更新。该设计即与我们双模式之一类似。

② 我们可以与 CPU 缓存的标签（tag）类比：小的缓存块（细粒度）造成较大标签开销，而大的缓存块（粗粒度）需要较小的标签开销。

NVM 中持久化，检查点生成变得非常快捷：我们只需要在生成检查点的过程中将元数据持久化即可。然而，这种将工作副本保存在 NVM 中并做原地更新的方式，要求我们在收到应用的写操作时将工作副本重映射到一个新的与检查点数据不同的 NVM 地址上。这样，应用可以在这个新的数据副本上操作而不会破坏检查点。而重映射操作的速度取决于追踪数据的粒度（即检查点粒度）：细粒度可以实现较快的重映射，使得应用可以快速更新数据；而粗粒度会导致缓慢的重映射（因为要求拷贝大量数据，如整个页），进而导致写操作的延迟，使得应用长时间停顿。

综上，表5.1总结了检查点粒度和工作副本位置对检查点生成延迟和元数据开销的影响。

表 5.1　选择不同的检查点粒度和工作副本位置构成的四种组合，以及不同组合的优势和劣势。粗体部分是决定一个组合高效或低效的主要因素。

		检查点粒度	
		小（缓存块）	大（页）
工作副本位置	DRAM（回写）	❶ *低效* ✘ **元数据开销大** ✘ **检查点生成延迟大**	❷ *部分高效* ✔ **元数据开销小** ✘ 检查点生成延迟大
	NVM（重映射）	❸ *部分高效* ✘ 元数据开销大 ✔ **检查点生成延迟小** ✔ 重映射快	❹ *低效* ✔ 元数据开销小 ✔ 检查点生成延迟小 ✘ **重映射慢**（处于关键路径）

选择小检查点粒度和在 DRAM 中保存工作副本（❶）被认为低效的原因是，该组合会带来较大的元数据开销（相对于大检查点粒度❷），而且造成较大的检查点生成延迟（相对于在 NVM 保存工作副本❸）。

我们认为大检查点粒度和在 NVM 中保存工作副本（❹）构成的组合也是低效的，因为它会造成过高的重映射操作的延迟。该延迟会实时地体现在对 NVM 的写操作中，处于应用执行的关键路径上。例如，如果我们在页粒度上跟踪数据，为了重映射一个写入 NVM 的缓存块粒度的更改，我们需要一并拷贝目标页中其他所有没有更改的缓存块到新的副本中，而且只有在这些完成后，应用才可以在该页继续进行写操作。这个重映射的开销在处理应用 store 指令的关键路径上。

基于对这些复杂的权衡关系的观察，我们的结论是，没有单一的检查点粒度或工作副本位置的选择可以很好地满足所有的需求。在 ThyNVM 的检查点生成模式中，我们设计了一个多粒度的检查点生成机制来实现多个检查点粒度和不同的工作副本位置所带来的最优特性。特别地，我们利用表5.1中两个具有互补权衡的部分高效的组合：（1）在小粒度上生成检查点，将工作数据保存在 NVM 中并执行 NVM 写的重映射（❸）。该组合对应于我们块重映射模式（第5.2.3节）。（2）在

大粒度上生成检查点，将工作数据保存在 DRAM 并回写到 NVM 中（❷）。该组合对应于我们的页回写模式（第5.2.4节）。我们设计了两种模式间精巧的合作机制以实现两者的优势（第5.2.5节），即同时获得较小的元数据开销和较短的检查点生成延迟。

5.2.2　系统定义

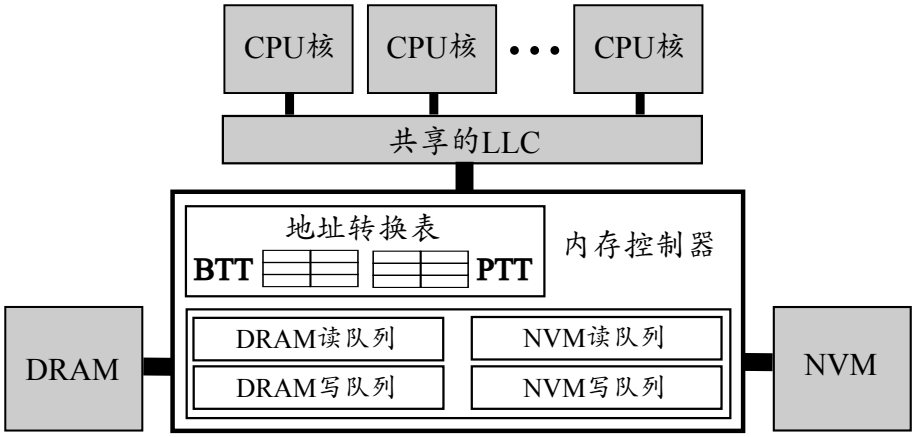


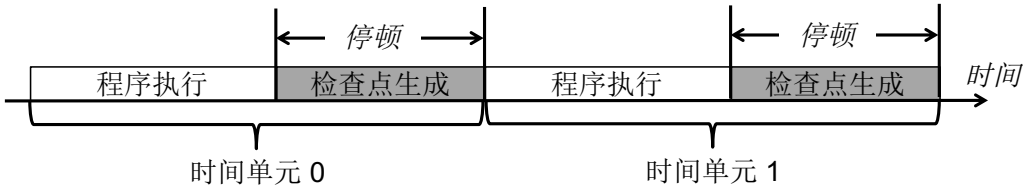
图 5.2　ThyNVM 的总体架构图。

系统架构：ThyNVM 采用 DRAM 和 NVM 的混合架构，基于硬件实现，提供对软件透明的故障时数据一致性保障。图5.2绘制了 ThyNVM 架构总览。DRAM 和 NVM 部署在内存总线上，并映射到同一个物理内存地址空间中。该地址空间暴露给操作系统。我们更改了内存控制器，增加了一个检查点生成控制器，和两个用于维护内存访问元数据的地址转换表。这两个表分别是在缓存块粒度和页粒度管理元数据的块地址转换表（block translation table，BTT）和页地址转换表（page translation table，PTT）。

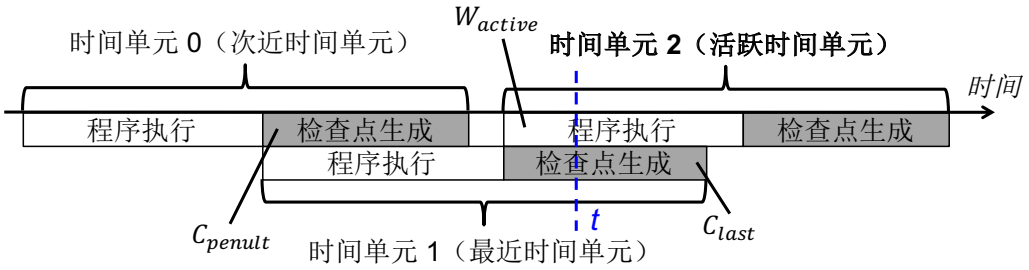
失效模型：ThyNVM 允许软件应用在诸如系统死机、掉电等故障发生后，从一个一致的内存数据检查点开始，继续 CPU 执行。为此，我们需要定期将内存中的数据的更新和 CPU 状态生成检查点，这其中包括 CPU 寄存器、写缓冲、脏缓存块以及内存控制器的写队列。我们的检查点生成机制保护内存数据和 CPU 状态的一致性不因系统故障而被污染。

时间单元模型：我们逻辑上把程序的执行时间分成连续的时间区段，称为时间单元（epoch，如图5.3）。每个时间单元包括一个执行阶段和一个检查点生成阶段。执行阶段更新工作数据，而检查点生成阶段持久化内存中的数据和 CPU 状态。

让执行阶段和检查点生成阶段顺次交替（如图5.3（a）所示）会导致显著的性能下降。对于内存访问频繁的负载，检查点生成可以消耗高达 35.4% 的程序运行



(a) 采用检查点生成时暂停全系统的时间单元模型



(b) ThyNVM 采用的时间单元模型，可将检查点生成和程序执行相互重合。该模型维护三个版本的数据：活跃的工作数据（W_{active}）、最近检查点数据（C_{last}），和次近检查点数据（C_{penult}）。

图 5.3　检查点生成系统的时间单元模型。

时间。所以，ThyNVM 采用了一种将检查点生成阶段和程序执行阶段重合的时间单元模型[105]，以降低该性能损失。我们定义三个连续的时间单元依次为活跃的、最近的和次近的时间单元。如图5.3（b）所示，为了消除由于检查点生成带来的程序停滞时间，在最近时间单元（时间单元 1）刚进入检查点生成阶段时，活跃时间单元（时间单元 2）即开始其执行阶段。注意，只有在次近时间单元（时间单元 0）完成检查点生成后，最近时间单元（时间单元 1）才可以进入检查点生成阶段。如果一个时间单元的检查点生成阶段的时间短于与之重合的下一个时间单元的执行阶段，那么该检查点生成阶段就可以不体现在程序执行的关键路径上，即图5.3（b）所示的情况。

然而，在时间单元的执行阶段，系统需要定位并更新与各时间单元对应的正确的数据版本（下文将详细讨论）。

数据版本：执行阶段和检查点生成阶段重合允许两个相邻的时间单元同时访问相同的数据。这对数据一致性的维护带来了两个挑战。第一，相互重合的执行阶段和检查点生成阶段可能会覆盖对方的数据更改，所以我们需要隔离不同时间单元的数据更改；第二，系统失效发生时，活跃时间单元和最近时间单元的数据可能被同时损坏，所以我们需要维持次近时间单元的数据版本的安全。为了应对这些挑战，ThyNVM 维护了连续时间单元中数据的三个版本：*活跃的工作副本* W_{active}、*最近检查点* C_{last} *和次近检查点* C_{penult}（C_{last} 和 C_{penult} 既包括内存数据也包括 CPU 状态）。以图5.3（b）所示情况为例，ThyNVM 在执行时间单元 2（更新 W_{active}）的时候，保留时间单元 0（C_{penult}）和时间单元 1（C_{last}）中生成的检查点。

ThyNVM 只有在时间单元 1 完成检查点生成后才丢弃时间单元 0 生成的检查点。在时间点 *t* 发生的系统故障可能同时破坏时间单元 2 正在更新的工作副本和时间单元 1 正在生成的检查点。这就是为什么我们需要维护时间单元 0 生成的检查点（C_{penult}）。这样，ThyNVM 总是可以回滚到时间单元 0 结束时的状态，使用 C_{penult} 作为一个安全和一致的内存数据副本。

5.2.3　基于块重映射的检查点生成

块重映射模式在缓存块粒度上生成对 NVM 的更新的检查点。该模式使得应用程序可以在执行阶段直接对 NVM 上的工作副本进行更新。我们通过将一个数据块的工作副本（来自活跃时间单元）重映射到 NVM 上的另一个地址来实现上述目的，而在检查点生成阶段仅需要持久化定位检查点数据所需的元数据。因此，生成一个数据块的检查点不需要移动数据本身。相反，在生成检查点时，已经更新在 NVM 上的数据块可以由作为工作副本的版本直接转成作为最近检查点的版本。所以，块重映射模式可以显著降低生成检查点的延迟。在本文工作中，我们提出对空间局部性低的数据更新采用块重映射模式生成检查点。此类更新大部分是随机的、分散的、粒度较小的（即通常仅涉及一个数据页中的少量块）因此，在块粒度上对这些更改各自生成检查点会比较高效。相比之下，如果在页粒度上生成这些更新的检查点，由于他们通常只变脏每个页的一小部分，就会比较低效。在一个页大部分没有被更改的情况下，还要将整个页回写到 NVM，这会对系统性能造成极大负面影响。

在该模式下，工作副本 W_{active} 在每个时间单元的执行阶段被直接写入 NVM。简单来说，ThyNVM 在块地址转换表 BTT 中记录从数据块到它们工作副本地址之间的映射关系。一个读请求可以通过查看 BTT 获得工作副本的有效地址。在活跃时间单元内，对同一个块的所有更新被重定向到同一个为当前时间单元当前数据块分配的新地址，即工作副本 W_{active} 的地址。这些更改在新地址上可以相继合并。在相应的检查点生成阶段，该工作副本随着 BTT 中映射关系被持久化，即转变为检查点版本 C_{last}。ThyNVM 在每个检查点生成阶段开始的时候将 BTT 持久化到 NVM 中。

5.2.4　基于页回写的检查点生成

页回写模式在页粒度上生成数据更新的检查点。在一个时间单元的执行阶段，热点页被缓存在 DRAM 中；其中的脏页在检查点生成阶段回写到 NVM 中。本工作中，我们提出对空间局部性高的数据更新采用页回写模式生成检查点，因

为这些更新通常是顺序的、聚集的，占用（近乎）完整的数据页。在页粒度而非块粒度上管理这些更新，可以减少硬件为管理这些更新需要追踪的元数据量（如第5.2.1节的描述）。

使用页回写模式管理 DRAM 与管理传统回写式 CPU 缓存不同，体现在如下两个方面。首先，在执行阶段的内存访问不触发任何页的替换或踢出。脏页只在检查点生成阶段被回写到 NVM。第二，在检查点生成阶段 ThyNVM 将 DRAM 中的脏页回写到 NVM 的过程中，不能覆盖现有的检查点数据，因为检查点生成阶段如果被系统故障阻断，我们需要恢复到最近的检查点。所以 ThyNVM 在生成检查点时需要将每个脏页重定向到一个不同的地址上。它使用页地址映射表 PTT 来为内存中的数据页追踪这些地址映射。在检查点生成阶段结束的时候，PTT 会自动被持久化到 NVM 上。这标志着一个时间单元的结束。

5.2.5　协调两种模式

我们设计了两种机制来协调双模式，以达到如下两个目标：（1）减少由于页回写模式生成检查点带来的应用程序停滞时间；（2）调整检查点生成模式来适应动态变化的内存访问行为。

减少由页回写导致的应用程序停滞时间：块重映射允许系统在检查点生成阶段仅持久化元数据。所以，它可以比页回写模式更快地完成一个时间单元的检查点生成阶段。在这种情况下，基于页回写的检查点生成通常会阻碍整个应用程序的执行，因为随后的时间单元无法更新 DRAM 中尚未回写到 NVM 的页。为了缓解这个问题，ThyNVM 使用块重映射模式临时吸纳新来的原本应由页回写模式处理的内存写（属于下一个时间单元），从而允许系统主动地开始下一个时间单元的执行。通过在页回写阻塞程序进度时转换到块重映射，我们隐藏了由页回写导致的停滞时间，实现并行地写 NVM 和 DRAM，并且增加内存的带宽利用率。

两种模式的切换：ThyNVM 动态地将检查点生成模式与数据不断变化的空间局部性和密集程度相匹配。内存控制器在每个时间单元开始的时候，依据页上近期更改所呈现的空间局部性，决定一些页是否需要从一个模式切换到另一个模式。切换检查点生成模式涉及元数据更新和数据迁移。从页回写模式切换到块重映射模式，我们需要剔除 PTT 中的相应的项，同时将 DRAM 中的页移动到 NVM 上由块重映射模式管理的区域。这样，下次该页的任意缓存块被更改时，内存控制器就会使用块重映射模式处理，并相应地将元数据信息添加至 BTT 中。从块重映射模式切换到页回写模式，我们需要在 PTT 加入一个项来存储页的元数据，同时将该页所有缓存块数据（使用 BTT 定位它们）集中拷贝至 DRAM 中新分配的位置。内存控制器处理所有上述操作，其性能开销可被执行阶段隐藏。

5.3　系统实现

本节描述 ThyNVM 的实现。我们首先描述地址空间布局和管理（第5.3.1节）以及元数据管理机制（第5.3.2节）。我们讨论 ThyNVM 如何执行内存的读和写（第5.3.3节）以及如何在生成检查点时刷出数据（第5.3.4节）。最后我们描述 ThyNVM 如何在系统恢复时处理内存数据（第5.3.5节）。

5.3.1　地址空间布局和管理

我们的 ThyNVM 实现基于一个特定的地址空间布局来存储和管理数据的多个版本。软件对*物理地址空间*的视图不同于内存控制器对*硬件地址空间*的视图，如图5.4所示。硬件地址空间比物理地址空间大，且其中部分区域仅对内存控制器可见，用来存储检查点数据和 CPU 状态等。下面我们描述硬件地址空间的不同区域及其目的。

我们的地址空间管理机制有两个关键目标：（1）在硬件地址空间的不同位置存放数据的多个版本，以及（2）提供一个有效的一致的数据版本给处理器。本节讨论 ThyNVM 如何实现这两个目标。

内存中的数据区域及版本存储：ThyNVM 中存在两种类型的数据。第一种类型是*不涉及检查点生成*的数据，即在过去三个时间单元内均未更新的数据。这类数据因为不涉及更新，只在系统中存有一个版本。所以它们不被 BTT 和 PTT 记录。当这些数据被处理器访问时，它们的物理地址被直接使用（不经过 BTT/PTT 的重映射）以定位内存中的数据。我们把保存此类数据的内存部分称作*家区域*（如图5.4）。例如，只读数据即总是位于硬件地址空间中的家区域。

第二种类型是*涉及检查点生成*的数据，即被最近的三个时间单元中的至少一个更新过。这些数据会有多个版本，例如 W_{active}、C_{last} 或 C_{penult}（参见第5.2.2节）。BTT/PTT 中的元数据记录着这些数据版本的位置。BTT/PTT 将这些数据的物理地址映射到与物理地址不同的硬件地址上。这样，当这些数据被处理器访问时，ThyNVM 的机制会将处理器重定向到合适的数据版本。

ThyNVM 在 DRAM 和 NVM 中维护三个数据区域，每个保存一个数据版本：W_{active}、C_{last} 或者 C_{penult}。如图5.4所示，这些区域包括一个 DRAM 中[①] 的*工作数据*区域和两个 NVM 中的*检查点区域*。工作数据区域主要保存由页回写模式管理的活跃工作数据 W_{active}^{page}。注意，在一个时间单元的检查点生成阶段，这些被页回写模式管理的数据可能改由块重映射模式处理以避免应用程序停滞（参见第5.2.5节）。

① 在当前描述的实现中，我们假设工作数据区域被映射到 DRAM，用于活跃更新的数据，以获得 DRAM 的性能优势。其他 ThyNVM 的实现方式可以将该区域在 DRAM 和 NVM 上进行分配或者完全使用 NVM。

此时，对这些数据的更新由 BTT 记录，并且同样被重映射到该工作数据区域。

检查点区域 A 和 B 以交错的方式保存检查点数据 C_{last} 和 C_{penult}：如果 C_{last} 保存在检查点区域 A，ThyNVM 就把 C_{penult} 写入检查点区域 B；反之亦然。注意，检查点区域 B 与家区域相同，以节省内存空间和 BTT/PTT 表项：如果数据不涉及检查点生成，这个区域保存这些数据的唯一副本，不需要任何 BTT/PTT 中的地址映射；如果数据涉及检查点生成，该区域保存合适的检查点数据版本，而元数据相应地保存在 BTT 或 PTT 中。

另外，如果上一个时间单元已经完成检查点生成（即全内存 C_{last} 已经完整生成，可以安全地用于系统恢复），我们以与 C_{last} 交错的方式直接将 W_{active}^{block} 写入检查点区域，即用 W_{active}^{block} 覆盖 C_{penult}。①

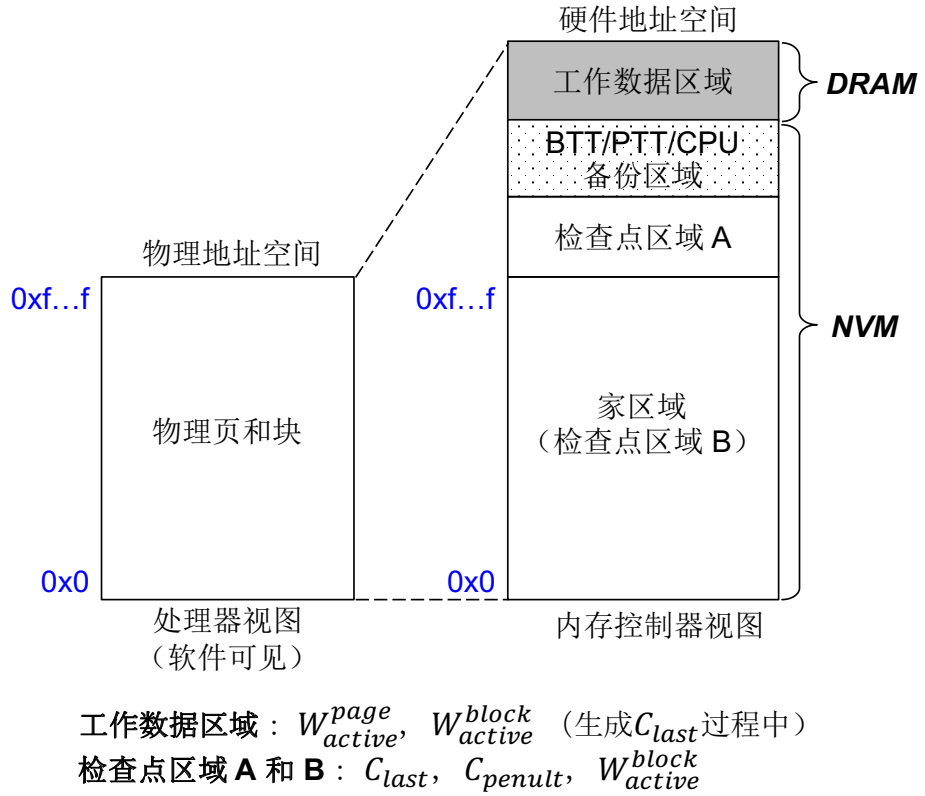


图 5.4　ThyNVM 的地址空间布局

ThyNVM 硬件地址空间留出部分 NVM 来保存 BTT/PTT 和 CPU 状态备份，称作 BTT/PTT/CPU 备份区域。

物理和硬件地址空间映射：为了提供一致的软件可见的数据版本②，ThyNVM 创建两个地址空间之间的映射关系。硬件地址空间是 DRAM 和 NVM 实际的设备

① 我们不在 NVM 中单独设置用于保存 W_{active}^{block} 的区域是因为 W_{active}^{block} 在其元数据持久化后即变为检查点数据。
② 软件可见的数据版本指可通过 load 和 store 指令读写的数据版本。

地址空间，由内存控制器管理，仅对内存控制器可见。物理地址空间是由内存控制器提供给软件的内存空间的逻辑视图，映射到硬件地址空间中适当的部分。下面我们讨论软件可见的数据版本的构成。

对于不涉及检查点生成的数据，软件可见版本位于硬件地址空间的家区域。这些数据的硬件地址即为物理地址的值（或辅以固定的偏移量）。对于涉及检查点生成的数据，软件可见版本定义如下（BTT/PTT 确保物理地址被映射到硬件地址空间中恰当的位置）。

$$\begin{cases} W_{active}, & \text{如果} W_{active} \text{存在} \\ C_{last}, & \text{如果} W_{active} \text{不存在} \end{cases}$$

当数据在活跃时间单元内未被更改时，我们没有其对应的工作副本（W_{active}），而最近检查点（C_{last}）实际上保存着应当被软件访问的最新数据。

从系统故障恢复的时候，DRAM 中保存的 W_{active} 会丢失。下面的软件可见版本会被恢复（版本选择取决于数据对应的 BTT/PTT 检查点的状态）。

$$\begin{cases} C_{last}, & \text{如果上一个检查点生成完毕} \\ C_{penult}, & \text{如果上一个检查点未完整保存} \end{cases}$$

在恢复的过程中，ThyNVM 使用 BTT/PTT 检查点来定位 C_{last} 或 C_{penult}。

5.3.2　元数据管理

保存在 BTT 和 PTT 中的元数据为内存控制器提供必需的信息，以完成如下三个目标：（1）将处理器发出的每个内存请求的物理地址转换成硬件地址；（2）将生成检查点的 NVM 写指向正确的硬件地址；（3）确定何时在两个检查点生成模式之间迁移数据。ThyNVM 在两个地址转换表 BTT 和 PTT 中维护元数据。两个表都包含五个列（表5.2），包含一个物理地址高位的块/页索引（BTT 需要 42 比特，PTT 需要 36 比特）、一个 2 比特的版本 ID、一个 2 比特的可见内存区域 ID、一个 1 比特的检查点内存区域 ID 和 6 比特的表征空间局部性的写计数器。①

为了达到第一个目标，必要的信息包括：（a）一个请求应当指向的内存区域，以及（b）在该内存区域内物理地址到硬件地址的映射。处理器只能够访问可见数据，所以我们用可见内存区域 ID 确定对应可见数据所处的内存区域。为了获得硬

① 因为不是所有 (版本 ID, 可见内存区域 ID, 检查点内存区域 ID) 的组合都存在，它们可以被压缩至 7 个状态，参见第6章。该状态机协议节约了硬件空间，同时简化了硬件逻辑。

表 5.2　BTT 和 PTT 表项的各个列（第 2 ～ 4 列可以压缩至 7 个状态，参见第6章）

物理块/页索引	版本 ID	可见区域 ID	检查点区域 ID	写计数
42 比特（BTT）/36 比特（PTT）	2 比特	2 比特	1 比特	6 比特

件地址，我们确保每个表项到表首的偏移数等于相应内存区域中块或页与首块或首页的偏移数（不适用于家区域）。所以，我们可以使用表项的偏移计算出块或页在内存区域中的硬件地址。

为了达到第二个目标，内存控制器使用检查点内存区域 ID 来确定一个检查点写的目的内存区域。我们采用和上文相同的方法，通过表项的偏移数来确定硬件地址。

为了达到第三个目标，内存控制器依据 BTT 和 PTT 的写计数来确定数据的空间局部性。BTT/PTT 中的每个写计数器记录了上一个时间单元中对相应物理块/页的写操作的数据；内存控制器在每个时间单元开始时读取并重置写计数器。一个时间单元内对同一个物理页较大数量的写操作（超过预定义的阈值）意味着这个页有较高的空间局部性和写强度。为此，我们下一个时间单元即采用页回写机制来操作对该页的写。写计数器的值如果小于预定义的阈值，表明我们下一个时间单元内应采用块重映射机制。我们实验了不同的阈值，发现从块重映射切换到页回写以及相反切换的阈值分别取 22 和 16 时最为有效。

生成 BTT 和 PTT 的检查点：元数据定位最近的一致的检查点，所以需要在系统故障前后保持持久化。在每个检查点生成阶段，我们在 NVM 的 BTT 和 PTT 备份区域生成 BTT 和 PTT 表项的检查点。如何原子性地将 BTT/PTT 写入 NVM 是一个需要处理的问题。我们使用一个表征 BTT/PTT 检查点生成是否完成的比特位（存储于 BTT/PTT/CPU 备份区域）来维护备份操作的原子性。

BTT 和 PTT 的大小：BTT 表项的数目取决于工作集合的写强度，而 PTT 表项的数目由 DRAM 的大小决定。如果有些负载要求 GB 级的 DRAM，巨大的 PTT 存储代价可以通过虚拟化 PTT 得到缓解，即在 DRAM 存储 PTT 而在内存控制器中缓存部分热点表项[156–158]。图5.15显示了系统性能对不同表大小的敏感度。本文系统测评中使用的 BTT 和 PTT 总大小约为 37KB。当前硬件的发展和创新使得内存控制器可以拥有复杂的功能和更大的空间，例如，HMC[159] 的逻辑层，或者 HP 的 The Machine[160] 的媒介控制器。这些技术让我们的硬件开销正在变得可行。

5.3.3　服务读写请求

为了服务一个内存读请求，我们用该请求的物理地址在 PTT 和 BTT 中查找。如果该物理地址在两张表中都没有，那么对应的可见数据位于 NVM 的家区域中。

为了服务一个内存写请求，如图 5.5（a）所示，我们首先使用 PTT 来确定是使用块重映射模式还是页回写模式——我们对所有 PTT 不包含的请求采用块重映射，不论它是否包含在 BTT 中。如果两张表中都不包含，说明对应的数据块位于家区域；这种情况下，我们需要在 BTT 中添加一个表项，记录该请求带来的元数据更新。对于 BTT/PTT 溢出，即没有表项空或者可以被替换的情况，我们只得生成一个检查点并开始一个新的时间单元。这样，我们可以安全地将那些表明其最新检查点在家区域的表项清除。

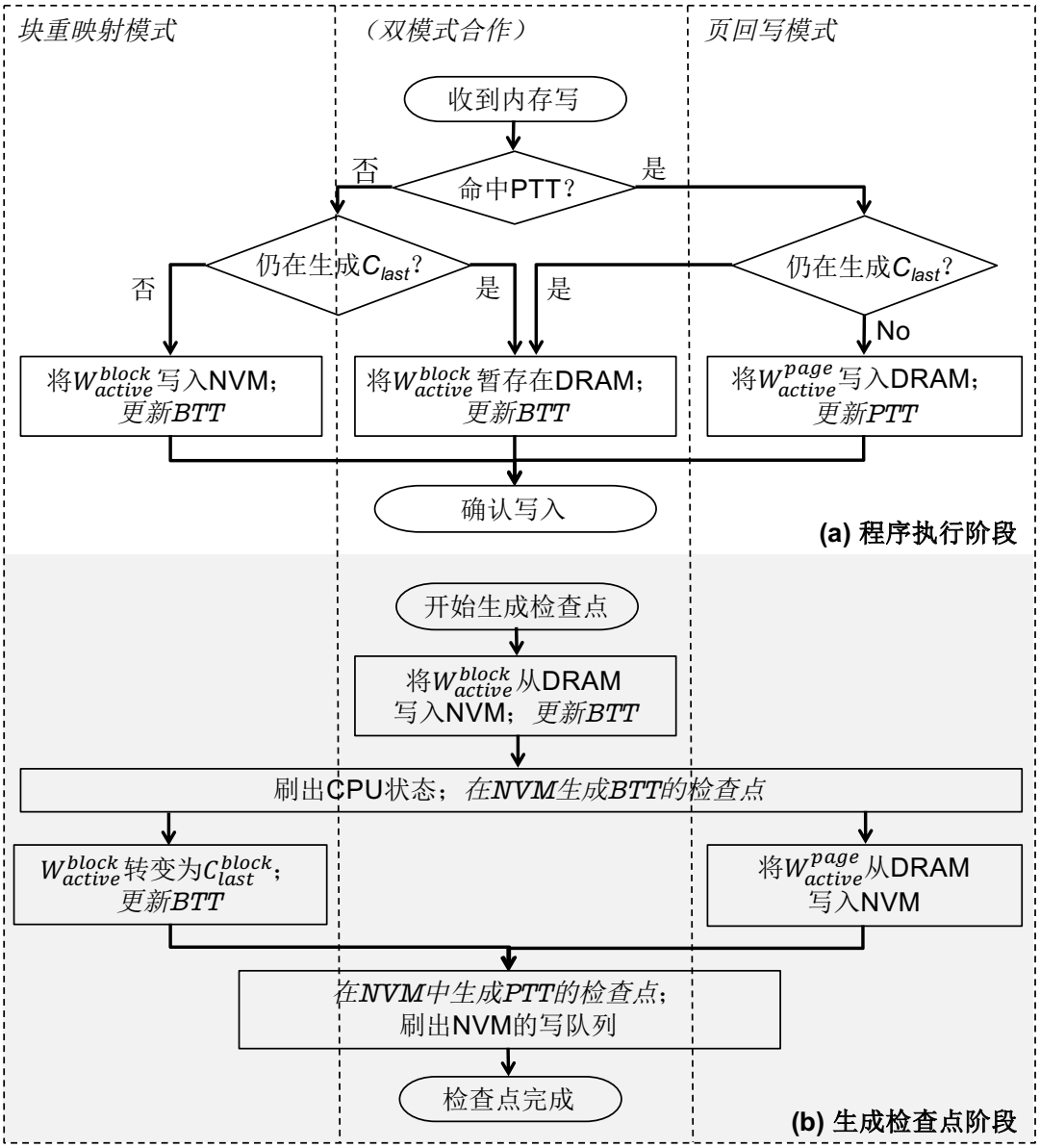


图 5.5　流程图：（a）在某个时间单元的执行阶段服务一个写请求；（b）在该时间单元的检查点生成阶段生成检查点。为了显示（a）中写入的数据和（b）中检查点数据的关联关系，我们用三个背景方格标记出两种检查点生成模式和它们之间的合作：每个背景方格内，在（a）中存储的数据，在生成检查点的时候，会经历（b）中对应的处理过程。

检查点生成：当我们生成内存数据的检查点时，我们确保元数据在数据之后生成检查点。特别的，我们采用如下的生成检查点的顺序：（1）将缓冲在 DRAM 中的 W_{active}^{block} 写入 NVM；（2）在 NVM 生成 BTT 的检查点；（3）将 DRAM 中的 W_{active}^{page} 写入 NVM；（4）在 NVM 生成 PTT 的检查点。

接收初始的数据访问：初始地，当系统开始执行，两个地址转换表都是空的；所有可见数据均位于 NVM 的家区域。因为 PTT 是空的，我们在第一个时间单元内仅采用块重映射来操作数据更新。之后，ThyNVM 将空间局部性高的数据迁移到页回写模式。系统第一次进入稳定状态（即在两种检查点生成模式间迁移的页数下降到一个相对小而稳定的数值）所需要的时间依赖于 PTT 的大小以及负载的内存访问行为。我们观察到，实验中使用的负载（第5.4节）的最坏情况，可能需要几十个时间单元使 2048 个 PTT 表项达到这种稳态。

5.3.4　数据刷出

为了保证检查点的一致性，在每个时间单元的生成检查点阶段，我们需要将处理器中的数据刷出（flush）到 NVM。为此，我们在每个检查点生成阶段的起始，将处理器寄存器保存到 NVM，并且刷出写缓冲和脏数据块。我们采用基于硬件的机制来执行数据刷出。内存控制器在一个执行阶段结束时通知 CPU。在收到通知后，CPU 中止应用软件的执行，发出一个刷出指令将所有寄存器的值写入一个 NVM 上的特别区域，并将写缓冲队列及脏缓存块写入内存控制器。注意我们并不使缓存中的脏数据块失效，类似于 Intel 的 CLWB 操作，从而后续的时间单元可以继续访问这些脏数据块。

5.3.5　系统恢复

对于采用 ThyNVM 维护故障时数据一致性的系统，数据恢复涉及三个步骤使持久化内存回滚到最新的检查点。首先，我们将 BTT 和 PTT 重新载入内存控制器，把它们对应于可见数据版本（定义见第5.3.1节）的备份从 NVM 拷贝出来。照此，我们恢复了可以用于恢复数据的元数据信息。第二，我们需要恢复页回写模式管理的可见数据。因为这些页的可见数据版本应当被保存在 DRAM 中，我们需要将 NVM 中的检查点拷贝到 DRAM 中。当元数据和数据都恢复完毕后，我们执行最后一步，从 NVM 的专属空间重新载入 CPU 寄存器。

一旦内存数据和处理器状态被恢复，我们即继续 CPU 的执行。然而，因为我们并不恢复外部设备（如网卡）的状态，恢复的程序很有可能遇到由于丢失这些设备状态而导致的错误。在我们的故障模型中，系统故障事件以设备错误的形式

对软件程序可见。我们依赖软件固有的异常或错误处理机制来应对这些设备错误。在第5.5节有关于外部状态（如网卡中的状态）丢失的讨论。

5.4　系统测评

5.4.1　实验环境

本节描述我们使用的模拟器、处理器和内存配置，以及标准测试集。我们的实验基于周期精准的模拟器 gem5[161]。我们使用 gem5 的提供详细时间模型的处理器，并扩展原有经典内存模型[162]。表5.3列举了我们模拟的处理器和内存的架构配置和主要详细参数。我们的 DRAM 和 NVM 都是采用 DDR3 接口进行模拟。我们模拟了 16 MB 的 DRAM，以及包含 2048/4096 个表项的 BTT/PTT。我们更改了 gem5 的实现来加入在第5.3节描述的 ThyNVM 的各项机制。每个时间单元的长度被限制在 10 ms（与论文[104,105] 的配置相近），因为用户或系统软件总是希望系统故障时的数据损失最小。

表 5.3　系统配置及参数[163]。

处理器	3 *GHz*，按序执行
L1 指令/数据缓存	私有 32 *KB*，8 路，64 *B* 块；4 周期命中延迟
L2 缓存	私有 256 *KB*，8 路，64 *B* 块；12 周期命中延迟
L3 缓存	共享 2 *MB* 每核，16 路，64 *B* 块；28 周期命中延迟
内存	DDR3-1600，64 位内存通道，2 ranks，8 banks，8 *KB* 行缓存
时间参数	DRAM：40 (80) 纳秒行命中（错失）延迟
	NVM：40 (128/368) 纳秒行命中（clean/dirty miss）延迟
	BTT/PTT：3 纳秒查找延迟

参照系统。ThyNVM 和如下四个不同的系统进行了对比。

1. 理想化的 DRAM 系统：一个只有 DRAM 作为主存的系统，并假设不需要任何代价即可实现数据一致性。DRAM 的大小等于 ThyNVM 的物理地址空间大小。
2. 理想化的 NVM 系统：一个只有 NVM 作为主存的系统，并假设不需要任何代价即可实现数据一致性。NVM 的大小等于 ThyNVM 的物理地址空间大小。
3. 日志系统：一个混合的 NVM 系统，使用日志[30,164,165] 来保证数据的一致性。我们的实现遵照文献[166]。DRAM 中设置一个日志缓冲来收集和归并更新的数据块。每个时间单元结束时，缓冲数据被回写到 NVM 的备份区域，然后

再在原目标地址更新数据。该机制使用一个表来追踪 DRAM 中被缓冲的脏数据块。这个表的长度与 ThyNVM 中 BTT 和 PTT 两个表的长度和相同。

4. 影子页系统：一个混合的 NVM，使用影子页机制[167] 来保证数据一致性。该机制对 NVM 实行写时拷贝，并在 DRAM 中保存这些页作为缓冲。当 DRAM 缓冲被填满时，脏页被写入 NVM，但不会覆盖原目标地址的数据。该配置中 DRAM 的大小和 ThyNVM 中的 DRAM 大小一样。

标准测试集。该测试集包括如下三组不同的负载。

1. 微型标准测试集涵盖不同的内存访问模式。为了显示我们一致性机制对不同访问模式的适应性，我们测评了三组代表典型访问模式的微型标准测试集。（i）随机：随机访问一个大数组。（ii）流式：顺序访问一个大数组。（iii）滑动：在大数组上虚拟一个工作数据集并滑动。每一步，随机访问数组中的一段特定区域，然后转移到另一个区域。上述每个负载都包含 1:1 的读写操作数目。
2. 面向存储的内存中标准测试集。持久性内存可通过将传统存储的数据组织在内存来优化基于磁盘存储的应用。为了测评 ThyNVM 对此类应用的影响，我们运行了两个标准测试集（类似于[23,89]）。它们包含代表典型内存存储的键值存储的实现，分别在哈希表和红黑树的内存数据结构上执行查找、插入和删除等操作。
3. SPEC CPU 2006。因为 ThyNVM 为支持多种类型的遗留代码设计，所以我们也测评了 SPEC 标准测试集。运行此类负载的进程因为可以假定一个可持久化的有一致性保障的地址空间而获益。我们从 SPEC 套件中选择了 8 个内存访问最集中的应用，每个应用执行 10 亿条指令。对于其他内存访问不太集中的应用，ThyNVM 的影响可以忽略不计，几乎等同于提供零代价一致性保证的全 DRAM 理想系统。

5.4.2　微型标准测试集

整体性能：图 5.6展示了各参测系统与零代价一致性的全 DRAM 理想系统的程序运行时间比。通过这些结果，我们可以得出三点观察结果。第一，ThyNVM 的性能在微型标准测试集的所有访问模式下均优于其他一致性机制。我们的性能平均比日志和影子页技术分别高 10.2% 和 14.8%。第二，与已有一致性机制不同，ThyNVM 可以灵活地适应不同的访问模式。例如，影子页机制在随机访问的模式下性能非常糟糕，因为即使内存中一个页只有很少的脏块，整个页都需要在生成检查点时写入 NVM。另一方面，影子页在其他两类访问模式下表现要好于日志机制，因为这些负载下有大量的顺序写被 DRAM 吸收，减小了检查点生成的代

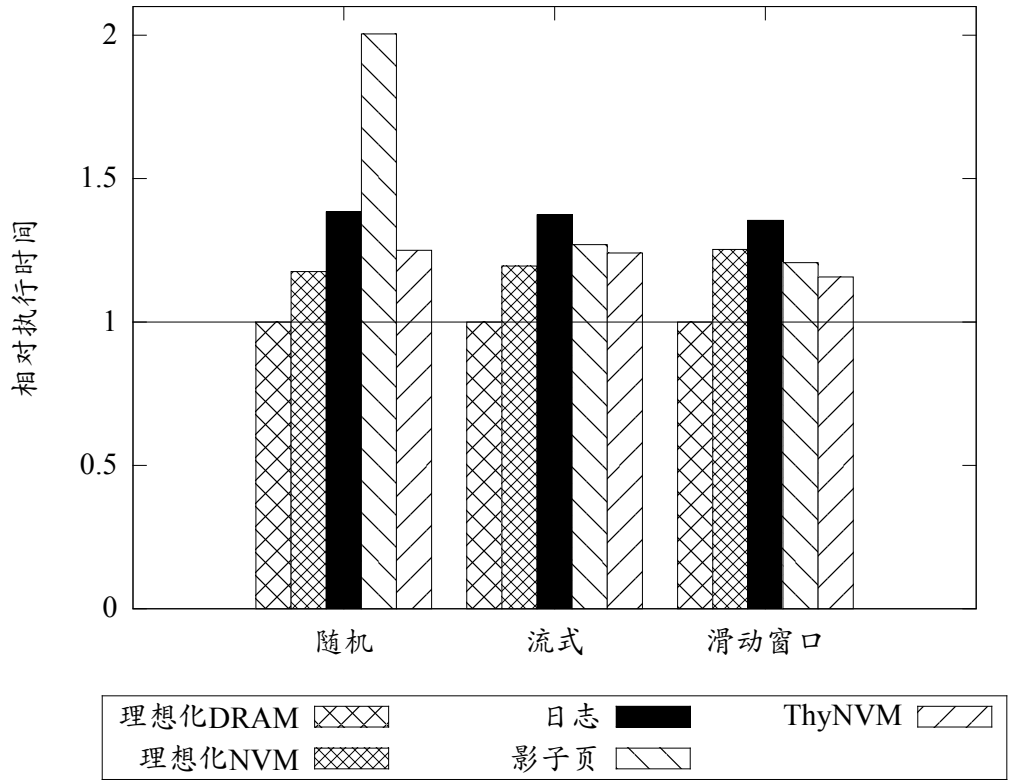


图 5.6　微型标准测试集执行时间。

价。然而，ThyNVM 可以适应不同的访问模式，没有任何表现不良的情况。第三，ThyNVM 的平均性能损失可以限制在理想 DRAM 系统的 14.3%，整体性能比理想 NVM 系统高出 5.9%。

总结来看，我们可以得出两个结论：

1. ThyNVM 可以灵活地适应不同的访问模式，其性能在所有微标准测试中高于传统日志或影子页一致性机制。
2. ThyNVM 引入了可以接受的较小的性能损耗以达成数据一致性保障，其性能和零代价一致性的理想系统相差无几。

NVM 写入量。图5.7到图5.9展示了各参评系统在不同负载下的 NVM 写入数据的总量。我们分析了三个组件：（1）最后一级缓存的回写，代表 CPU 向 NVM 输出的数据量；（2）检查点生成带来的 NVM 数据写入量；（3）页迁移带来的 NVM 数据写入量。我们可以得到关于数据传输的三点观察结果。

1. 日志和影子页无法自适应于不同的访问模式，导致在至少一种负载下的巨大的 NVM 传输量。相比之下，ThyNVM 在任意负载下都没有显示极端的 NVM 传输量。平均来看，ThyNVM 的 NVM 写入量，相比日志和影子页，分别降低了 10.8% 和 14.4%。
2. 平均来看，ThyNVM 减少了对所有访问模式平均的 NVM 写入量，但它比单

个访问模式下拥有最低 NVM 写入量的一致性机制会传输更多的数据。这个原因主要在于 ThyNVM 将检查点生成和程序执行重合，需要存储更多的数据版本，有些情况下需要为检查点生成更多的数据。所以，在一致性机制中存在一个性能和带宽之间的权衡。

3. 我们发现，对于不同的访问模式，ThyNVM 的页迁移可能产生不同的 NVM 写入量。例如，流式访问模式下，页不断地从 NVM 迁移到 DRAM 再被写回 NVM 而没有实质上的数据归并，所以产生的 NVM 写入量很大。与之相反，在滑动窗口访问模式下，因为工作数据集缓慢移动，许多页在被迁移回 NVM 前可以得到多次重复使用。

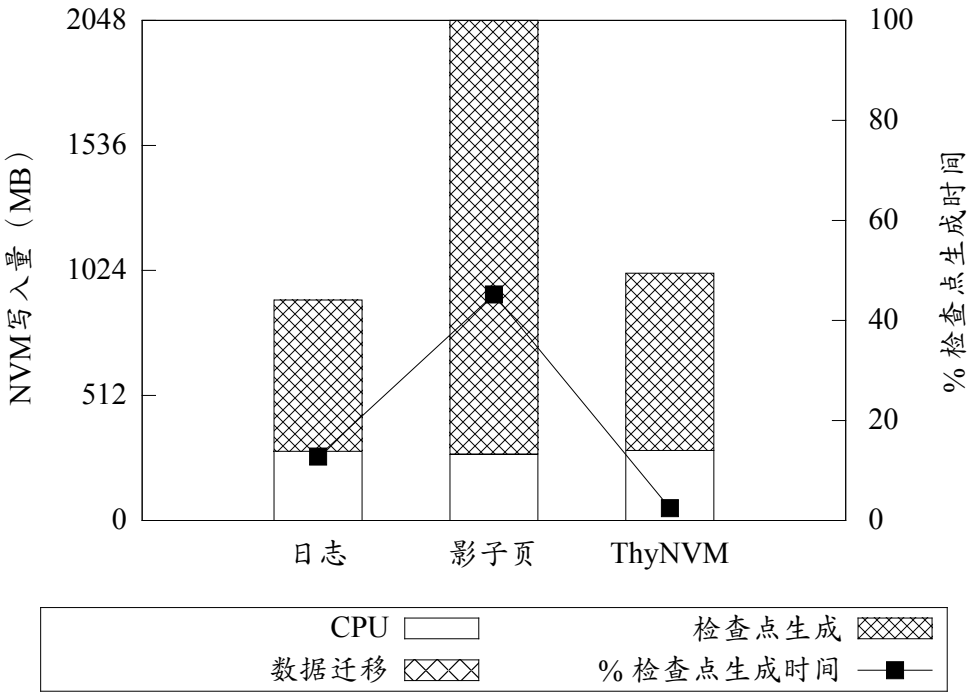


图 5.7　微型标准测试集随机访问模式产生的 NVM 写入量和检查点生成延迟（%）。“CPU”表示从 CPU 写入 NVM 的数据量。

总结起来，我们得出如下两个结论。

1. ThyNVM 对不同访问模式的适应性减少了各个访问模式下平均的 NVM 写入量。
2. 通过将检查点生成和程序执行重合，ThyNVM 可以减少程序执行时间，但对于特定的单一类型的负载，可能会引入比对应的最优传统一致性机制更多的 NVM 写入量。

检查点生成延迟。图5.7到图5.9同时展现了每个负载在检查点生成上花费了多大比例的运行时间。日志和影子页分别花费了平均 18.9% 和 15.2% 的时间用于生

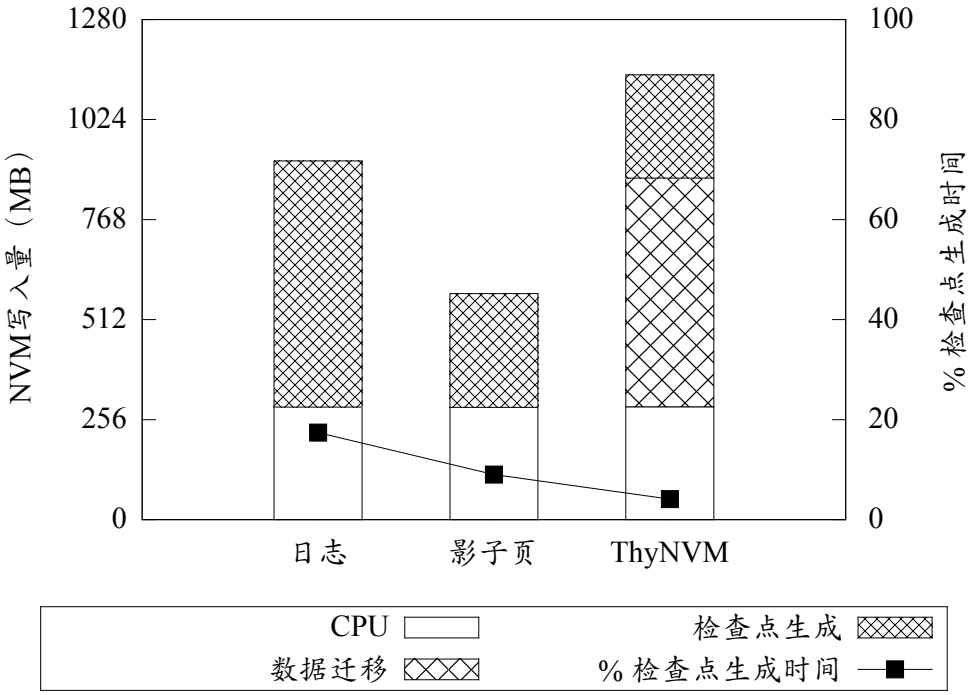


图 5.8　微型标准测试集流式访问模式产生的 NVM 写入量和检查点生成延迟（%）。“CPU”表示从 CPU 写入 NVM 的数据量。

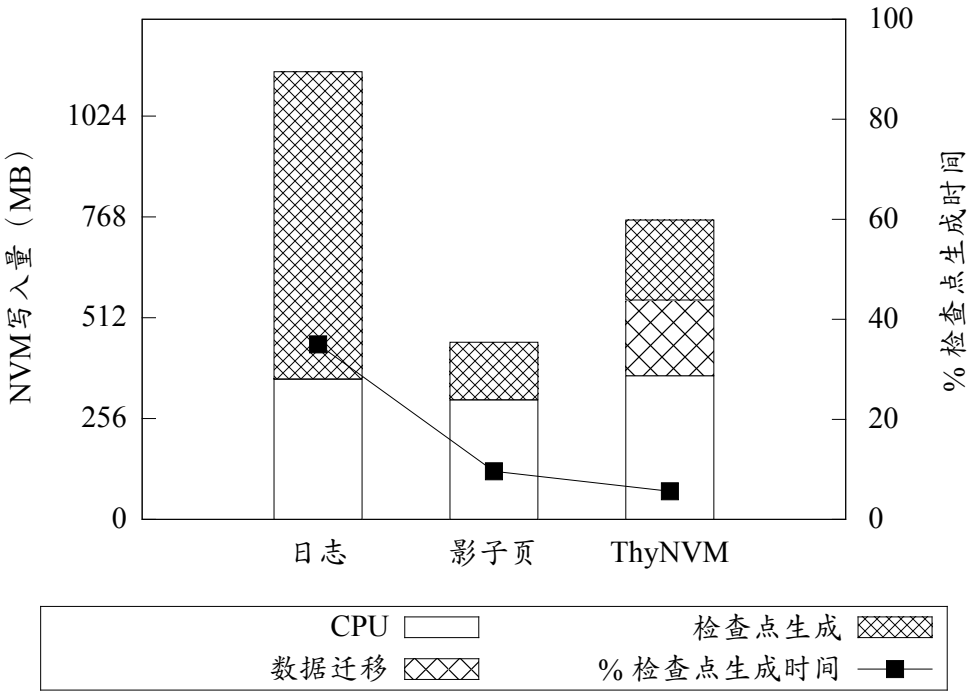


图 5.9　微型标准测试集滑动访问模式产生的 NVM 写入量和检查点生成延迟（%）。“CPU”表示从 CPU 写入 NVM 的数据量。

成检查点，而 ThyNVM 将该比例降低到平均 2.5%。我们可以得出结论，ThyNVM 可以通过将检查点生成和程序执行重合有效地避免给应用带来的停滞时间。

5.4.3　面向存储的标准测试集

内存存储应用通常访问数据结构的随机位置，连续的写较少。我们通过面向存储的标准测试集获得的实验结果，反映了 ThyNVM 在真实负载情况下的表现。

吞吐率性能：图5.10和图5.11展示了两个内存存储负载的事务吞吐率。实验中，发向两个键值存储的请求的大小由 16B 变化到 4KB。我们从该图可以得出两个观察结果。

1. ThyNVM 的吞吐率总是好于传统一致性机制。总体上，ThyNVM 在哈希表和红黑树上实现了比日志机制分别高 8.8% 和 4.3% 的吞吐率，以及比影子页高 29.9% 和 43.1% 的吞吐率。
2. ThyNVM 实现了和理想化 DRAM 系统接近的吞吐率，以及和理想化 NVM 系统相似的吞吐率。ThyNVM 在哈希表和红黑树上的平均吞吐率分别是理想化 DRAM 系统的 95.1% 和 96.2%。

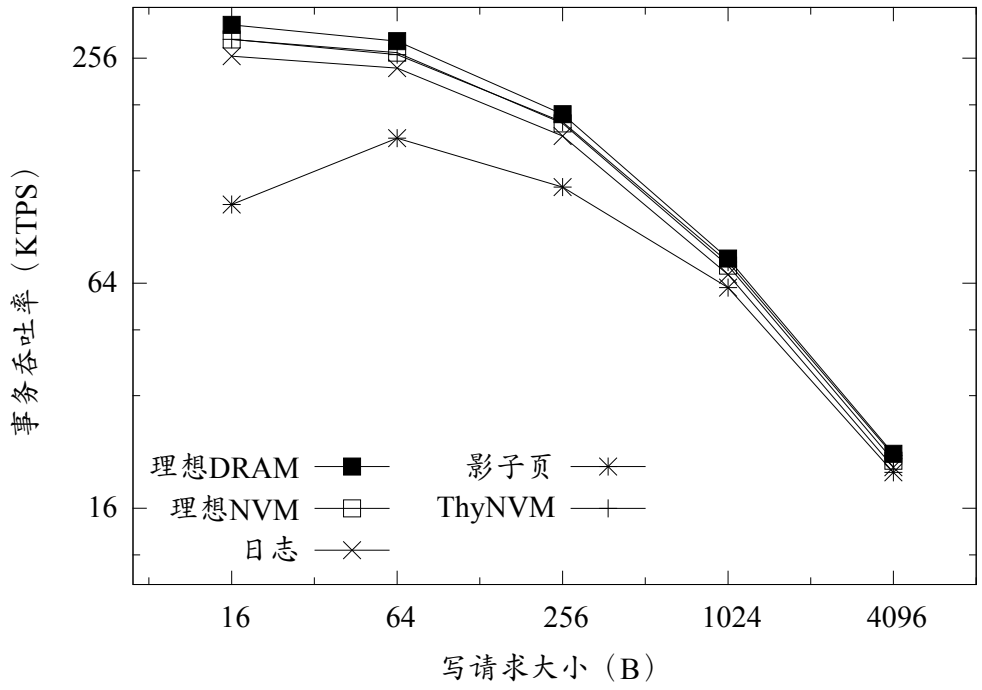


图 5.10　基于哈希表的键值存储的事务吞吐率。

总结起来，ThyNVM 在实际的内存存储负载下，比传统一致性机制性能更优，而且相对理想化的 DRAM/NVM 系统造成的吞吐率降低很小。

内存带宽消耗：图5.12和图5.13展示了两个内存存储负载在不同请求大小情况下的带宽消耗。通过该图我们可以得出两个观察结果。

1. ThyNVM 总比影子页机制占用更少的带宽，因为这些负载展现出随机写的

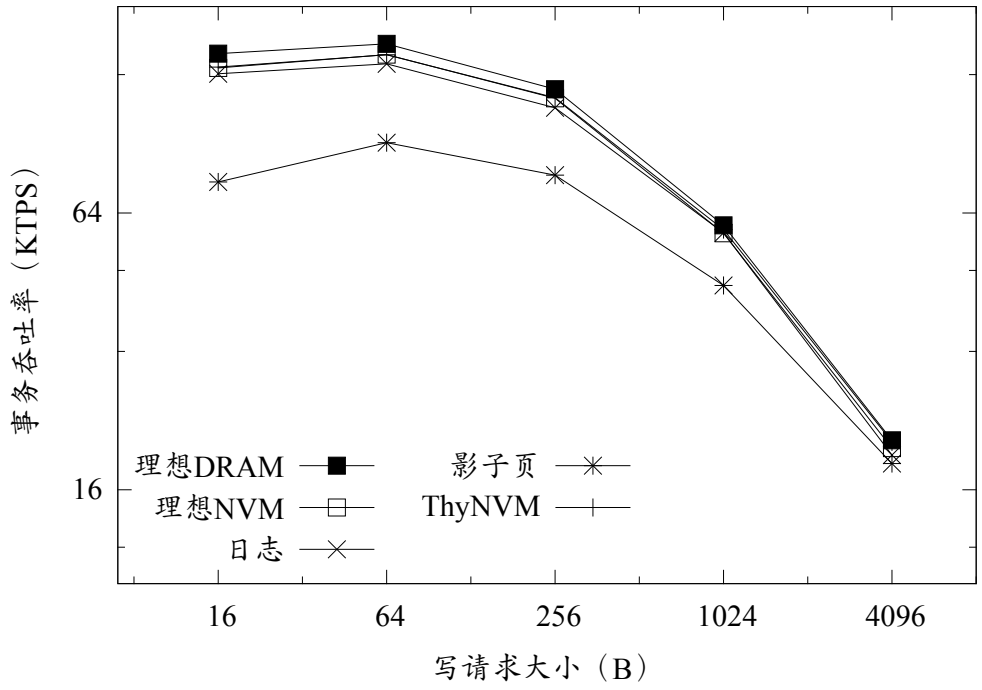


图 5.11　基于红黑树的减值存储的事务吞吐率。

行为。这种情况下，影子页机制由于需要写回只有少数脏数据块的页，生成检查点的带宽会增加。ThyNVM 在哈希表和红黑树上的带宽消耗分别比影子页机制减少 43.4% 和 64.2%。

2. ThyNVM 在这些负载下，会比日志机制带来更多的带宽消耗。原因是 ThyNVM 需要为了将检查点生成和程序执行重合而维护更多的数据版本。这个结果符合我们之前关于带宽消耗和性能之间权衡的观察结果。日志机制在哈希表和红黑树的负载下，分别比 ThyNVM 减少 19.0% 和 14.0% 的 NVM 写入带宽。

我们可以看到，ThyNVM 的写带宽消耗接近日志机制，而远小于影子页机制。

5.4.4　面向计算的标准测试集

图5.14显示内存使用集中的 SPEC CPU 标准测试集负载的 IPC。这些数据根据理想化的 DRAM 系统进行了归一化处理。从图中我们可以得出两点观察结果。第一，相对于理想化的 DRAM 系统，ThyNVM 使标准测试集的执行时间增加了 3.4%。第二，相对于理想化的 NVM 系统，ThyNVM 将标准测试集的执行时间减少了 2.7%。总结起来，ThyNVM 可以显著减少检查点生成的性能损耗，提供和零代价一致性系统相近的性能。

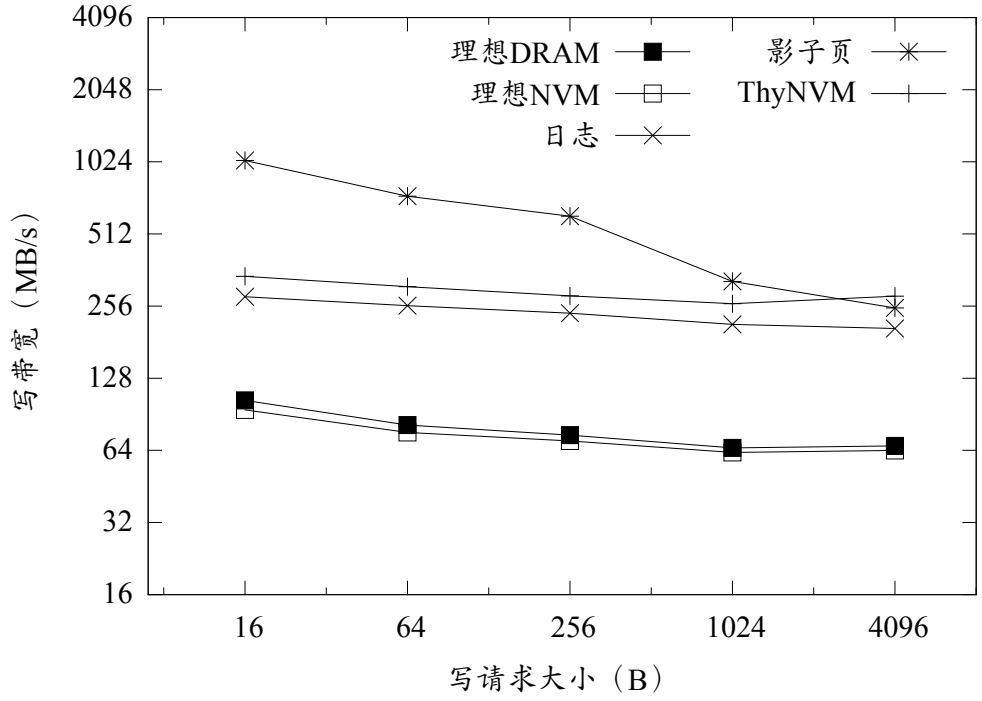


图 5.12　基于哈希表的键值存储的写带宽消耗。“理想化 DRAM” 对应 DRAM 的写入带宽，其余对应 NVM 写入带宽。

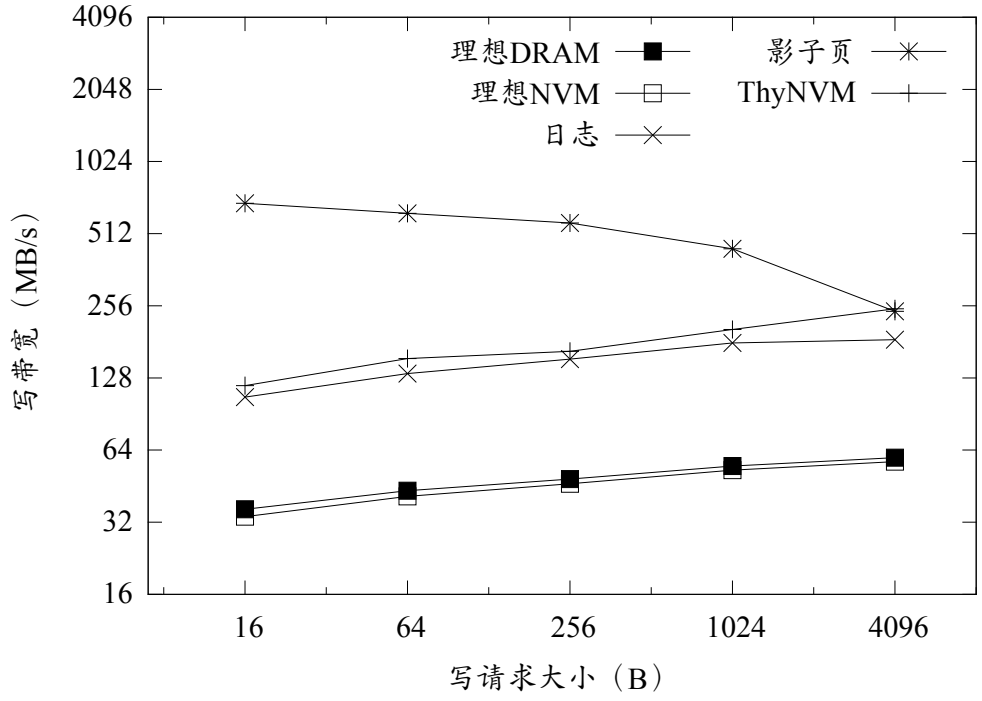


图 5.13　基于红黑树的键值存储的写带宽消耗。“理想化 DRAM” 对应 DRAM 的写入带宽，其余对应 NVM 写入带宽。

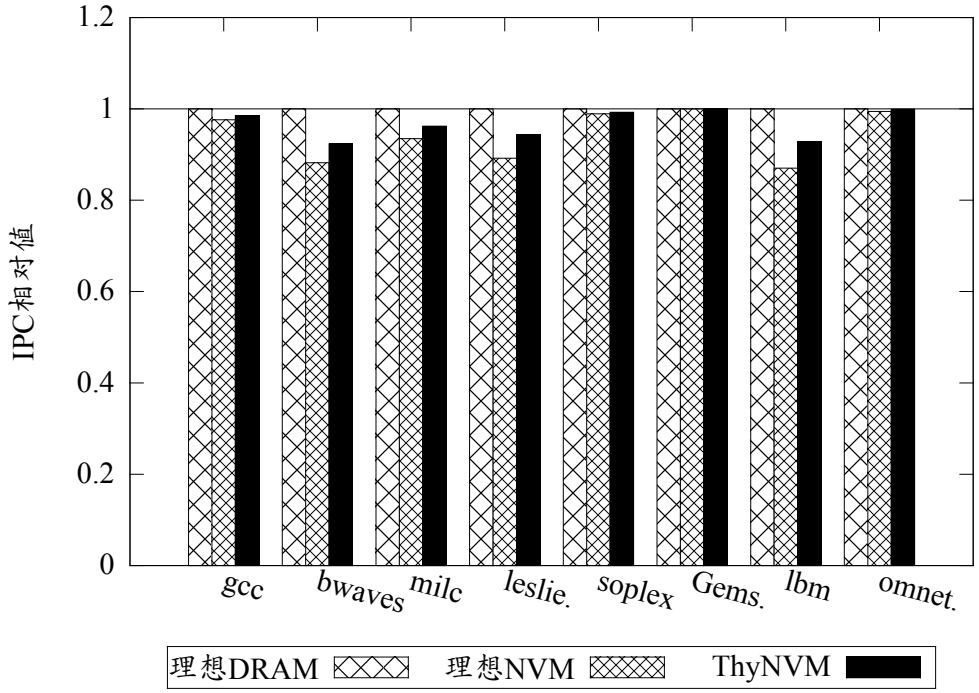


图 5.14　SPEC 2006 标准测试集的 IPC（数值经过归一化处理）。

5.4.5　敏感度分析

图5.15展示了 ThyNVM 在运行基于哈希表的内存存储负载时，对于 BTT 表项数目的敏感度。我们可以从图中得到两点结论。第一，基本上 BTT 越大，NVM 写入量越小，因为减小的检查点生成次数相应减小了到 NVM 的写。第二，基本上 BTT 越大，负载吞吐率越高。这是因为较高的内存带宽消耗和较小的 BTT 组合在一起时，会阻塞内存带宽，导致服务内存请求的延迟。

5.5　讨论

与全系统持久化的对比： ThyNVM 的模型和目标与全系统持久化（WSP）[100]的模型和目标有很大不同。WSP 的目标是支持全系统状态的恢复，包括设备状态。然而，ThyNVM 的目标是仅支持内存数据的故障时一致性。ThyNVM 不处理设备状态，将外部设备状态的恢复交由系统软件（如下所述）。WSP 假设一个全 NVM 配置从而仅需要在电源故障时刷出 CPU 的状态而不必要管非易失的 DRAM 状态。ThyNVM 采用 DRAM+NVM 的混合内存架构，可以提供比单独 NVM 更好的性能[17,96]，但要求定期生成 CPU 状态和内存数据的检查点以保证故障时的数据一致性。与 WSP 不同的是，ThyNVM 提供内存数据的故障时一致性保证，而不要求或假设全 NVM 的配置。

故障时外部状态丢失问题： ThyNVM 不保护外部状态（例如，网卡状态）。这

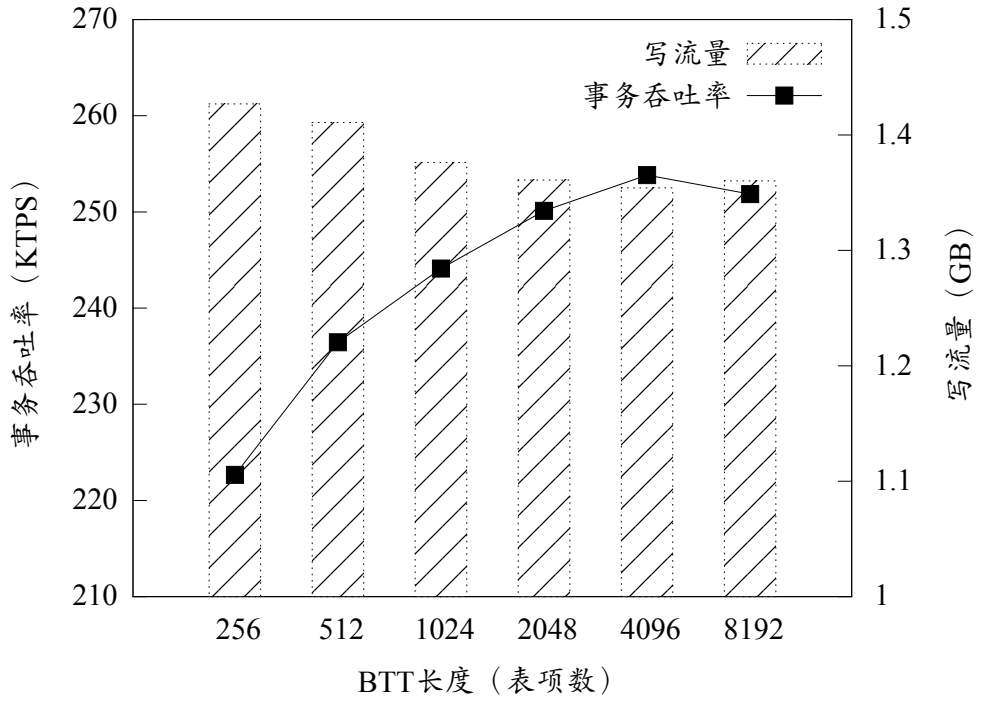


图 5.15　在面向存储的标准测试集的负载下变化 BTT 长度。

些状态在系统重启后丢失，可能在系统恢复过程中导致设备错误。因此，系统故障以设备错误的形式对程序可见。设备错误在计算机系统中相当普遍[168,169]。商用系统已经包含相应机制来处理此类错误[170–172]。我们假设的这个故障模型也与若干前人工作[104,105,173,174] 相一致。

软件程序错误： 软件错误可能导致内存状态被污染和系统崩溃。但是，故障时数据一致性和软件错误容忍是两个不同的问题。任何维护故障时数据一致性的系统（包括数据库和文件系统）都可能受到软件错误的伤害，而 ThyNVM 也不例外。例如，常见软件 Eclipse（一个广泛使用的开发套件）和 Chromium（一款流行的浏览器）中的软件错误分别曾导致数据库和文件系统的数据污染[175,176]。ThyNVM 提供故障时数据一致性保证，但无法避免这些软件错误，也并不是为容忍这类错误而设计。然而，它可以经过扩展帮助增强系统的错误容忍能力。例如，可以定期将部分检查点拷贝到二级存储中，并设计某种机制来定位和恢复到之前没有错误的检查点。对此类机制的研究可能会开辟新的研究方向。

显式持久化接口： 我们的检查点生成机制可以无缝地与可配置持久化保证的系统[27,28,177] 整合。此类系统仅允许丢失过去最多 n 毫秒内的数据更改，其中 n 是可配置的。ThyNVM 可以配置为每 n 毫秒产生检查点，从而可以在故障时回滚到最近的一致的检查点。与此同时，应用程序还可以通过在 CPU 指令集中加入的某个新指令强制 ThyNVM 结束一个时间单元，即显式地触发内存数据的持久化。

5.6　本章小结

我们为混合的持久性内存系统引入了一个对软件透明的提供内存数据故障时一致性的机制。我们的设计自动地定期生成内存数据的检查点，并最小化对应用执行的影响。为此，我们采用一种新的双模式检查点生成技术，可以有效地将程序执行和检查点生成过程重合。这种有效性基于（1）依据内存更改的空间局部性适配生成检查点的粒度以及（2）提供机制来协调两种检查点粒度下的内存修改。我们的测评涵盖多种不同的访问模式、真实的存储类负载和计算密集型负载。测评结果显式，ThyNVM 以很低的代价提供内存数据的故障时一致性。我们相信 ThyNVM 对软件透明的高效的故障时数据一致性机制可以促进（1）对持久性内存系统更加便捷和广泛的使用，以及（2）更加高效的支持持久性内存的软件栈设计。我们希望本章工作鼓励更多研究聚焦在为管理持久性或混合内存提供自动的对程序开发人员友好的机制。为此，我们将 ThyNVM 在如下地址开源：http://persper.com/thynvm。

第 6 章　软件透明的一致性协议及其证明

本章描述 ThyNVM 双模式检查点生成技术使用的基于状态机的数据一致性协议，以及协议的形式化证明。

6.1　地址空间管理

ThyNVM 使用两个地址转换表，*块转换表*（BTT）和*页转换表*（PTT），来维护从物理地址空间到硬件地址空间的映射。

每个物理地址在 NVM 上有一个对应的固定的*主硬件地址*。所以，确定一个物理地址的主硬件地址不需要经过 BTT 或 PTT 进行地址转换。不失一般性，我们假定主硬件地址和物理地址相等。与此同时，任何物理地址都可以通过 BTT 或 PTT *被动态地*映射到一个 NVM 上的不同的硬件地址，映射方式取决于 ThyNVM 检查点生成模式的规则。

为了方便内存管理，我们将硬件地址空间划分为六个有不同用途的区域。图 6.1描绘了 ThyNVM 地址空间的组织形式。下面我们描述每个区域的用途。

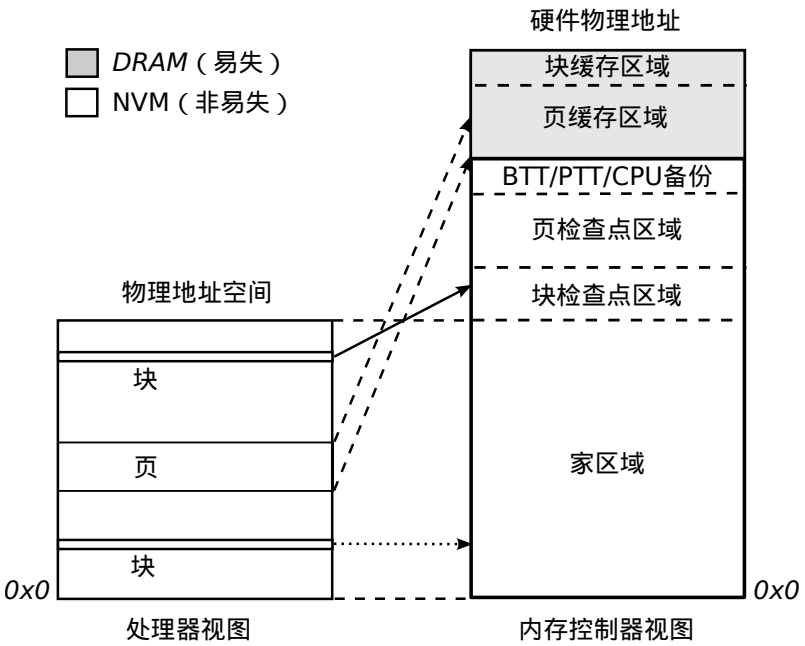


图 6.1　ThyNVM 的物理和硬件地址空间。

家区域：该区域位于 NVM，包含所有主硬件地址。每个物理地址对应于一个本区域内的静态的主硬件地址。该区域的大小与物理地址空间的大小相同。

块检查点区域：该区域位于 NVM，被 BTT 用来为检查点数据（ C_{last}^{block} 或 C_{penult}^{block} ）分配以缓存块大小为单位的存储空间。在块重映射模式中，一个块数据的工作副本（ W_{active}^{block} ）在其对应的 BTT 表项持久化后无需任何数据移动即可变为 C_{last}^{block} 。所以，该区域同时保存着其 BTT 表项尚未持久化的 W_{active}^{block} 数据。这样，一旦 BTT 完成持久化，这些数据可以直接变成 C_{last}^{block} 而不进行数据移动。

页检查点区域：该区域位于 NVM，被 PTT 用来保存页粒度的检查点数据，即 C_{last}^{page} 或 C_{penult}^{page} 。家区域和页检查点区域交替保存 C_{last}^{page} 和 C_{penult}^{page} ：如果 C_{penult}^{page} 保存在家区域中，那么 C_{last}^{page} 则保存在本区域；反之亦然。

页缓存区域：该区域位于 DRAM，是热点页工作副本（ W_{active}^{page} ）的缓存。在每个时间单元的检查点生成阶段，本区域中的脏页被回写到 NVM 作为 C_{last}^{page} （位于上文描述的页检查点区域或家区域）。

块缓存区域：该区域位于 DRAM，保存着一些块粒度的数据的临时副本。这些数据属于正在生成检查点的页。当程序执行和检查点生成过程重合时，对页缓存区域的写由块重映射模式处理，即被重映射到该区域做临时保存。

BTT/PTT/CPU **备份区域**：每个检查点生成阶段，BTT 和 PTT 对应于 C_{last} 的版本都会保存在这个位于 NVM 中的区域。这样，当系统故障发生时，我们可以恢复出检查点数据所处的位置。检查点生成阶段开始时的 CPU 状态也保存在本区域，并与对应的 BTT 和 PTT 备份相关联。早于 C_{penult} 版本的 BTT/PTT/CPU 备份可以被销毁。

这里的块检查点区域和页检查点区域共同对应于第5.3.1节描述的检查点区域A，而页缓存区域和块缓存区域则共同对应于工作数据区域。

6.2　地址转换状态机

我们使用一个状态机来决定一个内存写请求应当访问的位置，即将内存写请求的物理地址转换成一个合适的硬件地址。除了保存这个从物理地址到硬件地址的映射，每个 BTT 表项同时保存着 7 个状态之一，该状态驱动着块重映射模式的控制流。PTT 重用与该状态机相同的逻辑，但只需要这些状态中的 4 个。下面我们聚焦于描述 BTT 的行为，而后讨论 PTT 如何适配到这个状态机的一个子集。图 6.2描绘了状态机的全貌。为了清楚地展示，我们将状态分组而后逐组进行介绍。

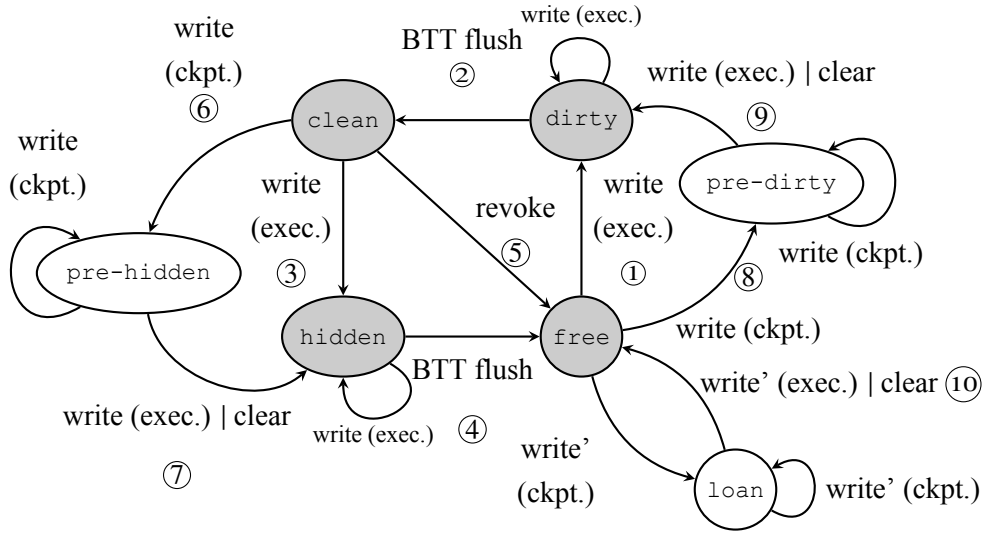


图 6.2　在块重映射模式下，一个 BTT 表项的状态。每个内存写（“write”）后注明“exec.”或“ckpt.”，分别意味着该内存写到来的时间在程序执行阶段（无检查点生成）或在检查点生成阶段；“write'”指原本应到页缓存的写。

6.2.1　执行时的状态

该组状态主要协调模式在程序执行（没有检查点生成）时的行为。为了决定内存写的正确地址，我们需要一个状态记录关于地址映射的两方面信息：（1）该物理地址在当前活跃时间单元内是否被写过。如果是的话，我们可以直接覆写对应的硬件地址，因为同一个时间单元内对相同物理地址的内存写可以合并。（2）硬件地址是一个主硬件地址还是位于检查点区域中。在某些情况下，如果我们需要避免覆写其中一个的话，我们将选择另外一个。据此，我们有四个状态来涵盖上述两个选择的所有组合。

- **`Free` 态**：标志一个空的无有效映射的表项。同时我们将没有包含在 BTT 中的物理地址默认为 `free` 态。这意味着其物理地址在当前活跃时间单元内没有被更改过，且其检查点数据位于家区域中的主硬件地址。新来的对该物理地址的写请求必须在 BTT 中开辟一个新的表项将这个写重映射到块检查点区域。
- **`Clean` 态**：该状态表明其物理地址在当前活跃时间单元没有被更改过，且其检查点数据位于块检查点区域。新来的对该物理地址的写请求必须被 BTT 重映射到家区域的主硬件地址以保护检查点数据。
- **`Dirty` 态**：该状态表明其物理地址在当前活跃时间单元内被更改过，且对应的硬件地址在块检查点区域。该映射可以继续有效直到当前时间单元结束，用以合并到相同物理地址的内存写。在当前时间单元结束后，该硬件地址上的数据应当变为检查点数据，所以这个表项会被备份到 NVM，将映射关系

持久化，以备系统故障时定位检查点数据。

- **Hidden 态**：这个状态表明其物理地址在当前活跃时间单元内被更改过，且对应的硬件地址是位于家区域的主硬件地址。这里实际上并没有地址转换，因为主硬件地址和物理地址是相等的。然而，该表项会一直存在直到当前时间单元结束，目的是合并同时间单元内对该物理地址的其他写。在当前时间单元结束后，这个表项会被清除。

实例 为了展示上述状态之间的动态转换，我们构造了图 6.3中的实例。我们首先忽略检查点生成期间的步骤（步骤 A 和 B），并简单假设它们没有发生。在这个实例中，我们展示了三个连续的时间单元，记为时间单元 0 到 2。从时间单元 k 收到的内存写会产生数据版本 v_k，其中 $k = 0, 1, 2$。

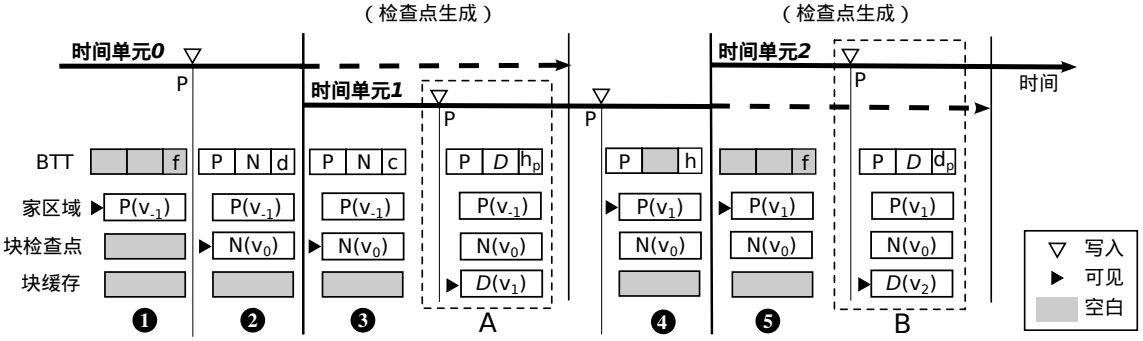


图 6.3　连续时间单元中对物理地址 P 进行多次内存写的实例。

❶ 初始状态，我们假设在物理地址 P 的数据位于其主硬件地址（家区域），所以在 BTT 中没有对应的表项，且不占用块检查点区域或块缓存区域的位置。P 位置的数据可以认为是最近检查点的版本（标记为 v_{-1}，即 v_0 之前的一个版本）。

❷ 在执行时间单元 0 的过程中，当一个对物理地址 P 的内存写到达时，我们不能覆写家区域中的检查点数据 v_{-1}。所以，一个新的映射 P-N 被添加到 BTT 中，将数据 v_0 导向块检查点区域中的硬件地址 N。我们将该表项标记为 `dirty` 态（图 6.2中的状态转换①）以表明该物理地址在当前活跃时间单元内已经被更改过。这个时间单元内后续的任意对物理地址 P 的内存写都可以合并在硬件地址 N。

❸ 在时间单元 0 之后，上述 `dirty` 表项被冲刷到 BTT/PTT/CPU 备份区域。该表项在 BTT 中的状态变为 `clean`（状态转换②）以表明该映射已经持久化且在新的时间单元 1 中尚无对该物理地址的写。

❹ 时间单元 0 完成检查点生成之后，我们不再需要家区域中的数据 v_{-1}。所以对物理地址 P 的内存写可以直接映射到主硬件地址。与此同时，相应的 BTT 表项状态变为 `hidden` 态（状态转换③）。该状态表明主硬件地址保存着数据 v_1，而且这

个时间单元内对 P 的后续写可以在这个位置上更新数据。如果这一步出现系统故障，ThyNVM 则会回退到块重映射区域保存的数据 v_0。

❺ 当时间单元 2 开始时，BTT 冲刷操作已经将 hidden 状态清除，而这个表项被释放以表明家区域中的主硬件地址保存着物理地址 P 的数据的最近检查点版本 v_1（状态转换④）。

我们可以看到，该 BTT 表项在连续多个时间单元收到内存写之后会变为 free 态。所以，多个时间单元之间的时间局部性可以减少使用中的表项，进而缓解对有限的 BTT 的存储空间的压力。在最坏状态下，如果 BTT 没有 free 状态的表项，我们必须撤销一个 hidden 态或者 clean 态的表项为新的内存写提供空间。Hidden 态的表项无论如何都会在当前活跃时间单元结束时被撤销；Clean 态的表项可以被安全撤销（状态转换⑤）是因为他们不记录当前活跃时间单元内的新的更新。然而，被 clean 态表项引用的数据块（位于块检查点区域）需要和其主硬件地址上的数据互换。

6.2.2　检查点生成时的状态

第二个状态组在检查点生成阶段发挥作用。该组包含两个状态，pre-hidden 态和 pre-dirty 态，分别处理如下两种情况。

- **Pre-hidden** 态：该状态处理一个 clean 态的物理地址在检查点生成阶段收到内存写的情况。我们不能简单地像程序执行阶段一样执行状态转换③，因为家区域和块检查点区域的数据在检查点生成阶段都必须被保留。以图 6.3中的步骤 A 为例。因为我们当前处于时间单元 1，块检查点区域的数据 v_0 是不可更改的，因为它是最近检查点 C_{last} 的一部分；同样地，家区域中的数据 v_{-1} 也不可更改，因为完整的一致的 C_{last} 尚未完成——如果此时系统故障终断了检查点生成进程，系统会回滚到 v_{-1}（即 C_{penult}）。因此，我们需要临时将工作数据（W_{active}）放入另外一个地方，块缓存区域，并相应地把这个特别的状态标记为 pre-hidden（状态转换⑥）。Pre-hidden 状态是 clean 状态和 hidden 状态之间的一个垫脚石。它只在当前活跃时间单元结束前存在。之后，如果该状态没有被一个内存写驱动转换到 hidden（达到图 6.3中步骤 4 相同的结果），它会被显式地清理（状态转换⑦)。不论哪种情况，因为时间单元 0 的检查点（C_{last}）已经完成，版本 v_{-1}（C_{penult}）都会被工作数据 W_{active} 安全地覆写，并相应收回块缓存区域里 W_{active} 的位置。
- **Pre-dirty** 态：该状态处理一个 free 态的物理地址在检查点生成阶段收到内存写的情况。与 pre-hidden 的处理逻辑类似，我们不能像程序执行

阶段一样直接执行状态转换①。以图 6.3中的步骤 B 为例。当时间单元 1 的检查点（ C_{last} ）正在生成的时候，我们不仅要保留 v_1，还要保留 v_0（ C_{penult} ），因为 C_{penult} 是我们仅有的完整一致的检查点。所以，我们把数据 v_2 放入块缓存区域，并向 BTT 添加一个 `pre-dirty` 态的表项来记录该映射（状态转换⑧）。`Pre-dirty` 状态或者被下一个程序执行阶段的内存写命中转换到 `dirty` 态，或者在那个阶段结束时被清理转换到 `dirty` 态（状态转换⑨）。

因为 `hidden` 态和 `dirty` 态在每个检查点生成阶段刚刚开始时伴随 BTT 冲刷操作被清理，他们永远不会在检查点生成阶段与内存写相遇。所以，我们没有其他情况需要处理。

6.2.3　用于双模式合作的状态

最后我们有一个用于两个模式间合作的状态。当页回写模式生成脏页的检查点时，对这些页的各个内存写暂由块重映射模式重定向到块缓存区域（与第6.2.2节介绍的操作类似）。我们不把这些 BTT 表项混入上文的那些状态和转换，所以这些表项被标记为 `loan`。当检查点生成结束这些页面重新可以被直接修改的时候，`Loan` 态的表项最终将数据移回原页中（状态转换⑩）。

6.3　正确性定义

本节给出一致性协议正确性证明的定义。我们这里聚焦于 ThyNVM 检查点生成模式的故障时一致性，亦即可恢复性——任何时候系统发生故障，ThyNVM 可以恢复最近的一致的检查点。特别地，假设 t_k 为时间单元 k 结束执行的时刻，那么在 t_{k-1} 和 t_k 之间，ThyNVM 顺次经历两个时间阶段，即一个检查点生成阶段（与时间单元 $k+1$ 的程序执行阶段重合）和一个程序执行阶段（指没有与检查点生成重合的部分）；t_k 时刻的数据快照记为版本 v_k，亦即时间单元 k 的检查点版本。我们将证明如下两个不变式，在状态机的转换中一直保持成立。

1. 在时刻 t_{k-1} 与 t_k 之间的检查点生成阶段，数据版本 v_{k-1} 和 v_{k-2} 及其在地址转换表中的映射都被保留。
2. 在时刻 t_{k-1} 与 t_k 之间的程序执行阶段，数据版本 v_{k-1} 及其在地址转换表中的映射都被保留。

显然，如果上述两个不变式成立，那么系统在检查点生成阶段或程序执行阶段发生故障时，可以分别恢复到 C_{penult} 或 C_{last} 。

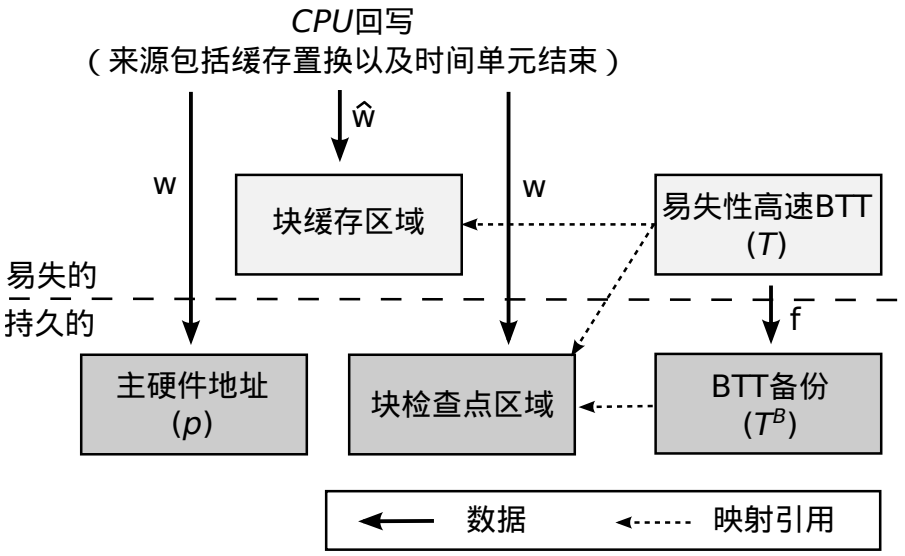


图 6.4　ThyNVM 中的数据结构及其符号。

6.4　块重映射模式的正确性证明

ThyNVM 的块重映射模式由第6.2节的状态机描述。对于任何由状态机转换形成的状态序列，我们使用数学归纳法[178] 验证上节两个不变式的正确性。

6.4.1　数据抽象

为了实施形式化证明，我们使用变量来代表 ThyNVM 中的数据结构，如图6.4所示。由于我们只关心数据版本而非具体的数据，不妨认为这些变量保存着数据版本值；大写的变量值表示集合。因为所有 BTT 表项互斥，我们仅关心任意一个 BTT 表项，并假设该表项当前是用于物理地址$\underline{\mathrm{P}}$。

- p 保存着主硬件地址 $\overline{\mathrm{P}}$ 上的数据块的版本号。
- T 保存着对应于$\underline{\mathrm{P}}$ 的 BTT 表项。该表项是易失的。如果没有对应于$\underline{\mathrm{P}}$ 的 BTT 表项，那么 $T = \varnothing$，此时可认为$\underline{\mathrm{P}}$ 的状态为 `free`；否则，我们恒有 $|T| = 1$，其中唯一的项保存着在块检查点区域中对应数据块的版本号。
- T^B 保存着持久化的 BTT 表项，位于 BTT 备份区域。类似地，我们有 $\left|T^B\right| \leq 1$。特别地，对于时间单元 0，我们仅出于方便计算的目的认为 $T^B \subset \{-1, -2\}$。

另外，下面的字母代 BTT 表项状态机的输入。

- **w** 和 **ŵ** 分别表示在程序执行阶段和检查点生成阶段向物理地址$\underline{\mathrm{P}}$ 的写操作。
- **r** 和 **r̂** 分别表示在程序执行阶段和检查点生成阶段撤销物理地址$\underline{\mathrm{P}}$ 对应的 BTT 表项。
- **f** 表示刷出 BTT 到备份区域。该事件标志一个检查点阶段的开始。
- **c** 表示在刷出 BTT 之前清理 `pre-dirty` 态和 `pre-dirty` 态表项的操作。

6.4.2　行为抽象

基于如上的变量和符号，我们将主要的状态转换过程表述为过程1到4。相应地，BTT 表项的状态机也可改写为图6.5。注意 `loan` 态没有在图中画出，而是在第6.5节单独讨论。

过程 1 转换至 `dirty` 态

Require: 处于程序执行阶段。

1: 在块检查点区域分配地址 $\overline{\mathtt{N}}$
2: 将数据块 v_0 写入 $\overline{\mathtt{N}}$
3: if 表项是 `pre-dirty` 态，即映射关系为$\underline{\mathrm{P}}$-$\overline{\mathrm{D}}$ then
4: 　　从块缓存区域撤销位于 $\overline{\mathrm{D}}$ 的数据块
5: end if
6: 设置表项为 `dirty` 态，即映射关系为$\underline{\mathrm{P}}$-$\overline{\mathtt{N}}$ ▷ $T \leftarrow \{0\}$

过程 2 临时缓存数据块

Require: 处于检查点生成阶段。

1: 在块缓存区域分配地址 $\overline{\mathrm{D}}$
2: 将数据块 v_0 写入 $\overline{\mathrm{D}}$
3: if 表项是 `clean` 态，即映射关系为$\underline{\mathrm{P}}$-$\overline{\mathtt{N}}$ then
4: 　　将位于 $\overline{\mathtt{N}}$ 的数据标记为在下一个检查点生成阶段被撤销
5: 　　将表项转换至 `pre-hidden` 态
6: else if 表项是 `free` 态 then
7: 　　将表项转换至 `pre-dirty` 态
8: end if
9: 设置映射关系为$\underline{\mathrm{P}}$-$\overline{\mathrm{D}}$ ▷ $T \leftarrow \{0\}$

6.4.3　证明步骤

每一个输入的写操作驱动状态机进行一步转换。因此，我们可以为所有变量添加一个下标 n，以标示变量值对应于状态机状态转换序列中的哪一步。我们采用数学归纳法的证明基于 n 进行推导。

命题：不变式 2和不变式 1可以分别翻译为如下两个命题 $P(n)$ 和 $Q(n)$。

过程 3 转换至 `hidden` 态

Require: 处于程序执行阶段。

1: 将数据块 v_0 写入主硬件地址　▷ $p \leftarrow 0$
2: if 表项是 `clean` 态，即映射关系为$\underline{\mathrm{P}}$-$\overline{\mathtt{N}}$ then
3:　　将位于 $\overline{\mathtt{N}}$ 的数据标记为在下一个检查点生成阶段被撤销
4: else if 表项是 `pre-hidden` 态，即映射关系为$\underline{\mathrm{P}}$-$\overline{\mathrm{D}}$ then
5:　　从块缓存区域撤销位于 $\overline{\mathrm{D}}$ 的数据块
6: end if
7: 将表项转换至 `hidden` 态　▷ $T \leftarrow \{p\}$

过程 4 撤销 `clean` 态的映射关系$\underline{\mathrm{P}}$-$\overline{\mathtt{N}}$

1: if 处于程序执行阶段 then
2:　　将位于 $\overline{\mathtt{N}}$ 的数据块 v_{-1} 写入 $\overline{\mathrm{P}}$，覆盖数据块 v_{-2}　▷ $p \leftarrow -1$
3:　　将位于 $\overline{\mathtt{N}}$ 的数据标记为在下一个检查点生成阶段被撤销
4: else　▷ 处于检查点生成阶段
5:　　交换位于 $\overline{\mathrm{P}}$ 的数据块 v_{-2} 和位于 $\overline{\mathtt{N}}$ 的数据块 v_{-1}　▷ $T^B \leftarrow (T^B - T) \cup \{p\}$　▷ $p \leftarrow -1$
6:　　在 BTT 备份（对应于 v_{-1}）中标记$\underline{\mathrm{P}}$-$\overline{\mathtt{N}}$ 为已置换
7:　　将位于 $\overline{\mathtt{N}}$ 的数据块标记为在下一个检查点生成阶段被撤销
8: end if
9: 将表项转换至 `free` 态，即撤销该表项　▷ $T \leftarrow \{p\}$

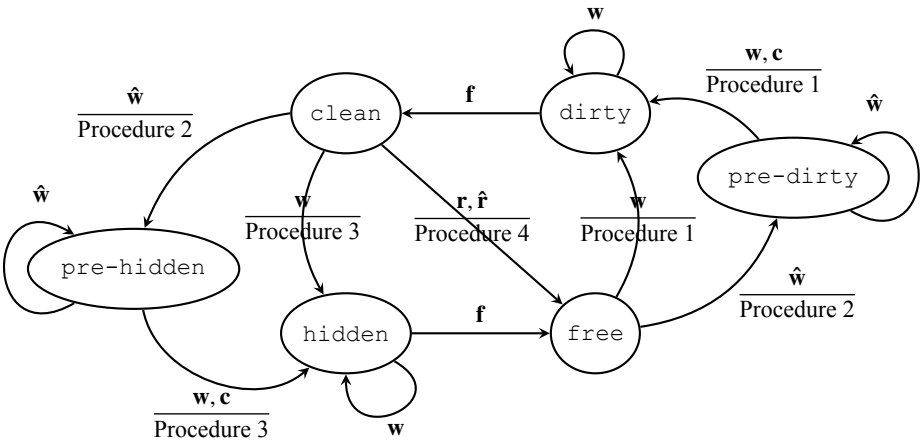


图 6.5　BTT 表项的状态机的抽象描述，用于实现块重映射模式。

在程序执行阶段，我们需要保证 $P(n)$ 一直成立：

$$P(n) = (-1 \in T_n^B) \vee (-1 \notin T_n^B \wedge p_n = -1).$$

我们假设当前活跃时间单元为 0。上式隐含了两个可以接受的情况。第一个是 BTT 备份包含一个指向块检查点区域的映射（即 $-1 \in T_n^B$）。第二个是 BTT 备份不包含对 v_{-1} 的映射关系（即 $-1 \notin T_n^B$）但 C_{last} 处于主硬件地址（即 $p_n = -1$）。只要两个情况任一成立，那么系统故障发生时，从 BTT 备份中恢复出 BTT 可使 ThyNVM 找到地址P 的正确的 C_{last}，即 v_{-1}。

在检查点生成阶段，我们需要保证 $Q(n)$ 一直成立：

$$Q(n) = (p_n = -2 \wedge -1 \in T_n^B) \vee (p_n = -1 \wedge -2 \in T_n^B).$$

与 $P(n)$ 类似，$Q(n)$ 意味着两个正确的情况：BTT 备份包含 C_{last}（v_{-1}）的映射并指向块检查点区域（即 $-1 \in T_n^B$），而 C_{penult}（v_{-2}）位于主硬件地址，依然得到保留（即 $p_n = -2$）；或者两个版本所处位置相反的情况——如果主硬件地址保存着 C_{last}，那么 BTT 备份必须记录着 C_{penult} 的位置（即 $p_n = -1 \wedge -2 \in T_n^B$）。

综上，我们需要证明如下这个命题：

$$S(n) = (\alpha_n \in \{\mathbf{w}, \mathbf{r}, \mathbf{c}\} \wedge P(n)) \vee (\alpha_n \in \{\hat{\mathbf{w}}, \hat{\mathbf{r}}, \mathbf{f}\} \wedge Q(n))$$

我们通过区分输入的字母 α_n 来判断处于程序执行阶段还是检查点生成阶段。**wr** 和 **c** 在程序执行阶段输入，而其余的在检查点生成阶段出现。

归纳基础：在整个系统最初始的状态，我们可以认为所有数据均位于主硬件地址。不妨假设初始步骤前存在虚拟的时间单元 -1，那么有 $p_0 = -1$；另外，由于 BTT 为空，我们有 $T_0^B = \varnothing$。所以，$P(0)$ 成立。进而，由于处在程序执行阶段，$S(0)$ 是成立的，值为真。

归纳步骤：假设对任意的 $n \geq 1$，$S(n-1)$ 为真，下面我们论证 $S(n)$ 也为真。

首先，我们统一考虑状态机中所有自循环。对于状态 `dirty`、`pre-dirty` 和 `pre-hidden`，ThyNVM 只更新 T_n，所以 $S_n = S_{n-1}$ 为真。对于状态 `hidden`，ThyNVM 令 $p_n = 0$，但根据将状态转换至 `hidden` 的过程3，在自循环前即有 $p_{n-1} = 0$，所以 ThyNVM 并没有更改该值。所以我们也可以得到 $S_n = S_{n-1}$。

其次，我们看两个包含 BTT 冲刷 **f** 的状态转换。

`dirty` → `clean`：转换前的状态为 `dirty` 表明 $p_{n-1} = -1$。转换后，新的时间单元与检查点生成阶段重合，所以我们有 $p_n = -2$[①]。另外，ThyNVM 将 `clean` 态的映射写入 BTT 备份，意味着 -1 被加入 T_n^B。所以，我们有 $p_n = -2 \wedge -1 \in T_n^B$，也就是 $Q(n)$ 成立。进而，$S(n)$ 为真。

`hidden` → `free`：转换前的状态是 `hidden` 意味着 $p_{n-1} = 0$，而在程序执行阶段，$S(n-1) \implies P(n-1)$，所以 $-1 \in T_{n-1}^B$ 必为真。状态转换之后，由于处在新的时间单元，我们有 $p_n = -1$ 以及 $-2 \in T_n^B$。所以 $Q(n)$ 为真，进而 $S(n)$ 成立。

除去上述情况，步骤 n 必然经历下面的过程。

过程1管理着状态转换 `free` → `dirty` 和 `pre-dirty` → `dirty`。不论哪个状态转换，该过程保护 $p = -1 \wedge -1 \notin T^B$：转换前，状态 `free` 和 `pre-dirty` 都有 $p_{n-1} = -1 \wedge -1 \notin T_{n-1}^B$，而该过程避免覆写主硬件地址，所以我们有 $p_n = -1$。而且，该过程不刷出表项到 BTT 备份区域，所以有 $-1 \notin T_n^B$。因此，$P(n)$ 成立。又因为处在程序执行阶段（即 $\alpha_n \in \{\mathbf{w}, \mathbf{r}, \mathbf{c}\}$），我们有 $S(n-1) \implies S(n)$。

过程2管理着状态转换 `clean` → `pre-hidden` 和 `free` → `pre-dirty`。不论哪个状态转换，该过程将数据写入块缓存区域，而不是更改 p 或 T^B。这么做的目的是在检查点生成阶段保护 v_{-1} 和 v_{-2}。因此，不难得出 $S(n) = S(n-1)$。

过程3管理着状态转换 `clean` → `hidden` 和 `pre-hidden` → `hidden`。转换前，状态 `clean` 和 `pre-hidden` 必然处在检查点生成阶段，所以有 $p_{n-1} = -2 \wedge -1 \in T_{n-1}^B$（根据 BTT 刷出行为和过程2）。在该过程中，ThyNVM 设置 $p_n = 0$ 但不涉及 T^B，所以我们有 $p_n = 0 \wedge -1 \in T_n^B$。又因为处于程序执行阶段，$S(n)$ 为真。

过程4管理着状态转换 `clean` → `free`，但有可能处于程序执行阶段，也可能处于检查点生成阶段。状态转换器前，依据 BTT 刷出行为我们有 $P_{n-1} = -2 \wedge -1 \in T_{n-1}^B$，且被 T 引用的最近的版本是 v_{-1}。如果处于程序执行阶段，该过程设置 $p_n = -1$，但保持 $-1 \in T_n^B$，所以 $P(n)$ 成立。进而，$S(n)$ 为真。如果处于检查点生成阶段，ThyNVM 交换 p 和 T^B 的值，从而导致 $p_n = -1 \wedge -2 \in T_n^B$。相应地，$Q(n)$ 成立，且 $S(n)$ 为真。

综上，对于状态机在所有可能的转换中所遵循的路径，恒有 $S(n-1) \implies S(n)$。考虑到初始步骤 0 符合 $S(0)$，我们可以得出结论 $S(n)$ 对所有的 $n \geq 0$ 成立。也就是说，不变式1和2对任何物理地址均成立。

① 只有 BTT 刷出操作会将所有的版本号减一。任何其他过程中，版本号或者保持或者被显式赋值

6.5　页回写模式的正确性证明

为了简便，我们在证明中忽略不重要的部分。首先，我们认为如下独立的回写模式的正确性自明：（1）在检查点生成阶段停止一切对内存的修改；（2）将缓存的脏页写入 NVM 作为检查点，但是不覆写它们之前的版本；（3）原子性地将记录所有检查点页位置的表写入 NVM。显然，不变式1和2都是成立的。

对于上述步骤（2），我们重用块重映射模式的状态机来维护故障时数据一致性。特别地，我们只需做如下替换：第一，块重映射协议中的程序执行阶段和检查点生成阶段在这个过程中都被认为是程序执行阶段，所以只有 `free`、`dirty`、`clean` 和 `hidden` 四个状态有效。第二，块检查点区域由页检查点区域代替。第三，所有写变为页粒度，而不再是块粒度。因此，这个过程可以视作之前论证的重映射模式的一个特例，其正确性可以得到保证。

进而，基于上述独立的回写模式的正确性，我们可以得出 ThyNVM 采用的页回写模式的正确性。当生成脏页的检查点时，块重映射模式会临时负责代理应用程序写往页缓存区域的新的请求，所以 ThyNVM 的页回写模式与独立的页回写模式等价，当且仅当满足如下两个条件：（1）页缓存区域保存着在生成检查点开始前保存着数据的最新的一致的版本；（2）页缓存区域的这一状态在整个检查点生成阶段得以保持。

第一个条件可以满足，因为在程序执行阶段页缓存区域是直接更新的，而保存在块缓存区域中的数据会在检查点生成阶段前移回页缓存区域（由一个清理操作完成）。第二个条件也成立，因为在检查点生成阶段，应用程序的新的写都被转移到块缓存区域而不会污染页缓存区域。综上，ThyNVM 的页回写模式可证明是正确的。

参考文献

[1] Galvin P B, Gagne G, Silberschatz A. Operating System Concepts. 9th ed., New York, NY, USA: John Wiley & Sons, Inc., 2013.

[2] Tanenbaum A S, Bos H. Modern Operating Systems. 4th ed., Upper Saddle River, NJ, USA: Prentice Hall Press, 2014.

[3] Lang T, Wood C, Fernández E B. Database buffer paging in virtual storage systems. ACM Trans. Database Syst., 1977, 2(4):339–351.

[4] Yu Y J, Shin D I, Shin W, et al. Optimizing the block I/O subsystem for fast storage devices. ACM Trans. Comput. Syst., 2014, 32(2):6:1–6:48.

[5] Takeuchi K, Tanaka T, Tanzawa T. A multipage cell architecture for high-speed programming multilevel NAND flash memories. IEEE Journal of Solid-State Circuits, 1998, 33(8):1228–1238.

[6] Jung T S, Choi Y J, Suh K D, et al. A 117-mm2 3.3-v only 128-mb multilevel nand flash memory for mass storage applications. IEEE Journal of Solid-State Circuits, 1996, 31(11):1575–1583.

[7] Kawahara T, Takemura R, Miura K, et al. 2 mb spram (spin-transfer torque ram) with bit-by-bit bi-directional current write and parallelizing-direction current read. IEEE Journal of Solid-State Circuits, 2008, 43(1):109–120.

[8] Kultursay E, Kandemir M, Sivasubramaniam A, et al. Evaluating STT-RAM as an energy-efficient main memory alternative. Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on, 2013. 256–267.

[9] Loke D, Lee T H, Wang W J, et al. Breaking the speed limits of phase-change memory. Science, 2012, 336(6088):1566–1569.

[10] Choi Y, Song I, Park M H, et al. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. IEEE International Solid-State Circuits Conference Digest of Technical Papers, 2012. 46–48.

[11] Raoux S, Burr G W, Breitwisch M J, et al. Phase-change random access memory: A scalable technology. IBM Journal of Research and Development, 2008, 52(4):465–479.

[12] Lee B C, Zhou P, Yang J, et al. Phase-change technology and the future of main memory. IEEE Micro, 2010, 30(1):143–143.

[13] Akinaga H, Shima H. Resistive random access memory (reram) based on metal oxides. Proceedings of the IEEE, 2010, 98(12):2237–2251.

[14] Meza J, Luo Y, Khan S, et al. A case for efficient hardware/software cooperative management of storage and memory. Proceedings of Fifth Workshop on Energy Efficient Design, 2013.

[15] Mutlu O, Subramanian L. Research problems and opportunities in memory systems. Supercomputing Frontiers and Innovations, 2014, 1(3).

[16] NVM Express, Inc. NVM Express. http://www.nvmexpress.org/.

[17] Qureshi M K, Srinivasan V, Rivers J A. Scalable high performance main memory system using phase-change memory technology. Proceedings of the 36th annual international symposium on Computer architecture, New York, NY, USA: ACM, 2009. 24–33.

[18] Zhou P, Zhao B, Yang J, et al. A durable and energy efficient main memory using phase change memory technology. Proceedings of the 36th annual international symposium on Computer architecture, New York, NY, USA: ACM, 2009. 14–23.

[19] Cho S, Lee H. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009. 347–357.

[20] Ipek E, Condit J, Nightingale E B, et al. Dynamically replicated memory: building reliable systems from nanoscale resistive memories. Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems, New York, NY, USA: ACM, 2010. 3–14.

[21] Azevedo R, Davis J D, Strauss K, et al. Zombie memory: Extending memory lifetime by reviving dead blocks. Proceedings of the 40th Annual International Symposium on Computer Architecture, 2013. 452–463.

[22] Volos H, Tack A J, Swift M M. Mnemosyne: lightweight persistent memory. Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, New York, NY, USA: ACM, 2011. 91–104.

[23] Coburn J, Caulfield A M, Akel A, et al. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA: ACM, 2011. 105–118.

[24] Nightingale E B, Veeraraghavan K, Chen P M, et al. Rethink the sync. Proceedings of the 7th symposium on Operating systems design and implementation, Berkeley, CA, USA: USENIX Association, 2006. 1–14.

[25] Chidambaram V, Pillai T S, Arpaci-Dusseau A C, et al. Optimistic crash consistency. SOSP, 2013.

[26] Ma D, Feng J, Li G. LazyFTL: a page-level flash translation layer optimized for nand flash memory. Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, New York, NY, USA: ACM, 2011. 1–12.

[27] Mickens J, Nightingale E B, Elson J, et al. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, Berkeley, CA, USA: USENIX Association, 2014. 257–273.

[28] Ports D R K, Clements A T, Zhang I, et al. Transactional consistency and automatic management in an application data cache. Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, Berkeley, CA, USA: USENIX Association, 2010. 1–15.

[29] DeBrabant J, Pavlo A, Tu S, et al. Anti-caching: A new approach to database management system architecture. Proc. VLDB Endow., 2013, 6(14):1942–1953.

[30] DeWitt D J, Katz R H, Olken F, et al. Implementation techniques for main memory database systems. Proceedings of the 1984 ACM SIGMOD international conference on Management of data, New York, NY, USA: ACM, 1984. 1–8.

[31] Kallman R, Kimura H, Natkins J, et al. H-store: A high-performance, distributed main memory transaction processing system. Proc. VLDB Endow., 2008, 1(2):1496–1499.

[32] Ongaro D, Rumble S M, Stutsman R, et al. Fast crash recovery in ramcloud. Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, New York, NY, USA: ACM, 2011. 29–41.

[33] Kim Y S, Jin H, Woo K G. Adaptive logging for mobile device. Proc. VLDB Endow., 2010, 3(1-2):1481–1492.

[34] Satyanarayanan M, Mashburn H H, Kumar P, et al. Lightweight recoverable virtual memory. Proceedings of the fourteenth ACM symposium on Operating systems principles, New York, NY, USA: ACM, 1993. 146–160.

[35] Dai H, Neufeld M, Han R. ELF: An efficient log-structured flash file system for micro sensor nodes. Proceedings of the 2nd international conference on Embedded networked sensor systems, New York, NY, USA: ACM, 2004. 176–187.

[36] Kim H, Ryu M, Ramachandran U. What is a good buffer cache replacement scheme for mobile flash storage? Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems, New York, NY, USA: ACM, 2012. 235–246.

[37] Lv Y, Cui B, He B, et al. Operation-aware buffer management in flash-based systems. Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, New York, NY, USA: ACM, 2011. 13–24.

[38] Coburn J, Bunker T, Schwarz M, et al. From aries to mars: Transaction support for next-generation, solid-state drives. SOSP, 2013.

[39] Hitz D, Lau J, Malcolm M. File system design for an nfs file server appliance. Proceedings of the USENIX Winter 1994 Technical Conference. San Francisco, CA, 1994. 235–246.

[40] Lee E, Kang H, Bahn H, et al. Eliminating periodic flush overhead of file I/O with non-volatilebuffer cache. IEEE Transactions on Computers, 2014, (99).

[41] Wu M, Zwaenepoel W. eNVy: a non-volatile, main memory storage system. Proceedings of the sixth international conference on Architectural support for programming languages and operating systems, New York, NY, USA: ACM, 1994. 86–97.

[42] Zhong K, Wang T, Zhu X, et al. Building high-performance smartphones via non-volatile memory: The swap approach. The ACM SIGBED International Conference on Embedded Software, 2014.

[43] Luo H, Tian L, Jiang H. qNVRAM: quasi non-volatile ram for low overhead persistency enforcement in smartphones. 6th USENIX Workshop on Hot Topics in Storage and File Systems, 2014.

[44] Jeong S, Lee K, Lee S, et al. I/O stack optimization for smartphones. Proceedings of the 2013 USENIX Conference on Annual Technical Conference, Berkeley, CA, USA: USENIX Association, 2013. 309–320.

[45] Kim H, Ramachandran U. Fjord: Informed storage management for smartphones. IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST), 2013.

[46] Koller R, Marmol L, Rangaswami R, et al. Write policies for host-side flash caches. Proceedings of the 11th USENIX conference on File and Storage Technologies, Berkeley, CA, USA: USENIX Association, 2013.

[47] Nightingale E B, Flinn J. Energy-efficiency and storage flexibility in the blue file system. Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation, Berkeley, CA, USA: USENIX Association, 2004. 25–25.

[48] Nguyen D T, Zhou G, Qi X, et al. Storage-aware smartphone energy savings. Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing, New York, NY, USA: ACM, 2013. 677–686.

[49] Mathur G, Desnoyers P, Ganesan D, et al. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. Proceedings of the 4th international conference on Embedded networked sensor systems, New York, NY, USA: ACM, 2006. 195–208.

[50] Nguyen D T. Improving smartphone responsiveness through I/O optimizations. Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication, New York, NY, USA: ACM, 2014. 337–342.

[51] Chun B G, Curino C, Sears R, et al. Mobius: Unified messaging and data serving for mobile apps. Proceedings of the 10th international conference on Mobile systems, applications, and services, New York, NY, USA: ACM, 2012. 141–154.

[52] Papathanasiou A, Scott M. Energy efficiency through burstiness. Proceedings of the Fifth IEEE Workshop on Mobile Computing Systems and Applications, 2003. 44 – 53.

[53] Weissel A, Beutel B, Bellosa F. Cooperative i/o: a novel i/o semantics for energy-aware applications. Proceedings of the 5th symposium on Operating systems design and implementation, New York, NY, USA: ACM, 2002. 117–129.

[54] Go Y, Agrawal N, Aranya A, et al. Reliable, consistent, and efficient data sync for mobile apps. FAST, 2015.

[55] Lee C, Sim D, Hwang J, et al. F2fs: A new file system for flash storage. FAST, 2015.

[56] Klug B. Moto x review. http://www.anandtech.com/show/7235/moto-x-review/9, 2013.

[57] Josephson W K, Bongo L A, Flynn D, et al. DFS: A file system for virtualized flash storage. Proceedings of the 8th USENIX Conference on File and Storage Technologies, Berkeley, CA, USA: USENIX Association, 2010. 7–7.

[58] Lee S W, Moon B. Design of flash-based dbms: An in-page logging approach. Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, New York, NY, USA: ACM, 2007. 55–66.

[59] Lee S W, Moon B, Park C, et al. A case for flash memory SSD in enterprise database applications. Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, New York, NY, USA: ACM, 2008. 1075–1086.

[60] Lee S W, Moon B, Park C. Advances in flash memory ssd technology for enterprise database applications. Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, New York, NY, USA: ACM, 2009. 863–870.

[61] Chen S. FlashLogging: exploiting flash devices for synchronous logging performance. Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, New York, NY, USA: ACM, 2009. 73–86.

[62] Johnson R, Pandis I, Stoica R, et al. Aether: A scalable approach to logging. Proc. VLDB Endow., 2010, 3(1-2):681–692.

[63] Johnson R, Pandis I, Stoica R, et al. Scalability of write-ahead logging on multicore and multisocket hardware. The VLDB Journal, 2012, 21(2):239–263.

[64] Pandis I, Johnson R, Hardavellas N, et al. Data-oriented transaction execution. volume 3. VLDB Endowment, 2010. 928–939.

[65] Welsh M, Culler D, Brewer E. Seda: An architecture for well-conditioned, scalable internet services. Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, New York, NY, USA: ACM, 2001. 230–243.

[66] Behren R, Condit J, Zhou F, et al. Capriccio: Scalable threads for internet services. Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, New York, NY, USA: ACM, 2003. 268–281.

[67] Zheng W, Tu S, Kohler E, et al. Fast databases with fast durability and recovery through multicore parallelism. Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, Berkeley, CA, USA: USENIX Association, 2014. 465–477.

[68] Huang J, Schwan K, Qureshi M K. NVRAM-aware logging in transaction systems. Proc. VLDB Endow., 2014, 8(4):389–400.

[69] Wang T, Johnson R. Scalable logging through emerging non-volatile memory. Proc. VLDB Endow., 2014, 7(10):865–876.

[70] Fang R, Hsiao H I, He B, et al. High performance database logging using storage class memory. IEEE 27th International Conference on Data Engineering, 2011. 1221–1231.

[71] Arulraj J, Pavlo A, Dulloor S R. Let's talk about storage & recovery methods for non-volatile memory database systems. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, New York, NY, USA: ACM, 2015. 707–722.

[72] Kannan S, Gavrilovska A, Schwan K. Reducing the cost of persistence for nonvolatile heaps in end user devices. Proceedings of the 20th International Symposium On High Performance Computer Architecture, 2014. 1–12.

[73] Pelley S, Chen P M, Wenisch T F. Memory persistency. Proceedings of the International Symposium on Computer Architecture, 2014. 1–12.

[74] Chakrabarti D R, Boehm H J, Bhandari K. Atlas: Leveraging locks for non-volatile memory consistency. Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, New York, NY, USA: ACM, 2014. 433–452.

[75] Saxena M, Shah M A, Harizopoulos S, et al. Hathi: Durable transactions for memory using flash. Proceedings of the Eighth International Workshop on Data Management on New Hardware, New York, NY, USA: ACM, 2012. 33–38.

[76] Sears R, Brewer E. Stasis: Flexible transactional storage. Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Berkeley, CA, USA: USENIX Association, 2006. 29–44.

[77] Santry D, Voruganti K. Violet: A storage stack for IOPS/capacity bifurcated storage environments. Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, Berkeley, CA, USA: USENIX Association, 2014. 13–24.

[78] Hwang T, Jung J, Won Y. Heapo: Heap-based persistent object store. Trans. Storage, 2014, 11(1):3:1–3:21.

[79] Balakrishnan M, Malkhi D, Wobber T, et al. Tango: Distributed data structures over a shared log. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, New York, NY, USA: ACM, 2013. 325–340.

[80] Huang J, Badam A, Qureshi M K, et al. Unified address translation for memory-mapped ssds with flashmap. Proceedings of the 42Nd Annual International Symposium on Computer Architecture, New York, NY, USA: ACM, 2015. 580–591.

[81] Stoica R, Athanassoulis M, Johnson R, et al. Evaluating and repairing write performance on flash devices. Proceedings of the Fifth International Workshop on Data Management on New Hardware, New York, NY, USA: ACM, 2009. 9–14.

[82] Chen F, Koufaty D A, Zhang X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems, New York, NY, USA: ACM, 2009. 181–192.

[83] Jung M, Kandemir M. Revisiting widely held ssd expectations and rethinking system-level implications. Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems, New York, NY, USA: ACM, 2013. 203–216.

[84] Caulfield A M, De A, Coburn J, et al. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA: IEEE Computer Society, 2010. 385–395.

[85] Kang W H, Lee S W, Moon B, et al. Durable write cache in flash memory ssd for relational and nosql databases. Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, New York, NY, USA: ACM, 2014. 529–540.

[86] Venkataraman S, Tolia N, Ranganathan P, et al. Consistent and durable data structures for non-volatile byte-addressable memory. Proceedings of the 9th USENIX conference on File and stroage technologies, Berkeley, CA, USA: USENIX Association, 2011. 5–5.

[87] Intel, a collection of linux persistent memory programming examples, https://github.com/pmem/linux-examples.

[88] SNIA. NVM programming model (NPM), version 1, 2013. http://snia.org/sites/default/files/NVMProgrammingModel_v1.pdf.

[89] Zhao J, Li S, Yoon D H, et al. Kiln: Closing the performance gap between systems with and without persistence support. Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, New York, NY, USA: ACM, 2013. 421–432.

[90] Moraru I, Andersen D G, Kaminsky M, et al. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. Proceedings of the ACM Conference on Timely Results in Operating Systems, 2013. 1–17.

[91] Liu R S, Shen D Y, Yang C L, et al. NVM Duet: Unified working memory and persistent store architecture. Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA: ACM, 2014. 455–470.

[92] Shapiro J S, Smith J M, Farber D J. EROS: a fast capability system. Proceedings of the seventeenth ACM symposium on Operating systems principles, New York, NY, USA: ACM, 1999. 170–185.

[93] Chen P M, Ng W T, Chandra S, et al. The Rio file cache: Surviving operating system crashes. Proceedings of the seventh international conference on Architectural support for programming languages and operating systems, New York, NY, USA: ACM, 1996. 74–83.

[94] Condit J, Nightingale E B, Frost C, et al. Better I/O through byte-addressable, persistent memory. Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, New York, NY, USA: ACM, 2009. 133–146.

[95] Pelley S, Wenisch T F, Gold B T, et al. Storage management in the NVRAM era. Proceedings of the VLDB Endowment, 2013, 7(2).

[96] Yoon H, Meza J, Ausavarungnirun R, et al. Row buffer locality aware caching policies for hybrid memories. IEEE 30th International Conference on Computer Design, 2012. 337–344.

[97] Lu Y, Shu J, Sun L, et al. Loose-ordering consistency for persistent memory. 32nd IEEE International Conference on Computer Design, 2014. 216–223.

[98] Zhao J, Mutlu O, Xie Y. FIRM: Fair and high-performance memory control for persistent memory systems. 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014. 153–165.

[99] Yoon H, Meza J, Muralimanohar N, et al. Efficient data mapping and buffering techniques for multilevel cell phase-change memories. ACM Trans. Archit. Code Optim., 2014, 11(4):40:1–40:25.

[100] Narayanan D, Hodson O. Whole-system persistence. Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA: ACM, 2012. 401–410.

[101] Flynn D, Nellans D, Strasser J, et al. Apparatus, system, and method for auto-commit memory, June 14, 2012. US Patent App. 13/324,942.

[102] Wang Y M, Huang Y, Vo K P, et al. Checkpointing and its applications. Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995. 22–31.

[103] Moody A, Bronevetsky G, Mohror K, et al. Design, modeling, and evaluation of a scalable multi-level checkpointing system. International Conference for High Performance Computing, Networking, Storage and Analysis, 2010. 1–11.

[104] Prvulovic M, Zhang Z, Torrellas J. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. Proceedings of 29th Annual International Symposium on Computer Architecture, 2002. 111–122.

[105] Sorin D, Martin M, Hill M, et al. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. Proceedings of 29th Annual International Symposium on Computer Architecture, 2002. 123–134.

[106] Gao S, He B, Xu J. Real-time in-memory checkpointing for future hybrid memory systems. Proceedings of the 29th ACM on International Conference on Supercomputing, 2015. 263–272.

[107] Gefflaut A, Banatre M, Kermarrec A M, et al. COMA: An opportunity for building fault-tolerant scalable shared memory multiprocessors. 23rd Annual International Symposium on Computer Architecture, 1996. 56–56.

[108] Islam T, Mohror K, Bagchi S, et al. Mcrengine: A scalable checkpointing system using data-aware aggregation and compression. International Conference for High Performance Computing, Networking, Storage and Analysis, 2012. 1–11.

[109] Oliner A J, Rudolph L, Sahoo R K. Cooperative checkpointing: A robust approach to large-scale systems reliability. Proceedings of the 20th Annual International Conference on Supercomputing, New York, NY, USA: ACM, 2006. 14–23.

[110] Rajachandrasekar R, Potluri S, Venkatesh A, et al. Mic-check: A distributed check pointing framework for the intel many integrated cores architecture. Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, New York, NY, USA: ACM, 2014. 121–124.

[111] Kang J U, Jo H, Kim J S, et al. A superblock-based flash translation layer for nand flash memory. Proceedings of the 6th ACM &Amp; IEEE International Conference on Embedded Software, New York, NY, USA: ACM, 2006. 161–170.

[112] Lee S W, Park D J, Chung T S, et al. A log buffer-based flash translation layer using fully-associative sector translation. volume 6, New York, NY, USA: ACM, 2007.

[113] Desnoyers P. What systems researchers need to know about nand flash. Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems, 2013.

[114] Kim H, Agrawal N, Ungureanu C. Revisiting storage for smartphones. Proceedings of the 10th USENIX Conference on File and Storage Technologies, Berkeley, CA, USA: USENIX Association, 2012.

[115] Lee K, Won Y. Smart layers and dumb result: IO characterization of an android-based smartphone. Proceedings of the tenth ACM international conference on Embedded software, New York, NY, USA: ACM, 2012. 23–32.

[116] Li J, Badam A, Chandra R, et al. On the energy overhead of mobile storage systems. FAST, 2014.

[117] Xu F, Liu Y, Moscibroda T, et al. Optimizing background email sync on smartphones. MobiSys, 2013.

[118] IEEE and The Open Group. IEEE Std 1003.1-2008 (POSIX.1-2008). http://pubs.opengroup.org/onlinepubs/9699919799/, 2015.

[119] Recovery Specialties, LLC. Data consistency: Explained. http://recoveryspecialties.com/dc01.html, 2015.

[120] Bailis P, Venkataraman S, Franklin M J, et al. Probabilistically bounded staleness for practical partial quorums. Proc. VLDB Endow., 2012, 5(8):776–787.

[121] Kim W H, Nam B, Park D, et al. Resolving journaling of journal anomaly in android I/O: Multi-version b-tree with lazy split. FAST, 2014.

[122] Nguyen D T, Peng G, Graham D, et al. Smartphone application launch with smarter scheduling. Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication, 2014.

[123] Rodeh O, Bacik J, Mason C. BTRFS: The linux b-tree filesystem. ACM Trans. Storage (TOS), 2013, 9(3).

[124] HTTP archive trends. http://httparchive.org/trends.php, 2014.

[125] Wang A I, Reiher P L, Popek G J, et al. Conquest: Better performance through a disk/persistent-ram hybrid file system. Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA: USENIX Association, 2002. 15–28.

[126] Carroll A, Heiser G. An analysis of power consumption in a smartphone. Proceedings of the 2010 USENIX conference on USENIX annual technical conference, 2010. 21–21.

[127] Mathur A, Cao M, Bhattacharya S, et al. The new ext4 filesystem: current status and future plans. Proceedings of the Linux Symposium, 2007.

[128] Monsoon power monitor. http://www.msoon.com/LabEquipment/PowerMonitor/, 2015.

[129] The monkeyrunner tool. http://developer.android.com/tools/help/monkeyrunner_concepts.html, 2015.

[130] Takeuchi K. Novel co-design of NAND flash memory and NAND flash controller circuits for sub-30 nm low-power high-speed solid-state drives (SSD). IEEE Journal of Solid-State Circuits, 2009, 44(4):1227–1234.

[131] Kawahara A, Azuma R, Ikeda Y, et al. An 8 mb multi-layered cross-point ReRAM macro with 443 mb/s write throughput. IEEE Journal of Solid-State Circuits, 2013, 48(1):178–185.

[132] Zuloaga S, Liu R, Chen P Y, et al. Scaling 2-layer RRAM cross-point array towards 10 nm node: A device-circuit co-design. 2015 IEEE International Symposium on Circuits and Systems (ISCAS), 2015. 193–196.

[133] Kimura H. Foedus: Oltp engine for a thousand cores and nvram. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, New York, NY, USA: ACM, 2015. 691–706.

[134] Wang Z, Yi H, Liu R, et al. Persistent transactional memory. Computer Architecture Letters, 2015, 14(1):58–61.

[135] Johnson R, Pandis I, Stoica R, et al. Scalability of write-ahead logging on multicore and multisocket hardware. The VLDB Journal, 2012, 21(2):239–263.

[136] Ren C, Lo E, Kao B, et al. On querying historical evolving graph sequences. Proceedings of the VLDB Endowment, 2011, 4(11):726–737.

[137] Cheng R, Hong J, Kyrola A, et al. Kineograph: Taking the pulse of a fast-changing and connected world. Proceedings of the 7th ACM European Conference on Computer Systems, New York, NY, USA: ACM, 2012. 85–98.

[138] Watson H, Wixom B H. The current state of business intelligence. Computer, 2007, 40(9):96–99.

[139] Corbett J C, Dean J, Epstein M, et al. Spanner: Google's globally-distributed database. Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, Berkeley, CA, USA: USENIX Association, 2012. 251–264.

[140] Daudjee K, Salem K. Lazy database replication with snapshot isolation. Proceedings of the 32Nd International Conference on Very Large Data Bases. VLDB Endowment, 2006. 715–726.

[141] Fekete A, Liarokapis D, O'Neil E, et al. Making snapshot isolation serializable. ACM Trans. Database Syst., 2005, 30(2):492–528.

[142] Ports D R K, Grittner K. Serializable snapshot isolation in postgresql. Proc. VLDB Endow., 2012, 5(12):1850–1861.

[143] Litz H, Cheriton D, Firoozshahian A, et al. SI-TM: Reducing transactional memory abort rates through snapshot isolation. Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA: ACM, 2014. 383–398.

[144] Belay A, Bittau A, Mashtizadeh A, et al. Dune: Safe user-level access to privileged cpu features. Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, Berkeley, CA, USA: USENIX Association, 2012. 335–348.

[145] Cooper B F, Silberstein A, Tam E, et al. Benchmarking cloud serving systems with ycsb. Proceedings of the 1st ACM Symposium on Cloud Computing, New York, NY, USA: ACM, 2010. 143–154.

[146] Intel Threading Building Blocks. concurrent_unordered_map and concurrent_unordered_multimap Template Classes. https://www.threadingbuildingblocks.org/docs/help/reference/containers_overview/concurrent_unordered_map_cls.htm, 2015.

[147] Intel Threading Building Blocks. concurrent_hash_map Template Class. https://www.threadingbuildingblocks.org/docs/help/reference/containers_overview/concurrent_hash_map_cls.htm, 2015.

[148] Lamb C, Landis G, Orenstein J, et al. The ObjectStore database system. Commun. ACM, 1991, 34(10):50–63.

[149] Copeland G, Keller T, Krishnamurthy R, et al. The case for safe RAM. Proceedings of the 15th International Conference on Very Large Data Bases, 1989. 327–335.

[150] Cascaval C, Blundell C, Michael M, et al. Software transactional memory: Why is it only a research toy? Queue, 2008, 6(5):40:46–40:58.

[151] Pankratius V, Adl-Tabatabai A R. A study of transactional memory vs. locks in practice. Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures, New York, NY, USA: ACM, 2011. 43–52.

[152] Dice D, Lev Y, Moir M, et al. Early experience with a commercial hardware transactional memory implementation. Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA: ACM, 2009. 157–168.

[153] Minh C C, Chung J, Kozyrakis C, et al. STAMP: Stanford transactional applications for multi-processing. Proceedings of the IEEE International Symposium on Workload Characterization, 2008.

[154] Felber P, Fetzer C, Riegel T. Dynamic performance tuning of word-based software transactional memory. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008. 237–246.

[155] Transactional memory in GCC. https://gcc.gnu.org/wiki/TransactionalMemory, 2012.

[156] Meza J, Chang J, Yoon H, et al. Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management. Computer Architecture Letters, 2012..

[157] Yoon D H, Erez M. Virtualized and flexible ECC for main memory. Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, 2010.

[158] Burcea I, Somogyi S, Moshovos A, et al. Predictor virtualization. Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, 2008.

[159] Pawlowski J T. Hybrid memory cube (hmc). Hot Chips, 2011.

[160] Labs H. The machine: A new kind of computer, 2015. http://www.hpl.hp.com/research/systems-research/themachine/.

[161] Binkert N, Beckmann B, Black G, et al. The gem5 simulator. SIGARCH Comput. Archit. News, 2011, 39(2):1–7.

[162] Hansson A, Agarwal N, Kolli A, et al. Simulating dram controllers for future system architecture exploration. IEEE International Symposium on Performance Analysis of Systems and Software, 2014. 201–210.

[163] Lee B C, Ipek E, Mutlu O, et al. Architecting phase change memory as a scalable DRAM alternative. Proceedings of the 36th Annual International Symposium on Computer Architecture, New York, NY, USA: ACM, 2009. 2–13.

[164] Hagmann R. Reimplementing the Cedar file system using logging and group commit. Proceedings of the eleventh ACM Symposium on Operating systems principles, New York, NY, USA: ACM, 1987. 155–162.

[165] Linux Community. Ext4 (and ext3) filesystem wiki. https://ext4.wiki.kernel.org, 2014.

[166] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. Crash consistency: FSCK and journaling, 2014. http://pages.cs.wisc.edu/ remzi/OSTEP/.

[167] Bernstein P A, Newcomer E. Principles of transaction processing. Morgan Kaufmann, 2009.

[168] Ghemawat S, Gobioff H, Leung S T. The google file system. Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, New York, NY, USA: ACM, 2003. 29–43.

[169] Ford D, Labelle F, Popovici F I, et al. Availability in globally distributed storage systems. Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, Berkeley, CA, USA: USENIX Association, 2010. 1–7.

[170] C++ Tutorials. Exceptions. http://www.cplusplus.com/doc/tutorial/exceptions/, 2015.

[171] Robillard M P, Murphy G C. Analyzing exception flow in java programs. Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, London, UK, UK: Springer-Verlag, 1999. 322–337.

[172] Ogasawara T, Komatsu H, Nakatani T. EDO: Exception-directed optimization in Java. volume 28, New York, NY, USA: ACM, 2006. 70–105.

[173] Goodenough J B. Exception handling: Issues and a proposed notation. Commun. ACM, 1975, 18(12):683–696.

[174] Fu C, Ryder B. Exception-chain analysis: Revealing exception handling architecture in java server applications. 29th International Conference on Software Engineering, 2007. 230–239.

[175] Platt R. Undo/redo and save actions corrupting files. https://bugs.eclipse.org/bugs/show_bug.cgi?id=443427, 2015.

[176] Bean@gmailcom. Unrecoverable chrome.storage.sync database corruption. https://code.google.com/p/chromium/issues/detail?id=261623, 2014.

[177] Cipar J, Ganger G, Keeton K, et al. LazyBase: Trading freshness for performance in a scalable database. Proceedings of the 7th ACM European Conference on Computer Systems, New York, NY, USA: ACM, 2012. 169–182.

[178] Hopcroft J E, Motwani R, Ullman J D. Introduction to Automata Theory, Languages, and Computation (Third Edition). Prentice Hall, 2006.

致 谢

衷心感谢导师郑纬民教授和武永卫教授。他们在学术上悉心指导，在生活上关心照顾，为学生的发展构建了广阔的平台。忘不了和老师们的一次次讨论，忘不了老师们的谆谆教诲，也忘不了武老师在我的婚礼上作为主婚人的温馨致辞。

我还要感谢以往论文的其他合作者和指导者，他们的智慧亦融合在我的毕业论文中。微软亚洲研究院的 Thomas Moscibroda 和 Mike Liang 研究员为 MobiFS 项目的策略设计给予了宝贵建议。卡内基•梅隆大学（Carnegie Mellon University）的 Onur Mutlu 教授帮助我挖掘想法中价值，和时任惠普实验室（HP Labs）研究员的 Jishen Zhao 博士、时任英特尔研实验室（Intel Labs）研究员的 Samira Khan 博士一起反复修改我的论文。斯坦福大学（Stanford University）的 David Cheriton 教授和 Heiner Litz 博士多次斧正我的想法，他们组的工作为我的论文提供了很好的基础。我合作的论文中，当时威斯康星大学麦迪逊分校（University of Wisconsin–Madison）的 Shan Lu 教授给予的帮助和指导，在此一并感谢；原 SanDisk 副总裁 John Butsh 博士、Facebook 工程师 Justin Meza 博士对论文工作的判断和帮助也令我受益匪浅。此外，特别感谢清华实验室的陈康老师、蒋进磊老师、黄小猛老师、杨广文老师等多年来寄予的指导和照顾；师兄弟刘立坤、张扬、祝美祺、章明星、王博、郭维超、赵勋、牟帅、王秋平、叶丰、苏茂萌、高品等同学，不断和我分享着他们的创见、努力和一起玩乐的美好时光。

感谢我的妻子，她无私无畏的支持和先于我工作所给予的经济援助，帮我度过了无数难关。儿子的到来，赐予我生命的灵感和前行的动力。更要感谢我的父母和岳父母，他们构筑了我多年科研工作的坚实后方，是我毕业论文能够成文的不可磨灭的功臣。

最后，感谢自然科学基金、国家 863 计划、国家 973 计划、清华大学博士生短期出国访学基金、清华大学信息科学技术学院登峰基金、斯坦福大学计算机科学系等对本人工作的资助。感谢答辩委员会专家及匿名评审专家的宝贵时间和指导。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

个人简历、在学期间发表的学术论文与研究成果

个人简历

1986 年 9 月 12 日出生于河北省保定市。

2006 年 9 月考入东北师范大学软件学院软件工程专业，2010 年 7 月本科毕业并获得工学学士学位。

2010 年 9 月免试进入清华大学计算机科学与技术系攻读硕士学位；2013 年 7 月转为提前攻读博士学位至今。

博士在读期间，2013 年 11 月至 2014 年 5 月在卡内基梅隆大学（Carnegie Mellon University）电子与计算机工程系做访问研究；2014 年 10 月至 2015 年 8 月在斯坦福大学（Stanford University）计算机科学系做访问研究。

发表的学术论文

[1] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems. In Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 672 - 685, Dec. 2015.（CCF 推荐 A 类会议）

[2] Jinglei Ren, Chieh-Jan Mike Liang, Yongwei Wu, and Thomas Moscibroda. Memory-Centric Data Storage for Mobile Systems. In Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC), pp. 599 - 611, Jul. 2015.（CCF 推荐 A 类会议）

[3] Jinglei Ren, Yongwei Wu, Meiqi Zhu, and Weimin Zheng. Quatrain: Accelerating Data Aggregation Between Multiple Layers. IEEE Transactions on Computers (TC), Vol. 63, No. 5, pp. 1207 - 1219, May 2014.（CCF 推荐 A 类期刊，SCI 检索）

[4] Mingxing Zhang, Yongwei Wu, Shan Lu, Shanxiang Qi, Jinglei Ren, and Weimin Zheng. AI: A Lightweight System for Tolerating Concurrency Bugs. In Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), pp. 330 - 340, Nov. 2014.（CCF 推荐 A 类会议）

[5] Yongwei Wu, Weichao Guo, Jinglei Ren, Xun Zhao, and Weiming Zheng. NO2: Speeding Up Parallel Processing of Massive Compute-Intensive Tasks. IEEE Transactions on Computers (TC), Vol. 63, No. 10, pp. 2487 - 2499, Oct. 2014.（CCF

推荐 A 类期刊，SCI 检索）

研究成果

[1] 武永卫、郑纬民、陈康、任晶磊: 远程调用方法及系统. (中国专利申请号：201310646738.0)