CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Edinburgh Research Explorer

THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

DAPPER: a database-inspired approach to persistent memory

Citation for published version:
Cintra, M, Chatzistergiou, A, Joshi, A, Nagarajan, V & Viglas, SD 2015, DAPPER: a database-inspired approach to persistent memory. in The 6th Annual Non-Volatile Memories Workshop (NVMW 2015).

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Publisher's PDF, also known as Version of record

Published In:
The 6th Annual Non-Volatile Memories Workshop (NVMW 2015)

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. Apr. 2019

DAPPER: a database-inspired approach to persistent memory

Marcelo Cintra
Intel, Germany
marcelo.cintra@intel.com

Andreas Chatzistergiou, Arpit Joshi, Vijay Nagarajan, Stratis D. Viglas
University of Edinburgh, UK
{{a.chatzistergiou, a.joshi}@sms, {vnagaraj, sviglas}@inf}.ed.ac.uk

Abstract

Persistent memory collapses the boundaries between the in-memory and secondary storage representations of data structures, and enables the programmer to process data directly from an imperative runtime. We present the early results of the DAPPER project, which takes a database-inspired approach to persistent memory. It supports recoverable data structures in persistent memory at the imperative language level, algorithms optimized for the performance characteristics of the new medium and a runtime to support them, and workload-driven adaptive data placement.

1 Introduction

Fast byte-addressable persistent memory brings the possibility of a universal persistent memory device that can replace both volatile memory and persistent storage. But how should it be used? Treating persistent memory as persistent storage means that we using algorithms and techniques that have been designed for a different medium. Treating persistent memory as volatile memory means that we will need to rethink our data structures in light of the new available capacity and assume that everything can be memory-resident. In both alternatives, we are effectively reusing a prior abstraction that has been originally built for different performance characteristics. We argue that we need to collapse these abstractions into a single universal interface.

Our work is also fuelled by the observation that if these abstractions are not collapsed, then applications will need to account for different representations based on whether data resides in volatile DRAM or persistent memory. Consider a multi-tier application: the programmer decides on the application-level control and data structures, and then decides on the persistent representation of the data structures. Specialized APIs translate data between the two runtimes, in a cumbersome and sometimes error-prone process. Moreover, data may be replicated in both DRAM and non-volatile memory (NVM[1]), while the byte-addressability of NVM is not leveraged. In the future, what is more likely to be the case is that NVM will be accessed directly from the programming language. This is a highly disruptive model when compared to contemporary systems. We argue that we need a solution that is not intrusive to the programmer and seamlessly integrates the application's data structures with their persistent representation at the system level.

We present the preliminary results of the DAPPER project, which aims to deliver a database-inspired approach to persistent memory. We assume a hybrid system with both DRAM and NVM and we identify the best use of both substrates when dealing with typical data management use-cases in the context of an imperative programming language. Our approach is shown in Figure 1. We use compiler support to identify transactional code fragments and a lightweight data management library to provide transactional semantics and recoverability to arbitrary data structures (Section 2). We then move on to mitigating the write inefficiency of NVM for two typical operations commonly encountered in data management applications: sorting and hash-based relational join processing (Section 3). Finally, we give methods for choosing what data to store in either DRAM or NVM depending on the workload and the data access patterns (Section 4). In what follows, we will briefly present each problem and our solution before concluding and identifying future work directions (Section 5).

[1] We use the terms 'persistent' and 'non-volatile' interchangeably.

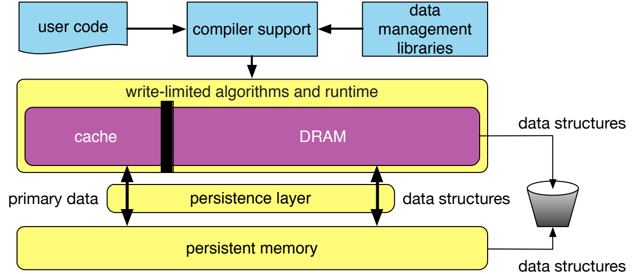

Figure 1: The overarching DAPPER approach

2 Recoverable data structures

The first step in enabling data processing over NVM is ensuring that persistence is readily accessible to the higher level programming substrates. This means that the data structures chosen by the programmer are persistent and recoverable. We have designed and implemented a recovery substrate for imperative languages termed REWIND, which stands for *REcovery Write-ahead system for In-memory Non-volatile Data structures* [1].

Our processing model is that persistent data is on byte-addressable NVM, accessible directly from user code through CPU loads and stores. Traditionally, data updates are first performed in volatile memory. It is thus possible to delay making log entries persistent until the transaction commits or the data updates are purged from main memory. In REWIND, updates are done directly on NVM data: the log entries must be made persistent immediately, and ahead of the data updates. We achieve this through enhanced versions of memory fences (*i.e.,* barriers that enforce ordering and persistence to preceding instructions), cacheline flushes and non-temporal stores (*i.e.,* direct to NVM stores that bypass the cache) with persistence guarantees.

REWIND uses physical logging as it fits better with imperative languages and allows easier compiler support. The log itself must be manipulated atomically in a recoverable way. Traditionally, the log is maintained in volatile memory and pushed to persistent storage through system calls. In REWIND, the log itself resides in persistent main memory and updates are made in-place. Transactional handling of failure of log updates is attained with carefully crafted data structures and code sequences. Furthermore, performance is relative to a baseline with the low cost of individual memory operations. Thus, logging must be optimized to incur only a small increase in the cost of a memory operation. In REWIND, we guarantee this with minimalist data structures and code sequences.

While contemporary systems offer record-level locking, they use coarse-grained page-level latching internally. REWIND employs fine-grained latching at a log record granularity: this enables more efficient and flexible locking mechanisms. The majority of recovery managers based on ARIES [5] are implemented within DBMSs [2]. Thus, they hide data management behind some data model (*e.g.,* relational) and allow data manipulation through a query language (*e.g.,* SQL). REWIND is implemented as a user-mode library that can be linked to any native application, allowing the programmer to access the data using an arbitrary sequence

of imperative commands. Moreover, the design of REWIND itself is such that it can be straightforwardly embedded into the compiler so that the disruption to user code is further minimized.

REWIND currently provides strong ACID semantics: at the time of crash, we guarantee that whatever transactions are visible at that instant are also made durable. We are currently working on relaxing this with a scheme in which durability trails visibility: upon a crash, the transactions that are durable may not correspond to the current state of visible transactions, but a past (yet consistent) state. In other words, we tradeoff some loss of work (during recovery) for gains in logging performance. We are also working on adding architectural support to speed up the logging.

3 Write-limited algorithms and supporting runtime

With the mechanics of recoverability in place, we next incorporate higher-level workflows that will leverage persistent data structures for more elaborate processing. We have focussed on two types of data-centric operations: sorting and hash-based relational join processing. Sorting is ubiquitous in a host of data processing algorithms and solutions. Whereas hash-based relational join processing builds on the powerful technique of splitting a dataset in disjoint partitions. Sorting and hash partitioning are used in data mining, (*e.g.,* producing association rules), machine learning (*e.g.,* clustering), and graph management (*e.g.,* nearest neighbor search).

We have devised a family of algorithms that we term *write-limited* that focus on mitigating the write cost of persistent memory for sorting and hash-partitioning [6]. The algorithms are based on a simple observation. Consider the simple process of reading an input dataset and then writing it—perhaps by applying a total ordering on it, as is the case of sorting; or by applying a hash function, as is the case for partitioning. Assume now a write-to-read cost of λ, meaning that writing is λ times more expensive than reading. Then for the cost of writing the output, we can afford λ extra reads. We therefore leverage this ratio to trade writes for reads. The result is that we can achieve the same I/O performance as well-known algorithms (*e.g.,* external merge-sort) but at a fraction of its write cost. Or, alternatively, the developer can tune the write intensity of the algorithms for a small hit on performance. With the algorithms in place, we implement these algorithms by proposing a flexible API. Our API records a blueprint of each algorithm's computation and enables the system to dynamically decide whether to trade writes for reads. The key notion is that the generation of new datasets (be they results of computation, or intermediate structures) is deferred by default. The system keeps track of the accumulated savings and the potential cost associated with generating a dataset. It then performs a dynamic cost-benefit analysis to decide if materializing the dataset would be more cost-effective than deferring its materialization.

We have implemented all these algorithms in terms of different potential deployments of non-volatile memory in contemporary systems, ranging from treating NVM as standard memory through persistent regions; to using block-level I/O; to a thin persistent-memory-aware filesystem. Our work has quantified the impact of implementations on performance and our results show that there are various tradeoffs that one needs to take into account in choosing the correct implementation alternative. But, at any rate, a dynamic cost-benefit approach based on write-limited algorithms is necessary to tune and maximize performance.

4 Adaptive data (re)placement

Next, we enhance memory utility through informed data placement. Data structures are placed in NVM for long-term storage, but may also be cached in DRAM. Data structures are tagged according to their intended semantics by the user. This tagging is used to inform the system whether the data structures are primary, which indicates long-term usage, or temporary, which indicates short-term access. We classify data structures and use a rule-based approach to decide where they are to be placed. We further build upon flow schemes [4, 3] to capture the flow of data between the memory regions of the system.

DRAM is split in two sections: a cache and a data structure store. The system decides the size of each section. The cache only caches primary data that is stored in NVM. The data structure store maintains various temporary auxiliary data structures that are by-products of data processing: *e.g.,* a version of an input dataset sorted on some particular key, or a hash partitioning of some dataset. The system employs replacement algorithms to evict data from the cache to NVM. The system also manages the data structure store by pushing temporary data structures to NVM if they are deemed useful. Doing so frees DRAM for more data structures but also incurs the cost of expensive NVM writes. Alternatively, the system may decide to simply delete a temporary data structure from DRAM or NVM if it is no longer useful (for instance, when it is not in the 'hot-set' of the current workload).

5 Conclusions and outlook

In the DAPPER project, we aim to integrate non-volatile memory into the data management stack, by following a database-inspired approach. As NVM becomes mainstream, the boundary between volatile and persistent memory will gradually disappear. This means that the in-memory and the persistent representations of data will no longer be separate. As such, we need mechanisms that enable such seamless manipulation. We have presented our approach to address these issues: (*a*) by proposing a recovery runtime that uses write-ahead logging to provide persistence and recoverability for arbitrary data structures stored in persistent memory regions; (*b*) by introducing write-limited algorithms that trade expensive writes for cheaper reads for key data processing operations; (*c*) by incorporating these algorithms into a runtime that leverages a cost-benefit analysis to dynamically support them; (*d*) by integrating DRAM and NVM into a single storage layer that decides on data placement based on the high-level semantics and access patterns of the data structures it serves.

There is more work to be done. One might extend the recovery runtime with architectural support for zero-overhead concurrency primitives; or enrich the write-limited algorithms with generalized approaches and indexes. It would also be interesting to see the tradeoffs involved in using only persistent memory and doing away with DRAM altogether. This becomes especially important if the latency of persistent memory matches that of DRAM. At any rate, we believe our approach to be a solid first step towards optimizing data management for persistent memory.

References

[1] A. Chatzistergiou, M. Cintra, and S. D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(1), 2015.

[2] R. Fang, H.-I. Hsiao, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *ICDE*, 2011.

[3] I. Koltsidas and S. D. Viglas. Flashing Up the Storage Layer. *PVLDB*, 1(1), 2008.

[4] I. Koltsidas and S. D. Viglas. Designing a flash-aware two-level cache. In *ADBIS*, 2011.

[5] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1), 1992.

[6] S. D. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5), 2014.