Karlsruher Institut für Technologie

An Analysis of DMA Interference Using Synthetic Load from an NVMe Device

Bachelorarbeit
von

Lukas Werling

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	Dipl.-Inform. Marius Hillenbrand

Bearbeitungszeit: 19. Mai 2015 – 18. September 2015

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 18. September 2015

Abstract

With the general availability of PCI Express 3.0 slots in both consumer and server hardware today, more and more devices making use of the fast transfer speed are being released. These devices read and write directly into the main memory using Direct Memory Access (DMA).

Other works [2] have already determined that memory-intensive applications running in parallel on a multi-core platform may slow each other down due to the memory controller reordering requests. In this work, we analyze interference of high-bandwidth DMA operations with applications running in parallel on the CPU. To generate DMA load, we design and implement a load generator based on NVMe solid-state drives.

Our experiments show that high-bandwidth DMA transfers indeed have an influence on memory-intensive applications. Additionally, we characterize bandwidth and command rate of our NVMe device and analyze last-level cache usage with performance counters.

Contents

Chapter 1

Introduction

All applications running on modern computers are accessing the main memory more or less extensively. With multicore CPUs, multiple programs will access the memory at the same time. Due to the organization of DRAM, some requests can be fulfilled faster than others. Making use of this fact, the memory controller reorders memory requests, improving overall memory throughput. However, this may reduce performance of a process whose memory requests are fulfilled less often.

This effect is similar to issues with preemptive CPU scheduling: A “fair” usage share of CPU time for every process is often desired. Priorities can usually be given to processes by different criteria, for example to make sure that an interactive application runs more often than a background process.

Managing memory bandwidth on the application level is already possible with MemGuard [2]. This method relies on an estimate of the worst-case memory bandwidth available, distributing the bandwidth to processes based on hardware performance counters built into newer CPUs. These counters measure the bandwidth each CPU core is using.

The maximum memory bandwidths of current CPUs vary a lot. Consumer CPUs usually have a maximum bandwidth of about 25 GB/s [4], while upcoming high-end server CPUs support up to 100 GB/s [9]. Our two test systems used for this thesis (see Section 5.1) support 21 GB/s (Sandy Bridge) and 59 GB/s (Haswell). However, the actual achievable bandwidth depends a lot on the memory access pattern. For example, randomly accessing small chunks of memory has less throughput than a long sequential read.

The MemGuard authors focused on the memory usage of applications and did not take another important contender to memory bandwidth into account: Modern I/O devices transfer data to the CPU by writing directly to main memory (Direct Memory Access, DMA).

For their system, the MemGuard authors estimated a worst-case bandwidth of only 1.2 GB/s, which is roughly equal to the rate of 10 Gigabit Ethernet. Consequently, their algorithm may not be able to distribute memory bandwidth properly while a DMA transfer from the network card is in progress.

With devices such as solid-state drives (SSD) and Ethernet controllers getting faster, DMA transfers can now take a significant proportion of the available memory bandwidth shared with the CPU.

Newer SSD models based on NVMe [12] are promising data transfers with up to 3.2 GB/s [15]. On a desktop computer with a quad-core CPU and a memory bandwidth of 25.6 GB/s, each core usually has 6.4 GB/s available. Thus, a full-speed DMA transfer would cut a single core's memory bandwidth in half. As real-world limits are likely smaller depending on the access patterns, the actual influence may be even larger.

The main contribution of this work is a flexible and extendible DMA load generator based on NVMe. In the following Chapter 2, we provide background information on technologies we used, such as DMA and NVMe. In Chapter 3, we discuss the design, and in Chapter 4 the implementation of our load generator. Chapter 5 contains our evaluation: We analyze the maximum bandwidth of our SSD, evaluate the use of performance counters to characterize NVMe traffic, and test interference with several benchmark programs. Finally, we summarize our results in Chapter 6, presenting a conclusion and proposing future work.

Chapter 2

Background

This chapter provides background information on technologies and tools used in this work: We first explain some details on DMA (Section 2.1). In Section 2.2, we then give an overview on the NVMe interface. Finally, we present tools we used for the evaluation: Intel's Performance Counter Monitor (Section 2.3), the PARSEC benchmark suite (Section 2.4) and the Silo in-memory database (Section 2.5).

2.1 DMA

Direct memory access (DMA) allows external devices to access the main memory independently from the CPU. Usually, the CPU requests a transfer from an external device. The device then can transfer its data directly from or to the main memory. During the transfer, the CPU is not blocked and can do other work. When the transfer finishes, the device notifies the CPU using an interrupt.

During the transfer, requests from both the CPU and the external device may reach the memory controller. Thus, memory accesses by the CPU may be slowed down. We analyze this interference in Section 5.5.

Another issue with DMA is cache coherency: In contrast to the CPU with its cache hierarchy, the external devices access the main memory directly. Consequently, devices may read stale data, which the cache has not written back yet, or the caches may serve the CPU data that changed in memory in the meantime. To solve these issues, the cache needs to be invalidated as part of the DMA operations, slowing down reads and writes by the CPU and adding load to the memory controller.

Intel's DDIO technology [3] aims to improve this situation. It makes the last-level cache (LLC) the primary target of DMA operations. This solves the cache coherency issue, as mechanisms for coherency with the higher-level caches, which are specific to the individual CPU cores, are already in place for multicore systems. Memory controller contention is also improved: Especially for frequent, small transfers such as those from network cards, the main memory is not involved in the transfers at all, speeding up processing considerably.

Having DMA writing to the LLC directly may reduce the amount of cache available to applications. However, Intel also greatly increased the total amount of last-level cache, making up to 20 MB cache possible [5]. Thus, this LLC usage is unlikely to slow down applications compared to CPUs without DDIO.

Our Sandy Bridge test system does not support the DDIO technology, but the Haswell system does (see Section 5.1 for an overview of our systems). In Section 5.4.3, we analyze LLC usage on the Haswell system.

2.2 NVMe

Non-Volatile Memory Express (NVMe) [12] is a communication interface designed for solid-state drives (SSD) connected via PCI Express. It allows for higher transfer rates with more parallelism compared to the older SATA/AHCI interface, which was primarily designed for hard disk drives. The NVMe specification defines a command set consisting of Admin and IO commands.

The Admin commands are primarily used by the NVMe driver to set up drives for data transfers. They allow querying drive-specific information as well as setting options, such as power management.

The IO commands are used for the actual data transfers. The NVMe specification defines three mandatory commands: *Read*, *Write*, and *Flush*. The *Read* and *Write* commands take a location in main memory as well as a logical block address and transfer data to or from the SSD. The *Flush* command instructs the SSD to write any changes from previous commands from its caches to non-volatile media.

All communication with the SSD happens with submission and completion queues located in main memory. The driver submits its commands to a submission queue. The SSD accesses commands from this queue via DMA. After processing a command, the SSD submits a completion entry to the completion queue, again with DMA. The driver usually waits for the completion entry, allowing other threads to run in the meantime.

Each NVMe controller has a single Admin Submission Queue and an Admin Completion Queue. IO commands are managed in separate IO Submission and Completion queues. With Admin commands, the NVMe driver can create multiple of them on demand.

Multiple queues improve performance as they allow the SSD to process requests in parallel. The NVMe Linux driver usually creates one IO Submission and one IO Completion Queue per CPU core.

IO operations work analogous to the Admin operations: The driver submits an IO command to one of its IO Submission Queues. The SSD then performs the data transfer and notifies completion to the driver by posting a completion entry to one of the IO Completion Queues. So in addition to the user data, an NVMe command of 64 B is transferred to the SSD and a completion entry of 16 B is transmitted from the SSD per operation.

Each IO operation can transfer multiple logical SSD blocks. The maximum number of blocks is specific to the SSD; our model (see Section 5.1) allows transferring 32 blocks of 4096 B per command.

With hard drive disks, the transfer block size is given by the organisation of the physical disk in sectors. In contrast, the logical block format is not fixed by the NVMe specification. With the *Format* admin command, it is possible to select a block format from a list of supported formats specific to the SSD. The *Identify* Admin command lists the available formats, indicating the expected relative performance of the formats to each other. Our SSD supports both 512 B and 4096 B blocks. It indicates "Good" performance for 512 B blocks and "Best" performance for the larger block size of 4096 B. Additionally, up to 64 B of metadata can be stored per block. This metadata is not used by the SSD; applications such as the file system can write any additional data there. However, we do not use any metadata for our DMA transfers.

Intel provides a Linux tool for sending both Admin and IO commands to an SSD from the command-line [10]. This tool is very useful for setting the SSD's block format. NVMe SSDs can be organized in multiple namespaces, dividing the available storage. However, our SSD only supports a single namespace. The command `nvme id-ns -h /dev/nvme0n1` lists the available formats for the first NVMe namespace of the first NVMe device on the system. For example, `nvme format -l 3 /dev/nvme0n1` then formats the first namespace to use the third format, deleting all data. Figure 2.1 shows the formats as reported for our SSD.

2.3 Performance Counter Monitoring

The Performance Counter Monitor (PCM) [11] is a technology by Intel for monitoring specific CPU-related events. For our purposes, Uncore Perfor-

```
# nvme id-ns -h  /dev/nvme0n1
[...]
LBA Format  0 : Metadata Size: 0   bytes - Data Size: 512 bytes
        - Relative Performance: 0x2 Good
LBA Format  1 : Metadata Size: 8   bytes - Data Size: 512 bytes
        - Relative Performance: 0x2 Good
LBA Format  2 : Metadata Size: 16  bytes - Data Size: 512 bytes
        - Relative Performance: 0x2 Good
LBA Format  3 : Metadata Size: 0   bytes - Data Size: 4096 bytes
        - Relative Performance: 0 Best (in use)
LBA Format  4 : Metadata Size: 8   bytes - Data Size: 4096 bytes
        - Relative Performance: 0 Best
LBA Format  5 : Metadata Size: 64  bytes - Data Size: 4096 bytes
        - Relative Performance: 0 Best
LBA Format  6 : Metadata Size: 128 bytes - Data Size: 4096 bytes
        - Relative Performance: 0 Best
```

Figure 2.1: Abridged output of the `nvme id-ns` command for our SSD.

mance Monitoring counters located at the last-level cache (LLC) are interesting. These counters count read and write accesses to the LLC. As with DDIO (see Section 2.1) all main memory accesses by the CPU and by external devices need to pass the LLC, we can use these counters to estimate the memory bandwidth used by applications and devices. In contrast to the more common CPU counters, the Uncore counters are only available for some CPUs [7].

A performance counter is essentially a simple integer variable. It can be tuned to a specific event.[1] Whenever this event occurs in a cache line, the variable is incremented by one. The events are further divided in cache hits and cache misses. In case of a cache miss, the LLC fetches the cache line from main memory, recording a cache hit afterwards (see Section 5.4.3). The CPU can read the counter's current value at any time. However, a counter may only monitor a single event at a time. Consequently, we either have to switch the counter rapidly between multiple events or run the measured program multiple times with different events.

As the events are triggered per cache line, which are 64 B, a single event usually means a read or write of 64 B of data. However, some events are also triggered for partial reads or writes, occurring when data is unaligned or when the access size is smaller than 64 B. Therefore, a single event indicates

[1] In fact, we are only looking at a single event in the sense of the Uncore documentation. This event is filtered by Opcode and Thread-ID. When we say "event" here, we are actually referring to a specific combination of Opcode and Thread-ID.

a data transfer of 64 B or less.

On our Haswell platform (see Section 5.1), the following events are available:

PCIeRdCur "PCIe read current transfer" – This event is triggered when a PCIe device reads from a cache line. It does not distinguish between full and partial reads, so the reported number is likely higher than estimates from transfer size.

RFO "PCIe partial write" – This event is shared with the CPU, but can be filtered for activity from PCIe. In this context, it counts partial writes (i.e., writes smaller than 64 B) to cache lines.

ItoM "PCIe write full cache line" – Just like RFO, this event is shared with the CPU and can be filtered. It counts full writes of 64 B to cache lines. The sum of RFO and ItoM corresponds to what PCIeRdCur reports for reading.

CRd "Demand Code Read"

DRd "Demand Data Read" – CRd and DRd are read events occurring when the CPU reads code or data from the last-level cache. These events are not directly related to PCIe traffic.

PRd "MMIO read"

WiL "MMIO write" – PRd and WiL events are recorded when the CPU does memory-mapped IO (MMIO) read or write. While this mechanism is used for communication with devices, it does not convey information about NVMe activity where the SSD device communicates using queues in memory.

2.4 PARSEC

PARSEC [1] is a benchmark suite with a focus on multithreaded applications. It consists of 13 workloads, of which we tested all except `facesim` and `fluidanimate` (see Section 5.5.1).

As we are using the benchmark programs to analyze interference of memory accesses, we are especially interested in workloads that have high levels of main memory usage. Figure 4.7 of the PARSEC overview paper [1] gives a breakdown of off-chip traffic (i.e., memory accesses that cannot be satisfied by a cache) of the PARSEC workloads. Note that the `raytrace` workload is missing from the graph.

The top workloads regarding memory usage in the graph are `streamcluster`, `canneal`, and `facesim`. The `streamcluster` and `canneal` workloads, along with `raytrace`, are also the workloads that showed interference in our evaluation in Section 5.5.1.

2.5 Silo

Silo [18] is an in-memory database designed to achieve high performance on multicore systems. Its authors demonstrate that Silo scales almost linearly to multiple cores.

An in-memory database is well suited to analyze DMA interference as its core objective is to store and retrieve large amounts of data from main memory as quickly as possible.

Chapter 3

Design

This chapter describes the design of a DMA load generator based on NVMe. The load generator needs to satisfy several goals: We want it to support different memory access patterns (Section 3.1.1) and it needs to support configuration options such as the number of workers used (Section 3.1.2). Additionally, it should support using performance counters to evaluate its performance (Section 3.2).

3.1 DMA Load Generator

In this section, we describe our DMA load generator.

The standard way of transferring data from and to an SSD works by reading and writing files on a file system. While this works fine for generating some load, there is little control over the exact size and memory location of the transfers as the operating system provides a cache between the device and client applications. Additionally, having as little overhead as possible is important as the load generation runs in parallel to the benchmarking applications.

Therefore, we decided to bypass the operating system and issue NVMe commands to the SSD directly. The core of the NVMe command set are the three commands `read`, `write`, and `flush`. The `read` and `write` commands transfer data from or to the SSD. The `flush` command instructs the SSD to persist anything written in previous commands and does not transfer any data.

In addition to these I/O commands, NVMe specifies an admin command set whose commands are used to identify features specific to the SSD and to configure the SSD.

In order to generate DMA load, we allocate a memory buffer of fixed size in our load generator, which is then used directly as target of the NVMe commands. We can then transfer data using `read` or `write` commands. These commands share a memory layout and are both given a block range on the SSD identified by a logical block address and count, and an address in memory. Switching between reading and writing thus is only a matter of changing the opcode.

The maximum number of blocks that can be transferred per command is constant per SSD and can be determined using the `identify` admin command. In our case, the SSD allows transferring up to 32 blocks of 4096 byte per command.

3.1.1 Memory Access Patterns

For our purpose, not only the bandwidth used by the load generator is important, but also the pattern of memory accesses. Supporting different patterns allows imitating different devices other than SSDs, such as network cards.

For our tests, we used three major patterns. All patterns can be set to either use `read` or `write` commands with a single buffer of configurable size.

single The *single* pattern transfers a single block with each NVMe command. It accesses the memory buffer sequentially. In addition to the data transfer, this pattern produces a large number of NVMe commands per second. It is meant to be similar to the way networking devices write incoming packets into a buffer, although network packets are usually a lot smaller than the 4 kiB blocks the SSD uses.

random This pattern is meant to resemble the way the operating system caches transfers from block devices. As there is no dedicated memory region for these transfers, they end up in random memory locations. The *random* pattern chooses a pseudo-random position in the memory buffer as source or destination for transfers. To achieve the maximum possible transfer rate, it always instructs the SSD to transfer the maximum number of blocks per command.

flush The *flush* pattern does not transfer any user data. Instead, it only sends `flush` commands to the SSD. This results in an even larger command rate than with the *single* pattern. The bandwidth is only used by the NVMe commands sent to the SSD and completion queue entries sent back.

3.1.2 Configuration Options

In addition to access pattern and buffer size, the load generator allows further shaping of transfers with the following options:

Caching In systems with Intel DDIO (see Section 2.1), DMA transfers go directly to the last-level cache (LLC) instead of the main memory. This can speed up both the transfers and the subsequent CPU access in case of written data. The load generator can ensure parts of the memory buffer are in the LLC already by reading the memory region before issuing a transfer command.

Parallelism A major feature of the NVMe command set is its ability to process commands in parallel by having multiple command submission queues. The load generator allows issuing multiple commands in parallel, increasing the overall bandwidth.

Bandwith and command limiting When analyzing the effects of the different access patterns, it may be desirable to keep bandwidth or command rate constant between the patterns. The load generator allows this by limiting the bandwidth or command rate to a level that is possible with both patterns.

3.2 Performance Counter Analysis

In this section, we amend the design of the load generator with specific performance counters provided by the hardware.

When generating artificial load, we have precise control and feedback about the amount of data transferred to and from the SSD. In addition to the user data stored on the SSD, this also includes the commands sent to, and transfer completion entries received from the SSD.

In other applications, especially when a file system is involved, this is not possible. Raw NVMe commands will only be generated in the driver, translating higher-level read or write system calls. One way of inspecting those transfers is by employing the PCIe performance counters available in Intel's newer Xeon processors. These counters can be analyzed using the Intel Performance Counter Monitor (PCM) tool, which is provided as free software by Intel [11].

All PCIe counters used by us are located at the last-level cache (LLC). They count different sources of access to the cache lines. Thus, the readings are only accurate to multiples of the cache line size 64 B. See Section 2.3 for more information about the counters available on our system.

In its standard operation mode, the PCM tool shows several event types related to PCIe transfers aggregated per second. While our hardware supports having multiple performance counters running at the same time, the tool will only use a single one at a time. Consequently, it has to switch rapidly between the available events, extrapolating the readings to the full second. Although this is sufficient to get a general feeling of the events involved, an exact reading is desirable to get more accurate results.

For this purpose, we added a performance counter mode to the load generator. In this mode, a counter is set to monitor a single event source. Analysis of the different counters is then possible over separate runs, each of which can last multiple seconds. This is possible because our generated load is uniform over time and between runs.

Chapter 4

Implementation

In this chapter, we present our implementation for Linux. Section 4.1 introduces our load generation tool `nvme-memload`. During development, we became concerned that our tool may be slowed down by its interface with the NVMe Linux driver, so we implemented a workaround described in Section 4.2. Finally, Section 4.3 details the integration of performance counter measuring in `nvme-memload`.

4.1 DMA Load Generation

In Section 3.1 we described the design of a DMA load generator. In this section, we will present our implementation called `nvme-memload`, published on GitHub [19].

Using the `ioctl` system call, the Linux NVMe driver allows bypassing the usual kernel stack by sending NVMe commands directly to the SSD. It only does minimal processing: The driver sets up the source or target memory pages for DMA access, puts the command in one of the SSD's NVMe command submission queues and then blocks, waiting for a response in the completion queue.

`nvme-memload` uses this system call to send raw NVMe I/O commands as fast as possible. It allocates a single buffer of configurable size. It then instructs the SSD to read or write to parts of the buffer according to the pattern loaded.

In `nvme-memload`, reading always starts at the SSD's first block. In contrast, permanently writing to the same SSD blocks has a big performance impact as SSDs have to delete blocks before they can be rewritten. Thus, `nvme-memload` picks a random location on the SSD to keep the influence low.

The SSD can be reset using the *Format NVM* command. This will quickly erase all data written on the SSD, returning it to a fresh state. Doing this between tests is advisable when writing to the SSD. Otherwise, write performance severely degrades during long test runs, as the SSD runs out of free blocks, to which it can write without erasing.

4.1.1 Patterns and Parallelization

The patterns described in 3.1.1 are implemented as shared objects, allowing them to be loaded independently from the main program. The worker threads call into the shared object, receiving a structure describing the next operation. The parameters are the NVMe operation (read, write, flush), the memory location, and the number of blocks.

Patterns such as the *single* pattern are required to hold state for sequential access, as they have to remember their previous position. This can lead to race conditions when multiple worker threads try to retrieve their next operation at the same time. To keep the patterns simple, access to them is protected by a mutex. This does not have a negative effect on performance, as the worker threads are blocked on their NVMe commands most of the time.

When multiple worker threads are active, the SSD will receive commands in parallel on its command queues. Consequently, no guarantee can be made about the actual ordering of operations. For `read` commands, the SSD is unlikely to benefit from reordering though as all requests are for the same SSD block.

4.1.2 Bandwidth and Command Limiting

`nvme-memload` allows limiting the throughput by data bandwidth or by command count, as described in 3.1.2.

The limit is implemented by counting commands for a configurable subsecond timespan, blocking when the configured limit is reached. As all worker threads need to reliably stop when the limit is reached, this involves another mutex and a condition variable to restart the threads for the next time slot.

With this design, the workers only have to do little additional work, keeping the maximum possible throughput high. The configurable timeslot size allows for short, focussed load bursts as well as continuous load with small pauses in between.

4.1.3 Caching

`nvme-memload` implements two caching modes designed to force usage of the last-level cache in the SSD's DMA operations. These modes are intended to work in conjunction with the Intel DDIO technology (see Section 2.1); it should not have an influence on other systems without this feature. When the CPU reads a memory location, it is fetched into its caches. A subsequent DMA operation on the same memory location will then only operate on the LLC until the cache lines are evicted from the LLC.

In the **once** mode, the whole buffer is read once before starting to send NVMe commands. This mode obviously only makes sense as long as the whole buffer can fit into the LLC. As will be seen in Section 5.4, this mode does not actually make a difference.

With the **always** mode, the worker threads will read the relevant part of the memory buffer before each command is sent off to the SSD. Consequently, the SSD will always operate on memory in the LLC, even when the whole buffer is too large to fit. With this mode, cache misses only occur in memory accesses by the CPU.

4.2 NVMe Linux Driver Customization

One concern with `nvme-memload` was the high overhead from the `ioctl` system call. When writing single blocks per command to memory, up to 300 000 commands are issued per second; each command will use a separate system call. To rule out slowdown from a system call and accordingly trap limit, we added another `ioctl` command to the NVMe Linux driver. Instead of a single NVMe instruction, this command accepts an arbitrarily-sized array of instructions, similar to the `readv` family of system calls [13]. When detecting the custom driver, `nvme-memload` will use batching, calling `ioctl` at a reduced rate. The driver then loops over the array and processes them one-by-one, just like the regular command.

While the custom driver successfully shifts a lot of the CPU time to the kernel, it does not have any positive or negative impact on the performance (see 5.3). Thus, the system call limit is not an issue for our work.

4.3 Performance Counter Analysis

In Section 3.2, we described the addition of performance counter monitoring to `nvme-memload`. In this section, we explain the actual implementation and operation of this feature.

With a few tweaks, Intel's PCM tool can also be built as a library. We use this and link it into `nvme-memload` to provide exact readings synchronized with the actual transfers. As the performance counters are not available on all platforms, the library is only included when requested to allow operation without the counters.

Our tool allows specifying a single event source per run. On Haswell, the following event sources are available: `PCIeRdCur`, `RFO`, `CRd`, `DRd`, `ItoM`, `PRd`, and `WiL`. See Section 2.3 for an explanation of the individual events. For each event source, either last-level cache hits or misses can be counted.

For precise measurements, `nvme-memload` can stop after a time limit or after a certain number of commands or blocks has been transferred. This makes comparison of the different event sources possible, even when the actual command rate differs between runs.

Chapter 5

Evaluation

In this chapter, we present our experimental results with high-bandwidth DMA load. First, we analyze the possible bandwidth with `nvme-memload` on our test systems (Section 5.2) and evaluate our NVMe driver modification (Section 5.3). In Section 5.4, we then correlate bandwidth and command counts with performance counters. Finally, we analyze interference from DMA load using the PARSEC benchmark suite (Section 5.5.1) and some in-memory databases (Section 5.5.2).

5.1 Test Systems

In our tests, we generated DMA load using the **Intel SSD 450** with a capacity of 400 GB. The SSD is connected via four lanes of PCI Express 3.0. The SSD supports two block sizes: 512 B and 4096 B. Using the NVMe `Identify` admin command, the SSD indicates that 4096 B is its preferred block format for best performance. Consequently, we formatted it to use the larger block size of 4096 B. The NVMe SSD was exclusively used for DMA load generation; the operating system and benchmark programs resided on a conventional SATA SSD.

We ran our tests on two systems with different hardware configurations:

1. A 2011 **Sandy Bridge** system with an Intel Xeon CPU E3-1230 with four cores clocked at 3.20 GHz [6]. It has 8 MB of cache available. This CPU only supports the older PCIe 2.0 standard, limiting possible maximum bandwidth from the SSD to 2 GB/s. DMA on this machine does not hit the last-level cache.

2. A 2014 **Haswell** system with two Intel Xeon CPU E5-2630 v3 with eight cores clocked at 2.40 GHz each [8]. Each CPU has 20 MB of

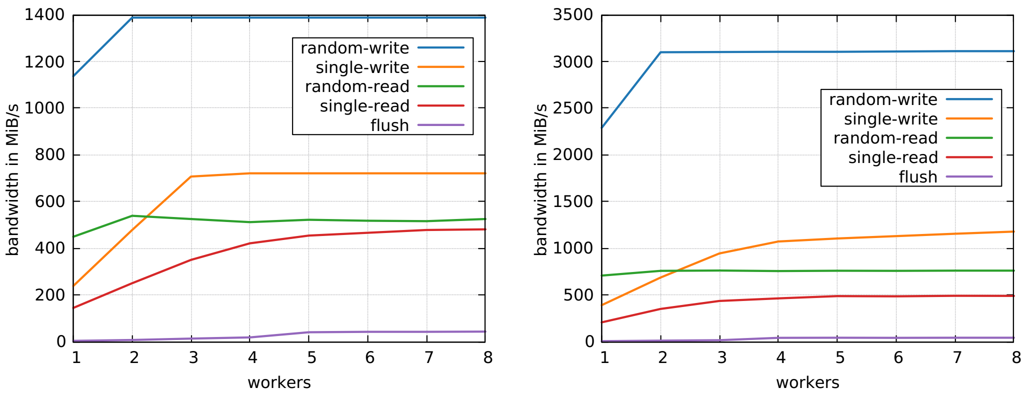
cache available and supports the PCIe 3.0 standard. All our tests were pinned to the first socket for both CPU and memory. The performance counters used in Section 5.4 were only available on this system. Additionally, it supports Intel DDIO, so DMA transfers go to the last-level cache (see Section 2.1).

We disabled Hyper-Threading on both systems to reduce noise in results.

5.2 DMA Load

In this section, we analyze the achievable bandwidth with our three access patterns *single*, *random*, and *flush* from Section 3.1.1 with different levels of parallelism.

As the Linux driver processes NVMe commands synchronously, blocking the calling user process, we are able to precisely measure the bandwidth used by each worker thread. In the following experiment, we ran each of our patterns for 10 s, summing the number of commands and the number of blocks transferred. We reset the SSD before each run with the NVMe `Format` command.



(a) Sandy Bridge (b) Haswell

Figure 5.1: DMA bandwidth used by `nvme-memload` with different amounts of worker threads sending NVMe commands.

Figure 5.1 shows the results as average bandwidth per second.

For the patterns transferring data, both a *read* and *write* variant is listed. These are given relative to the main memory, so *read* means reading from main memory and writing to the SSD.

In addition to data blocks transferred, the bandwidth displayed also includes transfers from NVMe commands. For each operation, the SSD reads a

64 B NVMe command from main memory and then writes a 16 B completion queue entry back to main memory. In summary, the bandwidth is calculated as follows:

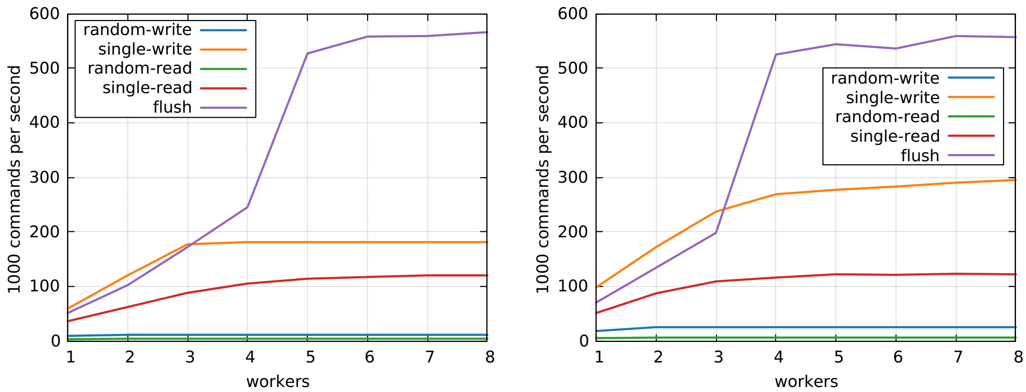
$$\text{avg bandwidth} = \frac{\text{number of blocks} * 4096B + \text{number of commands} * (64B + 16B)}{\text{runtime}}$$

The PCIe 2.0 bandwidth limit on the Sandy Bridge system is clearly visible: *random-write* transfers are capped below 1400 MiB/s, whereas the same configuration on the PCIe 3.0 Haswell system achieves more than 3000 MiB/s.

The patterns behave very differently when given more commands in parallel. On both systems, the *random-write* pattern reaches maximum bandwidth usage with two workers. It stays constant as more workers are added. In contrast, the *single-write* pattern, which transfers only a single block per command, only caps on the Sandy Bridge system with three or more workers. On Haswell, *single-write* performance keeps gradually increasing. Still, the *random-write* pattern performs at roughly double the bandwidth of the *single-write* pattern on both systems.

This gap is a lot smaller for the *read* variants. These both perform below their *write* counterparts. They also do not have a clear cap on either system.

The *flush* pattern's bandwidth usage only becomes noticeable compared to the actual data transfers with more than four workers. As our goal is to analyze interference from high-bandwidth DMA transfers, we only use the high-bandwidth patterns *single* and *random* for the experiments in Section 5.5.



(a) Sandy Bridge (b) Haswell

Figure 5.2: NVMe command rate of `nvme-memload` with different amounts of worker threads.

In contrast to the bandwidth graphs, the command rates in Figure 5.2 look very similar on both our systems. The *random* pattern only sends very few commands as most of the time is spent waiting for the SSD to transfer the data. The two *single* patterns show very clearly where the PCIe 2.0 limit on Sandy Bridge comes into play: The command rates for *single-read* are roughly identical on both systems, whereas the *single-write* pattern starts similarly on both systems but clearly caps out on Sandy Bridge after three workers.

The *flush* pattern has the most striking curve. Along with the *single-write* pattern, it has the highest command rate among our test set. The `flush` NVMe command instructs the SSD to commit data from previous I/O commands to non-volatile media. However, as this is the only command we send, it is essentially a no-op and can thus be answered very quickly. Similarly, the *single-write* commands finish fast because only little data needs to be transferred for them. In contrast to the *single-write* pattern, the *flush* curve shows a huge jump in command rate between 4 and 5 (Sandy Bridge) or 3 and 4 (Haswell) workers. The final command rate is identical on both systems.

Considering the results above, we decided to use 4 worker threads on the Sandy Bridge system and 8 worker threads on the Haswell system for the DMA interference experiments in Section 5.5. With this configuration, we miss only very little bandwidth on the Sandy Bridge system and go for maximum bandwidth on Haswell. We did not consider more threads as all load generator threads will be pinned to a single CPU core for the interference experiment.

5.3 Syscall Batching

In Section 4.2, we described a modification to the NVMe Linux driver where the `ioctl` system call used to send commands to the SSD is amended with batching functionality. This reduces the rate of system calls our application does and may improve performance. To verify this, we repeated the tests from the previous section, sending sets of 1000 commands to the driver.

In Figure 5.3 we see the command rate with our modified driver. Comparing to Figure 5.2a, it is clearly visible that even with the *flush* pattern, having the highest command throughput, there is no improvement with our modifications. Thus, for our application, the system call rate limit is not an issue.

In the following benchmarks, we use the standard driver without modifications.

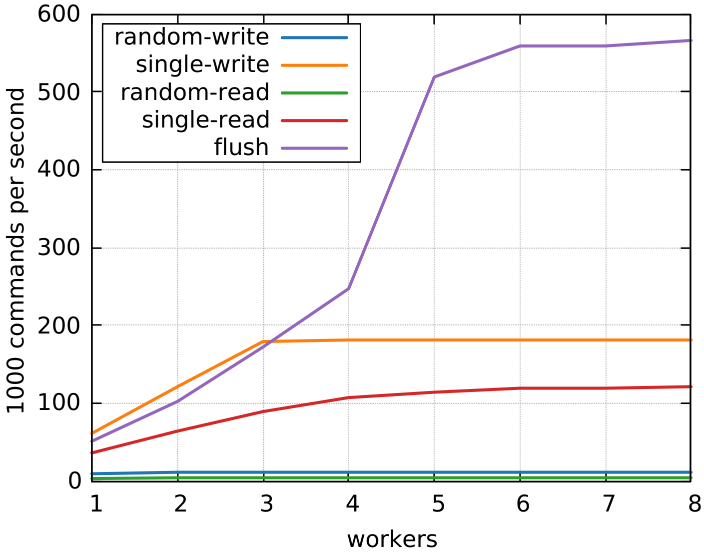


Figure 5.3: NVMe command rate on Sandy Bridge with our modified Kernel.

5.4 Performance Counter Monitoring

In this section, we analyze results from the Performance Counter Monitor integrated in `nvme-memload` as described in Section 4.3. We only did those tests on the Haswell system as the Sandy Bridge system does not have the necessary performance counters.

In the following experiments, we ran `nvme-memload` with Performance Counter monitoring enabled. For each run, it sets up a single counter, then issues a fixed number of NVMe commands. At last, it records the final number in the counter. Consequently, separate runs of the same configuration are necessary to test multiple events.

We tested our *single* and *random* patterns for both reading and writing, and the *flush* pattern as described in Section 3.1.1. To assess the base load, we also added a *noop* pattern, which just runs the performance counters without doing transfers.

Figure 5.4 shows the results from the *noop* pattern run over 5 s. The `CRd` (Demand Code Read) and `DRd` (Demand Data Read) events are shared with the CPU and cannot be filtered for PCIe traffic. As explained in Section 2.1, the `PRd` (MMIO Read) and `WiL` (MMIO Write) events carry no usable information about NVMe transfers.

The remaining events are `PCIeRdCur` (PCIe read current transfer), `RFO` (PCIe partial write), and `ItoM` (PCIe write full cache line). The last two of them are shared with the CPU as well, but can be filtered for PCIe traffic.

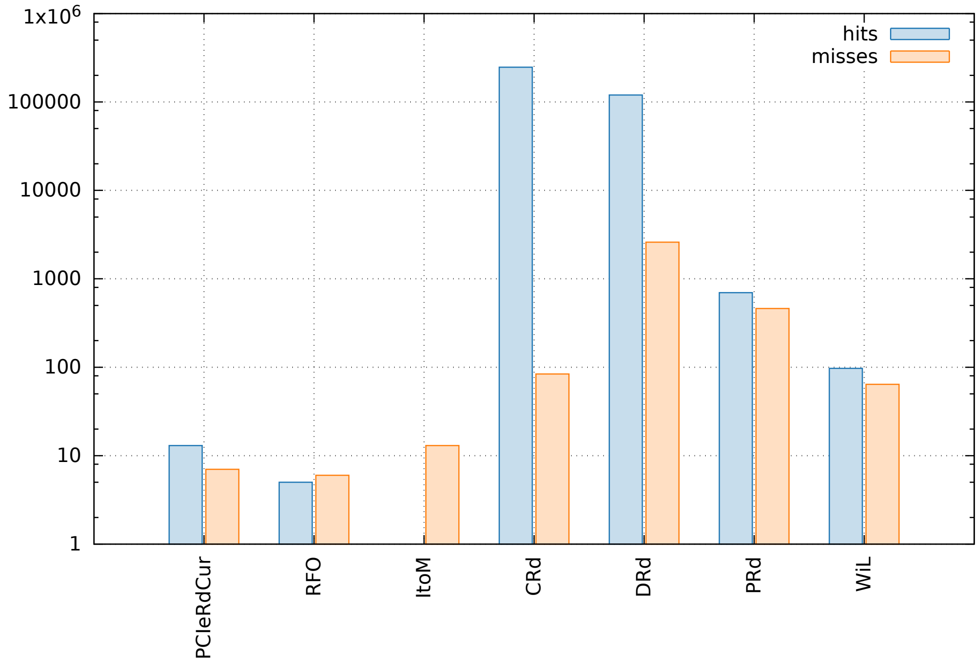


Figure 5.4: Performance counter results of running the counters for 5 s without any load on a logarithmic scale.

Only little base activity is visible for these events. We decided to keep the test duration between 5 s and 10 s to keep this base load low. For this runtime, we can expect less than 50 unrelated events for each event type.

5.4.1 NVMe Command Counts

Immediately visible from the results is the correlation between the command count and the `RFO-hits` event. Figure 5.5 compares command count and RFO-hits for each pattern. This correlation only occurs when the target memory buffer the SSD writes to is aligned at 64 B boundaries, matching LLC cache lines.

The `RFO` event counts partial cache line writes. Our SSD works with 4096 B blocks, fitting exactly in 64 cache lines. Accordingly, no `RFO` events are recorded when all transfers are properly aligned.

However, there is another data transfer: The SSD writes completion entries into memory after every NVMe operation. These are 16 B and emit a single `RFO` event each. Consequently, we are actually counting completion entries here.

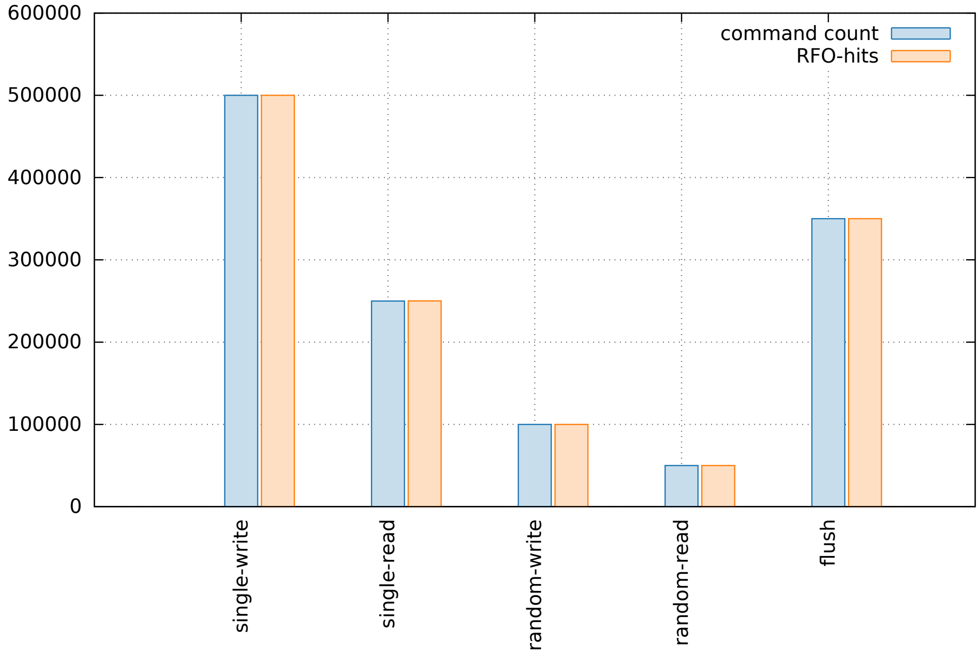


Figure 5.5: Comparison of the command count and `RFO-hits` events for each pattern.

5.4.2 Bandwidth

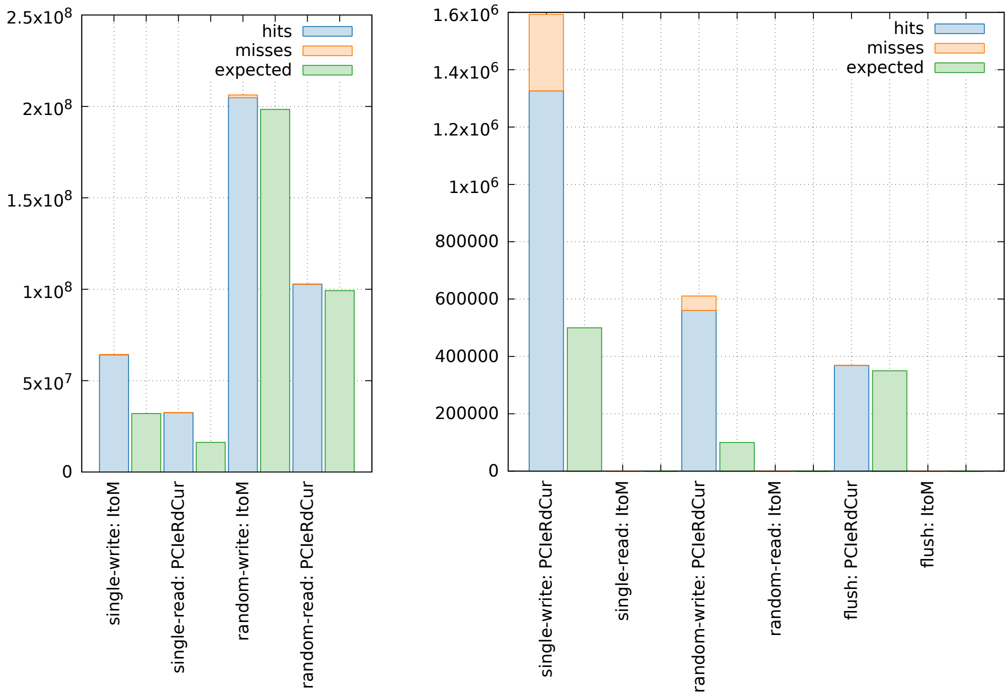
In addition to NVMe command counts, we can also estimate transfer sizes using performance counters. In Section 5.2, we give a formula for calculating the total bandwidth. The performance counters distinguish between traffic to the SSD (`PCIeRdCur`) and traffic from the SSD (`ItoM`). `PCIeRdCur` counts both full and partial cache line, so it includes NVMe commands (64 B each) regardless of their alignment. The completion entries (16 B each) are smaller than a cache line. Thus, they are not included in the `ItoM` events, which are only triggered for full cache line writes, but in the `RFO` events (see Section 5.4.1).

The performance counters work with cache lines, which are 64 B in size. Thus, the expected counter values are as follows:

$$\text{PCIeRdCur} = \frac{\text{blocks read from memory} * 4096B + \text{commands} * 64B}{64B}$$

$$\text{ItoM} = \frac{\text{blocks written to memory} * 4096B}{64B}$$

The counters further divide these events into cache hits and misses. The graphs in Figure 5.6 compare our estimated event count with the combined number for cache hits and misses.



(a) Events that include data blocks.

(b) Events that do not include data blocks.

Figure 5.6: Comparison of actual and expected counter values.

For the events including data blocks (Figure 5.6a), the accuracy of our estimate depends on the pattern. With the *random* pattern, which transfers multiple blocks per NVMe command, the counter values and our estimate are very close. On the other hand, the counter results for the *single* pattern are twice our estimates.

In Figure 5.6b, only bandwidth from NVMe commands and completion entries is recorded. For the `ItoM` event, the performance counters do not record any events. This confirms that the completion queue entries are not included here. For the `PCIeRdCur` event, the performance counters match the expected number of events with the *flush* pattern. For the other patterns that transfer data, the actual counter value is a lot higher. Thus, when writing from SSD to memory, additional `PCIeRdCur` events are generated.

5.4.3 Caching

The performance counters distinguish between cache hits and misses. By reading the target memory area with the CPU beforehand, we can reduce the amount of cache misses significantly; cache misses then only occur when the CPU accesses the memory. DMA memory accesses triggering a cache miss will record both a cache miss and a cache hit in the performance counters.

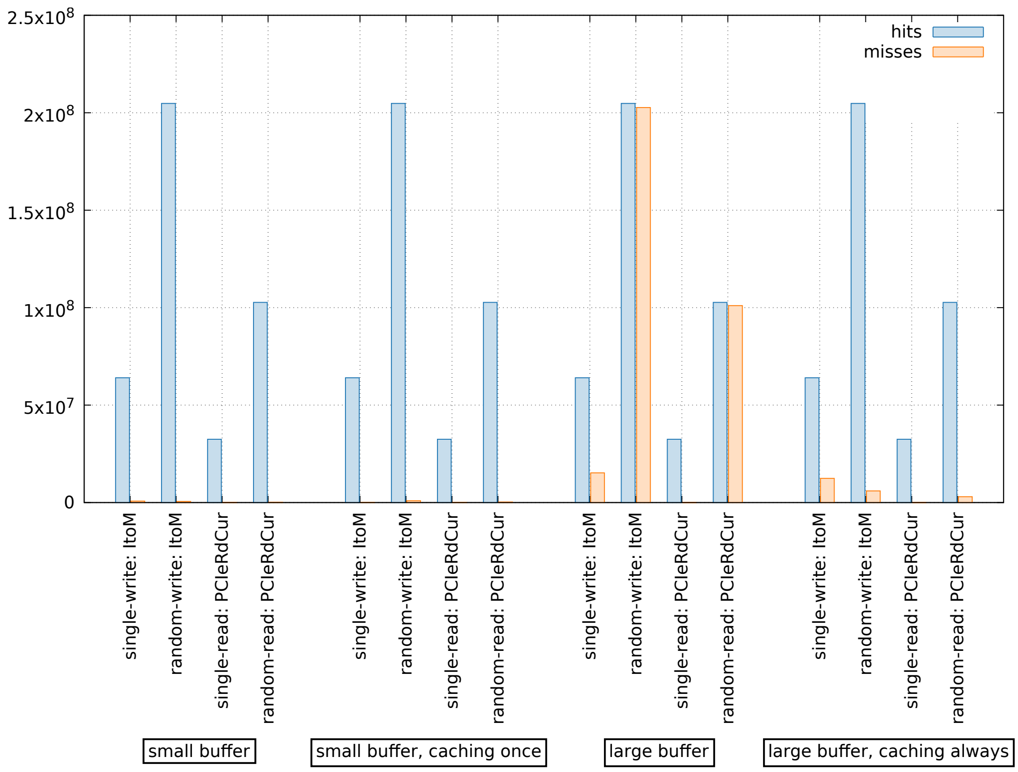


Figure 5.7: Comparison of PCM event counts with different buffer sizes and caching modes in `nvme-memload`. Only events including transferred data blocks are shown.

Figure 5.7 shows counter results for different configurations of `nvme-memload`. The small buffer, which was also used in the tests above, has a size of 1000 blocks (approximately 3 MiB). Thus, it fits completely in the 20 MB last-level cache of our Haswell test system and only few cache misses occur.

In the second configuration, the whole buffer is read once by the CPU before any NVMe commands are sent, putting it completely into the cache. However, this does not change the number of cache misses significantly. Consequently, DMA transfers always allocate cache lines in the LLC; a *write allocate* write policy is used.

The large buffer used in the other two configurations has exactly 1 GiB and thus cannot be cached completely. Without any additional work by `nvme-memload`, a large number of cache misses is recorded. The observed numbers are consistent with how the patterns operate: With the *single* pattern, the SSD reads or writes its blocks sequentially in memory. Thus, blocks are more likely to be in memory already due to prefetching. In contrast, the *random* pattern always picks a random location in the memory buffer. This location is unlikely to be cached, so almost every memory access produces a cache miss.

In the last configuration, each memory location is read by the CPU before issuing an NVMe command. Consequently, cache misses mainly happen when the CPU accesses the memory and not as part of the DMA transfer. Still, the number of cache misses recorded is larger than with the small memory buffer.

The values of the *hits* bars do not change between our configuration options. They only depend on the number of blocks transmitted by each pattern. Therefore, even for cache misses, the performance counters record a cache hit after the memory area has been loaded into the cache. This also reveals that all DMA operations are processed via the LLC, occupying cache rows even when the CPU never accesses the data.

5.5 DMA Interference

In the following sections, we analyze interference from our DMA load generator. As test programs, we used several benchmarking programs.

We ran the benchmark programs in parallel with multiple configurations of `nvme-memload`. We pinned the load generator process to a single CPU core, excluding the benchmark processes from that core. For most configurations, this did not have a significant effect on the achieved DMA load, while eliminating all influence of CPU scheduling. Some of the benchmark programs we used overwrite their pinning configuration on startup. Specifically, they pin some of their threads to the first CPU core. By pinning `nvme-memload` to the last core, we were able to prevent conflicts.

An additional run per configuration without any DMA load but with the same pinning is used as baseline.

We varied the following parameters through several configurations:

Number of memory channels We can make different amounts of memory channels available to the CPU by physically adding or removing RAM. On the Sandy Bridge system, there are two memory channels available;

on the Haswell system four. For both systems, we tested the minimum (1) and the maximum (2 or 4) number of memory channels. Note that with fewer memory channels, there is less memory available for the system. We had to adjust some benchmarks to reduce memory usage.

Last-level cache usage Although we determined in Section 5.4 that putting the whole buffer in cache once does not improve cache usage on the Haswell system, reading the buffer before issuing each NVMe commands does. Consequently, we tested with and without doing this. On the Sandy Bridge system, the caching configuration does not affect the DMA transfer itself, but still adds load by the CPU to the memory controller.

Patterns From the access patterns introduced in Section 3.1.1, we used both the *single* and *random* patterns for our tests. We only set the patterns to write to main memory as higher bandwidth can be achieved compared to reading. The *flush* pattern does not transfer any data besides the commands and thus will not cause any meaningful interference.

Buffer size We tested two buffer sizes. The small buffer has 1000 blocks (about 3 MiB) and will thus always fit in the LLC of both our test systems. The large buffer has 1 GiB, providing a very large target area. As seen in Section 5.4.3, this configuration causes a large number of cache misses due to bad cache locality.

We combined the parameters above into four configurations. For each configuration, we first did a baseline run without any load ("plain"), then a run with the *random* pattern and finally one with the *single* pattern. To detect random noise in the results, we ran this cycle ten times for each configuration. The other parameters of the four configurations are listed below:

default The maximum number of memory channels, no caching, and the small buffer size.

big buffer Same as *default*, but with the large buffer of 1 GiB.

caching Same as *big buffer*, but caching enabled as explained above.

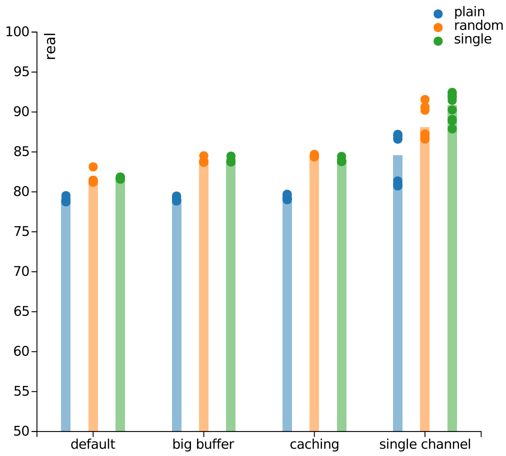
single channel Same as *default*, but with only a single memory channel.

5.5.1 PARSEC

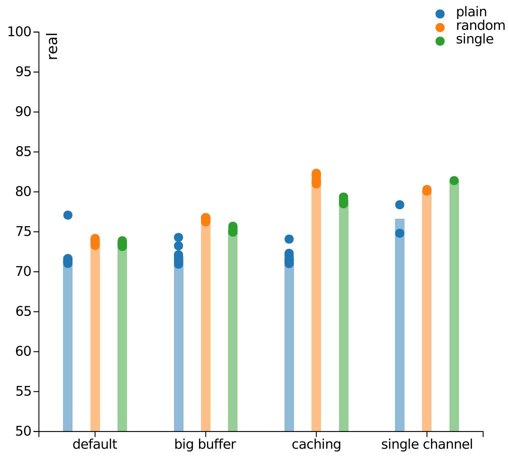
PARSEC is a benchmark suite with a focus on multithreaded applications. Its authors intended its workloads to be similar to real-world tasks [1]. Therefore, it provides a good basis for estimating the impact of high-bandwidth DMA transfers to applications.

The full benchmark results are in Appendix A. *Plain* is the benchmark result without any load, *random* is with the *random* pattern, and *single* is with the *single* pattern. The graph bars indicate averages; individual results are noted as dots. We measured wall-clock time from launch to exit of the benchmark programs; the graphs depict the time in seconds.

For most of the benchmarks on both systems, no difference is visible when adding DMA load. The interesting benchmarks that show a difference are `canneal`, `raytrace`, and `streamcluster`. The `swaptions` benchmark shows a lot of variation, which is not helpful for our analysis; it only shows a clear difference in a single configuration on Haswell.



(a) Sandy Bridge



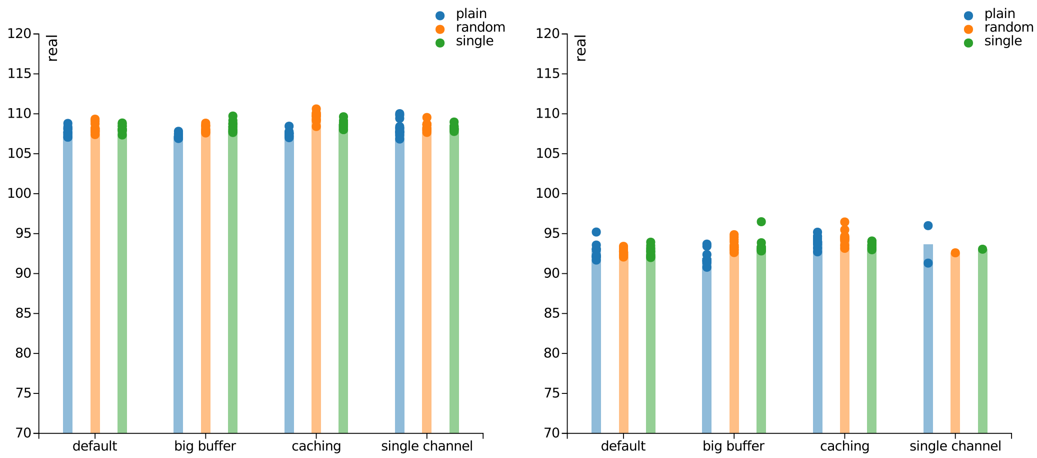
(b) Haswell

Figure 5.8: Results of the `canneal` workload.

The `canneal` benchmark shows an average difference of 3.2% on Sandy Bridge and 2.7% on Haswell when adding either DMA load pattern in the default configuration (Figure 5.8). When increasing the buffer size to 1 GiB, the difference increases to 6.0% on Sandy Bridge; keeping the target buffer in cache does not change this. As DMA transfers never go to the cache on our Sandy Bridge system, we expect the “caching” configuration to be equivalent to the “big buffer” one.

On Haswell with the large buffer, `canneal` gets slower by 6.5% with the *random* pattern and 4.6% with the *single* pattern. When adding caching, the difference increases significantly to 14% with the *random* pattern and 10% with the *single* pattern. This big difference to the results on Sandy Bridge confirms that DMA transfers work differently on the Haswell system due to Intel DDIO.

With only a single memory channel, the *random* and *single* pattern results diverge: On Sandy Bridge, the *random* pattern slows down `canneal` by 4.2% and the *single* pattern by 7.4%. On Haswell, the numbers are similar (4.7% and 6.3% respectively).



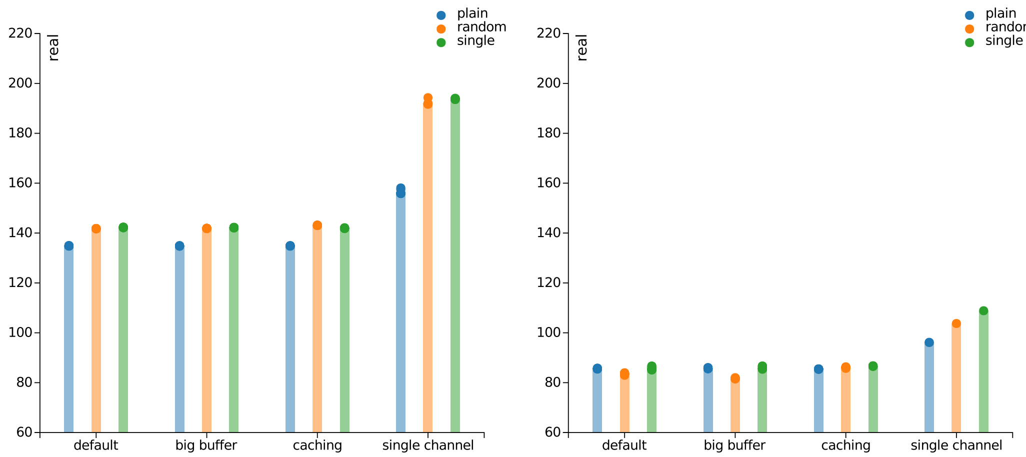
(a) Sandy Bridge (b) Haswell

Figure 5.9: Results of the `raytrace` workload.

Figure 5.9 shows the results with the `raytrace` workload. On both systems, it shows no significant difference in the default and single channel configurations.

On the Sandy Bridge system, a slim difference of 1.2% is visible with the large buffer and the *single* pattern, and of 2.0% with caching and the *random* pattern. On the Haswell system, the difference is slightly larger with about 1.8% for both patterns in the big buffer configuration. When adding caching, this difference disappears.

The `streamcluster` benchmark shown in Figure 5.10 has the biggest differences. On the Sandy Bridge system, there is a difference of 5.1% and of 5.4% for the *random* and *single* patterns in the first two configurations. With caching, this difference rises slightly by one percentage point for the *random* pattern. Limiting to a single memory channel increases the difference a lot to 23% for the *random* pattern and 24% for the *single* pattern. This suggests that the benchmark program is indeed slowed down because of limitations of the memory bus.



(a) Sandy Bridge (b) Haswell

Figure 5.10: Results of the `streamcluster` workload.

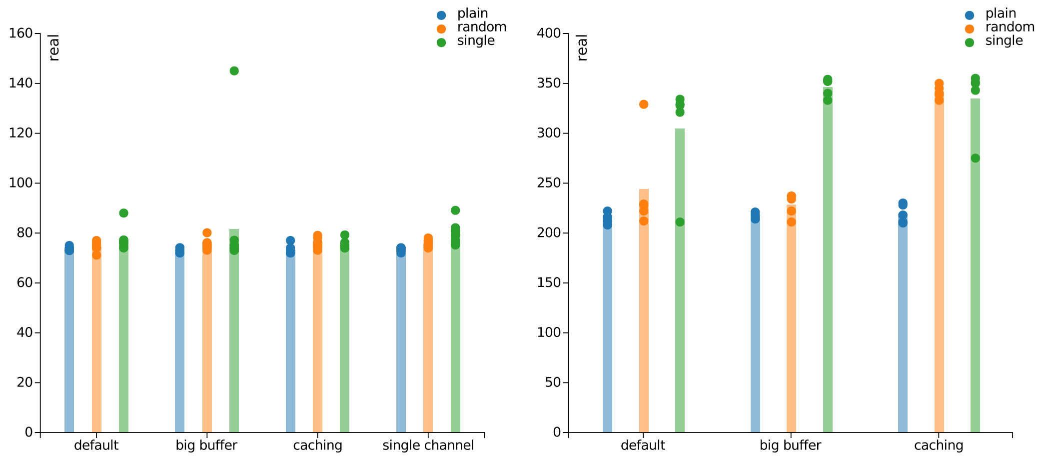
In contrast, on the Haswell system (Figure 5.10b), the results are counter to our expectations. While the *single* pattern shows almost no difference to the plain time, the *random* pattern actually improves the benchmark result by 2.5%. With the large buffer, the improvement doubles to 5.0%. With caching enabled, this effect disappears and the *random* and *single* patterns are slightly slower than the baseline. With only a single memory channel, we get results similar to the Sandy Bridge system. However, the *single* pattern has a bigger influence than the *random* pattern with 13.2% to 7.9%.

In conclusion, our results coincide with the evaluation of the PARSEC authors (see Section 2.4): The DMA load only affects the memory intensive workloads. With `streamcluster` suffering the most from interference, we were also able to roughly reproduce their relative ranking.

5.5.2 In-memory Databases

In addition to the PARSEC benchmark suite, we also tested interference of some in-memory databases. As multithreading was the main concern for the PARSEC benchmarks, they tend to be heavy on CPU usage, but not necessarily on memory accesses. In contrast, in-memory databases usually do very little processing on the CPU but are optimized to read and write large amounts of data from memory as quickly as possible. Consequently, we expect that any interference from DMA would be especially heavy on those applications.

Redis [16] is an in-memory datastore. It handles requests to store or



(a) Sandy Bridge (b) Haswell

Figure 5.11: Results of the `memtier-redis` benchmark. The default and single channel configurations are with the small buffer of 1000 blocks; big buffer and caching are with the 1 GiB buffer.

retrieve data synchronously on a single thread. The **memtier** benchmark program [14] acts as client to Redis, sending requests to store or retrieve data concurrently on multiple threads.

We ran Redis in its default configuration, but with persistence disabled. The memtier benchmark was set to do a fixed number of requests. The resulting runtimes are shown in Figure 5.11. For all configurations, the baseline run without DMA load is the fastest and the run with the *single* pattern the slowest.

On Sandy Bridge (Figure 5.11a), the difference between patterns is only small. However, the *single* pattern creates a lot of variation in the results, with a single run being almost two times slower than the median of the big buffer configuration. The biggest difference is visible with the reduced memory channel configuration: The *random* pattern slows down Redis by 2.8%, and the *single* pattern by 8.4%.

On Haswell (Figure 5.11b), the differences are more significant. We ran the benchmark with more worker threads there, increasing the overall command count and runtime. In the default configuration, the *random* pattern is 13% slower and the *single* pattern is 41% slower than the baseline. With the big buffer, these numbers are similar. When enabling caching in addition to the big buffer, the *random* pattern slows the benchmark down as much as the *single* pattern does. The caching does not have an effect on the *single* pattern though.

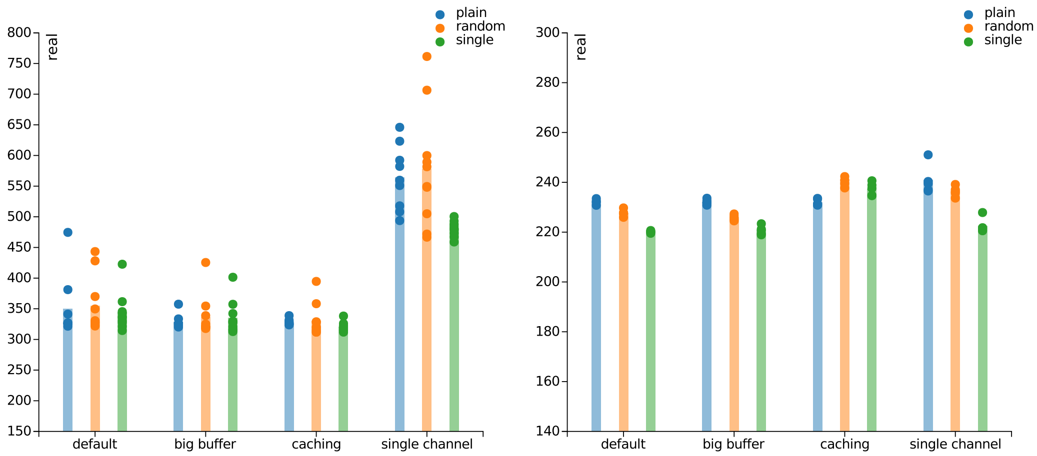
Silo [17] is another in-memory database. It is designed to be very efficient on modern multicore systems, achieving high memory throughput (see Section 2.5). Silo's source includes a couple of benchmarks; we used the TPC-C and YCSB benchmarks here.

The TPC-C benchmark, shown in Figure 5.12, shows a lot of noise on our Sandy Bridge system, but is very consistent on the Haswell system. Nevertheless, a curious pattern is visible on both systems: with the DMA load, Silo actually gets slightly faster than without DMA load in all configurations. On the Haswell system in the default configuration, Silo is 1.9% faster with the *random* pattern and 5.2% faster with the *single* pattern. The numbers are similar for the big buffer configuration. In the caching configuration, the benchmark is once again slightly slower with DMA load. With only a single memory channel, the plain and *random* times get slower compared to the default configuration, while the *single* time stays roughly at the same level, increasing the difference to 8.2%.

Due to the noise, the differences are a lot less visible on the Sandy Bridge system (Figure 5.12a). The clearest difference is in the single channel configuration, where the *single* pattern speeds up Silo by 17%.

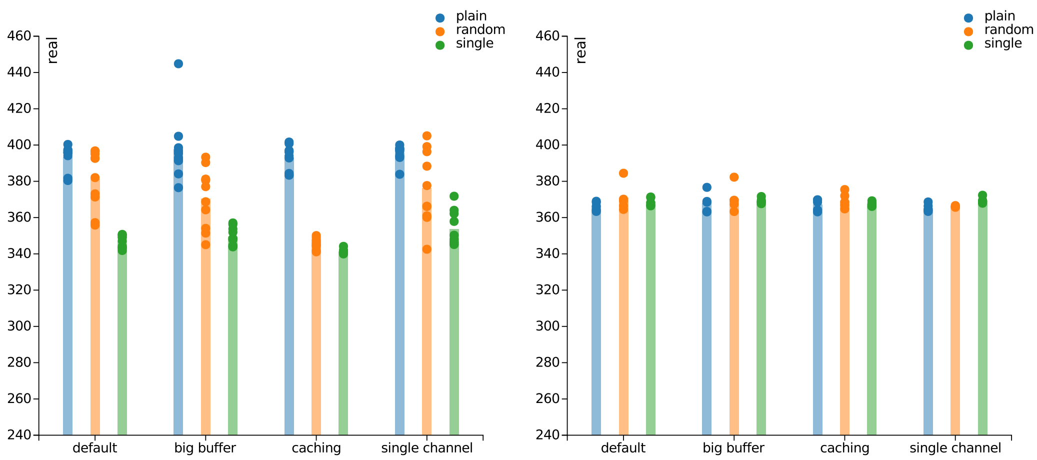
With the YCSB benchmark, the runtimes are all within 1% of each other on the Haswell system, showing little interference. In contrast, on the Sandy Bridge system the DMA load speeds up the benchmark again. In the default configuration, the *random* pattern makes the benchmark faster by 3.2% and the *single* pattern by 13%. Apart from noise, the results are similar with the big buffer and single channel configurations. When adding caching, the benchmark with the *random* pattern also drops down and is 14% faster than the baseline. This significant change is curious, as DMA transfers on the Sandy Bridge system do not go to the cache (see Section 2.1).

In conclusion, the in-memory databases indeed all show interference from our artificial DMA load. However, the databases often get faster with added load, which is the opposite of what we expected to see.



(a) Sandy Bridge (b) Haswell

Figure 5.12: Results of the `silo-tpcc` benchmark.



(a) Sandy Bridge (b) Haswell

Figure 5.13: Results of the `silo-ycsb` benchmark.

Chapter 6

Conclusion

In this work, we analyzed the influence of high-bandwidth DMA transfers on running programs. We chose solid-state drives based on NVMe to generate the synthetic DMA load and used raw NVMe commands to shape traffic precisely. This allowed us to simulate access patterns resembling both block storage devices (with the *random* pattern) and networking cards (with the *single* pattern). Making sure that the load generator is only limited by our hardware, we also modified the Linux kernel to eliminate a potential bottleneck in the NVMe driver; this did however not improve performance.

In our evaluation, we first analyzed the DMA bandwidth our load generator can achieve. Using two test systems, we showed the bandwidth differences of PCI Express 2.0 and 3.0, and presented the performance characteristics of our SSD at different levels of parallelism. We revealed that when aiming for maximum transfer bandwidth, only two parallel workers are currently required; this is likely to change with future devices though.

We used Uncore performance counters to further evaluate our synthetic DMA load, showing ways to estimate NVMe command counts as well as data transfer sizes without modifying the program issuing the commands. Additionally, we were able to see effects of Intel's DDIO technology, which uses the last-level cache as end point of DMA transfers instead of the main memory. When working with small amounts of data on an SSD, the main memory is never directly involved; this likely improves overall performance.

Finally, we used the PARSEC benchmark as well as the in-memory databases Redis and Silo to assess interference with our synthetic DMA load. We found that only processes doing very large and frequent memory transfers are affected at all. Our expectation that DMA transfers would only slow down processes was not confirmed; in fact, several benchmarks showed a significant speedup.

Our results show that, if contention of memory accesses is a concern—for example in the situation the MemGuard authors describe in their evaluation [2]—then high-bandwidth DMA transfers will indeed influence the running processes as well. A bandwidth reservation system thus should not only manage the memory bandwidth that applications use directly, but also the bandwidth used by I/O operations.

6.1 Future Work

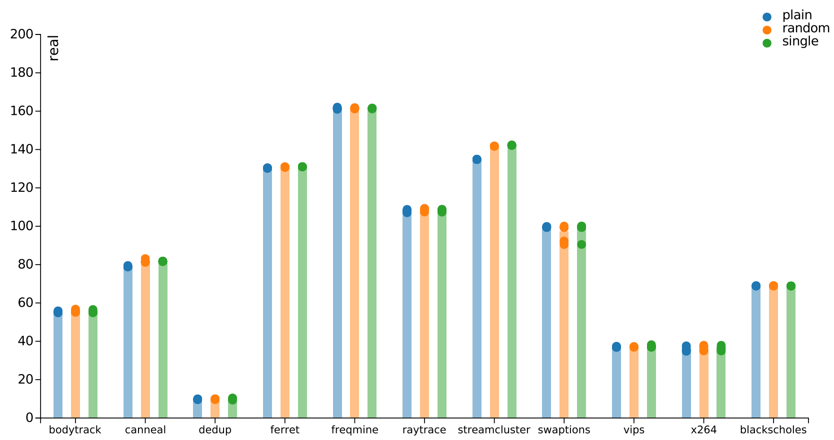
This work was mainly concerned with the generation of DMA load. Our results with the interference warrant further analysis of the causes. Performance counters located at the memory controller may help identifying the bottlenecks in the memory system. Especially the benchmarks speeding up with added DMA load need additional investigation.

In our evaluation, we only used a single solid-state drive per system. On our Haswell system, the DMA load was only limited by the read and write performance of the SSD. Additional tests with multiple DMA devices could explore limitations of DMA and the memory bus. Such a setup is interesting because of its resemblance with real-world servers receiving requests from a networking card and serving data from an SSD.

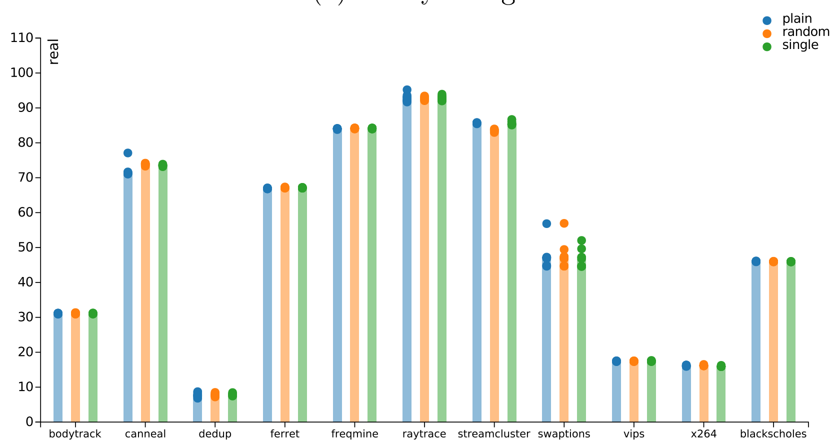
While one of our test systems had two sockets, we did not test any effects from non-uniform I/O access (NUIOA). For our experiments, we always pinned both the load generator and the benchmark to the socket the SSD was connected to. When the load generator runs on the other socket instead, the DMA I/O first has to traverse a QPI link, which may be another bottleneck causing interference. Additionally, the Intel DDIO technology does not work in this setup [5].

Appendix A

Full PARSEC Results

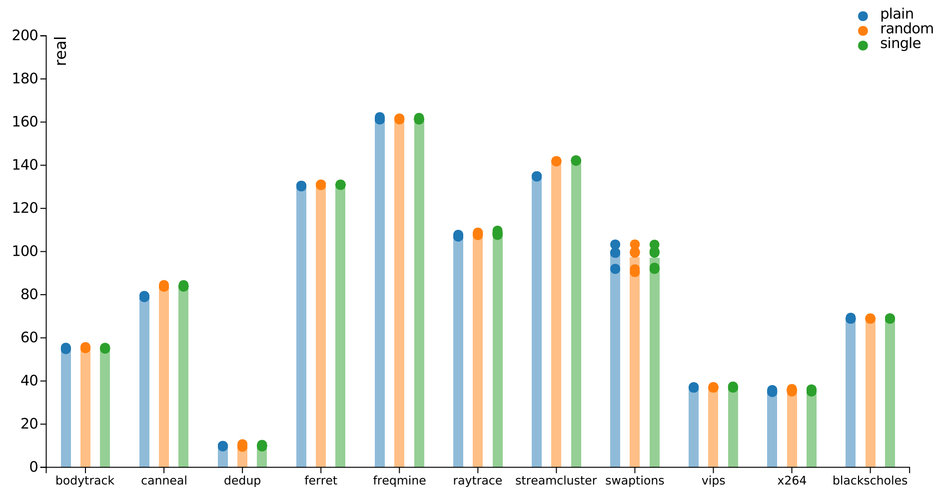


(a) Sandy Bridge

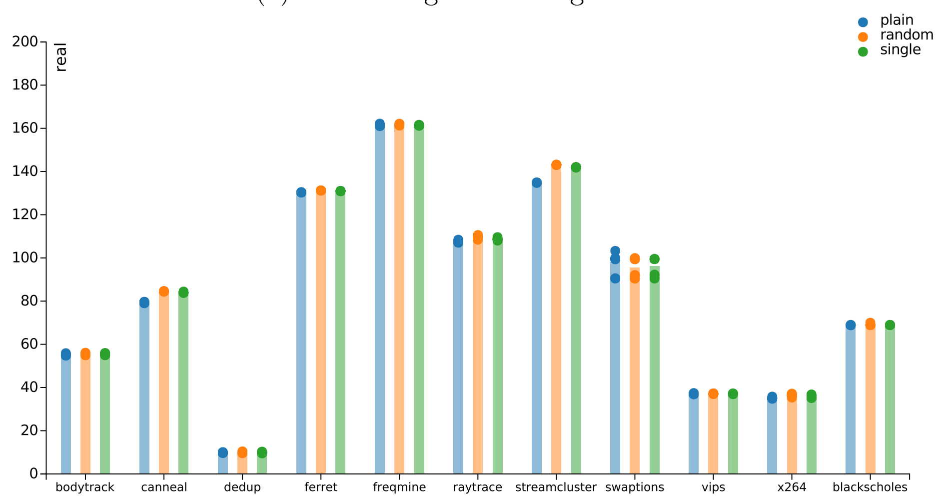


(b) Haswell

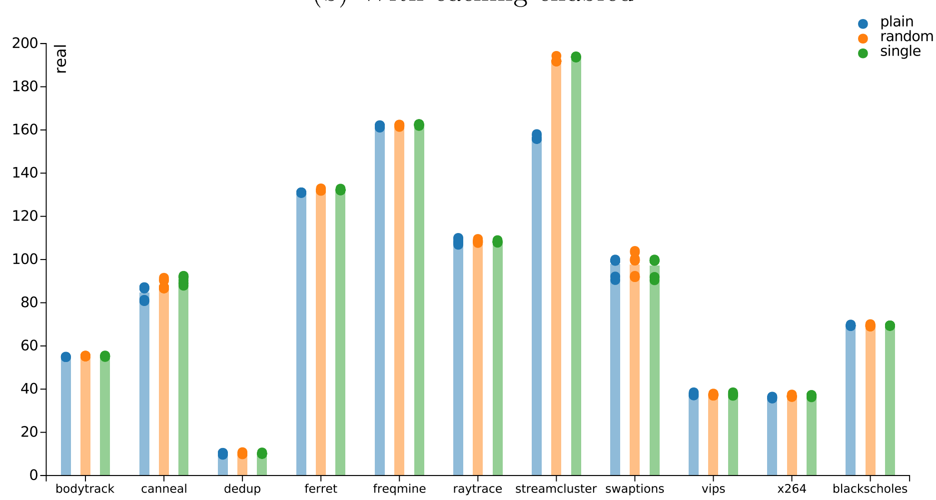
Figure A.1: PARSEC benchmark results in the default configuration.



(a) With a big 1 GiB target buffer

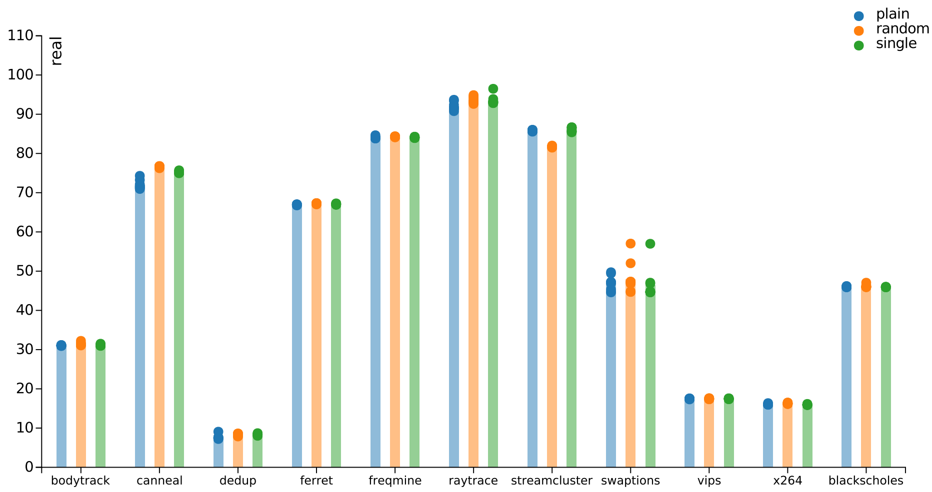


(b) With caching enabled

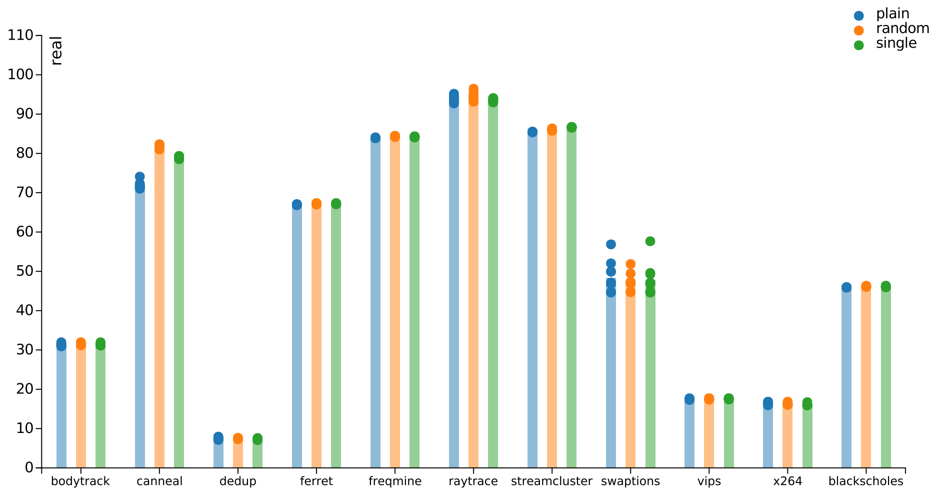


(c) With only a single memory channel

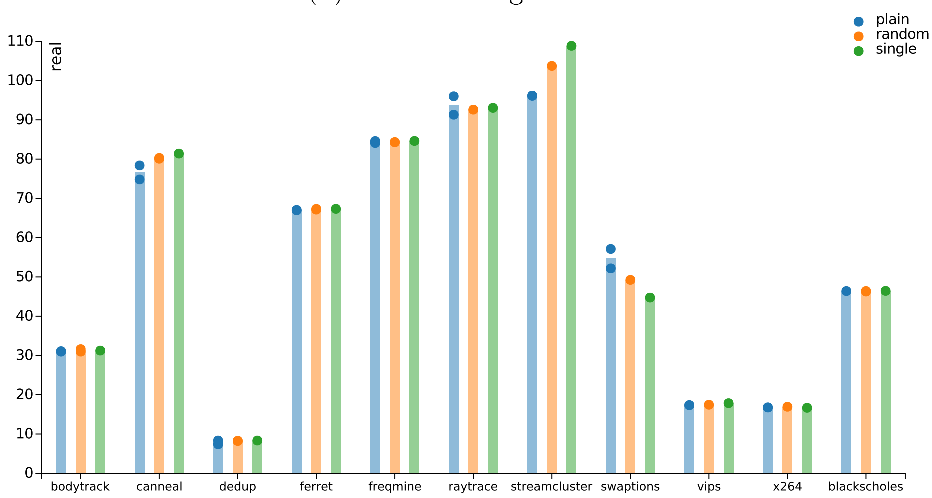
Figure A.2: PARSEC benchmark results on the Sandy Bridge system.



(a) With a big 1 GiB target buffer



(b) With caching enabled



(c) With only a single memory channel

Figure A.3: PARSEC benchmark results on the Haswell system.

Bibliography

[1] Christian Bienia. “Benchmarking Modern Multiprocessors.” PhD thesis. Princeton University, Jan. 2011.

[2] Marco Caccamo et al. “MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms.” In: RTAS ’13. 2013.

[3] Intel. *Data Direct I/O Technology.* URL: `https://www-ssl.intel.com/content/www/us/en/io/direct-data-i-o.html`.

[4] Intel. *Intel® Core^{TM} i7-4790K Processor.* URL: `http://ark.intel.com/products/80807/`.

[5] Intel. *Intel® Data Direct I/O Technology (Intel® DDIO): A Primer.* Tech. rep. Feb. 2012. URL: `http://www.intel.de/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf`.

[6] Intel. *Intel® Xeon® Processor E3-1230.* URL: `http://ark.intel.com/products/52271/`.

[7] Intel. *Intel® Xeon® Processor E5 and E7 v3 Family Uncore Performance Monitoring Reference Manual.* June 2015. URL: `https://www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-e5-v3-uncore-performance-monitoring.html`.

[8] Intel. *Intel® Xeon® Processor E5-2630v3.* URL: `http://ark.intel.com/products/83356/`.

[9] Intel. *Intel® Xeon® Processor E7-8890 v3.* URL: `http://ark.intel.com/products/84685/`.

[10] Intel. *nvme-cli.* URL: `https://github.com/linux-nvme/nvme-cli`.

[11] Intel. *Performance Counter Monitor.* Dec. 18, 2014. URL: `https://software.intel.com/en-us/articles/intel-performance-counter-monitor/`.

[12] *NVM Express 1.2 Specification*. 2014-11-03. URL: http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_2-Gold-20141209.pdf.

[13] *READV(2) Linux Programmer's Manual*. Aug. 19, 2014.

[14] RedisLabs. *Memtier Benchmark*. URL: https://github.com/RedisLabs/memtier_benchmark.

[15] Samsung. *NVMe SSD*. URL: http://www.samsung.com/global/business/semiconductor/product/flash-ssd/nvmessd.

[16] Salvatore Sanfilippo. *Redis*. URL: http://redis.io/.

[17] Stephen Tu. *Silo*. URL: https://github.com/stephentu/silo.

[18] Stephen Tu et al. "Speedy Transactions in Multicore In-memory Databases." In: SOSP '13. 2013.

[19] Lukas Werling. *nvme-memload*. URL: https://github.com/lluchs/nvme-memload.