Profiling and Performance

Introduction

This chapter first discusses the general concepts for analyzing memory and storage performance and how to identify opportunities for using persistent memory for both high-performance persistent storage and high-capacity volatile memory. We then describe the tools and techniques that can help you optimize your code to achieve the best performance.

Performance analysis requires tools to collect specific data and metrics about application, system, and hardware performance. In this chapter, we describe how to collect this data using Intel VTune Profiler. Many other data collection options are available; the techniques we describe are relevant regardless of how the data is collected.

Performance Analysis Concepts

Most concepts for performance analysis of persistent memory are similar to those already established for performance analysis of shared memory programs or storage bottlenecks. This section outlines several important performance considerations you should understand to profile and optimize persistent memory performance and defines the terms and situations we use in this chapter.

Compute-Bound vs. Memory-Bound

Performance optimization largely involves identifying the current performance bottleneck and improving it. The performance of compute-bound workloads is generally limited by the number of instructions the CPU can process per cycle. For example, an application doing a large number of calculations on very compact data without many dependencies is usually *compute-bound*. This type of workload would run faster if the CPU were faster. Compute-bound applications usually have high CPU utilization, close to 100%.

In contrast, the performance of *memory-bound* workloads is generally limited by the memory subsystem and the long latencies of fetching data from caches and system memory. An example is an application that randomly accesses data from data structures in DRAM. In this case, adding more compute resources would not improve such an application. Adding persistent memory to improve performance is usually an option for memory-bound workloads as opposed to compute-bound workloads. Memory-bound workloads usually have lower CPU utilization than compute-bound workloads, exhibit CPU stalls due to memory transfers, and have high memory bandwidth.

Memory Latency vs. Memory Capacity

This concept is essential when discussing persistent memory. For this discussion, we assume that DRAM access latencies are lower than persistent memory and that the persistent memory capacity within the system is larger than DRAM. Workloads bound by memory capacity can benefit from adding persistent memory in a volatile mode, while workloads that are bound by memory latency are less likely to benefit.

Read vs. Write Performance

While each persistent memory technology is unique, it is important to understand that there is usually a difference in the performance of reads (loads) vs. writes (stores). Different media types exhibit varying degrees of asymmetric read-write performance characteristics, where reads are generally much faster than writes. Therefore, understanding the mix of loads and stores in an application workload is important for understanding and optimizing performance.

Memory Access Patterns

A memory access pattern is the pattern with which a system or application reads and writes to or from the memory. Memory hardware usually relies on temporal locality (accessing recently used data) and spatial locality (accessing contiguous memory addresses) for best performance. This is often achieved through some structure of fast internal caches and intelligent prefetchers. The access pattern and level of locality can drastically affect cache performance and can also have implications on parallelism and distributions of workloads within shared memory systems. Cache coherency can

also affect multiprocessor performance, which means that certain memory access patterns place a ceiling on parallelism. Many well-defined memory access patterns exist, including but not limited to sequential, strided, linear, and random.

It is much easier to measure, control, and optimize memory accesses on systems that run only one application. In the cloud and virtualized environments, applications within the guests can be running any type of application and workload, including web servers, databases, or an application server. This makes it much harder to ensure memory accesses are fully optimized for the hardware as the access patterns are essentially random.

I/O Storage Bound Workloads

A program is I/O bound if it would go faster if the I/O subsystem were faster. We are primarily interested in the block-based disk I/O subsystem here, but it could also include other subsystems such as the network. An I/O bound state is undesirable because it means that the CPU must stall its operation while waiting for data to be loaded or unloaded from main memory or storage. Depending on where the data is and the latency of the storage device, this can invoke a voluntary context switching of the current application thread with another. A voluntary context switch occurs when a thread blocks because it requires a resource that is not immediately available or takes a long time to respond. With faster computation speed being the primary goal of each successive computer generation, there is a strong imperative to avoid I/O bound states. Eliminating them can often yield a more economic improvement in performance than upgrading the CPU or memory.

Determining the Suitability of Workloads for Persistent Memory

Persistent memory technologies may not solve every workload performance problem. You should understand the workload and platform on which it is currently running when considering persistent memory. As a simple example, consider a compute-intensive workload that relies heavily on floating-point arithmetic. The performance of this application is likely limited by the floating-point unit in the CPU and not any part of the memory subsystem. In that case, adding persistent memory to the platform will likely have little impact on this application's performance. Now consider an application

that requires extensive reading and writing from disk. It is likely that the disk accesses are the bottleneck for this application and adding a faster storage solution, like persistent memory, could improve performance.

These are trivial examples, and applications will have widely different behaviors along this spectrum. Understanding what behaviors to look for and how to measure them is an important step to using persistent memory. This section presents the important characteristics to identify and determine if an application is a good fit for persistent memory. We look at applications that require in-memory persistence, applications that can use persistent memory in a volatile manner, and applications that can use both.

Volatile Use Cases

Chapter 10 described several libraries and use cases where applications can take advantage of the performance and capacity of persistent memory to store non-volatile data. For volatile use cases, persistent memory will act as an additional memory tier for the platform. It may be transparent to the application, such as using Memory Mode supported by Intel Optane DC persistent memory, or applications can make code changes to perform volatile memory allocations using libraries such as libmemkind. In both cases, memory-capacity bound workloads will benefit from adding persistent memory to the platform. Application performance can dramatically improve if its working dataset can fit into memory and avoid paging to disk.

Identifying Workloads That Are Memory-Capacity Bound

To determine if a workload is memory-capacity bound, you must determine the "memory footprint" of the application. The memory footprint is the high watermark of memory concurrently allocated during the application's life cycle. Since physical memory is a finite resource, you should consider the fact that the operating system and other processes also consume memory. If the footprint of the operating system and all memory consumers on the system are approaching or exceeding the available DRAM capacity on the platform, you can assume that the application would benefit from additional memory because it cannot fit all its data in DRAM. Many tools and techniques can be used to determine memory footprint. VTune Profiler includes two different ways

to find this information: *Memory Consumption* analysis or *Platform Profiler* analysis. VTune Profiler is a free download for Linux and Windows, available from `https://software.intel.com/en-us/vtune`.

The Memory Consumption analysis within VTune Profiler tracks all memory allocations made by the application. Figure 15-1 shows a VTune Profiler bottom-up report representing memory consumption of the profiled application over time. The highest value on the y-axis in the Memory Consumption timeline indicates the application footprint is approximately 1GiB.
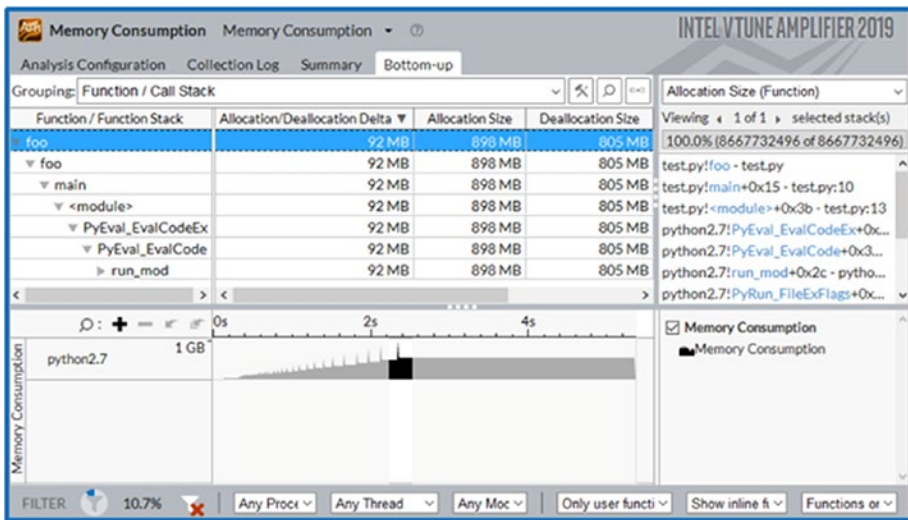

Figure 15-1. *The VTune Profiler bottom-up analysis showing memory consumption with time and the associated allocating call stacks*

The Memory Utilization graph in the Platform Profiler analysis shown in Figure 15-2 measures the memory footprint using operating system statistics and produces a timeline graph as a percentage of the total available memory.
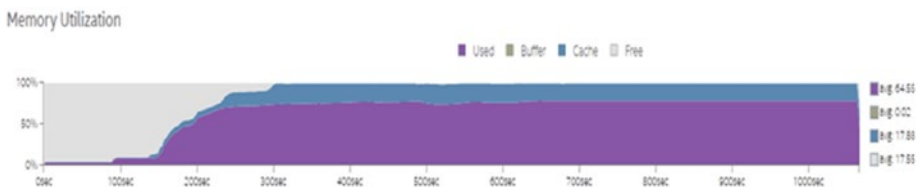

Figure 15-2. *The VTune Platform Profiler Memory Utilization graph as a percentage of total system memory*

The results in Figure 15-2 were taken from a different application than Figure 15-1. This graph shows very high memory consumption, which implies this workload would be a good candidate for adding more memory to the system. If your persistent memory hardware has variable modes, like the Memory and App Direct modes on Intel Optane DC persistent memory, you will need some more information to determine which mode to use first. The next important information is the hot working set size.

Identifying the Hot Working Set Size of a Workload

Persistent memory usually has different characteristics than DRAM; therefore, you should make intelligent decisions about where data will reside. We will assume that accessing data from persistent memory has higher latency than DRAM. Given the choice between accessing data in DRAM and persistent memory, we would always choose DRAM for performance. However, the premise of adding persistent memory in a volatile configuration assumes there is not enough DRAM to fit all the data. You need to understand how your workload accesses data to make choices about persistent memory configuration.

The working set size (WSS) is how much memory an application needs to keep working. For example, if an application has 50GiB of main memory allocated and page mapped, but it is only accessing 20MiB each second to perform its job, we can say that the working set size is 50GiB and the "hot" data is 20MiB. It is useful to know this for capacity planning and scalability analysis. The "hot working set" is the set of objects accessed frequently by an application, and the "hot working set size" is the total size of those objects allocated at any given time.

Determining the size of the working set and hot working set is not as straightforward as determining memory footprint. Most applications will have a wide range of objects with varying degrees of "hotness," and there will not be a clear line delineating which objects are hot and which are not. You must interpret this information and determine the hot working set size.

VTune Profiler has a Memory Access analysis feature that can help determine the hot and working set sizes of an application (select the "Analyze dynamic memory objects" option before data collection begins). Once enough data has been collected, VTune Profiler will process the data and produce a report. In the bottom-up view within the GUI, a grid lists each memory object that was allocated by the application.

Figure 15-3 shows the results of a Memory Access analysis of an application. It shows the memory size in parenthesis and the number of loads and stores that accessed it. The report does not include an indication of what was concurrently allocated.

Grouping: Memory Object / Function / Call Stack			
Memory Object / Function / Call Stack	Loads ▼	Stores	LLC Miss Count ≫
▷ matrix.c:116 (128 MB)	161,578,247,202	0	0
▷ matrix.c:121 (128 MB)	15,043,951,305	0	0
▷ matrix.c:126 (128 MB)	2,196,965,907	70,028,400,789	2,250,135
▷ [vmlinux]	117,903,537	65,701,971	0

Figure 15-3. *Objects accessed by the application during a Memory Access analysis data collection*

The report identifies the objects with the most accesses (loads and stores). The sum of the sizes of these objects is the working set size – the values are in parentheses. You decide where to draw the line for what is and is not part of the hot working set.

Depending on the workload, there may not be an easy way to determine the hot working set size, other than developer knowledge of the application. Having a rough estimate is important for deciding whether to start with Memory Mode or App Direct mode.

Use Cases Requiring Persistence

Use cases that take advantage of persistent memory for persistence, as opposed to the volatile use cases previously described, are generally replacing slower storage devices with persistent memory. Determining the suitability of a workload for this use case is straightforward. If application performance is limited by storage accesses (disks, SSDs, etc.), then using a faster storage solution like persistent memory could help. There are several ways to identify storage bottlenecks in an application. Open source tools like dstat or iostat give a high-level overview of disk activity, and tools such as VTune Profiler provide a more detailed analysis.
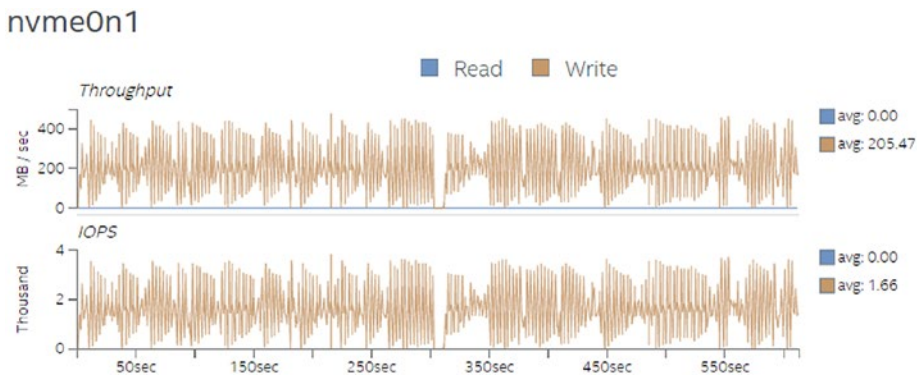
Figure 15-4.   *Disk throughput and IOPS graphs from VTune Profiler's Platform Profiler*

Figure 15-4 shows throughput and IOPS numbers of an NVMe drive collected using Platform Profiler. This example uses a non-volatile disk for extensive storage, as indicated by the throughput and IOPS graphs. Applications like this may benefit from faster storage like persistent memory. Another important metric to identify storage bottlenecks is I/O Wait time. The Platform Profiler analysis can also provide this metric and display how it is affecting CPU Utilization over time, as seen in Figure 15-5.
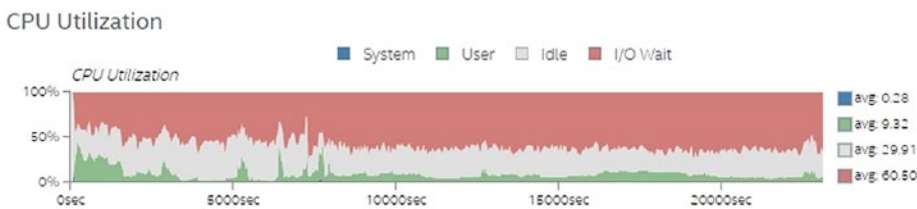

Figure 15-5.   *I/O Wait time from VTune Profiler's Platform Profiler*

Performance Analysis of Workloads Using Persistent Memory

Optimizing a workload on a system with persistent memory follows the principles similar to those of optimizing a workload performance on a DRAM-only system. The additional factors to keep in mind are:

- The writes to persistent memory may impact performance more than the reads.

- Applications can allocate objects on DRAM or persistent memory. If done indiscriminately, this can negatively impact performance.

- In Memory Mode (specific to Intel Optane DC persistent memory), users have the option of varying the near-memory cache size (DRAM size) to improve workload performance.

Keeping these additional factors in mind, the approach to workload performance optimization will follow the same process of characterizing the workload, choosing the correct memory configuration, and optimizing the code for maximum performance.

Characterizing the Workload

The performance of a workload on a persistent memory system depends on a combination of the workload characteristics and the underlying hardware. The key metrics to understand the workload characteristics are:

- Persistent memory bandwidth

- Persistent memory read/write ratio

- Paging to and from traditional storage

- Working set size and footprint of the workload

- Nonuniform Memory Architecture (NUMA) characteristics

- Near-memory cache behavior in Memory Mode (specific to Intel Optane DC persistent memory)

Memory Bandwidth and Latency

Persistent memory, like DRAM, has limited bandwidth. When it becomes saturated, it can quickly bottleneck application performance. Bandwidth limits will vary depending on the platform. You can calculate the peak bandwidth of your platform using hardware specifications or a memory benchmarking application.

The Intel Memory Latency Checker (Intel MLC) is a free tool for Linux and Windows available from https://software.intel.com/en-us/articles/intelr-memory-latency-checker. Intel MLC can be used to measure bandwidth and latency of DRAM and persistent memory using a variety of tests:

- Measure idle memory latencies between each CPU socket

- Measure peak memory bandwidth requests with varying ratios of reads and writes

- Measure latencies at different bandwidth points

- Measure latencies for requests addressed to a specific memory controller from a specific core

- Measure cache latencies

- Measure b/w from a subset of the cores/sockets

- Measure b/w for different read/write ratios

- Measure latencies for random and sequential address patterns

- Measure latencies for different stride sizes

- Measure cache-to-cache data transfer latencies

VTune Profiler has a built-in kernel to measure peak bandwidth on a system. Once you know the peak bandwidth of the platform, you can then measure the persistent memory bandwidth of your workload. This will reveal whether persistent memory bandwidth is a bottleneck. Figure 15-6 shows an example of persistent memory read and write bandwidth of an application.
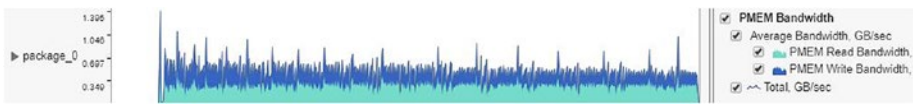

Figure 15-6. *Results from VTune Profiler persistent memory bandwidth measurements*

Persistent Memory Read-Write Ratio

As described in "Performance Analysis Concepts," the ratio of read and write traffic to the persistent memory plays a major role in the overall performance of a workload. If the ratio of persistent memory write bandwidth to read bandwidth is high, there is a good chance the persistent memory write latency is impacting performance. Using the Platform Profiler feature in VTune Profiler is one way to collect this information. Figure 15-7 shows the ratio of read traffic vs. all traffic to persistent memory. This number should be close to 1.0 for best performance.
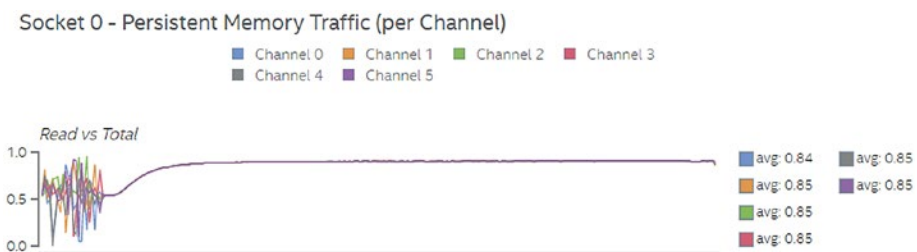

Figure 15-7. *Read traffic ratio from VTune Profiler's Platform Profiler analysis*

Working Set Size and Memory Footprint

As described in "Determining the Suitability of Workloads for Persistent Memory," the working set size and memory footprint of the application are important characteristics to understand once a workload is running on a system with persistent memory. Metrics can be collected using the tools and processes previously described.

Non-Uniform Memory Architecture (NUMA) Behavior

Multi-socket platforms typically have persistent memory attached to each socket. Accesses to persistent memory from a thread on one socket to another will incur longer latencies. These "remote" accesses are some of the NUMA behaviors that can impact performance. Multiple metrics can be collected to determine how much NUMA activity is occurring in a workload. On Intel platforms, data moves between sockets through the socket interconnect called the QuickPath Interconnect (QPI) or Ultra Path Interconnect (UPI). High interconnect bandwidth may indicate NUMA-related performance issues. In addition to interconnect bandwidth, some hardware provides counters to track local and remote accesses to persistent memory.

Understanding the NUMA behavior of your workload is another important step in understanding performance optimization. Figure 15-8 shows UPI bandwidth collected with VTune Profiler.
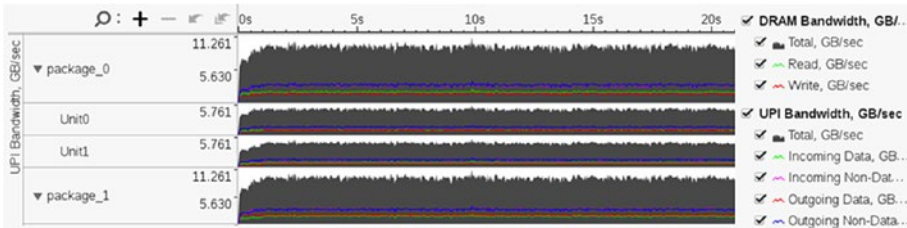

Figure 15-8. *UPI traffic ratio from VTune Profiler*

The Platform Profiler feature in VTune Profiler can collect metrics specific to persistent memory.

Tuning the Hardware

The memory configuration of a system is a significant factor in determining the system's performance. The workload performance depends on a combination of workload characteristics and the memory configuration. There is no single configuration that provides the best value for all workloads. These factors make it important to tune the hardware with respect to workload characteristics and get the maximum value out of the system.

Addressable Memory Capacity

The combined capacity of DRAM and persistent memory determines the total addressable memory available on the system. You should tune the size of persistent memory to accommodate the workload's footprint.

The capacity of DRAM available on the system should be large enough to accommodate the workload's hot working set size. A large amount of volatile traffic going to persistent memory while DRAM is fully utilized is a good indicator that the workload can benefit from additional DRAM size.

Bandwidth Requirements

The maximum available persistent memory bandwidth depends on the number of channels populated with a persistent memory module. A fully populated system works well for a workload with a high bandwidth requirement. Partially populated systems can be used for workloads that are not as memory latency sensitive. Refer to the server documentation for population guidelines.

BIOS Options

With the introduction of persistent memory into server platforms, many features and options have been added to the BIOS that provide additional tuning capabilities. The options and features available within the BIOS vary for each server vendor and persistent memory product. Refer to the server BIOS documentation for all the options available; most share common options, including:

- Ability to change power levels to balance power consumption and performance. More power delivered to persistent memory can increase performance

- Enable or disable persistent memory–specific features

- Tune latency or bandwidth characteristics of persistent memory

Optimizing the Software for Persistent Memory

There are many ways to optimize applications to use persistent memory efficiently and effectively. Each application will benefit in different ways and will need to have code modified accordingly. This section describes some of the optimization methods.

Guided Data Placement

Guided data placement is the most common avenue for optimizing volatile workloads on a persistent memory system. Application developers can choose to allocate a data structure or object in DRAM or persistent memory. It is important to choose accurately because allocating incorrectly could impact application performance. This allocation is usually handled via specific APIs, for example, the allocation APIs available in the Persistent Memory Development Kit (PMDK) and memkind library.

Depending on your familiarity with the code and how it works with production workloads, knowing which data structures and objects to store in the different memory/ storage tiers may be simple. Should those data structures and objects be volatile or persisted? To help with searching for potential candidates, tools such as VTune Profiler can identify objects with the most last-level cache (LLC) misses. The intent is to identify what data structures and objects the application uses most frequently and ensure they are placed in the fastest media appropriate to their access patterns. For example, an object that is written once but read many times is best placed in DRAM. An object that is updated frequently that needs to be persisted should probably be moved to persistent memory rather than traditional storage devices.

You must also be mindful of memory-capacity constraints. Tools such as VTune Profiler can help determine approximately how many hot objects will fit into the available DRAM. For the remaining objects that have fewer LLC misses or that are too large to allocate from DRAM, you can put them in persistent memory. These steps will ensure that your most accessed objects have the fastest path to the CPU (allocated in DRAM), while the infrequently accessed objects will take advantage of the additional persistent memory (as opposed to sitting out on a much slower storage devices).

Another consideration for optimizations is the load/store ratio for object accesses. If your persistent memory hardware characteristics are such that load/read operations are much faster than stores/writes, this should be taken into account. Objects with a high load/store ratio should benefit from living in persistent memory.

There is no hard rule for what constitutes a frequent vs. infrequently accessed object. Although behaviors are application dependent, these guidelines give a starting point for choosing how to allocate objects in persistent memory. After completing this process, start profiling and tuning the application to further improve the performance with persistent memory.

Memory Access Optimization

The common techniques for optimizing cache performance on DRAM-only platforms also apply to persistent memory platforms. Concepts like cache-miss penalties and spatial/temporal data locality are important for performance. Many tools can collect performance data for caches and memory. VTune Profiler has predefined metrics for each level of the memory hierarchy, including Intel Optane DC persistent memory shown in Figure 15-9.
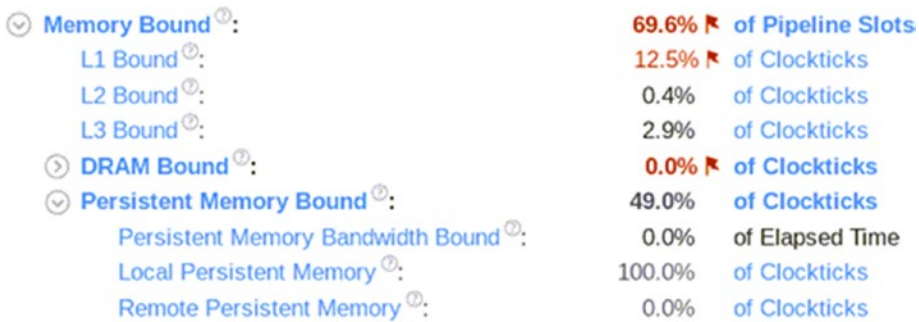
Figure 15-9. VTune Profiler memory analysis of a workload showing a breakdown of CPU cache, DRAM, and persistent memory accesses

These performance metrics help to determine if memory is the bottleneck in your application, and if so, which level of the memory hierarchy is the most impactful. Many tools can pinpoint source code locations and memory objects responsible for the bottleneck. If persistent memory is the bottleneck, review the "Guided Data Placement" section to ensure that persistent memory is being used efficiently. Performance optimizations like cache blocking, software prefetching, and improved memory access patterns may also help relieve bottlenecks in the memory hierarchy. You must determine how to refactor the software to more efficiently use memory, and metrics like these can point you in the right direction.

NUMA Optimizations

NUMA-related performance issues were described in the "Characterizing the Workload" section; we discuss NUMA in more detail in Chapter 19. If you identify performance issues related to NUMA memory accesses, two things should be considered: data allocation vs. first access, and thread migration.

Data Allocation vs. First Access

Data allocation is the process of allocating or reserving some amount of virtual address space for an object. The virtual address space for a process is the set of virtual memory addresses that it can use. The address space for each process is private and cannot be accessed by other processes unless it is shared. A virtual address does not represent the actual physical location of an object in memory. Instead, the system maintains a multilayered page table, which is an internal data structure used to translate virtual addresses into their corresponding physical addresses. Each time an application thread

references an address, the system translates the virtual address to a physical address. The physical address points to memory physically connected to a CPU. Chapter 19 describes exactly how this operation works and shows why high-capacity memory systems can benefit from using large or huge pages provided by the operating system.

A common practice in software is to have most of the data allocations done when the application starts. Operating systems try to allocate memory associated with the CPU on which the thread executes. The operating system scheduler then tries to always schedule the thread on a CPU that it last ran in the hopes that the data still remains in one of the CPU caches. On a multi-socket system, this may result in all the objects being allocated in the memory of a single socket, which can create NUMA performance issues. Accessing data on a remote CPU incurs a latency performance penalty.

Some applications delay reserving memory until the data is accessed for the first time. This can alleviate some NUMA issues. It is important to understand how your workload allocates data to understand the NUMA performance.

Thread Migration

Thread migration, which is the movement of software threads across sockets by the operating system scheduler, is the most common cause of NUMA issues. Once objects are allocated in memory, accessing them from another physical CPU from which they were originally allocated incurs a latency penalty. Even though you may allocate your data on a socket where the accessing thread is currently running, unless you have specific affinity bindings or other safeguards, the thread may move to any other core or socket in the future. You can track thread migration by identifying which cores threads are running on and which sockets those cores belong to. Figure 15-10 shows an example of this analysis from VTune Profiler.

Thread / Package / Core / Function / Call Stack	CPU Time ▼	Instructions Retired	Microarchitecture Usage	TID
▶ matrix.gcc (TID: 455334)	3.844s	1,255,500,000	9.6%	455334
▼ matrix.gcc (TID: 455336)	3.834s	1,215,000,000	11.8%	455336
▼ package_1	3.528s	1,053,000,000	9.8%	455336
▶ core_17	2.551s	729,000,000	7.1%	455336
▶ core_1	0.386s	121,500,000	16.9%	455336
▶ core_7	0.286s	81,000,000	28.0%	455336
▶ core_5	0.010s	0	0.0%	455336
▶ package_0	0.306s	162,000,000	30.1%	455336

Figure 15-10. *VTune Profiler identifying thread migration across cores and sockets (packages)*

Use this information to determine whether thread migration is correlated with NUMA accesses to remote DRAM or persistent memory.

Large and Huge Pages

The default memory page size in most operating systems is 4 kilobytes (KiB). Operating systems provide many different page sizes for different application workloads and requirements. In Linux, a *Large Page* is 2 megabytes (MiB), and a *Huge Page* is 1 gigabyte (GiB). The larger page sizes can be beneficial to workload performance on persistent memory in certain scenarios.

For applications with a large addressable memory requirement, the size of the page table being maintained by the operating system for virtual to physical address translation grows significantly larger in size. The translation lookaside buffer (TLB) is a small cache to make virtual-to-physical address translations faster. The efficiency of TLB goes down when the number of page entries increases in the page table. Chapter 19 describes this in more detail.

Persistent memory systems that are meant for applications with a large memory requirement will likely encounter the problem of large page tables and inefficient TLB usage. Using large page sizes in this scenario helps reduce the number of entries in the page table. The main trade-offs when using large page sizes is a higher overhead for each allocation and memory fragmentation. You must be aware of the application behavior before using large pages on persistent memory. An application doing frequent allocation/deallocation may not be a good fit for large page optimization. The memory fragmentation issue is somewhat abated by the large address space available on the persistent memory systems.

Summary

Profiling and performance optimization techniques for persistent memory systems are similar to those techniques used on systems without persistent memory. This chapter outlined some important concepts for understanding performance. It also provides guidance for characterizing an existing application without persistent memory and understanding whether it is suitable for persistent memory. Finally, it presents

important metrics for performance analysis and tuning of applications running on persistent memory platforms, including some examples of how to collect the data using the VTune Profiler tool.

Performance profiling and optimization are an iterative process that only ends when you determine that the investment required for the next improvement is too high for the benefit that will be returned. Use the concepts introduced in this chapter to understand how your workloads can benefit from persistent memory, and use some of the optimization techniques we discussed to tune for this type of platform.