1 Introduction

High performance computing is to process data and do calculation on that data at high speed. The amount of data that is being generated is increasing every year. The data that is being generated on the biggest projects exceeds the capacity of traditional memory. Traditional memory in this context is dynamic random access memory(DRAM). The way this is solved is to make use of the hard drive which is making the processing and calculation of data significant slower. traditional

A possible solution to this problem could be to use NVDIMM. This is a persistent random-access memory[ref], the difference between DRAM and NVDIMM is that data wont be deleted from NVDIMM when the computer shuts down. It has a faster load/store than a hard drive, but is ~~also~~ slower than traditional DRAM. Since NVDIMM is persistent it can also be used as a storage device the same way as traditional hard drives.

This thesis will explore several things. First is to explore how fast NVDIMM is compared to DRAM when ~~DRAM and~~ NVDIMM ~~are~~ is working alone. The speed of NVDIMM and DRAM will also be measured when they are working simultaneously, they will be made to work simultaneously using the openMP library. The thesis will also explore how the persistent memory performs in experiments that simulates what persistent memory might be used for in the real world. The goal is to come up with a set of advi~~ses~~ce that may help other people who are using NVDIMM in their projects.

The rest of this chapter will describe what persistent memory is and how it will be used in this thesis. There will also a explanation of the challenges and advantages of persistent memory such as cost, durability and persistent memory leak. ~~Persistent memory leak is a new problem that many programmers have not encountered before.~~ outline of thesis research questions

1.1 What is persistent memory

Persistent memory[**Rudoff2**] is a non-volatile storage memory[**Mahmut**] that is byte-addressable and has speed close to that of DRAM. ~~DRAM stands for direct random-access memory, this~~ which is a volatile storage system that will lose all its data when the computer is shut down or restarted. Applications and data used by the CPU are temporarily loaded into memory from a hard drive in order to reduce latency and increase bandwidth. Persistent memory is another layer between the CPU and the disk. The data the CPU has the most use for is stored in

DRAM

the L1-L3 caches. When the cache is full the data needs to be evicted the evicted data will be sent back to the memory. If the data usage of the program is so large that it exceeds the memory available on the computer then the computer will start using virtual memory on the disk which is a lot slower then DRAM. The reason virtual memory is so slow is because one must do an I/O block to read and write to disk which takes time. Persistent memory is a layer between the DRAM and the disk in which the CPU can access directly just like it would do a normal DRAM. Figure 1 illustrates where in the hierarchy the persistent memory is placed.

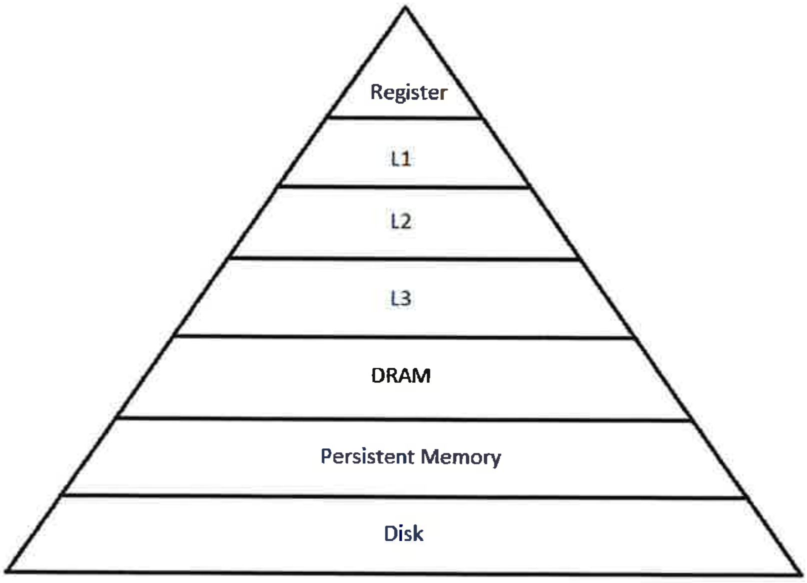

Figure 1: Persistent memory becomes a new level between DRAM and Disk[**optane**]

1.2 Challenges

1.2.1 Security

One Another challenge when it comes to persistent memory is security and privacy[**Badam**]. When an encryption key is used to unlock files, the key is stored in the memory. If it is DRAM then the key will disappear when the computer is shut down, but if it is stored in persistent memory it will persist and remain there until it is deleted. It will be very easy for someone to extract data from persistent memory if the person is able to gain access to it. The same concern also applies to personal information that is being stored in the persistent memory. When developing applications that will use persistent memory and handling sensitive information the programmer needs to remember that data he puts in persistent memory will stay there until it is deleted. It's also worth mentioning that when the data is deleted or deallocated in the memory, the OS makes the space available to another application. The data is only deleted when its overwritten.

1.2.2 Durability

Another challenge for persistent memory is durability. While persistent memory behaves more like DRAM it still has a considerably shorter lifespan than DRAM[**Badam**]. This is because persistent memory can only write data to a certain amount of time to a region before the region can no longer hold any data reliably. While the storage capacity of persistent memory has increased, so has the bit error rate increased even more. The solution to this is to either have the hardware mask all the regions with bit error from the software or have the hardware expose them to the software and let the software handle the rest. There is also the possibility of letting the hardware and software work together in order to mask regions with bit errors. This will expand the lifespan of persistent memory, but the challenge for hardware producers is to come up with new technologies that can increase the durability of the persistent memory.

1.2.3 Persistent memory leaks

A common problem one might have when programming in C ~~are~~ Concerns memory leaks. When the application is using normal DRAM, the memory consumed by the application can be freed just by restarting the application. If the application is using persistent memory on the other hand, then the memory consumed by the application will remain consumed even after the program has been restarted.[**Volos**][**Swanson**] Memory leaks will persist a shutdown just like persistent memory will. The technology must ensure that the memory occupied on the persistent memory can be tracked down to the application that allocated the memory. By doing this it is possible to track down and remove memory leak for the persistent memory.

1.3 Advantages

1.3.1 Cost

One of the biggest shortcomings of DRAM is that it is expensive. When the amount of memory increases in a system the cost of DRAM scales nonlinearly.[**Badam**] This has led to memory becoming a bottleneck in servers that runs programs where a lot of memory is needed. Persistent memory is more scalable in terms of cost then compared to DRAM.

This will enable servers to have more storage capacity that can be read at almost the same speed as DRAM.

1.3.2 Capacity, Larger physical memory

The memory capacity of persistent memory represents a drastic increase in the size of memory that is available to the user. The biggest size of a DRAM memory module that can be bought today is 64 Gigabytes. Intel has announced a new persistent memory called Intel Optane DC[**side1**]. This is a persistent memory that is compatible with a DDR4 socket and each memory module can contain 512 Gigabytes of memory. The bigger memory size will make it possible to keep more of the data the user is working on in the memory and reduce the traffic between the memory and the hard drive.

reference ???

1.3.3 Byte addressable, low latency

Since the persistent memory is connected to a DRAM slot it is also byte-addressable. When a program accesses traditional storage it must wait for the OS to do an I/O block in order to get access which takes a long time and read/write are done in 4 kB blocks. With persistent memory the program can skip the I/O block and access the data directly, this will dramatically decrease the latency. Typical latency when using DRAM would be around 10^{-7}[**lerebok**] while Intel has measured their Optane DC to have a latency of 4.1 ms[**optane2**]. Persistent memory is 40 times slower than normal DRAM, but it is still a lot better than SSD that can have 80 ms latency.

unit?

1.4 The rest of thesis

In Chapter two there will be an explanation of how to program with NVDIMM. There will be an explanation of some of the different typeS of libraries that exist. How to set up a memory pool that will be used by

Have you introduced the concept of NVDIMM?

the programmer and an explanation of method that will be most used by the programmer.

In chapter three will be ~~contain~~ several benchmarks that will show the performance of DRAM and NVDIMM when whey are working alone and when they are working simultaneously. There will also be an comparison to see if DRAM and NVDIMM are working faster or slower together when compared to DRAM working alone.

Chapter four ~~there~~ will be about a scenario that a programmer can encounter. That is when the data exceeds the the total capacity of DRAM and the programmer is forced to split the data in two part and place the part that exceeds DRAM on NVDIMM. There will be a ~~a~~ formula that calculates how many threads should be reallocated to work on NVDIMM.

Chapter five will ~~also~~ be about a different scenario. There will be a program with two parts. Part one is data is calculated on DRAM by a group of threads. Part two will also have a group of threads that will first transfer the calculated data from DRAM to NVDIMM and analyze the data on NVDIMM.

this paragraph must be improved

Chapter six will contain a summary and a conclusion.

1.5 Research questions

This thesis will try to answer the following research questions.

- What is the data transfer speed of NVDIMM compared to DRAM?
- In an competitive competition how ~~does~~ NVDIMM and DRAM affect each other?
- When the size of the data is higher than the capacity of the DRAM. How much data should be transferred to NVDIMM? How many threads should be allocated to work on the data on NVDIMM?
- While DRAM is working on a task, is it possible for NVDIMM to be working on a different type of task?

1 Basic programming with NVDIMM

1.1 Introduction~~s~~

This chapter will be about how to program with NVDIMM. In order to create code that is using NVDIMM the programmer must chose what library to use and there is many libraries to pick from and all are made for different purpose and have many different types of methods. This chapter exist so other can start on the right track and quickly learn how to use NVDIMM.

choose

The libraries and methods described in this chapter are relevant for NVDIMM devices that supports the libraries created by Intel at pmem.io.

1.2 Different types of libraries

1.2.1 Libpmemobj

meaning of "torn"?

Libpmemobj[19] allows objects to be stored in persistent memory without being torn by interruptions. The objects in question are not class objects one finds in C++, but instead they are variable-sized blocks of data that fall under the term object storage. The object has an object ID that is independent when it comes to location. The changes or updates to these objects are atomic because the library have transactions to make this happen. This library can be used for multithreading and ~~are~~ have been optimized for scaling when it comes to multithreading. The main author also mention that the C++ version of this library is the cleanest and least prone to error compared to all the other libraries[10]. He therefore recommends that programmers should start using this library if they are new to persistent memory programming.

what is object storage?

1.2.2 libpmemblk and libpmemlog

L

These libraries are made for specific cases. libpmemblk[17] is used for handling large arrays of persistent memory blocks. The blocks must be larger than 512 bytes in order to work. This library is useful if the program is made to manage a block cache. Libpmemlog[18] is used to append log files. If the program logs a lot of data, it might be better to use libpmemlog in order to avoid going through the traditional file system where most of the time would be spent waiting. Both libraries explained before?

are built upon the libpmem library, but unlike libpmem their updates can't be torn because of interruptions.

1.3 Creating pmempool

Before NVDIMM can be used, the user must create whats called a memory pool on the NVDIMM. The NVDIMM has several modes, in order to be able to create a memory pool the mode must be set to fsdax. On a server this must be done by the system administrator. To see what mode the NVDIMM is in can be done by the command ndctl-list. A program called pmempool must also be installed on the server, it is this program that will create the memory pool. The command used for creating the memory pool for this thesis is

```
pmempool create --layout Layout_name --size=170G obj
   pool.obj
```

The layout is a string stored in the memory pool. When a program accesses a memory pool it needs to send a string that matches the string in the memory pool in order to use it. The user can specify the size of the memory pool, if size is not specified the pmempool will create a pool with the lowest size allowed. There are three different types of memory pool to choose from, they are obj, log and blk. Which type of memory pool to use depends on which type of library is used in the program. In this thesis the libpmemobj library was used and that is why obj was used in the creating of memory pool. ~~For~~ log and blk are for the libraries libpmemlog and libpmemblk. The last part of the command line is the name and file address of the memory pool.

1.4 Methods

In this project it is the libpmemobj library that will be used. Below is a short description of all the methods that will be used in the thesis.

1.4.1 Open memory pool

The memory pool uses a pointer called PMEMobjpool. The memory pool is opened by using the method pmemobj_open that needs two arguments. The first argument is the path to the memory pool created in chapter 1.3. The second argument is a text string that identifies what data belongs to what program.

```
PMEMobjpool *pop = pmemobj_open(path, LAYOUT_NAME);
```

1.4.2 Declaring an array

When the programmer wants to declare an array, the follow command must be used.

```
TOID(Type) Array_name;
```

Type is the data type the programmer wants to use and the name is the name of the array.

1.4.3 Allocating array

When allocating the array the method called POBJ_ALLOC is used. The method have six arguements. The first argument is the memory pool created in chapter 1.4.1. The second argument is the the array the user want to allocate memory. Third argument is the data type and the fourth argument is the array length in bytes. The last two arguments is irrelevant in this context and can be given the value NULL.

```
//Allocating of the array
POBJ_ALLOC(Memory_pool, &Array_name, Type, sizeof(double)
    * ARRAY_LENGTH, NULL, NULL);
//Deallocating of the array
POBJ_FREE(&Array_name);
```

When deallocating the array the method POBJ_FREE that only need the array the user want to deallocate as argument.

← rephrase this sentence

1.4.4 Read/Write to array

Reading from the array is done by using the method D_RO that must have the array the user want to read from as argument. The method also uses square brackets after the argument that need the index of the element in the array the user want to read. The method D_RW is used when reading to the array. The use of this method is identical to D_RO.

```
//Reading an array variable.
var = D_RO(Array_name)[index];
```

What are "D_RO" and "D_RW"?

```
//Writing to an array variable.
D_RW(Array_name)[index] = var;
```

1.5 Coding example

This is an example on how to use ~~the~~ pmemobj library. The example will find the average of an array where the array is replaced with an NVDIMM array. The purpose is to show how easy it is to code with NVDIMM by having all the relevant methods in an easy example. The way of using the a NVDIMM library is a lot similar to using ordinary arrays. Once the programmer have chosen what NVDIMM library to use and included the library in the code the memory pool must be opened. The first thing the code need s is the path to the memory pool and a layout, which is a string that identifies the pool that the user can choose what it will be. This can either be a command line argument the user gives when starting the program or it can be hard coded into the code. This is what has been done ~~at~~ in listing 1 at line 5 and 11. These two strings are used as arguments when initiating the pool at line 14-15. The initiation is also followed up with an if-sentence at line 16-19 to check that the memory pool has been successfully created. If it has not the program will print out an error message and exit the program.

Next is to create a NVDIMM array, this is what happens at line 21. The NVDIMM array pointer is a void pointer that is casted to a double pointer. The array gets initiated at line 22. The method used is called POBJ_ALLOC, this method are similar to malloc for DRAM. The method have six arguments, the first argument is what memory pool the array will be assigned to. The second argument is the ~~the~~ address of the pointer. Third argument is the type of the elements in the array. The fourth argument is the length of the array, that is the size of type multiplied with the number of elements in array. The last two arguments are set to NULL.

When writing to an NVDIMM array the programmer must use the method called D_RW. It only have one argument which is the name of the array. It is followed up with a pair square brackets that contains the index of the element the programmer wants to write to, an example can be found at line 25.

D_RO is the name of the method one must use to read an element from an NVDIMM array. This method also have one argument which is the name of the array and the square brackets contains the index of the element that will be read. Line 29 is an example of how to add the value of an element in a NVDIMM array to a variable.

In order to deallocate the a NVDIMM array one must use the method POBJ_FREE and the only argument needed is the address of the NVDIMM pointer, an example can be found in line 34. If the programmer forgets to free up the NVDIMM array there will be a permanent memory leak that will last even after the program have stopped running. In order to get rid of the memory leak one must delete the memory pool and create a new one.

Lastly one must close the memory pool before the program is terminated. This is done with pmemobj_close, it only have the pointer for the memory pool as argument. Line 35 shows how to close the memory pool.

Listing 1: Example of coding with NVDIMM

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <libpmemobj.h>
4
5  POBJ_LAYOUT_BEGIN(array);
6  POBJ_LAYOUT_TOID(array, double);
7  POBJ_LAYOUT_END(array);
8
9  #define ARRAY_LENGTH 1000
10 #define LAYOUT_NAME "my_layout"
11 int main(int argc, char *argv[])
12 {
13   double average = 0.0;
14   int i;
15   //The path for the memory pool.
16   const char path[] = "/mnt/pmem0-xfs/pool.obj";
17
18   /* create the pmemobj pool or open it if it already
        exists */
19   PMEMobjpool *pop;
20   pop = pmemobj_open(path, LAYOUT_NAME);
21   if (pop == NULL) {
22     perror(path);
23     exit(1);
24   }
25   //Creation of NVDIMM array.
26   TOID(double) nvm_array;
```

```
  POBJ_ALLOC(pop, &nvm_array, double, sizeof(double) *
     ARRAY_LENGTH, NULL, NULL);
  //Writing to the array.
  for(i=0;i<ARRAY_LENGTH;i++){
    D_RW(nvm_array)[i] = i;
  }
  //Reading from the NVDIMM array.
  for(i=0;i<ARRAY_LENGTH;i++){
    average += D_RO(nvm_array)[i];
  }
  average = average / ARRAY_LENGTH;
  printf("%f\n", average);

  POBJ_FREE(&nvm_array);
  pmemobj_close(pop);
  return 0;
}
```

1 Benchmarks

1.1 Introduction

Persistence memory is slower than DRAM. But there is not much information on how much slower the NVDIMM is in comparison to DRAM. This chapter will test the performance of NVDIMM when it work s alone and when it works simultaneously with DRAM. The results will be presented with graphs and tables that will show the difference in performance. The chapter will start off with using the STREAM[1] benchmark in to find the performance of DRAM. The NVDIMM will also be tested with a STREAM benchmark that have been modified by the author of this thesis. Three original benchmarks will also test the NVDIMM when it works simultaneously with DRAM. In the first benchmark DRAM will copy an array from ~~DRAM~~-DRAM while NVDIMM copy and array from ~~NVDIMM~~-NVDIMM. In the second benchmark DRAM will copy from ~~DRAM~~-DRAM while NVDIMM will copy from ~~NVDIMM~~-DRAM. In the last benchmark DRAM will copy from DRAM-DRAM while NVDIMM will copy from DRAM-NVDIMM.

rephrase

1.1.1 Hardware

All the benchmarks

The program have been tested on a server with the following hardware.
Motherboard: Supermicro X11DPU-Z+
CPU: Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz, 16 cores
DRAM: Samsung RDIMM, 2666 MT/s.
NVDIMM: Micron Technology NV-DIMM , 2933 MT/s

The server have two CPU s both CPU s have twelve memory slots each. Each CPU have six channels. There are one DRAM and one NVDIMM sharing one channel. The compiler used to compile the code is gcc (Ubuntu 7.5.0-3ubuntu1 18.04) 7.5.0. The code have been optimized to level two. (O2)

All of the benchmarks have been tested on socket two. This is to avoid disturbances as much as possible since most of the other processes are running on socket one.

1.2 STREAM DRAM

The STREAM[1] benchmark is a synthetic and simple benchmark that is designed to measure bandwidth in MB/s. This benchmark is seen as the standard for measuring memory bandwidth and has not been modified in any way after it was downloaded from the creators web-sites. The benchmark test memory bandwidth by running four different tests. The first one test is copy where the elements in one array is copied to another array. The second test is called scale where each element are multiplied with a constant and the result is placed in a second array, the index of the element in the first array and the result in the second array is the same. Third test is add where the elements from two different arrays with the same index are added together and place in a third array where the index is the same as in the two other arrays. Last test is the triad where the one array is multiplied with a constant then added together with a second array and then placed in a third array.

The DRAM Stream benchmark run the test 32 times and only on one socket, every times it restart with one extra thread is added. The CPU has 16 cores and when the thread number surpass that number it starts using the hyper thread on the same core. The Linux program numactl is also used to manage the number of threads and what socket the benchmark is allowed to use.

The result shown in figure 1 and table 1 is what was expected, adding more threads in beginning will give a big increase in transfer speed. But at thread 5 the~~re~~ gains in transfer speed will start to diminish and at thread 11 there will be very little increase in transfer speed when adding more threads. This means that it might be possible to allocate five of the sixteen threads to work on NVDIMM and not loose a significant amount of performance for the eleven remaining threads that are still working on DRAM.

After sixteen threads the benchmark start to use the hyper-threads. There is a 10,000 MB/s decrease when the benchmark start to use the hyper-threads. When more and more hyper-threads are added the bandwidth will increase until it is almost at the same level when there was only sixteen threads. 32 threads have 2,000 MB/s lower bandwidth than sixteen threads.

1.3 STREAM NVDIMM

The stream NVDIMM benchmark measure the memory speed of the NVDIMM. This benchmark is the same as the STREAM benchmark has been descibed in chapter 1.2. The different is that the memory type have been changed from DRAM to NVDIMM. The code shown in

State that you have extended the original STREAM benchmark!

Threads	Copy	Scale	Add	Triad
1	11673.5	12180.6	12799.4	12745.3
2	22995.1	23892.4	24637.1	24807.8
3	33554.9	34206.9	36248.9	36070.7
4	42917.3	44315.0	46759.0	46333.7
5	50260.9	50853.1	55574.9	55784.3
6	53612.5	54305.3	60174.4	60129.4
7	56100.8	56671.2	63670.6	63425.1
8	58554.6	58888.6	66348.1	66607.5
9	60491.7	60947.7	69059.1	68923.9
10	62242.2	62368.3	71335.2	70900.6
11	64257.1	64270.1	72604.0	71854.1
12	64890.3	65611.6	73973.6	73866.7
13	65648.8	65805.9	74285.3	74204.8
14	65606.5	65943.6	74128.9	74158.9
15	65665.5	65897.7	73918.8	74199.9
16	65509.8	65770.4	73721.2	74312.2
17	55365.3	58578.1	63624.8	64728.6
18	57104.2	59481.5	65404.0	62472.2
19	59160.6	59279.3	64749.4	68522.2
20	61328.6	61453.3	67489.5	69020.7
21	62290.7	62453.6	68987.6	69178.2
22	63091.2	63146.4	70173.6	70239.2
23	63737.2	63887.6	71195.5	71235.8
24	64056.6	64108.0	71629.1	71669.4
25	64601.7	64685.7	72282.9	72167.3
26	64824.5	64850.8	72729.4	72623.9
27	64706.3	64890.3	72444.1	72525.6
28	64654.6	64743.1	72586.7	72656.9
29	64827.0	64750.6	72505.7	72481.2
30	64589.2	64659.6	72453.0	72472.8
31	64703.8	64714.4	72531.8	72356.7
32	64610.4	64721.9	72459.3	72212.9

Table 1: DRAM Stream, 100,000,000 elements

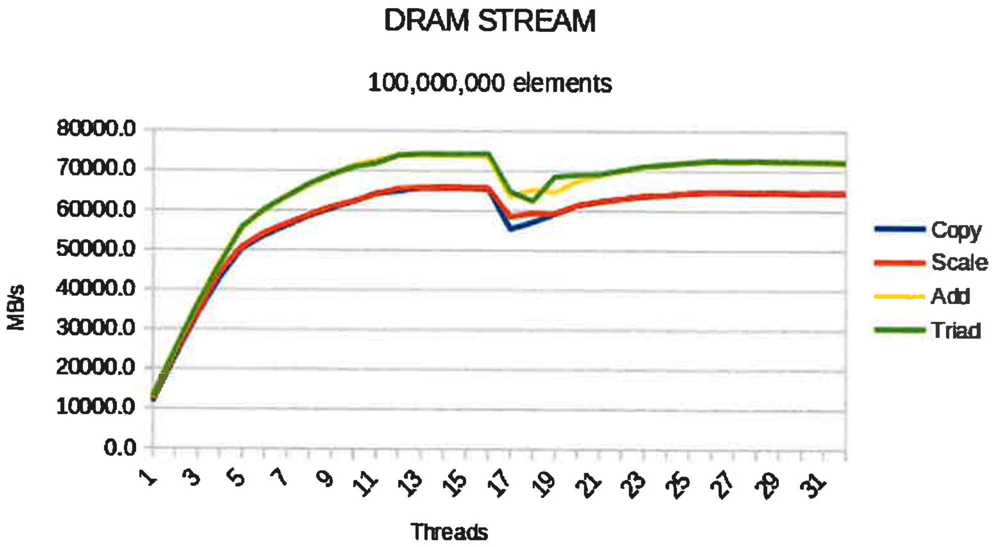

Figure 1: DRAM Stream, 100,000,000 elements

listing 1 is part of the original code that have been removed from the code.

Listing 1: Original STREAM benchmark code at line 175-181.

```
#ifndef STREAM_TYPE
#define STREAM_TYPE double
#endif

static STREAM_TYPE a[STREAM_ARRAY_SIZE+OFFSET],
                   b[STREAM_ARRAY_SIZE+OFFSET],
                   c[STREAM_ARRAY_SIZE+OFFSET];
```

It has been replaced with the code shown in listing 2. The code starts by opening the memory pool at line 21-27. The code will use a method called initiate at line 28 that will initiate the three arrays. Once this is done the code will continue executing the rest of the STREAM benchmark code which is identical to the original STREAM benchmark code.

Listing 2: Code that has replaced original code.

```
1 PMEMobjpool *pop;
2 POBJ_LAYOUT_BEGIN(array);
3 POBJ_LAYOUT_TOID(array, double);
4 POBJ_LAYOUT_END(array);
```

Don't you need D_RO and D_RW?

```
//Declearing the arrays
TOID(double) a;
TOID(double) b;
TOID(double) c;

void initiate()
{
  //Initiating the arrays.
  POBJ_ALLOC(pop, &a, double,
      (STREAM_ARRAY_SIZE+OFFSET)*sizeof(STREAM_TYPE), NULL,
      NULL);
  POBJ_ALLOC(pop, &b, double,
      (STREAM_ARRAY_SIZE+OFFSET)*sizeof(STREAM_TYPE), NULL,
      NULL);
  POBJ_ALLOC(pop, &c, double,
      (STREAM_ARRAY_SIZE+OFFSET)*sizeof(STREAM_TYPE), NULL,
      NULL);
}

int main()
{
  const char path[] = "/mnt/pmem0-xfs/pool.obj";
  pop = pmemobj_create(path, LAYOUT_NAME, 10737418240,
      0666);
  if (pop == NULL)
    pop = pmemobj_open(path, LAYOUT_NAME);
  if (pop == NULL) {
    perror(path);
    exit(1);
  }
  initiate();
  //The rest of the STREAM benchmark after this.
}
```

The result on the NVD~~V~~IMM Stream benchmark shown in figure 2 and table 2 is very different from the DRAM Stream benchmark. The ~~the~~ DRAM Stream benchmark had a steep increase in bandwidth in the beginning that started to taper off at thread five and almost no increase from thread eleven. The NVDIMM have a more linear increase in bandwidth when the threads are increased from one thread~~s~~ towards sixteen threads. This might be because the speed of one thread is half the bandwidth of one thread in the DRAM Stream benchmark. The max bandwidth reached by the NVDIMM Stream benchmark is 51,273

OK

MB/s and the DRAM Stream benchmark starts to taper off at around 55,000 MB/s. The NVDIMM Stream benchmark never reaches a speed high enough so it can start to taper off and therefore it look more like linear increase in bandwidth.

Thread seventeen and after are the hyper-threads and at thread seventeen there is a 15,000 MB/s decrease before the bandwidth overall start to increase with more hyper-threads added. The bandwidth swings up and down a lot. There is no evidence clear explanation on why the bandwidth fluctuates so much.

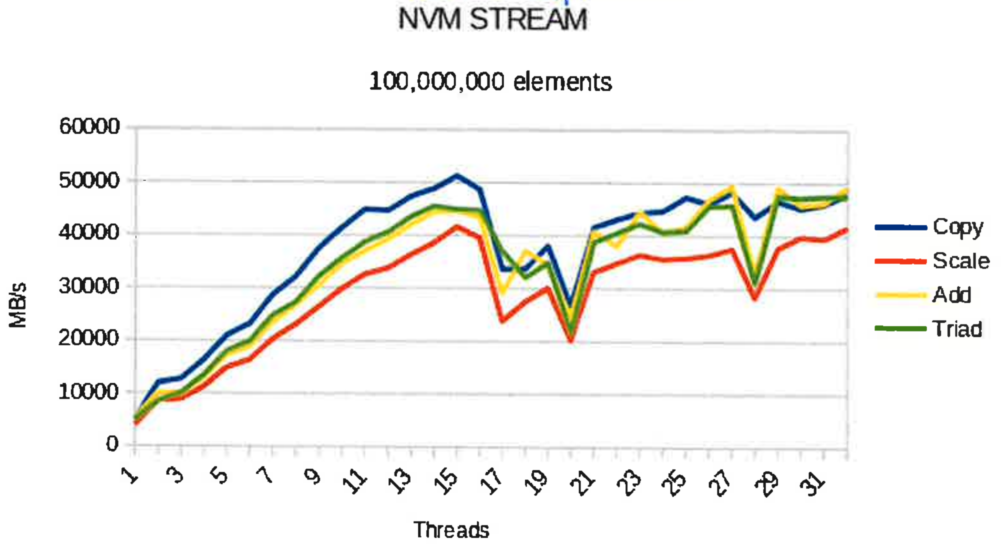

Figure 2: NVDIMM Stream, 100,000,000 elements

1.4 Competition benchmarks

This chapter are Section is about three ~~different~~ new benchmarks. In the first benchmark data will be copied for from a DRAM array to another DRAM array and from a NVDIMM array to another NVDIMM array simultaneously. In the second benchmark data will be copied from DRAM-DRAM arrays and from DRAM-NVDIMM arrays simultaneously. In the last benchmark data will be copied from DRAM-DRAM and NVDIMM-DRAM simultaneously.

The purpose of these benchmarks is to get an understanding of how performance will be affected when different threads generates traffic from DRAM and NVDIMM simultaneously. That is why there is three

Threads	Copy	Scale	Add	Triad
1	5036.7	4026.4	5151.3	4956.0
2	11970.2	8616.8	10120.7	8515.8
3	12715.1	8903.4	9972.6	10085.0
4	16349.6	11266.2	12922.7	13474.2
5	20935.0	14924.3	17282.1	18050.7
6	23119.7	16381.2	18887.2	19859.8
7	28694.5	20320.0	23509.8	24824.9
8	32104.6	23082.0	26744.5	27255.4
9	37491.8	26450.3	30517.3	32194.4
10	41394.8	29897.7	34575.6	35671.3
11	44856.8	32659.1	37032.5	38714.6
12	44695.1	33848.9	39292.6	40625.8
13	47377.9	36377.7	42050.6	43542.1
14	48853.3	38589.6	44440.4	45509.7
15	51273.9	41662.3	44663.6	44941.4
16	48704.8	39592.3	43615.7	44797.8
17	33638.9	23842.9	29161.2	37024.5
18	33712.5	27466.9	37166.7	32095.5
19	38073.6	30095.0	34539.9	34835.6
20	26627.1	20307.7	24445.4	21617.2
21	41575.9	33180.5	40777.5	38727.8
22	43078.1	34787.1	38100.6	40516.5
23	44306.5	36409.9	44583.9	42289.1
24	44679.1	35595.9	40602.3	40671.9
25	47327.4	35824.6	41718.8	40939.5
26	46085.6	36378.7	46893.2	45583.9
27	48237.8	37579.8	49538.8	45762.3
28	43507.7	28391.9	32673.6	31051.1
29	46562.0	37710.1	49211.6	47533.4
30	45188.7	39833.6	45820.8	47210.6
31	46011.6	39429.2	46247.3	47491.9
32	47961.6	41638.6	49278.3	47481.3

Table 2: NVDIMM Stream, 100,000,000 elements

types of benchmarks in order to test all possible combination of traffic. Does a combination of DRAM and NVDIMM threads exist where the total bandwidth of DRAM and NVDIMM threads exceeds what a DRAM alone can archive.

1.4.1 NVM-NVM

The code for this benchmark are described in listing 3. From line 2-14 the code is declaring variables and creating the arrays where the result from the benchmark will be stored. There will be declared two DRAM array s and two NVDIMM arrays that will be used in the benchmark. When the threads arrive at line 16 they will synchronize before they are split into two group s Threads with a thread id lower than the number of DRAM threads will pass the if-sentence at line 17, where they will initiate two DRAM array s and place values into each element. The rest of the threads will move on to line 27 and enter this bracket. These threads will initiate two NVDIMM arrays and place values into each element. All the threads will then synchronize at line 40 before they will divide into two group s at line 41 in the same fashion they did in line 17. All the threads will then start to copy data from one array to the other array. The DRAM threads will copy ~~from~~ the DRAM-DRAM array s and the NVDIMM threads will copy ~~from~~ the NVDIMM-NVDIMM array s. They will repeat this for as many times as the user of the benchmark has decided by defining the total_test variable as an argument in the command line when the program was started. The time measurement will be started at the beginning of the for-loop at line 45 or 54 and end at the end of the for-loop at line 50 or 60. When the benchmark testing is over the threads will free up their arrays and the benchmark will print out the result.

When the threads pass the barrier at line 40 and begin the benchmark test they will never synchronize another time. Because of this the DRAM threads will complete their tasks ~~a lot~~ earlier then NVDIMM threads because DRAM speed is faster then NVIDMM speed. This also means that once the fastest thread is done the rest of the threads will share more bandwidth among themself and become faster. When more and more threads complete their tasks the faster the remaining threads will become. In order to get a correct benchmark where all threads have been working the user need s to throw out the data where some threads are working when other threads have completed their tasks. Throwing out the last one third is usually enough. There is also a need to throw out at least the first 25 iterations. This is because the

meaning ???

than

NVDIMM is a lot slower to get started then DRAM.

Listing 3: NVM-NVM source code

```
#pragma omp parallel
{
  //Declearing variables.
  int thread_id = omp_get_thread_num();
  int i,j;
  double *drm_read_array;
  double *drm_write_array;
  TOID(double) nvm_read_array;
  TOID(double) nvm_write_array;
  srand((unsigned int)time(NULL));
  #pragma omp master
  {
    /* Creates array where the test result will be added.
       */
  }
  //Creates all the arrays needed for the test.
  #pragma omp barrier
  if(thread_id < totalThreads-nvmThreads){
    drm_read_array =
       (double*)malloc(ARRAY_LENGTH*sizeof(double));
     drm_write_array =
        (double*)malloc(ARRAY_LENGTH*sizeof(double));
    #pragma omp critical
    {
      for(i=0;i<ARRAY_LENGTH;i++){
        drm_read_array[i] =
           ((double)rand()/(double)(RAND_MAX));
        drm_write_array[i] =
           ((double)rand()/(double)(RAND_MAX));
      }
    }
  }else if(thread_id >= totalThreads-nvmThreads){
    POBJ_ALLOC(pop, &nvm_read_array, double,
       sizeof(double) * ARRAY_LENGTH, NULL, NULL);
    POBJ_ALLOC(pop, &nvm_write_array, double,
       sizeof(double) * ARRAY_LENGTH, NULL, NULL);
    #pragma omp critical
    {
      for(i=0;i<ARRAY_LENGTH;i++){
        D_RW(nvm_read_array)[i] =
            ((double)rand()/(double)(RAND_MAX));
        D_RW(nvm_write_array)[i] =
            ((double)rand()/(double)(RAND_MAX));
      }
      //printf("NVM thread_id: %d, %f\n", thread_id,
          D_RO(nvm_read_array)[11235]);
    }
  }
  //Doing the test.
  #pragma omp barrier
  if(thread_id < totalThreads-nvmThreads){
    //From DRAM to DRAM:
    for(i=0;i<total_tests;i++){
      //Time start
      test_time[thread_id][i] = mysecond();
      for(j=0;j<ARRAY_LENGTH;j++){
        drm_write_array[j] = drm_read_array[j];
      }
      //Time stop.
      test_time[thread_id][i] = mysecond() -
          test_time[thread_id][i];
    }
  }else if(thread_id >= totalThreads-nvmThreads){
    //From NVM to NVM:
    for(i=0;i<total_tests;i++){
      //Time start
      test_time[thread_id][i] = mysecond2();
      for(j=0;j<ARRAY_LENGTH;j++)
        D_RW(nvm_write_array)[j] =
            D_RO(nvm_read_array)[j];
      //Time stop.
      test_time[thread_id][i] = mysecond2() -
          test_time[thread_id][i];
    }
  }else
    printf("ERROR\n");
  /* Freeing up DRAM and NVDIMM arrays */
}
```

The Result of the benchmark is shown in figure 3 and table 3. They show that as the number of NVDIMM threads increases at the expense of the DRAM threads the NVDIMM bandwidth increases and DRAM bandwidth decreases. The sum of the DRAM and NVDIMM

Threads	DRAM	NVDIMM	Sum
1	61042.84	3038.16	64081.01
2	57761.80	6027.24	63789.03
3	53767.10	9223.43	62990.52
4	51257.79	12630.42	63888.22
5	47770.89	15202.48	62973.37
6	43771.07	18771.08	62542.15
7	39780.16	22085.75	61865.90
8	36275.70	25068.35	61344.05
9	31873.99	28520.43	60394.42
10	28691.82	30840.40	59532.22
11	24580.59	33991.75	58572.33
12	20550.17	38006.44	58556.61
13	15877.24	40011.90	55889.14
14	11010.71	43913.44	54924.15
15	5932.95	46352.16	52285.11

Table 3: NVM-NVM, 32 GB

start s at 64,081 MB/s and decreases with 1,100 MB/s when there are five NVDIMM threads When the number of NVDIMM threads increases even more, the bandwidth decreases at a higher rate. From five to fifteen NVDIMM threads the bandwidth have decreased with 10,000 MB/s. This might be because the NVDIMM are not fast enough to make use of all the bandwidth that becomes available as the number of DRAM threads decreases. The result is that the user loses performance if the number of NVDIMM is too high.

The graph in figure 4 is the result of the same benchmark as figure 3. The difference is that figure 4 now show s the number of seconds on x-axis and bandwidth on y-axis. Threads ends at different times. The last DRAM ends at 231 seconds and the last NVDIMM ends at around 319 seconds. The figure 4 also show s that all the DRAM threads have the same consistent speed except for the end of the DRAM. The NVDIMM threads on the other hand fluctuate a lot, they seem to have a speed at around 3,000 MB/s, but once in a while they drop down to 1,700 MB/s.

1.4.2 NVM-DRAM

In this version of the benchmark there will be one group of threads that will transfer data from DRAM-DRAM and another group of threads

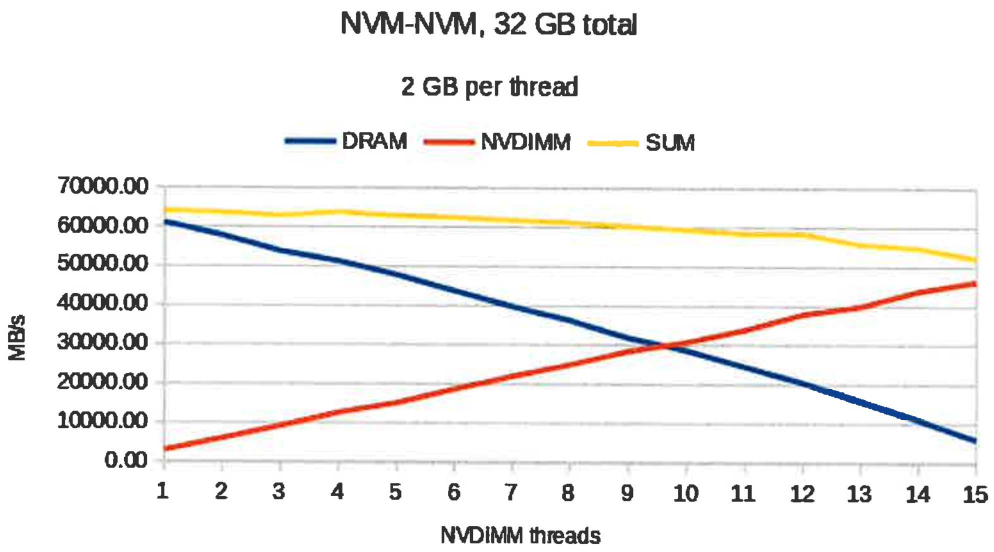

Figure 3: NVM-NVM, 32 GB

← more texts needed in the caption

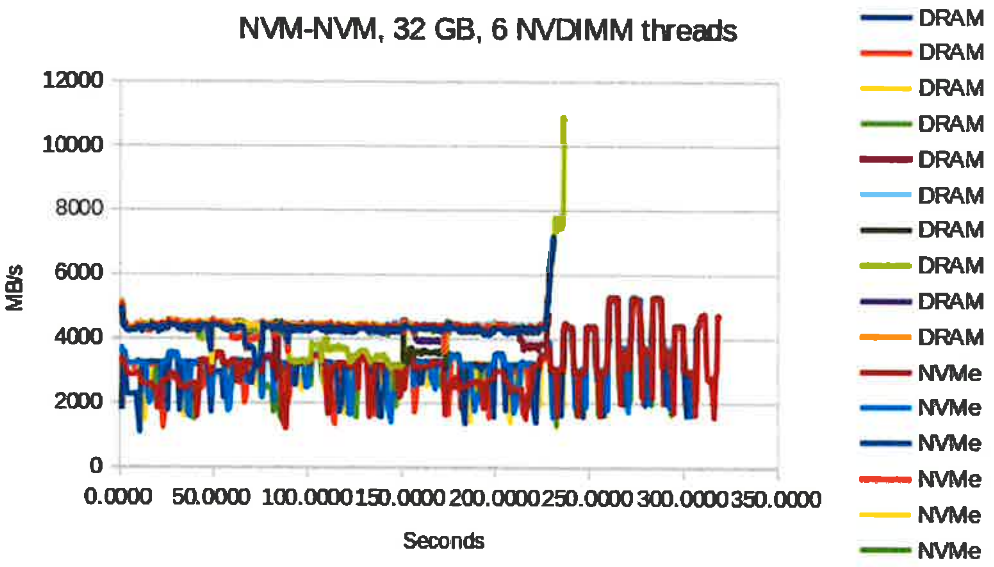

Figure 4: NVM-NVM, 32 GB, 6 NVDIMM Threads

10 DRAM threads

that will transfer data from NVDIMM-DRAM. This code is very similar to the previous code. The differences are at line 26-36 where the code will initiate and add values to one DRAM array and one NVDIMM array instead of two NVDIMM arrays. The other difference is at line 51-61 where the code will copy data from NVDIMM-DRAM instead of NVDIMM-NVDIMM.

Listing 4: NVM-DRAM source code

```
1  #pragma omp parallel
2  {
3    int thread_id = omp_get_thread_num();
4    int i,j;
5    double *drm_read_array;
6    double *drm_write_array;
7    TOID(double) nvm_read_array;
8    srand((unsigned int)time(NULL));
9    #pragma omp master
10   {
11     /* Creates array where the test result will be added.
         */
12   }
13   //Creates all the arrays needed for the test.
14   #pragma omp barrier
15   if(thread_id < totalThreads-nvmThreads){
16     drm_read_array =
          (double*)malloc(ARRAY_LENGTH*sizeof(double));
17     drm_write_array =
          (double*)malloc(ARRAY_LENGTH*sizeof(double));
18     #pragma omp critical
19     {
20       for(i=0;i<ARRAY_LENGTH;i++){
21         drm_read_array[i] =
              ((double)rand()/(double)(RAND_MAX));
22         drm_write_array[i] =
              ((double)rand()/(double)(RAND_MAX));
23       }
24     }
25   }
26   else if(thread_id >= totalThreads-nvmThreads){
27     drm_write_array =
          (double*)malloc(ARRAY_LENGTH*sizeof(double));
28     POBJ_ALLOC(pop, &nvm_read_array, double,
          sizeof(double) * ARRAY_LENGTH, NULL, NULL);
```

```
    #pragma omp critical
    {
      for(i=0;i<ARRAY_LENGTH;i++){
        D_RW(nvm_read_array)[i] =
           ((double)rand()/(double)(RAND_MAX));
        drm_write_array[i] =
           ((double)rand()/(double)(RAND_MAX));
      }
    }
  }
  //Doing the test.
  #pragma omp barrier
  if(thread_id < totalThreads-nvmThreads){
    //From DRAM to DRAM:
    for(i=0;i<total_tests;i++){
      //Time start
      test_time[thread_id][i] = mysecond();
      for(j=0;j<ARRAY_LENGTH;j++){
        drm_write_array[j] = drm_read_array[j];
      }
      //Time stop.
      test_time[thread_id][i] = mysecond() -
         test_time[thread_id][i];
    }
  }
  else if(thread_id >= totalThreads-nvmThreads){
    //From NVM to DRAM:
    for(i=0;i<total_tests;i++){
      //Time start
      test_time[thread_id][i] = mysecond();
      for(j=0;j<ARRAY_LENGTH;j++)
        drm_write_array[j] = D_RO(nvm_read_array)[j];
      //Time stop.
      test_time[thread_id][i] = mysecond() -
         test_time[thread_id][i];
    }
  }
  else
    printf("ERROR\n");
  /* Freeing up DRAM and NVDIMM arrays */
}
```

The result of this benchmark are shown in figure 5 and table 4. Fig-

Threads	DRAM	NVDIMM	Sum
1	61257.14	3742.24	64999.37
2	58419.89	6691.30	65111.18
3	54544.07	10520.34	65064.41
4	49997.69	15247.47	65245.16
5	46226.88	19189.81	65416.69
6	42942.39	22806.13	65748.52
7	38156.47	27257.45	65413.92
8	34613.22	30474.15	65087.37
9	30458.70	34533.35	64992.05
10	25489.86	39324.61	64814.47
11	22083.85	42763.51	64847.36
12	17704.90	47147.70	64852.60
13	13394.67	51380.63	64775.31
14	8947.75	56009.92	64957.67
15	4489.12	60031.12	64520.24

Table 4: NVM-DRAM, 32 GB

ure 5 show that when the number of NVDIMM threads increases the NVDIMM bandwidth also goes up while the DRAM threads and bandwidth goes down. The sum of ~~both~~ the DRAM and NVDIMM bandwidth is stable at around 64,000 MB/s for all numbers of NVDIMM threads.

Figure 6 shows the result of the same benchmark as figure 7, but with second on the x-axis. The last DRAM thread ends at 234 seconds and the last NVDIMM thread ends at 258 seconds. The DRAM threads have consistent speed through the entire test at around 4,300 MB/s, only at the end will the speed increase because other threads have finished before them and therefore made more bandwidth available for the remaining threads. The NVDIMM threads are also more consistent in this benchmark where five of the six threads have speed of 3,800 MB/s for most of the test. There is only one thread that is fluctuating between 3,000 MB/s and 3,600 MB/s for most of the benchmark.

1.4.3 DRAM-NVM

In this version of the benchmark there will be one group of threads that transfer data from DRAM-DRAM and another group of threads that will transfer data from DRAM-NVDIMM. This code only have one difference when compared to the code in 1.4.2. The difference is found

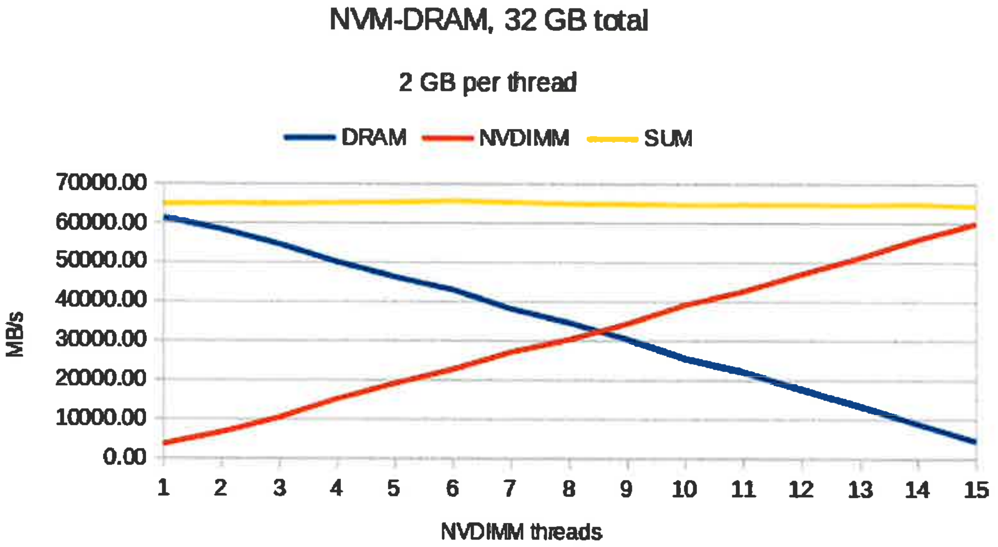

Figure 5: NVM-DRAM, 32 GB

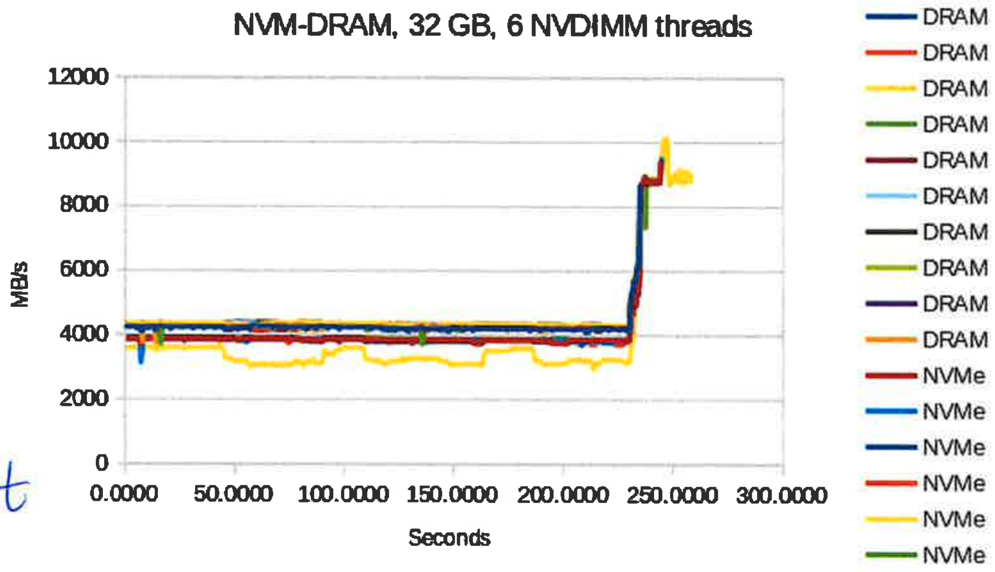

Figure 6: NVM-DRAM, 32 GB, 6 NVDIMM Threads

difficult to see the different curves

in line 51-61 where data is transferred from DRAM-NVDIMM insted of from NVDIMM-DRAM.

Listing 5: DRAM-NVM source code

```
1  #pragma omp parallel
2  {
3    int thread_id = omp_get_thread_num();
4    int i,j;
5    double *drm_read_array;
6    double *drm_write_array;
7    TOID(double) nvm_write_array;
8    srand((unsigned int)time(NULL));
9    #pragma omp master
10   {
11     /* Creates array where the test result will be added.
          */
12   }
13   //Creates all the arrays needed for the test.
14   #pragma omp barrier
15   if(thread_id < totalThreads-nvmThreads){
16     drm_read_array =
          (double*)malloc(ARRAY_LENGTH*sizeof(double));
17     drm_write_array =
          (double*)malloc(ARRAY_LENGTH*sizeof(double));
18     #pragma omp critical
19     {
20       for(i=0;i<ARRAY_LENGTH;i++){
21         drm_read_array[i] =
              ((double)rand()/(double)(RAND_MAX));
22         drm_write_array[i] =
              ((double)rand()/(double)(RAND_MAX));
23       }
24     }
25   }
26   else if(thread_id >= totalThreads-nvmThreads){
27     drm_read_array =
          (double*)malloc(ARRAY_LENGTH*sizeof(double));
28     POBJ_ALLOC(pop, &nvm_write_array, double,
          sizeof(double) * ARRAY_LENGTH, NULL, NULL);
29     #pragma omp critical
30     {
31       for(i=0;i<ARRAY_LENGTH;i++){
32         drm_read_array[i] =
           ((double)rand()/(double)(RAND_MAX));
        D_RW(nvm_write_array)[i] =
           ((double)rand()/(double)(RAND_MAX));
      }
    }
  }
  //Doing the test.
  #pragma omp barrier
  if(thread_id < totalThreads-nvmThreads){
    //From DRAM to DRAM:
    for(i=0;i<total_tests;i++){
      //Time start
      test_time[thread_id][i] = mysecond();
      for(j=0;j<ARRAY_LENGTH;j++){
        drm_write_array[j] = drm_read_array[j];
      }
      //Time stop.
      test_time[thread_id][i] = mysecond() -
         test_time[thread_id][i];
    }
  }
  else if(thread_id >= totalThreads-nvmThreads){
    //From DRAM to NVM:
    for(i=0;i<total_tests;i++){
      //Time start
      test_time[thread_id][i] = mysecond();
      for(j=0;j<ARRAY_LENGTH;j++)
        D_RW(nvm_write_array)[j] = drm_read_array[j];
      //Time stop.
      test_time[thread_id][i] = mysecond() -
         test_time[thread_id][i];
    }
  }
  else
    printf("ERROR\n");
  /* Freeing up DRAM and NVDIMM arrays */
}
```

The result of this benchmark are shown in figure 7 and table 5. The result in figure 7 is similar to the result in figure 5 where NVDIMM speed increases and DRAM speed decreases at the same around the same rate. The sum of DRAM and NVDIMM speed remain stable at around 63,000 MB/s. Only at eleven NVDIMM threads and higher does

Threads	DRAM	NVDIMM	Sum
1	61002.60	3581.48	64584.09
2	56598.08	7323.96	63922.04
3	53247.52	10954.06	64201.58
4	48681.71	15070.61	63752.32
5	44761.80	18725.90	63487.70
6	41022.39	22614.45	63636.84
7	36897.42	26557.71	63455.13
8	32548.18	30570.88	63119.07
9	28892.06	33996.95	62889.00
10	24941.87	38191.10	63132.97
11	20563.14	42263.28	62826.42
12	16803.83	45915.39	62719.21
13	12648.47	49341.90	61990.38
14	8386.48	53331.10	61717.57
15	4189.43	57348.14	61537.57

Table 5: DRAM-NVM, 32 GB

the speed decrease a little.

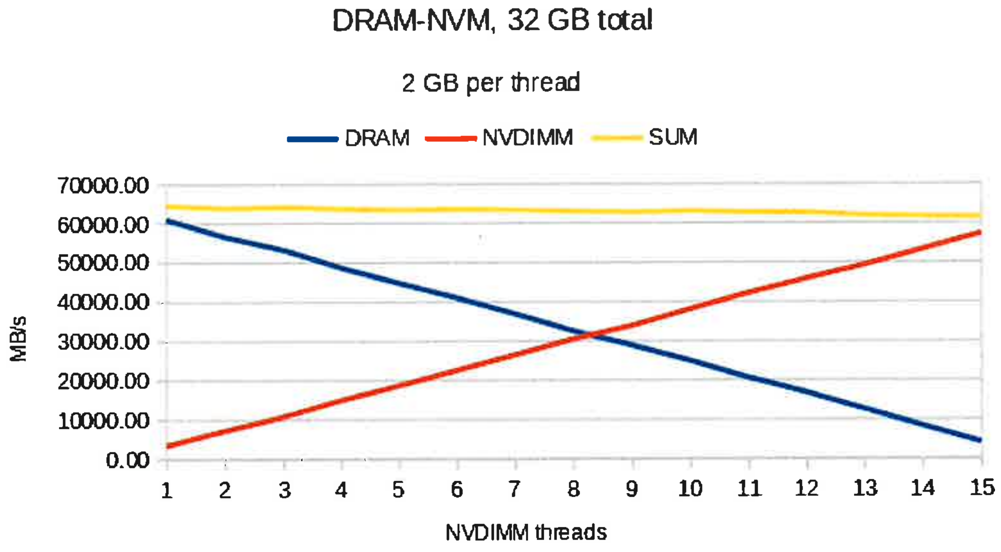

Figure 7: DRAM-NVM, 32 GB

Similar to ~~to~~ the other two benchmarks the DRAM threads in Figure 8 have more consistent speed than the NVDIMM. The speed of the is

DRAM threads are around 4,100 MB/s until the end where it increases sharply because other threads have completed their tasks and given the remaining threads more bandwidth. The speed of the NVDIMM is more unstable. The speed ~~is~~ remains at 3,800 MB/s, but all of the NVDIMM threads drops their speed several time during the test. Sometimes as low as 1,900 MB/s.

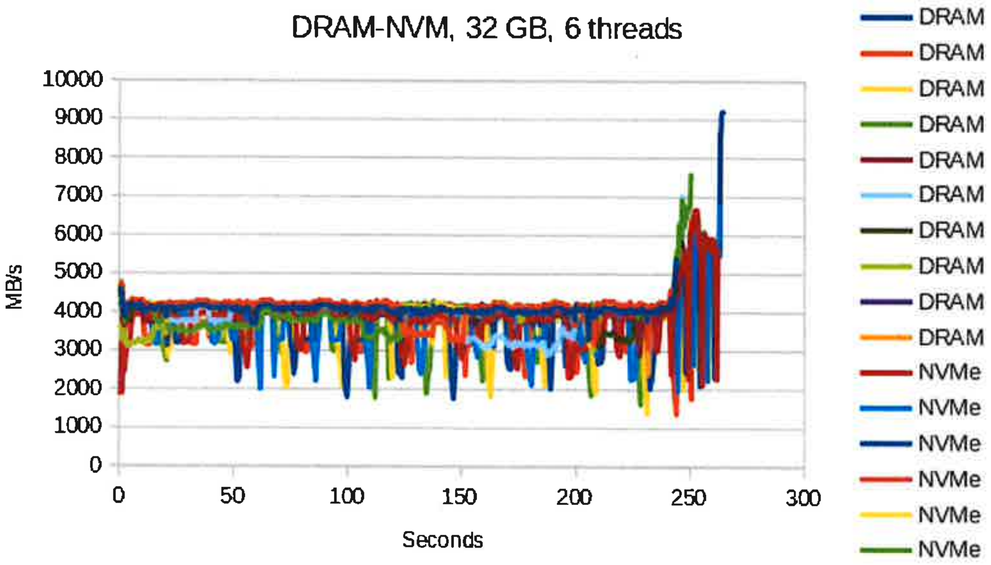

Figure 8: DRAM-NVM, 32 GB, 6 NVDIMM Threads

1.4.4 Observations

Section Section

The benchmarks at chapter 1.4.2 and chapter 1.4.3 have results more similar than the result in chapter 1.4.1. This is because the transfer speed DRAM to NVDIMM and NVDIMM to DRAM is almost identical. The DRAM-NVDIMM speed with one thread is 3581 MB/s and with fifteen thread the speed is 57348 MB/s. The NVDIMM-DRAM speed with one thread is 3742 an with fifteen thread the speed is 60031 MB/s. The speed with one and fifteen threads are almost the same. The rate the speed increase for each NVDIMM are also very similar.

Section

The speed of the NVDIMM to NVDIMM in chapter 1.4.1 is slower ~~than~~ the other two benchmarks. When there is five threads the total speed of both DRAM and NVDIMM starts to decrease and at fifteen threads the total speed have decreased by 10,000 MB/s. This means that if a program have two group of threads, one group only working on DRAM and the other only in NVDIMM. The program should not have more than five threads working on NVDIMM if the object is to objective maximize performance.

section section

Figure 4 in chapter 1.4.1 and 8 in chapter 1.4.3 shows the NVDIMM threads fluctuate a lot. The figure in 6 in chapter 1.4.2 on the other hand show that all the NVDIMM threads except for one is a lot more stable. This might mean that writing to NVDIMM is a lot more sensitive to disturbances than what reading data from NVDIMM is.

The NVDIMM Stream in chapter 1.3 also something similar where there are only NVDIMM threads and no traffic on DRAM. When each thread have their own core there is a smooth increase from one thread to sixteen threads. But when the hyper-treads are included and there are more than one thread per core the speed starts to fluctuate up and down for each new thread that is included.

Regardless of what the reason is for the fluctuations. The fluctuation might make it difficult to predict how much time it takes for an NVDIMM thread to complete a task.

References

[1] John D. McCalpin. *STREAM source code*. URL: `https://www.cs.virginia.edu/stream/FTP/Code/stream.c` (visited on 12/20/2020).