THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

An Early Evaluation of Intel's Optane DC Persistent Memory Module and its Impact on High-Performance Scientific Applications

Citation for published version:
Weiland, M, Brunst, H, Quintino, T, Johnson, N, Iffrig, O, Smart, S, Herold, C, Bonanni, A, Jackson, A & Parsons, M 2019, An Early Evaluation of Intel's Optane DC Persistent Memory Module and its Impact on High-Performance Scientific Applications. in *SC'19 Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis.*, 76, ACM, pp. 1-19, International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2019), Denver, Colorado, United States, 19/11/19. https://doi.org/10.1145/3295500.3356159

Digital Object Identifier (DOI):
10.1145/3295500.3356159

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
SC'19 Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis

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: 27. Sep. 2022

An Early Evaluation of Intel's Optane DC Persistent Memory Module and its Impact on High-Performance Scientific Applications

Michèle Weiland
EPCC, The University of Edinburgh
Edinburgh, United Kingdom
m.weiland@epcc.ed.ac.uk

Holger Brunst
TU Dresden
Dresden, Germany
holger.brunst@tu-dresden.de

Tiago Quintino
ECMWF
Reading, United Kingdom
tiago.quintino@ecmwf.int

Nick Johnson
EPCC, The University of Edinburgh
Edinburgh, United Kingdom
n.johnson@epcc.ed.ac.uk

Olivier Iffrig
ECMWF
Reading, United Kingdom
olivier.iffrig@ecmwf.int

Simon Smart
ECMWF
Reading, United Kingdom
simon.smart@ecmwf.int

Christian Herold
TU Dresden
Dresden, Germany
christian.herold@tu-dresden.de

Antonino Bonanni
ECMWF
Reading, United Kingdom
antonino.bonanni@ecmwf.int

Adrian Jackson
EPCC, The University of Edinburgh
Edinburgh, United Kingdom
a.jackson@epcc.ed.ac.uk

Mark Parsons
EPCC, The University of Edinburgh
Edinburgh, United Kingdom
m.parsons@epcc.ed.ac.uk

ABSTRACT

Memory and I/O performance bottlenecks in supercomputing simulations are two key challenges that need to be addressed on the road to Exascale. The recently released byte-addressable persistent non-volatile memory technology from Intel, DCPMM, promises to be an exciting opportunity to break with the status quo, with unprecedented levels of capacity at near-DRAM speeds. In this paper, we explore the potential of DCPMM in the context of high-performance scientific computing using two distinct applications in terms of outright performance, efficiency and usability for both its Memory and App Direct modes. In Memory mode, we show that it is possible to achieve equivalent performance and better efficiency for a CASTEP simulation that struggles with memory capacity limitations on conventional DRAM-only systems without needing to introduce any changes to the application. For IFS, we demonstrate that using a distributed object-store over the NVRAM devices reduces the data contention created in weather forecasting data producer-consumer workflows. In addition to presenting the impact on two applications, we also present results for achievable memory bandwidth performance using STREAM.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SC19, November 2019, Denver, US

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**; *Multicore architectures*; • **Hardware → Memory and dense storage**.

KEYWORDS

non-volatile memory, IO performance

ACM Reference Format:
Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. 2019. An Early Evaluation of Intel's Optane DC Persistent Memory Module and its Impact on High-Performance Scientific Applications. In *Proceedings of Supercomputing19 (SC19).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Application performance on supercomputers today is often not limited by the capabilities of the processing units, but rather by the systems' ability to feed those processing units with data. The performance characteristics of memory and storage play a key role in bringing the data to the compute, and attempts at filling the latency gap in the hierarchy between main memory (DRAM) and the network attached parallel filesystems include approaches such as burst buffers or SSD storage on the compute nodes [28]. Memory capacity and bandwidth, and I/O performance, can be performance limiting factors on today's systems and are often cited as key challenges to be addressed on the road to Exascale [36] [14] [15]. The recently announced Aurora system, the first Exascale system in the US, will include byte-addressable persistent memory in order to

further bridge this latency gap [13]; it is scheduled to come online in 2022. The technology that has been chosen for the Aurora system is Intel's *Optane Data Center Persistent Memory Module* (referred to from hereon as DCPMM).

DCPMM was launched as a product on the 2nd April 2019 [7] alongside Intel's Cascade Lake CPUs (needed to support DCPMM) and there is thus far scant information in the public domain on its performance characteristics. The Non-Volatile Systems Laboratory have published basic DCPMM performance measurements based on a range of low-level benchmarks and (primarily) database applications [23]. The potential impact of the persistent memory layer on the performance of traditional high-performance computing applications (as opposed to data-centric enterprise workloads such as in-memory databases) however is not explored and is an important aspect in understanding its potential in the context of Exascale computing.

This paper is the first evaluation of the impact and performance implications of byte-addressable persistent memory, specifically DCPMM, on HPC applications that are significantly memory or I/O bound. More specifically, our contributions include:

- Exploring in detail the different usage scenarios for DCPMM, in particular for applications that are limited by memory capacity or I/O performance on current supercomputer architectures;
- Demonstrating the impact of the increased memory capacity of the compute nodes using DCPMM's Memory mode (without changing the application) in terms of runtime, energy and power;
- Evaluating the performance impact of modifying an application to use DCPMM's App Direct mode to be able to directly load/store from/to persistent memory;
- Quantifying the impact of NUMA effects using the libvmmalloc library in App Direct mode; and
- Showing the performance of DCPMM using an established profiling toolchain and the memory-bandwidth benchmark STREAM.

2 DCPMM - BYTE-ADDRESSABLE PERSISTENT MEMORY

Byte-addressable persistent memory is a form of memory that is both non-volatile (i.e. data persists even after a power cycle) and that tcan at the same time be addressed directly by the CPU through load and store operations. Intel's *Optane Data Centre Persistent Memory Module* (DCPMM for short) offering is a particular type of byte-addressable persistent memory, based on the 3D XPoint technology.

DCPMM is delivered in DIMM form factor (DDR4 socket compatible) and thus the modules sit in the DIMM slots next to the CPU, alongside standard DRAM modules. The CPU's integrated memory controller addresses both the volatile and the non-volatile memory banks equally; note that DCPMM currently relies on Intel's Cascade Lake CPUs with large memory support (model suffix "M"). DCPMM can be operated in two main platform modes (Memory mode and App Direct mode, details below), which can be changed into by rebooting the compute nodes that host the memory. The current generation of DCPMM modules are available in three different sizes (128GB, 256GB and 512GB), which gives a ~5-10x increase in capacity (per module) over DDR4 DRAM. Although DCPMM is slower than DRAM, it delivers much higher memory capacity per node than is possible with a DRAM-only solution. DCPMM cannot replace DRAM entirely: each memory channel (there are 6 per CPU with two DIMMs slots each) must be populated with at least one DRAM DIMM.

2.1 DCPMM Memory mode

In *Memory* mode, also known as two-level memory mode, the byte-addressable persistent memory is transparent to applications and represents the main memory space, while DRAM effectively becomes the last level cache. In this mode, the persistent properties of the technology are not exploited because coherence between DRAM and persistent memory cannot be guaranteed. Applications do not have to be modified to use the persistent memory in this mode. All data objects are placed into DCPMM by default.

2.2 DCPMM App Direct mode

In *App Direct* mode, also referred to as one-level memory mode, the persistent memory is only accessible via direct load and store operations and its primary use is as very fast byte-addressable non-volatile local storage. In this mode, applications can only exploit the persistent memory either if they manage it directly or if system software provides an interface (e.g. through a file system that is mounted on the persistent memory).

2.3 libvmmalloc

When a compute node is in App Direct mode, it is also possible to use the persistent memory as if it were main DRAM memory by using the *libvmmalloc* library, which is part of the Persistent Memory Development Kit (PMDK) [11]. PMDK implements SNIA's Non-Volatile Memory (NVM) Programming Model standard [10]. By using *libvmmalloc*, typically by setting the `LD_PRELOAD` environment variable, all calls to dynamic memory allocations (e.g. *malloc*, *memalign* or *free*) are intercepted and replaced with persistent memory allocations without the need to modify the application. Instead of placing data structures on the system heap, they are placed into a memory mapped file that resides in persistent memory. Unlike in Memory mode, where all data objects are placed into the non-volatile memory, using libvmmalloc means that statically allocated objects will remain in DRAM.

2.4 Configuring and managing DCPMM

In order to be able to manage the DCPMMs, verify their status and modes, Intel provide a utility called *ipmctl* [8]. The command line interface tool can be used the check the health and performance of the modules, provision their configuration, and even update their firmware. For example, changing the mode from App Direct to Memory can be achieved using two simple *ipmctl* commands (first removing the existing setup and then creating a new one):

```
ipmctl delete -f -dimm -pcd
ipmctl create -f -goal MemoryMode=100
```

An API (*libipmctl*) is also provided to enable DCPMMs to be managed programmatically. The main limitation of the ipmctl tool is that it requires root level privileges on a system, which makes it impractical for most users. However checking the mode of the DCPMMs can also be done simply by looking at *meminfo*:

```
cat /proc/meminfo | grep MemTotal
```

If this command returns `3120802508 kB` on a system that is configured with 3TB of NVRAM per node, the DCPMMs on that node are in Memory mode.

3 MEMORY & I/O BOTTLENECK USE CASES

In this paper, we evaluate the impact of using byte-addressable persistent memory, specifically DCPMM, on two distinct use case exemplars that represent realistic challenges for supercomputing resources today:

- Firstly, a simulation that requires more memory than is available per core and that must therefore to be run on a large number of nodes to satisfy memory requirements. For this use case, we evaluate a large-memory simulation using the CASTEP materials modelling code (see Section 3.1).
- Secondly, a simulation that performs a large amount of I/O, to the point where this becomes a performance limiting factor. Here we use an IFS weather forecasting simulation (see Section 3.2).

The sections below introduce the applications in more detail and describe why their resource demands can be problematic. In addition to the two applications, we also use the STREAM benchmark to establish the baseline memory-bandwidth performance for DCPMM. It is briefly described in Section 3.3.

3.1 Large memory use case: CASTEP

CASTEP [17, 22, 27, 31] is a leading simulation code for calculating the properties of materials from first principles. Using density functional theory, it can simulate a wide range of material proprieties, including energetics, structures at the atomic level, vibrational properties, and electronic response properties. CASTEP is written in Fortran90, and parallelised with a hybrid MPI and OpenMP scheme.

The computational resource demands of CASTEP vary greatly depending on the type of simulation that is undertaken. Two different test cases are used for the experiments in this paper. The first is a small, 32-atom Titanium Nitride (TiN) surface, designed to be run on a small number of cores [4]. Because of its size and short runtime, this test case is well suited to demonstrating the functionality of the performance tools and showing the principles and impact of different system configurations. The second test case is a poly-A DNA simulation with 1356 atoms in a large simulation box [3]. This test requires a large amount of memory per MPI process, and we use it here as our main example of a simulation that exceeds the memory capacity offered by many HPC systems: on our test system with 34 compute nodes, each with 48 cores and 192GB of main memory (i.e. 4GB of memory per core; for a detailed description see Section 4.1), a fully populated run using 1632 MPI processes fails with "out of memory" errors (see Table 1). CASTEP provides its own internal estimator for per process memory requirements - the estimates are not necessarily entirely accurate (they often overestimate the amount of memory that is actually required), however they generally provide a good guideline. In order to fit the problem onto the 34-node system, it is necessary to underpopulate the nodes by reducing the number of MPI processes per node. A hybrid MPI-OpenMP parallelisation scheme was implemented in CASTEP in order to reduce the memory footprint of the application [21] while still being able to use all the cores for at least part of the simulation. Table 1 gives an overview of the memory per process and parallel efficiency rating estimates as computed by CASTEP, together with the runtime for 3 iterations of the self-consistent field (SCF) solver for a range of configurations. As can be seen from the numbers, the memory requirement per process decreases with increasing numbers of nodes, but so does the estimated parallel efficiency of the simulation, and the time to solution does not improve going from 20 to 24 nodes because the simulation does not scale.

3.2 I/O intensive use case: IFS

The Integrated Forecast System (IFS) [29] is the main numerical weather prediction application currently used by ECMWF, the European for Medium Range Weather Forecasts [6], for its daily operational weather forecasts. In four daily forecast cycles, global weather observations and satellite data are assimilated to produce the initial conditions of the atmosphere and oceans, and from there new global weather forecasts are computed up to 15 days ahead. Since these models inherently contain multiple sources of uncertainty, the deterministic high-resolution forecast (IFS HRES) is complemented with a parallel run of 51 probabilistic ensemble (IFS ENS) of forecasts (50 perturbed plus 1 control unperturbed). These models are part of a larger workflow that processes the forecasts and disseminates them to the member-states and worldwide to users of ECMWF data [5]. Importantly, this workflow is time-critical and subject to a strict schedule of data delivery. Any improvement in runtime allows delivery of weather forecasts precious minutes earlier. This is paramount as the value of the data decays rapidly with time (think of energy companies playing on the futures market).

The raw output of the model are fields layered in two-dimensional slices of the atmosphere, covering all of the globe. The HRES uses a reduced Gaussian grid of average spacing of 9km with 137 vertical levels and extends to 10 days, while the ENS average spacing is 18km with 91 vertical levels extending to 15 days. The time resolution is hourly for the first 90 forecast steps, then 3-hourly until 6 days, and 6-hourly there forth. For a detailed characterization of this data set, refer to Table 2, where a further distinction between the atmospheric and the wave model is presented.

All the model output fields are in a raw form, meaning they still require post-processing to meet the requirements of the data users. This is the responsibility the product generation application (PGEN) which processes the fields written by the model into client-specific products that are then disseminated (pushed) to client destinations world-wide, to a total current amount of 30 TiB per day. In order to meet a very strict delivery schedule, each product generation task is started as soon as the straggler model finishes writing the data for a given forecast step (a time slice) and sends its *step complete* event to the workflow manager. Each product generation task is a small scale parallel computation, typically using 4 to 8 HPC compute

Table 1: DNA test case baseline performance results, including CASTEP's own estimates of memory requirements and parallel efficiency, together with measured wallclock time for 3 SCF loop iterations, for different node and process counts.

Nodes	MPI x OpenMP	Active cores	Memory estimate per process	Runtime	Overall parallel efficiency
34	48x1	1632	5,486.6 MB	OOM	N/A
24	36x1	864	6,425.0 MB	5,509.70s	20%
24	24x2	1152	7,439.2 MB	4,977.56s	18%
20	36x1	720	7,001.9 MB	5,353.51s	25%
20	24x2	960	8,280.0 MB	4,916.26s	21%
18	36x1	648	7,355.5 MB	6,551.68s	23%
18	24x2	864	8,843.5 MB	5,710.15s	31%

Table 2: Characterization of ECMWF forecast model output fields.

Model	Avg. Resolution	Field Pts	Typical Field Size	Nb Fields / cycle	Total Size / cycle	Total Size / day
HRES Atmos	9 km	6.6 M	3.2 MiB	272 K	950 GiB	3.8 TiB
HRES Wave	14 km	938 K	1.4 MiB	170 K	325 GiB	1.3 TiB
ENS Atmos	18 km	1.7 M	804 KiB	10 M	15025 GiB	60.1 TiB
ENS Wave	28 km	234 K	340 KiB	10.5M	3475 GiB	13.9 TiB

nodes, however 122 steps require processing for each forecast cycle, making it a sizable part of the operational workflow.

To store the HRES and ENS model output and serve its numerous fields to the PGEN, the data is currently written to a parallel file-system. The data is stored in an internationally mandated format for weather data interchange, named GRIB [38]. For fast access to each individual field the data is indexed according to an indexing schema to and made accessible as an object by its scientific meta-data. This domain-specific object store for weather and climate data is called the Fields Database (FDB), and has been operational at ECMWF for multiple decades. Version 5 entered operations in 2018.

There are three main factors that compound to make the above workflow challenging: (1) the sheer size of the data sets written by the model HRES/ENS and read by the product generation PGEN as described in Table 2; and (2) the concurrent nature of the workflow, where all the forecast models (1 HRES + 51 ENS) run simultaneously writing data streams per I/O task per model, together with the PGEN for each of the forecast steps which reads accross all of these output streams, (3) the time-critical nature of the workflow, with a schedule service level agreement measured with resolution of minutes

This data exchange between the data producers (forecast models) and the data consumers (product generation) is currently one of the pain points of the operational workflow. Ongoing studies for the upgrade to the next model resolution have encountered significant bottlenecks. One particular case for a resolution of HRES 5km and ENS 10km, we have obtained a time to solution of 5765s without model output, which increased 17% when model output was added (not withstanding the usage of asynchronous I/O servers). The striking fact is that this timing was further increased by 26% by having the consumers simultaneously access the dataset in the parallel filesystem. The data production (HRES and ENS runs) was slower because the data consumers were active, even though they are completely independent parallel runs. There is some evidence this is caused by contention in the data movers, with read requests for recent data throttling the write request for new data.

To compound this I/O challenge and frame this problem for the Exascale era, we need to understand these figures are aggravated with an average yearly compound growth of 40% to 45% of data sizes. Putting this into context, in 1995 the operational weather forecast produced 14 TiB yearly, whereas it currently produces roughly 75 TiB daily. This growth rate is a direct function of the computational power available, and the relentless increase in model resolution, probabilistic ensembles and ever increasing complex physical models. The ambition is to continue this rate of progression, leading to Exascale numerical weather prediction [18].

To mitigate these I/O challenges, the FDB was redesigned to support storing its fields and its indexing meta-data to different storage back-ends. To continue supporting the current functionality we implemented a POSIX file-system back-end. In addition, to make use of the byte addressable non-volatile memory we implemented a back-end using PMDK. However, because each NVDIMM is only accessible from its compute node, the FDB was also further extended with a remote distributed front-end, based on a linearly scalable Rendezvous distributed hash table [35], which handles the dispatching of requests over the fabric between the nodes holding the storage and thus unifies them under a single FDB object store. All this work is explained in full detail in [34] and [33]. In section 5.2 we will demonstrate how this new FDB software (version 5), designed for the new DCPMM hardware (using App Direct), tackles the I/O bottleneck between the weather forecast data producers and data consumers by minimizing contention.

3.3 STREAM benchmark

The STREAM benchmark [30] is a synthetic application designed to measure the achievable memory bandwidth of a given processor or compute node. To evaluate memory bandwidth it measures the time taken for each of four common mathematical operations (copy,

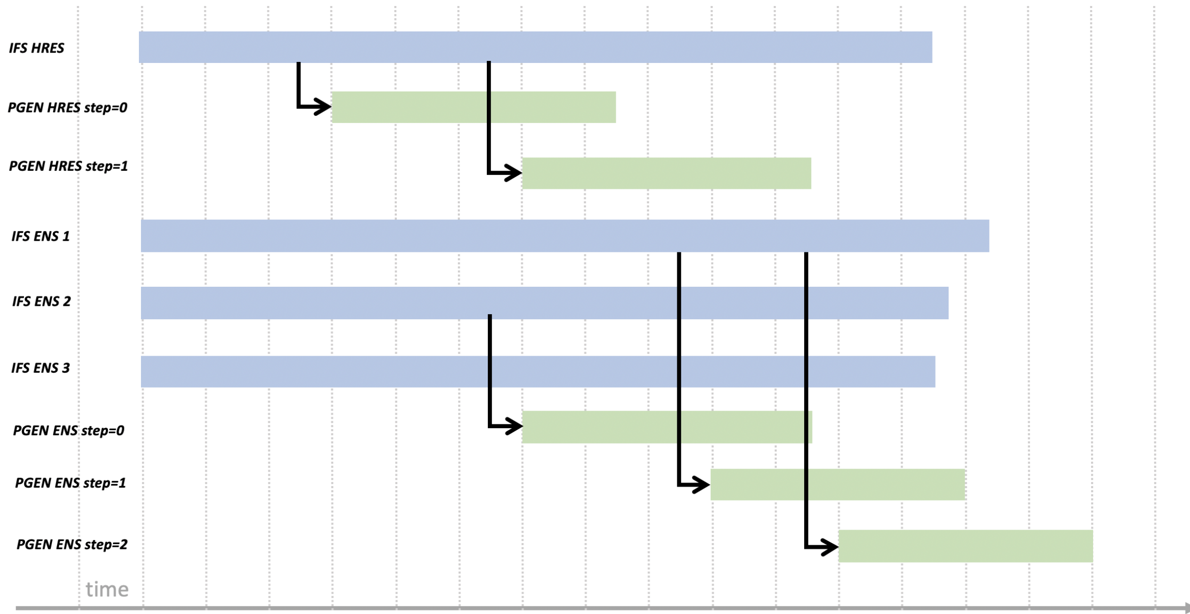

Figure 1: Simplified view of workflow dependencies between IFS HRES and ENS data producing jobs (blue) and the PGEN data consumer jobs (green). When a model output step is completed an event triggers the start up of PGEN (black arrows). The event manager interaction isn't shown. In the example, the PGEN jobs for ENS output are triggered from the straggler job, which initially is ENS 2 but then becomes ENS 1. Note that final PGEN jobs may overrun the model, as shown by the bottom PGEN ENS job. For simplicity we have omitted most steps and shown only 3 out of 51 ensembles.

scale, norm and triad) using up to three data arrays (a, b, c) and a single scalar variable (α). Operations are performed as a loop over elements in the arrays, with each array of equal size, preferably four times as big as the last level of cache in the hardware the benchmark is being run on.

For this paper we have used the standard STREAM benchmark to evaluate DRAM and DCPMM Memory mode performance as well as creating a new version of the STREAM benchmark that directly targets App Direct mode (more details in Section 5.3). This allows us to explore the achievable performance of DCPMM, and the overheads the different modes or ways of exploiting the memory may impose.

4 EXPERIMENTAL SETUP

In this section, we first give an overview of the system that was used for testing and then describe the tools used to support the analysis of DCPMM performance.

4.1 Prototype test system

The prototype test system consists of 34 1U dual-socket Intel Xeon (Cascade Lake generation) compute nodes that are built around a new motherboard designed specifically for the DCPMM memory which uses Intel's 3D XPoint technology. The compute nodes are supported by two login nodes, a boot node, a service node and two disk storage nodes. All nodes use the same motherboard and processor type, but vary in the amount of memory and number of processors; all the compute nodes have the same configuration. Each processor is an Intel Xeon 8260M with 24 cores (with a maximum of 48 hardware threads) and a base frequency of 2.4GHz. Hyperthreading is enabled on the system. Each compute node consists of two processors and 192GB of DDR4 RAM (12x16GB DIMMs) accompanied by 3,072GB of DCPMM memory (12x256GB DIMMs). The total DCPMM capacity of the system is 102TB, in addition to 6.5TB of DDR4 DRAM. The Intel Omni-Path high-speed interconnect, with 100Gbps port speed, was used to connect the compute nodes in a redundantly connected fabric providing 200Gbps per direction to other nodes. Each Omni-Path switch supports a maximum switching capacity of 9.6Tbps. An Omni-Path to Infiniband Gateway node has also been developed but was not used in this work. The disk storage nodes provide access to a 270TB Lustre filesystem. The system software consists of Linux CentOS 7.5, SLURM for job scheduling, Intel's *ipmctl* tool to manage the DCPMM modules (see Section 2.4), the *ndctl* [9] software to manage namespaces and the Persistent Memory Development Kit (PMDK) set of libraries and tools. The Intel 19 compiler suite, MPI and MKL libraries are also available on the prototype and were used to build the applications for this study.

4.2 Performance tools

For our performance evaluation we make use of the performance tools Score-P [26] and Vampir [25]. These two tools support the recording and processing of application-oriented performance metrics like computational intensity or communication overhead in the context of software modules, functions, or loops. At the same time,

system performance metrics such as memory, energy, or scheduling demands can be recorded accordingly to distinguish between internal and external causes. The tools workflow is organized as follows: Score-P is attached to an application and records selected performance metrics; Vampir then translates the recorded data to performance profiles and timelines for interactive graphical inspection and deduction.

The focus of this paper is primarily on studying memory and I/O performance, as well as energy consumption, effects resulting from the use of DCPMM. We configure the data pre-selection and filtering capabilities of Score-P to record representative function invocations in combination with energy (RAPL counters), I/O, memory allocation (rusage and malloc/free metrics), and memory access counters (PAPI counters). Combining this data with low overhead application sampling enables us to understand performance effects as they happen. Overall, we achieve low application perturbation with an overall measurement overhead less then two percent. We achieve this low overhead by measuring at an average rate of 4 kHz per thread. For example, for the CASTEP performance analysis (see Section 3.1), timing data was collected for 307 dominant functions out of 641.

4.3 Measuring energy and power consumption using system tools

The SLURM resource manager provides the *sacct* [12] feature, which keeps track of accounting data for all jobs inside a database. One of the data points that is collected is "ConsumedEnergy" - SLURM uses RAPL counters to get the energy consumed by the CPU and the memory, broken down by the node that made up a job. A second system-level method, which can also report the total power consumption of an node rather than only CPU and memory, uses the Intelligent Platform Management Interface (IPMI [1]) through Fujitsu's integrated Remote Management Controller (iRMC), taking a power reading every minute.

5 PERFORMANCE EVALUATION

In the subsequent sections we present results for the two applications, and STREAM, comparing their performance on the system with and without using DCPMM. In App Direct mode, DCPMM is not used unless it is explicitly exploited, and as such running an application without support for DCPMM means the system presents itself like a standard DRAM-only platform. This allows for a clean evaluation of the impact of DCPMM on the applications.

5.1 CASTEP

In the paragraphs below, we analyse the impact of expanding the usable memory capacity available to CASTEP through DCPMM. We are using the performance analysis tools on the small TiN test case only. The small TiN case fully fits into DRAM or DCPMM on a single identical node and it therefore allows us to study the application's pure memory performance without any performance effects resulting from the system interconnect or the memory to CPU mapping, which cannot be avoided for the DNA input data set. Table 3 compares key performance characteristics, such as runtime, Cycles per Instruction (CPI), stalls and loads from DCPMM.

5.1.1 *TiN - DRAM only baseline.*

As the TiN test case is small we limit our experiments to be within a single node. For the single socket experiments we pin the MPI processes to the cores on socket 0 using the `I_MPI_PIN_PROCESSOR_LIST` environment variable. The baseline results we observe are 435.57s using socket 0 only, and 193.63s using the full node. Figure 2 shows profile information over time of the run on socket 0 with DRAM only (white background) to be used as reference. The most dominant function is hamiltonian_diogonalise_ks() (highlighted in green), which consumes 67% of the total run time and varies between four and six seconds per call. As expected, there are no loads from DCPMM in this setup. The memory high watermark for the TiN test case is 18.658GB, well within DRAM capacity, which is important for our experiment in Memory mode below.

5.1.2 *TiN - App Direct mode with libvmmalloc.*

The TiN test case is small enough to easily fit into DRAM, however as mentioned earlier we use this example to demonstrate the functionality and the performance implications of using NVRAM in different configurations. As described in Section 2.3, libvmmalloc allows applications to use NVRAM without any implementation changes, and without the need to reboot compute nodes into Memory mode. It simply requires setting the size of the non-volatile memory pool (in bytes and per MPI process) and the location of the pool using the environment variables `VMMALLOC_POOL_SIZE` and `VMMALLOC_POOL_DIR`. The path to the non-volatile memory pool must point to a directory that is created in either `/mnt/pmem_fsdax0`, the DCPMMs located next to socket 0, or `/mnt/pmem_fsdax1`, located next to socket 1[1]. Once the environment variables have been set, the libvmmalloc library can be preloaded using `LD_PRELOAD` and the application can be launched as normal. We test three different scenarios:

(1) Executing the application on socket 0, we place the data onto the DCPMM next to socket 0;
(2) Keeping the application on socket 0, we now place the data onto the "remote" DCPMM next to socket 1 to quantify the effect of bad data locality;
(3) Using the full node and distributing the data to the local DCPMM according to the MPI process ranks (i.e. ranks below 24 will use the DCPMM on socket 0, ranks over 24 use the DCPMM on socket 1).

Tables 3 and 4 show the performance achieved when using libvmmalloc and compare them with the baseline. For the single socket example, it can be seen that the CPI and stall cycles increase in line with the runtime. Please note that DRAM is not used for caching in this mode of operation. When ensuring good data locality, the performance difference between using DRAM only or using a combination of DRAM and NVRAM is small, on the order of 10%. However, when deliberately forcing poor data locality by placing the data away from the processing cores onto remote DCPMM, the performance (as expected) drops. The impact of the non-uniform memory access effect between sockets is around 30%, which is also reflected in the reported stall cycles.

It is worth noting that the amount of data that is allocated dynamically in this example is very small (1.011GB) - Fortran applications often do most of their memory allocation statically, and therefore

[1] /mnt/pmem_fsdax0 and /mnt/pmem_fsdax1 are persistent memory namespaces that support DAX operations with a block-device based file-system.

using libvmmalloc means only a very small fraction of the data will end up being allocated on the DCPMM.

5.1.3 TiN - Memory mode. In order to use CASTEP in Memory mode, the compute nodes have to be rebooted - however no changes need to be made to the application and it can be run as is, i.e. not even a recompilation is required. In Memory mode, all the data is placed in DCPMM by default and DRAM becomes the last level cache. The memory controller ensures that the data is placed in the DRAM cache if required. Comparing the performance of the TiN test case in Memory mode vs App Direct mode with libvmmalloc on a full node, Memory mode exhibits better performance despite only a small amount of data being allocated to DCPMM in App Direct mode. One of the reasons is most likely the lack of data caching when using libvmmalloc: in Memory mode, DRAM is the cache, and after the initial allocation on NVRAM the data will therefore primarily be written to or read from DRAM; this caching functionality however does not exist when using libvmmalloc and accesses to the data in DCPMM come at a greater cost because of increased memory latency. This is supported by the DCPMM load instruction values in Table 3: the number is roughly halved for Memory mode, which means that the remainder of the data accesses is from DRAM (i.e. the cache). Having said that, it is surprising to detect that, running in Memory mode, CASTEP keeps accessing NVRAM throughout the entire application run at a small but relevant rate as depicted in the red graph at the bottom of Figure 2. One might expect that an application that fully fits into DRAM cache to asymptotically perform like our reference run in DRAM (white background), however the two performance graphs (green and red) at the bottom of Figure 2 tell us otherwise. We observe a performance degradation of 8% in all calls to hamiltonian_diagonalise_ks(), which is in line with the DCPMM loads observable in Memory mode throughout the entire run with peaks at the beginning and towards the end of the computation.

5.1.4 DNA - DRAM only. As DCPMM is not visible to the application in App Direct mode, only 192GB of DRAM can be allocated by default. The "Runtime" column in Table 1 shows the baseline performance that can be observed from the application when using the DNA test case. We only measure the first 3 iterations of the SCF loop as this gives us sufficient information to analyse the application from a performance analysis point of view; a fully convergent run requires more than 30 iterations. The best performance is achieved on 20 nodes; it is possible to run on only 18 nodes, however the runtime does increase significantly. For our further analysis of the application running in this mode, we focus on the 20 node performance as the baseline.

5.1.5 DNA - Memory mode. We executed the DNA test case on 4 nodes in Memory mode. From a memory capacity point of view, the test case would fit into the NVRAM of only 1 node (see Table 5), however the runtime becomes prohibitively large for experiments that use fewer than 4 nodes. We therefore decided to allocate more nodes than strictly necessary and test a range of different process and thread configurations per node: 36 MPI processes; 24 MPI processes with 2 OpenMP threads per process; 48 MPI processes; and finally 48 MPI processes with 2 OpenMP threads per process, ensuring that the MPI processes are placed on the physical cores, while the OpenMP threads are put on the logical cores.

Compared to the baseline performances, a DNA simulation running in Memory mode on 4 nodes using 24 MPI processes with 2 OpenMP threads is 3.75x slower than 20 nodes of its equivalent configuration in App Direct mode, and 3x times slower when using 36 MPI per node, while in both cases using 5x fewer nodes. Using all 48 cores per node results in the marginally better runtime. A sensitivity analysis, where we reduced the number of MPI processes used per socket by 1 at a time, showed that even better runtimes can be achieved in Memory mode. We found the optimum to be 22 cores per socket, or 44 cores per node, which results in a runtime of 15,057.01s (6% faster than using 48 MPI processes per node), after which point the trend reverses. This implies that CASTEP benefits from the additional memory bandwidth it gains for a short while, however this gain is fairly quickly negated by the reduction in processes available for computation. Table 6 details the performance results for the different configurations. As can be seen from these results, for a very large test case Memory mode delivers better performance per node than App Direct mode.

5.1.6 DNA - App Direct mode with libvmmalloc. Finally, we also tested the DNA simulation using libvmmalloc. As the volume of data that is dynamically allocated, and thus able to be intercepted by libvmmalloc, is negligible compared to the total, the impact on reducing the DRAM resource demands is insignificant. The usage case is therefore such that no gains are expected, however we add it here for completeness. The impact on the runtime is noticeable (in particular so when using more MPI processes per node) because of the lack of NVRAM data caching already described in Section 5.1.2 and 5.1.3.

5.1.7 Energy and power consumption. In terms of their utilisation of compute resources (i.e. CPU-hours) being used, Memory mode is more efficient than App Direct mode - a 5x difference in the number of nodes used results in an approximately 3.5x difference in runtime. According to the energy and power consumption of the simulation as reported by SLURM's *sacct*, Memory mode can draw more or less power than App Direct mode, depending on the configuration (see Table 6). In terms of energy usage however, Memory mode is more efficient because of its relatively better performance, as outlined earlier. In order to better understand the power usage of the nodes, we measured the power consumption of the memory (both DRAM and DCPMM) while running STREAM using 48 OpenMP threads, this time using the *perf* [2] tool to measure power:

```
perf stat -a -e power/energy-ram/ -I 1000
```

The total idle power draw for both the DRAM and DCPMM is ~23W per node (see Table 7). Because DRAM and DCPMM share memory controllers, the values returned by the *perf* tool (or indeed sacct or RAPL) do not distinguish between the two. In order to measure the difference (i.e. a node with 12 DRAM and 12 DCPMM DIMMs versus a node with 12 DRAM DIMMs only) we physically removed the DCPMM from a node and measured the idle power. This measurement states that the 256GB DCPMM idle power is ~12W per node, with DRAM accounting for the remaining ~11W.

For DRAM, the difference between read and write power consumption is very small, with write operations typically drawing

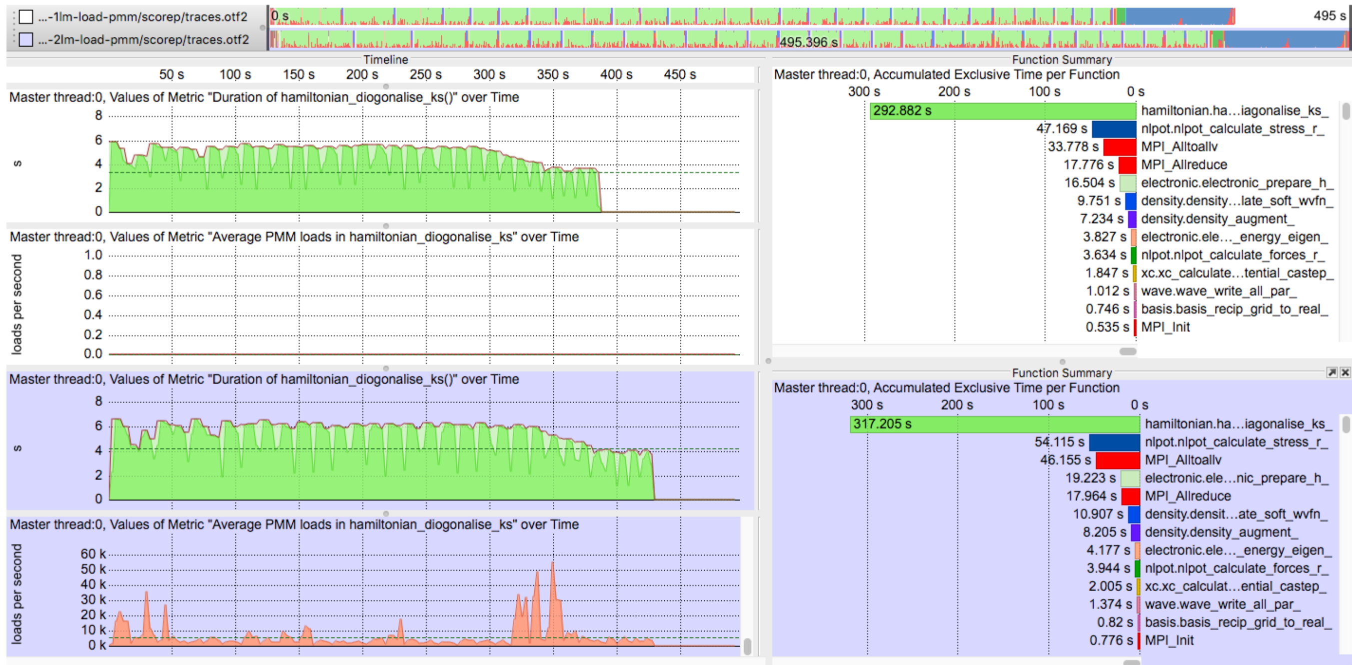

Figure 2: Timeline of DCPMM loads (red graph) during the execution of the most expensive routine (green graph) in the TiN test case. The top timelines (white background) show the application running in App Direct mode, with no loads from DCPMM registered. The bottom timelines (purple background) show the same application running in Memory mode, and here loads from DCPMM have been registered throughout the entire run although CASTEP and its data fully fit into DRAM cache.

Table 3: TiN test case, single socket (24 MPI processes) - comparing App Direct (DRAM only) vs App Direct with libvmmalloc (both socket local and socket remote DCPMM) vs Memory mode.

Mode	Runtime (s)	Slowdown	CPI	Stall cycles	Load instructions from DCPMM
App Direct (DRAM only)	442.30	-	1.17	1.62E+13	0
App Direct + libvmmalloc local	468.89	1.06x	1.25	1.84E+13	6.87E+08
App Direct + libvmmalloc remote	599.70	1.36x	1.57	2.45E+13	5.66E+08
Memory mode	484.90	1.10x	1.25	1.87E+13	3.81E+08

Table 4: TiN test case, 1 node (48 MPI). App Direct (DRAM only) vs App Direct with libvmmalloc vs Memory mode.

Mode	Runtime (s)	Slowdown
App Direct (DRAM only)	193.63	-
App Direct + libvmmalloc local	217.27	1.12x
Memory mode	204.79	1.06x

less power than reads. On DCPMM however the power consumption of writes can be up to 20W peak, whereas reads are closer to idle power. In Memory mode, DRAM acts as a write-back cache and thus covers writes to the DCPMM; as writes to DRAM are less power hungry than writes to DCPMM, and with reads being very power efficient, this accounts for the difference in the overall power draw.

Table 5: Measured memory high watermark ("MaxRSS") for the DNA test case, as reported by SLURM's sacct tool.

Number of nodes	MPI procs	Average mem per node (GBs)	Total mem (TBs)
1	48 (48 MPI per node)	1,735.40	1.69
2	96 (48 MPI per node)	882.16	1.72
4	192 (48 MPI per node)	452.10	1.77
20	720 (36 MPI per node)	103.21	2.01

5.1.8 CASTEP summary. For a simulation such as the DNA test case with CASTEP, the large capacity of the DCPMM can be highly beneficial if used in Memory mode. Although there are no performance gains in real terms of time to solution, the benefits lie in the much more efficient use of the resources that are available. It is possible to run the simulation on a smaller number of nodes while using

Table 6: DNA test case - comparing the runtime, total energy consumption (CPUs and memory only) and power draw (CPU and memory per node) for App Direct vs App Direct with libvmmalloc vs Memory mode.

Mode	Nodes	MPI x OpenMP (per node)	Hyperthreading (Y/N)	Runtime (seconds)	Energy. (MJoules, total)	Power (Watts, per node)
App Direct (DRAM only)	20	36x1	N	5,353.51	47.39	442
App Direct (DRAM only)	20	24x2	N	4,916.26	41.48	422
App Direct with libvmmalloc	20	36x1	N	5,953.80	51.52	433
App Direct with libvmmalloc	20	24x2	N	5,185.84	43.76	421
Memory mode	4	36x1	N	16,132.42	28.64	444
Memory mode	4	24x2	N	18,461.64	30.28	410
Memory mode	4	48x1	N	15,958.82	29.93	469
Memory mode	4	48x2	Y	26,130.83	43.11	412

Table 7: Memory power consumption per node - (1) DRAM only (no DCPMM), (2) DRAM plus DCPMM, (3) App Direct mode (DCPMM at idle), and (4) Memory mode. The size of the STREAM problem is given both as the total memory required and the array sizes.

	STREAM total memory (Array size)	Memory power per node
Idle - DRAM only		11W
Idle - with DCPMM		23W
App Direct mode	4.5GiB (200 million)	40W
Memory mode	1,117.6 GiB (50 billion)	24W

more cores per node, rather than wasting processing capabilities to satisfy memory capacity requirements. A secondary effect of using DCPMM in Memory mode is a smaller overall energy footprint per iteration, not because of reduced power consumption per node (depending on the configuration this can be higher), but because fewer nodes are involved in the simulation and the application displays greater parallel efficiency. For three of the four Memory mode configurations, the total energy consumption reported for the DNA simulation in Table 6 is much reduced compared to the best App Direct configuration. It is worth noting that this measurement only takes into account CPUs and memory and not any other energy consumers such as the network, which can be seen to be a fixed cost per node. From a system throughput point of view, it also means fewer idle resources and more densely utilised compute nodes. Using libvmmalloc does not prove beneficial in this particular case, for two reasons: firstly, CASTEP does not use malloc-style dynamic memory allocation sufficiently to reduce the demands on DRAM to be able to reduce the number of nodes that are needed for the simulation; and secondly, the lack of DRAM-caching, which in Memory mode is dealt with by the memory controller, means that repeated data accesses to DCPMM limit overall performance.

5.2 IFS

To assess the effectiveness of this new architecture in tackling the challenges for the IFS case, we have taken to analyze the architecture's effect on the FDB object-store that holds the model output data from which data consumers (PGEN) generate the forecast products. In Section 5.2.1 we analyze the raw performance of the object store whereas in Section 5.2.2 we analyze the effect on a simplified workflow using App Direct mode. Note that due to the very recent access to the system, we have not been able to trouble shoot and optimize the FDB backend based on PMDK, for which reason we have refrained from analyzing its impact in this paper. The backend discussed here relies therefore on a local (fsdax) file-system mounted on the DCPMM devices.

5.2.1 *FDB object-store performance.*

We ran a series of tests as summarized in Table 8, where we varied the number of object-store servers and the number of writing tasks. We have also varied the performance tests according to 2 characteristic field sizes (800KiB and 3.2MiB), to assess if the size of the objects matters in the object-store performance. This test case mimics the behaviour of a very large operational forecast with multiple ensembles, with burst behaviour following the time stepping procedure, including the instructions to flush data to persistent storage (which effectively act as barriers thus blocking progress). As a result the test is highly dependent on the application stack (and its bottlenecks) and the number of client tasks writing. We believe that driving this benchmark with application code is more realistic than simpler tests. and that the performance measured is much closer to what an operational application would expect to see from this architecture. The performance measure is an aggregate of taking all the data pushed into storage divided by the overall wall-clock time.

Table 8 shows two sets of results with the distributed object store being driven with varying sized workloads, with different numbers of client and server processes. The object store is configured to distribute the data across the server nodes according to a Rendezvous hash. In one case the data is stored to the DCPMM devices along with all the relevant indexing information. In the second case the bulk data itself is discarded at the final write operation, and only the indexing information is written. The routing of the data across the network and through the software infrastructure is identical in both cases. This allows us to attribute performance to either I/O, and network and/or software.

We see that the performance obtained is strongly correlated with both the number of servers and also the number of clients driving

the test by writing data. This suggests that there are significant bottlenecks in the application layer, restricting the ability of individual processes to push sufficient data to the servers.

The application issues `flush` requests to persist and consolidate the data indexes at regular intervals, coinciding with the completion of a forecast step. There is also the possibility that the network stack is playing a role, because the protocol is currently relying on IP over Omni-Path. Despite this, the FDB object store achieves a maximum of 40.17GiB/s of application level traffic over 32 server nodes.

When writing the data to the DCPMM devices is disabled (only indexes are updated), a small but consistent improvement in performance is observed, likely due to to the elimination of the (small) time taken to ensure that the data is persisted after each time step. Overall the performance of the two cases track each other. This demonstrates that the aggregated performance of this distributed object store on this platform is currently limited by the network connectivity and application software stack rather than the DCPMM devices. This is in significant contrast to all cases run using filesystems over HDD devices.

5.2.2 IFS workflow performance. Given the size of the prototype (34 nodes), for this numerical weather forecast test case we had to significantly reduce the size of the forecast workflow, which usually requires many thousands of nodes to compute in an hour's schedule. We have thus chosen to run 6 model ensembles (ENS) for half a day of forecast (12 forecast hours) and the accompanying PGEN post-processing for each of those steps. We configured this to use either the FDB storage backend on Lustre or distributed over 32 nodes, each using a single DCPMM mount point because our capacity requirements we did not require more. The Lustre file system uses 6 OSTs and we striped large data files over all 6. We then compared run times between simply running the models, with all model output enabled but no data consumers active (no PGEN), to runs where the 12 PGEN consumers gradually execute as the data becomes available.

From the results presented in Table 9 we observe that running over Lustre the difference between running with and without data consumers is a small slowdown from 1793s to 1928s (9%). This is somewhat smaller than the 11% as observed at scale (see Section 3.2). We believe that the system is not large enough to generate weather forecast data of sizes and intensity that would trigger magnitude of contention observed in operational systems.

Using a distributed FDB over NVRAM allows us to run the IFS jobs about 10% faster (17% with data consumers) than using the Lustre backend and notably to run the PGEN jobs concurrently to the model without noticeable impact on its performance (in fact the timing with PGEN was shorter, but under the run-time variability). It is important to underline that this eliminates the majority of the overall impact of I/O on the runtime of the workload. It is also worth noting that in this case the distributed FDB servers are located on the same nodes as the IFS computation and PGEN tasks are running, and that the additional overhead and jitter introduced by this collocation does not add a noticeable overhead to the runtime.

5.3 STREAM: Memory Bandwidth

Our App Direct mode benchmark uses largely the same code as the standard STREAM benchmark, but replaces the initial memory allocation with a call to `pmem_file_map` and some memory offset calculations. After these modifications the only requirement is to add the necessary "data persist" instructions to the benchmark to ensure data is fully stored on the DCPMM hardware before benchmarking ends.

We benchmarked a range of process and thread configurations and chose the one that gave the best performance, namely 48 MPI processes per node, each with 1 OpenMP thread. STREAM collects the bandwidth achieved by each process on a node to calculate the total node bandwidth. We collect the minimum, maximum, and median bandwidth achieved for each operation to enable evaluation of the variation of achieved bandwidth across a number of runs. We used an array size of 19MB, requiring 2.7GB of memory across all processes. The benchmark was set to repeat 10 times (with the timings collected for the first iteration not used to calculate the achieved bandwidth) and we ran each instance 5 times. The results for the Triad operation are presented in Table 10.

The theoretical peak memory bandwidth of the nodes is around 210GB/s (given the processors and memory used in the system). We can see from Table 10 that we are achieving a good fraction of the peak bandwidth even for the more involved Triad operation. There is less than 10% performance variation between the minimum and maximum achieved bandwidths for those benchmarks and configurations. The App Direct memory bandwidth is much lower than the DRAM bandwidth, but well within the advertised 3-10x performance difference between DRAM and DCPMM.

For this benchmark size, the Memory mode performance is very close to the DRAM performance and equally consistent. However this is a very small test case for Memory Mode and it violates the STREAM benchmark guideline that the array size should *be at least 4x the size of the sum of all the last-level caches used in the run.* Given that DRAM is used as the last-level cache for Memory mode, STREAM also needs to be tested with much larger array sizes for this configuration in order to get a complete performance picture. We re-ran the benchmark in Memory mode with an array size of 4GB per process (giving a total memory size for the benchmark of 768GB). For this configuration, STREAM achieved 12GB/s for the Triad operation with the same level of variation between minimum and maximum performance as seen before. This is a significant difference to the 146GB/s achieved in Memory mode for the smaller size, or even when accessing DCPMM directly (32GB/s), however it is to be expected: for sufficiently large problems, STREAM is designed to bypass the cache (which Memory mode relies on for performance) entirely. For this problem each memory access will result in a DRAM miss, which does not happen when using DCPMM directly, making this a pathologically bad case for Memory mode.

6 RELATED WORK

Although Intel Optane DCPMM was only launched as a product very recently, research into how to overcome memory limitations and how to accelerate I/O performance has been a topic of interest for several years. Almost a decade ago, Caulfield et al [16] discussed the potential impact of non-volatile memory technologies on both memory-centric and I/O intensive HPC applications, looking at NVRAM as an alternative to using large amounts of DRAM and or SSDs. Rudoff [32] also discusses the expected benefits of persistent

Table 8: Distributed FDB object store write performance, varying with the number of active servers. Each server is configured to either use both DCPMM mount points, or to discard all bulk data.

				DCPMM		Null (data discarded)	
Number of Servers	Client Tasks	Field Sizes [KiB]	Total Fields	Total Client Time [s]	Performance [GB/s]	Total Client Time [s]	Performance [GB/s]
4	32	803	768 k	2923	6.44	2264	8.31
4	64	803	1536 k	7405	10.17	6774	11.11
8	32	803	768 k	2111	8.92	1864	10.10
8	64	803	1536 k	5960	12.63	3893	19.34
8	128	803	3072 k	16800	17.93	11500	26.19
16	32	803	768 k	2100	8.96	2041	9.22
16	64	803	1536 k	4697	16.03	3799	19.82
16	128	803	3072 k	12998	23.17	8970	33.57
24	32	803	768 k	2562	7.35	2437	7.72
24	64	803	1536 k	4979	15.12	4858	15.50
24	128	803	3072 k	11562	26.05	9961	30.23
24	192	803	4608 k	20420	33.18	15611	43.40
32	128	803	3072 k	10919	27.58	4278	46.81
32	256	803	6144 k	29982	40.18	15589	51.39
4	16	3205	64 k	452	6.92	257	12.20
4	32	3205	128 k	1129	11.09	731	17.12
4	64	3205	256 k	4029	12.43	2341	21.39
8	16	3205	64 k	344	9.08	265	11.79
8	32	3205	128 k	1030	12.15	608	20.59
8	64	3205	256 k	2811	17.81	1855	27.00
8	96	3205	384 k	5059	22.27	2142	52.59
16	64	3205	256 k	2239	22.36	2861	17.50
16	96	3205	384 k	5289	21.30	2424	46.47
32	64	3205	256 k	2048	24.45	1735	28.86
32	128	3205	512 k	5276	37.96	4037	49.61

Table 9: Time to solution of the ENS model runs under the IFS workflow. Measurements with and without concurrent PGEN tasks and with two alternative storage backends Lustre and NVRAM. PGEN post processing of forecasts products for the 12 step hours generated. Each ENS run scheduled into 5 nodes and PGEN scheduled to 1 node each.

Workflow	FDB backend	Number Fields	Data set [GiB]	Runtime [s]
6 ENS	Lustre	257202	286.6	1793
6 ENS + 12 PGEN	Lustre	257202	286.6	1928
6 ENS	NVRAM	257202	286.6	1610
6 ENS + 12 PGEN	NVRAM	257202	286.6	1599

memory for HPC use cases, but also describes the challenges that must be overcome, in terms of system software support, in order to make the technology useful for as broad a range of applications as possible. In [24], Kim and Bahn propose an architecture designed to accelerate I/O by use of non-volatile memory as back-end storage partition rather than a cache device. Fernando et al [20] present an object store called Phoenix, specifically for persistent objects, designed to overcome the bandwidth differences between DRAM main memory and NVRAM - this is particularly relevant in a setting where (unlike with DCPMM) the memory spaces have to be managed separately. In [37], the authors describe the potential benefit (in terms of time to solution and efficient use of compute resources) of using byte-addressable storage class memory in particular with a

Table 10: STREAM Triad operation, using an array size of 19MB and 48 MPI processes each with 1 OpenMP thread.

Mode	Min BW (GB/s)	Med BW (GB/s)	Max BW (GB/s)
App Direct (DRAM only)	142	150	155
App Direct (DCPMM only)	32	32	32
Memory mode	144	146	147

view to HPC workflows, i.e. in scenarios where applications are run in a producer-consumer setting. The authors in [19] also address the topic of HPC workflows and introduce NVStream, a user-level data management system for producer-consumer applications that exploits the byte-addressable and persistent nature of NVRAM to enable streamed I/O for scientific workflows.

7 CONCLUSIONS

DCPMM is a new memory technology that, as a concept at the very least, has the potential to be highly disruptive to the status quo of the HPC landscape. The opportunities that are offered by the combination of capacity, byte-addressability, latency and persistence are vast and need to be explored very carefully. In its most easy to use form, Memory mode, DCPMM does not make use of the persistent nature of the 3D XPoint technology, but simply expands the available memory capacity of a system. This setup can deliver excellent performance for sufficiently large workloads, while not necessitating any changes to the application. As shown earlier, NUMA effects however do have the potential to impact performance considerably, and data locality must therefore be dealt with carefully through process pinning and local placement of data objects. Using DRAM effectively for caching is also a key requirement for achieving good performance in Memory mode. Unlike Memory mode, App Direct mode requires intervention, either by the system software (such as libvmmalloc) or by the application developers, to enable direct access to DCPMM (as described for IFS in the paper). To be able to benefit from libvmmalloc, three requirements should be met: the application must be limited by the DRAM capacity offered in App Direct mode; it should use a large proportion of dynamic memory allocations that can be intercepted by the library; and it must not be sensitive to the increased latency that DCPMM shows without the benefit of DRAM caching. In order to extract the most performance from DCPMM, applications should be modified to use the memory directly through load and store operations in App Direct mode. This scenario is the most invasive and complex to realise in terms of the changes that are required, however as demonstrated in this paper, this level of investment can pay off with considerable performance gains, such as effectively negating the impact of I/O on an application's overall performance.

ACKNOWLEDGMENTS

This work has been funded by the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 671951.

REFERENCES

[1] 2013. Intelligent Platform Management Interface Specification v2.0 rev. 1.1. Retrieved April 9, 2019 from https://www.intel.co.uk/content/dam/www/public/us/en/documents/product-briefs/ipmi-second-gen-interface-spec-v2-rev1-1.pdf

[2] 2015. perf: Linux profiling with performance counters. Retrieved May 22, 2019 from http://perf.wiki.kernel.org

[3] 2017. CASTEP DNA benchmark. Retrieved March 25, 2019 from http://www.castep.org/CASTEP/DNA

[4] 2017. CASTEP TiN benchmark. Retrieved April 5, 2019 from http://www.castep.org/CASTEP/TiN

[5] 2019. ECMWF Datasets. Retrieved April 10, 2019 from https://www.ecmwf.int/en/forecasts/datasets

[6] 2019. ECMWF website. Retrieved April 10, 2019 from https://www.ecmwf.int

[7] 2019. Intel Announces Broadest Product Portfolio for Moving, Storing and Processing Data. Retrieved April 8, 2019 from http://newsroom.intel.com/news-releases/intel-data-centric-launch

[8] 2019. IPMCTL. Retrieved April 4, 2019 from https://github.com/intel/ipmctl

[9] 2019. NDCTL - Utility library for managing the libnvdimm (non-volatile memory device) sub-system in the Linux kernel. Retrieved May 24, 2019 from https://github.com/pmem/ndctl

[10] 2019. NVM Programming Model (NPM), Version 1.2. Retrieved April 4, 2019 from https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf

[11] 2019. Persistent Memory Development Kit. Retrieved April 3, 2019 from http://pmem.io/pmdk/

[12] 2019. SLURM accounting. Retrieved April 5, 2019 from https://slurm.schedmd.com/sacct.html

[13] 2019. U.S. Department of Energy and Intel to deliver first exascale supercomputer. Retrieved April 8, 2019 from http://www.anl.gov/article/us-department-of-energy-and-intel-to-deliver-first-exascale-supercomputer

[14] Sadaf R. Alam, Hussein N. El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni. 2011. Parallel I/O and the Metadata Wall. In *Proceedings of the Sixth Workshop on Parallel Data Storage (PDSW '11)*. ACM, New York, NY, USA, 13–18. https://doi.org/10.1145/2159352.2159356

[15] Julian Borrill, Leonid Oliker, John Shalf, and Hongzhang Shan. 2007. Investigation of Leading HPC I/O Performance Using a Scientific-application Derived Benchmark. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*. ACM, New York, NY, USA, Article 10, 12 pages. https://doi.org/10.1145/1362622.1362636

[16] Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. 2010. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/SC.2010.56

[17] S. J. Clark, M. D. Segall, C. J. Pickard, P. J. Hasnip, M. J. Probert, K. Refson, and M.C. Payne. 2005. First principles methods using CASTEP. *Z. Kristall.* 220 (2005), 567–570.

[18] ECMWF. 2015. ECMWF Strategy 2016-2015, The strength of a common goal. http://www.ecmwf.int/sites/default/files/ECMWF_Strategy_2016-2025.pdf.

[19] Pradeep Fernando, Ada Gavrilovska, Sudarsun Kannan, and Greg Eisenhauer. 2018. NVStream: Accelerating HPC Workflows with NVRAM-based Transport for Streaming Objects. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18)*. ACM, New York, NY, USA, 231–242. https://doi.org/10.1145/3208040.3208061

[20] P. Fernando, S. Kannan, A. Gavrilovska, and K. Schwan. 2016. Phoenix: Memory Speed HPC I/O with NVM. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. 121–131. https://doi.org/10.1109/HiPC.2016.023

[21] Edward Higgins, Matt Probert, Phil Hasnip, Keith Refson, and Ian Bush. 2015. Hybrid OpenMP and MPI within the CASTEP code. *ARCHER eCSE Technical Report* (2015). http://www.archer.ac.uk/community/eCSE/eCSE01-017/eCSE01-017_Final_Report_technical.pdf

[22] P. Hohenberg and W. Kohn. 1964. Inhomogeneous electron gas. *Phys. Rev.* 136 (1964), B864–B871.

[23] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[24] J. Kim and H. Bahn. 2018. Accelerating Storage Performance with NVRAM by Considering Application's I/O Characteristics. In *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*. 383–389. https://doi.org/10.1109/BigComp.2018.00063

[25] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. 2008. The Vampir Performance Analysis Tool-Set. In *Tools for High Performance Computing*, Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 139–155.

[26] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–91.

[27] W. Kohn and L. J. Sham. 1965. Self-consistent equations including exchange and correlation effects. *Phys. Rev.* 140 (1965), A1133–A1138.

[28] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. 2012. On the role of burst buffers in leadership-class storage systems.

In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–11. https://doi.org/10.1109/MSST.2012.6232369

[29] S. Malardel, N. Wedi, W. Deconinck, M. Diamantakis, C. Kuehnlein, G. Mozdzynski, M. Hamrud, and P. Smolarkiewicz. [n. d.]. A new grid for the IFS. *ECMWF Newsletter* 146 ([n. d.]), 23–28.

[30] J. D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec 1995).

[31] M. C. Payne, M. P. Teter, D. C. Allan, T.A. Arias, and J. D. Joannopoulos. 1992. Iterative minimization techniques for ab initio total-energy calculations - molecular-dynamics and conjugate gradients. *Rev. Mod. Phys.* 64 (1992), 1045–1097.

[32] Andy Rudoff. 2017. Persistent Memory: The Value to HPC and the Challenges. In *Proceedings of the Workshop on Memory Centric Programming for HPC (MCHPC'17)*. ACM, New York, NY, USA, 7–10. https://doi.org/10.1145/3145617.3158213

[33] Simon D. Smart, Tiago Quintino, and Baudouin Raoult. 2017. A High-Performance Distributed Object-Store for Exascale Numerical Weather Prediction and Climate. In *Submitted to Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '19)*. ACM, New York, NY, USA.

[34] Simon D. Smart, Tiago Quintino, and Baudouin Raoult. 2017. A Scalable Object Store for Meteorological and Climate Data. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '17)*. ACM, New York, NY, USA, Article 13, 8 pages. https://doi.org/10.1145/3093172.3093238

[35] David Thaler and Chinya Ravishankar. 1996. A Name-Based Mapping Scheme for Rendezvous. http://www.eecs.umich.edu/techreports/cse/96/CSE-TR-316-96.pdf.

[36] Ananta Tiwari, Anthony Gamst, Michael A. Laurenzano, Martin Schulz, and Laura Carrington. 2014. Modeling the Impact of Reduced Memory Bandwidth on HPC Applications. In *Euro-Par 2014 Parallel Processing*, Fernando Silva, Inês Dutra, and Vítor Santos Costa (Eds.). Springer International Publishing, Cham, 63–74.

[37] Michele Weiland, Adrian Jackson, Nick Johnson, and Mark Parsons. 2018. Exploiting the Performance Benefits of Storage Class Memory for HPC and HPDA Workflows. *Supercomputing Frontiers and Innovations* 5, 1 (2018). http://superfri.org/superfri/article/view/164

[38] World Meteorological Organization. 2015. GRIB Format. http://www.wmo.int/pages/prog/www/WMOCodes/WMO306_vI2/Publications/2015editionUP2018/WMO306_vI2_en_ONLINE.pdf.