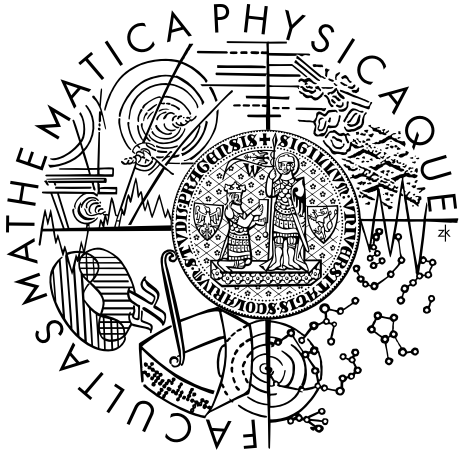
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Lukáš Marek

Parallel Processing and Software Performance

Department of Software Engineering

Supervisor: Doc. Ing. Petr Tůma, Dr.

Study Program: Computer Science, Software Systems

2008

I would like to thank my supervisor, *Doc. Ing. Petr Tůma, Dr.*, for his guidance, advices and time he spent helping me with this thesis. I would also like to thank all members of Distributed Systems Research Group for their help and encouragement. Last but not least, I thank my whole family for their endless support.

I declare that I have elaborated this thesis on my own and listed all used references. I agree with making this thesis publicly available.

Prague, Dec 12, 2008 Lukáš Marek

Contents

1 Introduction **1**

1.1 Assignment 1

1.2 Goals revisited 2

2 Measurement **3**

2.1 Framework 3

2.2 Experiments 6

2.2.1 RDTSC and PAPI 7

2.3 Tested platforms 8

2.3.1 Dell PowerEdge 1955 (Intel) 8

2.3.2 Dell PowerEdge SC1435 (AMD) 9

3 Benchmarks **10**

3.1 CPU 10

3.1.1 Cache coherency 11

3.1.2 Ping-pong 11

3.1.3 Producer - Consumer 19

3.1.4 Cache sharing 25

3.1.5 Shared cache - Bandwidth 25

3.1.6 Shared cache - Thrashing 30

3.2 Memory 35

3.2.1 Memory bus and memory controller 35

3.2.2 Eviction behavior in physically mapped cache 36
3.2.3 Memory bandwidth - single-pointer 38
3.2.4 Memory bandwidth - multi-pointer 43
3.2.5 Memory Saturation 48
3.3 Hard disk drive 48

4 Conclusion **50**
4.1 Future Work 51

A Memory content cache **54**

B Basic code constructs **57**

Název práce: Paralelní Zpracování a Výkonnost Software
Autor: Lukáš Marek
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: Doc. Ing. Petr Tůma, Dr.
e-mail vedoucího: petr.tuma@mff.cuni.cz

Abstrakt: Práce se zaměřuje na vliv sdílení vybraných prostředků, především paměťových cache, na výkon software na moderních procesorových architekturách. V práci je navrhnut framework pro spouštění experimentů, které vyhodnocují vliv sdílení na výkon. V práci je popsaný návrh, měření a vyhodnocení experimentů s použitím tohoto frameworku. Výsledky dávají odhad velikosti vlivu sdílení prostředků na výkon na zvolených platformách.

Keywords: paralelní zpracování, výkon, sdílení prostředků

Title: Parallel Processing and Software Performance
Author: Lukáš Marek
Department: Department of Software Engineering
Supervisor: Doc. Ing. Petr Tůma, Dr.
Supervisor's e-mail address: petr.tuma@mff.cuni.cz

Abstract: The thesis focuses on the effects of resource sharing on software performance for selected resources of modern processor architectures, especially shared memory caches. The thesis introduces a framework for conducting experiments that assess the impact on software performance, and, using this framework, designs, executes and evaluates the experiments. The results provide an estimate of the scale of the performance impact of resource sharing on the experimental platform.

Keywords: parallel processing, performance, resource sharing

Chapter 1

Introduction

When multi-core systems entered the mainstream, parallel processing ceased to be the domain of corporate servers and specialized software, and started to be an issue even in common programming projects. Parallel programming is much more difficult than single threaded programming. The most obvious difference between parallel programming and single threaded programming is that the data structures are shared between multiple threads. The threads therefore need to be synchronized and the programmer has to pay attention to race conditions and potential deadlocks.

Another aspect of parallel programming, perhaps less immediately visible than races and deadlocks, is the performance impact of resource sharing. Resources such as processor caches, memory or hard drives have always been slower than processor execution units. With parallel processing, the throughput of these components is even more important.

1.1 Assignment

This is the original text of the assignment composed at the time when the work on the thesis was initiated.

Design a set of experiments to identify and quantify the influence of resource sharing on software performance. Focus on parallel architectures (hyperthreading, multicore, multiprocessor) and concurrent execution of code. Include a scale of resources from processor caches to disk files.

Execute the experiments on common platforms and summarize the results. Include Intel processors, native processes as well as virtualized (CLI or JVM processes), Linux and Windows.

Analyze the applicability of the results in optimizing the placement in component deployment, finetuning the algorithms for process scheduling and in minimizing the imprecision in performance modeling.

1.2 Goals revisited

As the work on the thesis progressed, it has, however, turned out that the original assignment covers more area than it is possible to investigate in a single master thesis. Faced with the choice of whether to do a superficial assessment of a broader set of resources, or whether to do an in depth investigation of fewer resources, the decision – based on frequent consultations with the thesis advisor – has been to prefer depth over breadth. Here, we therefore present the revised goals of the thesis and summarize the subset of the assignment that has been done, as approved by the thesis supervisor.

This thesis is a continuation of the thesis Influence of Resource Sharing on Performance [9] written by Vlastimil Babka that was oriented on resource sharing where workloads were invoked sequentially while sharing the same CPU.

The goal of this thesis is to identify the shared resources and evaluate the performance impact introduced by parallel sharing. The goal is achieved by designing experiments that will describe the effect on the hardware level and will be independent of the programming language or the operating system.

An impact of resource sharing can be analyzed using either real or artificial workload. The advantage of using real workloads is that the results can be directly compared with real applications. However, real workloads are composed from variety of different actions and effects that cause the slowdown are generally not easy to track down. As we said above, our goal is to describe each effect individually. For that reason, we designed special artificial workloads that trigger only a few effects in one measurement.

We will also need a measurement framework as the tool for our experiments that will help execute the experiments and analyze the results.

The motivation for low level experiments is to quantify and discover all types of effects that can be encountered – it turns out that the effects of sharing are not trivial, and, especially because of lack of hardware documentation, to come up with a plausible explanation for the experiment results is not as easy as it could seem to be. By carefully explaining the experiment results, the thesis therefore provides a sort of a bonus result in the form of clarification and extension of many places in the hardware documentation, which would otherwise be difficult to obtain.

Chapter 2

Measurement

All the experiments described in this thesis follow the same design pattern – a set of workloads is executed in parallel over a shared resource. The workloads are designed artificially to provide a well defined pattern of resource usage, which can be adjusted as necessary. Using the experiments, the thesis evaluates the effect of sharing on the observed changes in performance of the workloads.

Since the basic structure of the experiments is similar, we have designed a common software framework for running the experiments, and a common presentation outline for explaining the results.

2.1 Framework

In the following text, the word benchmark is used to define one type of workload that runs on a single core. The word experiment defines a set of one, two or more benchmarks preset to generate a meaningful workload with measurable results. The experiment is described in a configuration file and specifies the benchmarks and all their parameter values.

When we want to start doing experiments, we need a tool that will help us. The first requirement is to create and compose variety of workloads. It is important to be able to combine different workloads together to create different scenarios. We also need to run the experiment with variety of parameters automatically because at the beginning we do not know precise values for experiment parameters. A simple extensibility for additional benchmarks and other modifications is also very helpful.

The framework should also measure, collect and store the results for a later use. The stored results are then processed in to graphs and statistics.

With these requirements we had to develop our own framework. The basic idea was taken from the RIB [6] framework created by Vlastimil Babka but the whole framework was developed from scratch to better suit our needs. The framework called RIP is mainly developed in C++ and R is used as a graph plotting back end.

The main focus of this framework is extensibility, ease of use and a precise measurement.

In the framework, benchmarks are implemented as individual classes that can later be composed. To create a new benchmark, we only need to derive one class, register it with simple macro and implement two methods. A substantial part of the new benchmark is a new constructor with parameter definitions and an implementation of one function that takes care of the measurement. These two methods are presented on Listing 2.1.

A very simple implementation creates the whole benchmark. The macro "DEFINE_PARAM_*" allows to define benchmark specific parameters. The macro "GET_CURRENT_*" retrives a current value for selected parameter. The measured code is placed between "MEASURED_BLOCK_BEGIN" and "MEASURED_BLOCK_END".

Listing 2.1: Simple benchmark creation

```
NewBench::NewBench() {
    // define parameters
    DEFINE_PARAM_NUMERIC("addition");
}

void NewBench::runMBench() {
    int sum = 0;
    // get parameter
    int addition = GET_CURRENT_NUMERIC("addition");
    // measure addition
    MEASURED_BLOCK_BEGIN {
        sum += addition;
    } MEASURED_BLOCK_END
}
```

The created "NewBench" benchmark is now prepared for a measurement. The measurement in RIP is driven by a configuration file and is done in five steps. Between each of the five steps all benchmarks are synchronized. This means that all of the benchmarks have to confirm a movement to the next step before all of them can do it. The first and last steps are just synchronization steps and they serve as a starting and stopping barrier. The next step after the start is a warm-up phase. The reason for this phase is to eliminate all undesirable effects that come with uninitialized data. Next two steps are measurement phases where the first of them

measures RDTSC and the second one measures PAPI. More about RDTSC and PAPI is in following sections. The important information for now is that RDTSC measures the number of CPU clocks and PAPI measures hardware counters like the number of the L2 cache misses.

Now let us look on a measurement configuration visible on Figure 2.2. First three parameters are global for all benchmarks. The parameter "benchmarks" is a list of all benchmarks that are running. The number of "runs" says how many times is whole experiment restarted.

The framework runs an experiment for all parameter combinations in one benchmark. The parameter "paramChange" sets algorithm for an exact parameter change. All possible values for the parameter change are separated with ",". Parameters where is possible only one value are the three parameters in "rip" section and also parameter "type" which determines benchmark class and "cpu" which binds benchmark on given core (CPU). These parameters cannot be changed.

The next six parameters are measurement parameters. These parameters are important for the MEASURED_BLOCK_BEGIN and MEASURED_BLOCK_END macros in Listing 2.1. The parameter "warmup" says how many times is the measured block executed before the real measurement takes its place. The number of "samples*" determines how many samples are taken and stored. Be aware that the total number of samples is computed like: the number of "runs" multiplied by the number of "samples*". The parameter "cyclesPer*" determines how many times is measurement block executed before one sample is taken. This means that at the beginning, the framework starts RDTSC measurement. Now because "cyclesPerRDTSC" is 100, the measurement block is done one hundred times and after that RDTSC measurement is stopped. As you can see, the result of our measurement is an average. The reason is that the start and the stop creates an overhead and adds a nondeterminism. To eliminate these effects, we get more samples in a row.

The last parameter "addition" is the user defined parameter with two possible values that will be both tested in two different experiment rounds.

Listing 2.2: Simple configuration file

```
[rip]
benchmarks=newbench1
runs=10
paramChange=AllBenchChange

[newbench1]
type=newbench
cpu=0

warmups=40K
```

```
samplesRDTSC=10
samplesPAPI=3
cyclesPerRDTSC=100
cyclesPerPAPI=1000
papiEvents=L1D_ALL_REF,RESOURCE_STALLS:ANY

addition=1,2
```

A plotter is the second part of the framework that is responsible for graph generation. The plotting is from a user perspective very similar to the measurement but now user does not need to implement any class or method. The plotter can handle most of the requirements only with a configuration file. The plotting has two phases. In the first phase, data are prepared by ripplot (C++) by merging data files as specified in the configuration file. Then ripplot runs R [5] with a special script which takes care about a graph generation. Supportive R scripts can additionally modify some aspects of the graph like axis naming or curve colors.

2.2 Experiments

Later in the text we will present experiments. Each experiment will have a defined structure that looks like this:

Experiment: Experiment name

Purpose Simple experiment description

Measured The main measured value and all benchmarks in experiment

Parameters All relevant experiment parameters and their values

Some parameters have a range specified like this: ("starting value" - "ending value" exponential step). Exponential step means that a new value is calculated like this: "new value" = "previous value" * "previous value" where "previous value" is at the beginning "starting value" and "new value" can be at the most "ending value".

All framework specific parameters that care about measurement are missing because they are always same. The description of framework parameters follows.

Expected Results A proper experiment should have estimated results for confrontation.

Measured Results Experiment measured results with figures and explanation.

Figures in all experiments show medians. If the y-axis shows the number of clocks measured by RDTSC, the medians are computed from 100 samples. If the y-axis shows some hardware counter measured by PAPI, the medians are computed from 30 samples. One median value is not a duration of one access (or the number of hardware events during one access) but it is an average over "X" accesses. The magic number "X" is included in each y-axis label as ["measured unit" - "X" Avg].

All these numbers can be also extracted from the experiment configuration file. Total number of samples measured by framework is computed as (runs * samplesRDTSC) or (runs * samplesPAPI). All three parameters have only one value for all experiments in this thesis and they are: runs(10), samplesRDTSC(10), samplesPAPI(3).

The magic number "X" is also in the configuration file as cyclesPerRDTSC(100) or cyclesPerPAPI(1000). They are also same for all experiments with one exception. The exception is the Producer - Consumer experiment 3.1.3 where the average is measured over whole transferred memory.

Open Issues All unresolved issues that we encounter

Effect Summary Summary of all major effects

In each experiment, we will also include the source code. We believe that it is necessary for understanding what is happening. In some cases the code is simplified to be more transparent. The original code, as it was measured, can be found on the included CD.

In Appendix B you can see some basic constructs which are not depended on a particular experiment but we thing that the reader might be interested in how are actually programmed. Every occurrence in the text contains link to the source code so it is not necessary to read the code now.

2.2.1 RDTSC and PAPI

We already know how RDTSC and PAPI are measured but we still do not know what exactly they are.

RDTSC [1, page B3 - B5] [2, 4-246 - 4-247] is a machine instruction returning the number of clocks from the start of the processor. By a simple subtraction, we can simply measure the number of clocks between two RDTSC calls. Some sources discourage from using this instruction as the source of precise timer because on

modern processors it does not need to give the stable results. But we do not know about any sufficient replacement and so far we do not encounter any serious issue with this counter.

PAPI [3] is a framework using perfctr [4] that provides a simple access to hardware performance counters on variety of platforms. The hardware performance counters make possible to measure events like number of cache misses, duration of TLB walks or processor resource stalls. These counters are supported directly by the processor so there is no overhead when samples are taken. The only overhead is when we need to setup these counters or collect the results. Because the events are measured directly by hardware, only a small number of events can be tracked simultaneously.

RDTSC and PAPI are not transparent for the measurement. Both of them introduce a non-trivial overhead that is also present in results. This overhead was not subtracted from final results and is still present in all graphs and values. The reason is that the overhead is not totally stable. RDTSC can be influenced by the out-of-order execution and the branch predictor. PAPI can only measure a few counters at the same time and the composition of counters influences the result values. Also a PAPI code base is not trivial and an execution can slightly vary between runs.

The overhead for RDTSC is something about 259 cycles. Be aware that all RDTSC results are an average over 100 access so in results it is around 2.6 cycles per access. The PAPI overhead is not included here but in 1000 samples is relatively small and because we use PAPI as a support for RDTSC results we do not need precise values. The only exception is Shared cache - Thrashing experiment 3.1.6 where we measure the number of L2 cache misses but PAPI does not need so much space in the cache so the number of the L2 cache misses is very small.

Because we have multi-core measurements, it is also good to know how long it takes to collect the samples. RDTSC collection cycle takes about 497 clocks and for PAPI it is around 8000 clocks.

2.3 Tested platforms

2.3.1 Dell PowerEdge 1955 (Intel)

Dell PowerEdge 1955 is the main platform for our experiments. It was picked as a representative of the common server hardware used today. Later in the text, it is referred to as Intel Server.

The configuration is the same as in Project Deliverable D3.3 [8].

Processor Dual Quad-Core Intel Xeon CPU E5345 2.33 GHz (Family 6 Model 15 Stepping 11), 32 KB L1 caches, 4 MB L2 caches

Memory 8 GB Hynix FBD DDR2-667, synchronous, two-way interleaving, Intel 5000P memory controller

Hard drive 73 GB Fujitsu SAS 2.5 inch 10000 RPM, LSI Logic SAS1068 Fusion-MPT controller

Operating system Fedora Linux 8, kernel 2.6.25.4-10.fc8.x86_64, gcc-4.1.2-33.x86_-64, glibc-2.7-2.x86_64

2.3.2 Dell PowerEdge SC1435 (AMD)

We have also planned to include results from an AMD platform, but AMD uses the NUMA memory architecture and our framework right now does not include NUMA support. Most of the tests are prepared and some measurements were done but there is still a lot of work left. All tests have to be revised, properly tuned and reruned. After that the hardest work still remains, explain the results.

Chapter 3

Benchmarks

In the previous chapter, we have described the measurement framework and the common code used in most benchmarks. We have also determined overhead produced by the measurement routines and explained the frequently appearing acronyms. Now we have all the needed tools for introducing experiments, which are designed to describe resource sharing effects on multi-core and multi-processor machines.

In recent years, processor developers have inserted more and more cores into the CPU. But twice more computing units do not mean that we have twice the computing power. Cores still share resources like caches, memory or hard disk drives. In the following sections, we will focus on each of the shared resource and describe effects and overheads that come with their sharing.

3.1 CPU

Before reading further, we recommend reading Appendix A about caches in general. Basic cache knowledge is necessary for understanding the following text.

The first shared resource is CPU itself. CPU is composed from execution units. These units are not normally shared but some Intel processors contain technology named Hyper Threading (HT) that makes it possible. HT technology is surrounded with a lot of discussion if it is actually performance supportive or not. For some time it even disappeared from mainline processors, but it reappears in the new Intel Nehalem architecture. HT is a special kind of resource sharing and tested Intel Server even does not support it so we decided to exclude it from our measurement.

CPU is not only cores with execution units but also caches. Processor contains lot of caches like instruction caches, data caches or TLB caches.

Instruction caches are not very interesting for us. They are not shared between multiple cores and we can ignore cache coherency state synchronization because code modifications are rare.

TLB caches hold data for virtual to physical address translation. The only way how these caches can be influenced by other cores is remote invalidation. The results presented in the Q-ImPrESS Project Deliverable D3.3 [8, page 51] show how invalidation affects performance. We think that the presented overhead is not the maximum we can get. When the experiment invalidates TLB, it does not care about the cache content. In a common application, the cache is full of data and in most cases TLB miss fetches the needed data directly from memory. But the experiment does not use a cache at all. The TLB data can be stored in the cache and because they are not modified, they can be reused again. Instead of fetching data from memory, data are fetched from the cache and overhead is much smaller. On the other hand, we think that TLB invalidation is so rare that it influences performance only very little.

Data (and unified) caches can contain shared data in multiple copies on many cores. This data has to stay consistent across all caches in every core and processor in the whole system and this requires the non-trivial effort. Also two or more cores can share one common cache and cores starts competing for space and bandwidth to this cache. The cache coherency and the cache sharing will be the first two big topics of this thesis.

3.1.1 Cache coherency

In SMP environment, the same data can be shared between multiple cores (processors). To keep data consistent in multiple copies, the processor uses a cache coherency protocol. One such protocol is the MESI protocol, used by Intel and also implemented in the tested Intel Server. This protocol guarantees that each processor works with up-to-date copy of data. But synchronization between processors is not for free. Actually, in some cases it can be a really costly process that requires cooperation even from memory.

3.1.2 Ping-pong

The first scenario simulates a frequently used shared variable (like synchronization primitives or shared resource counters) and tries to evaluate performance losses that come with the synchronization between two cores.

An experiment creates a shared variable between two cores and measures the time

to read or write this variable in different scenarios. The benchmark source code is the same for both cores and is presented in Listing 3.1. ppPlace represents the shared variable described above. Two variants of the code are used. In one, ppPlace is allocated (not visible in Listing) by SyncedPointer as shared a variable between cores but also as a private variable for performance comparison. The write access increments the variable using a lock instruction prefix. The lock prefix ensures that the increment will be atomic on a multiprocessor machine. The read access only moves the variable content to a register. This is also done in assembler because we only want to read the variable into the register to minimize access into the cache. But moving memory to register and then forgetting it is something that is difficult to achieve in C++.

Listing 3.1: Ping-pong read and write access

```
if(write) { // write
      MEASURED_BLOCK_BEGIN {

            //++(*ppPlace);

            // asm replacement with lock
            asm volatile ("lock incl (%0)"
                  : "=r" (ppPlace)
                  : "r" (ppPlace)
                  : ); // nothing used

      } MEASURED_BLOCK_END
}
else { // read
      MEASURED_BLOCK_BEGIN {

            //pingpong_t volatile tmpRead = *ppPlace;

            // asm replacement
            asm volatile ("movl (%0), %%eax"
                  : // no output
                  : "r" (ppPlace)
                  : "%eax");

      } MEASURED_BLOCK_END
}
```

The first variable in the experiment configures read or write access, creating four different scenarios. These are all combinations of read and write on two cores (read x read, read x write...). Another parameter decides if the shared variable is really shared or if it is a local copy for each core. This will help us compare the results with the fastest (non-sharing) scenario.

Experiment: Ping-pong

Purpose Determine the effect of sharing a variable between two cores.

Measured Time to read or write a frequently accessed shared variable (code 3.1).

Parameters data sharing: non-shared, shared; data access (first core x second core): read x read, read x write, write x read, write x write; ping-pong cores positions (relatively to each other): same package with the shared L2 cache, same package without the shared L2 cache, different package

Expected Results Read access on both cores is the simplest scenario. Reads are satisfied from the cache line with a shared state and no cache coherency messages need to be sent here. Results should be the same for shared and non-shared variable regardless of cores positions.

In the next situation, one core reads and the other core writes. Readers and writers perspective will be different but the resulting time should be almost the same. When a writer requests write to memory, the MESI protocol invalidates all cache lines in the other processors that hold the same data (have the same address). In our experiment, the writer holds a valid local copy of the data from previous runs (warm-up) in the cache and after the invalidation, it can start writing the data right away. But there is another delay besides the invalidation. This delay is caused by the reader and it also stops writer because of MESI protocol. After a write request, none of the other processors have the shared variable in their cache lines. So when another processor reads the shared variable, the data are transported from the writing processor to the reading processor and writer marks the data as shared. The writer cannot continue because it waits for reader to complete a data fetch. After the reader reads the data, the writer starts from the beginning with the invalidation. From this, we can see that writers and readers time should be almost the same because they are synchronized. For cores with the non-shared L2 cache and cores in a different package, times should be higher because of transport latencies.

Write core against write core scenario should be simple to explain now. Each core invalidates the cache line on the other core and steals the data from it. Also as mentioned before, cores located further from each other have greater latencies due to data transport.

Measured Results Even though we have described the expected results relatively in detail, modern processors are not so simple.

Note that the accessed variable is 4 byte large but the CPU transports the whole cache line that is 64 byte long. In this experiment, if we present duration

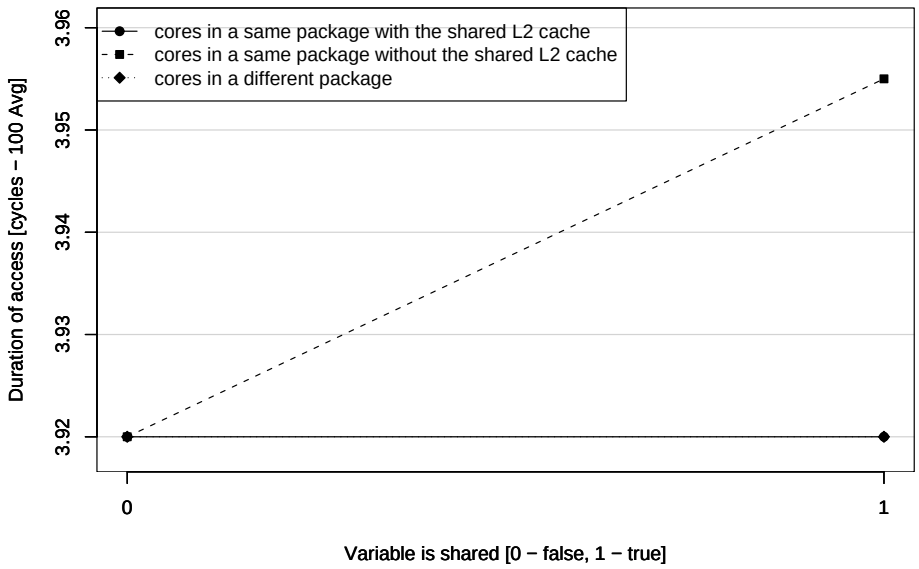


Figure 3.1: Time to read variable on the first core, the second core is also reading

in clocks, it therefore represents how long it takes to transfer the whole cache line.

Results from "read on both cores" scenario are presented on Figure 3.1. The little jump with shared variable is only a measurement error. Otherwise, the results are the same as we have predicted and simultaneous shared variable reading on two cores causes no slowdown.

Read on the first core and write on the second core is visible on Figure 3.2. Note that we are looking only at the reading core for now. As we can see, the results are exactly opposite than we have expected and the core with shared L2 cache is the slowest one. Reading local non-shared variable on single-core costs only 4 clocks but time to read the shared variable while second core writes is 60 clocks for cores that share the L2 cache. Average overhead for cores with no shared L2 cache is 18 clocks and for cores in a different package only 10 clocks. We believe that the core owning data can do more than one access before the other core snoops (we will support this hypothesis later with PAPI counters). Greater distances between cores are the cause of greater latencies in communication and this gives some time for another access. The subsequent access is much faster than an access where the CPU waits on the MESI protocol and such a fast access with no synchronization lowers the average access time. Be aware that one result time that comes from measurement is sum of multiple accesses (100 here) and one access showed in the graph is only a computed average.

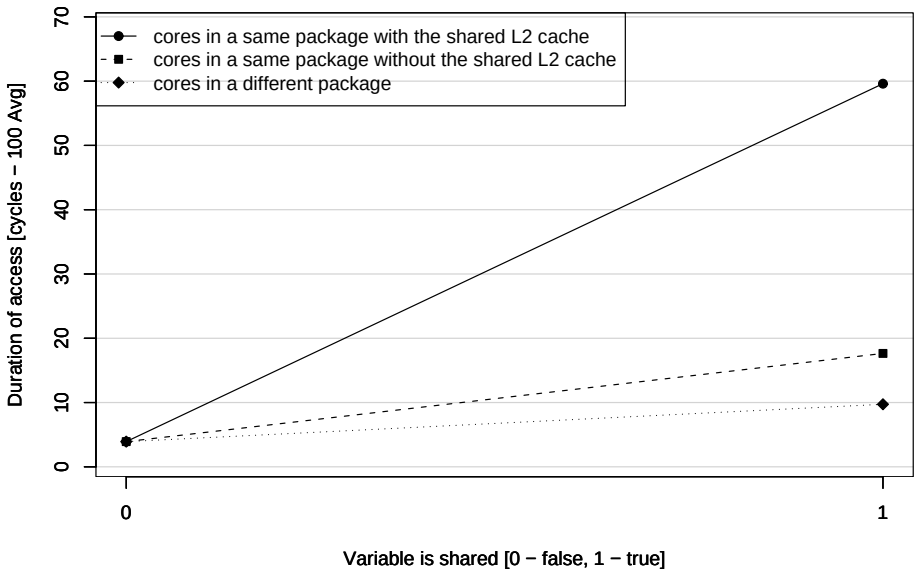


Figure 3.2: Time to read variable on the first core, the second core is writing

In the previous scenario, we come up with the hypothesis that if cores are further apart from each other, they can do more accesses after they acquire desired data. But how does this explain Figure 3.3 showing a write on the first core and a read on the second core? This is the same situation as the previous scenario but now we are looking at the writing core. A problematic part is that cores that do not share the L2 cache have higher access times than cores that share the L2 cache and are closer to each other. A look at Figure 3.4, showing number of accessed cache lines in the L2 cache that are in an Exclusive state, can help with answer to this question. When the cache is shared between two cores, according to Intel [7] when data are requested by the second core, the L1 cache on the first core only updates the values in the shared L2 cache. The update process does not take as much time as a transfer through memory bus and that is the difference we are looking at. The same effect will also appear in the experiment 3.1.3 that explains how data are transported from the L1 cache on one processor into the L1 cache on the other processor.

To make it even more interesting, let us compare Figure 3.2 with Figure 3.3. We have assumed that in the two previously described scenarios, the read and write accesses take the same time. However, this is true only for cores that share the L2 cache where both read and write times are around 60 clocks and as you will see later, we cannot disprove that the same time is a coincidence. In two other cases, the times are 18 clocks read and 97 clocks write for cores in the same package with the non-shared L2 cache and 10 clocks read and 48

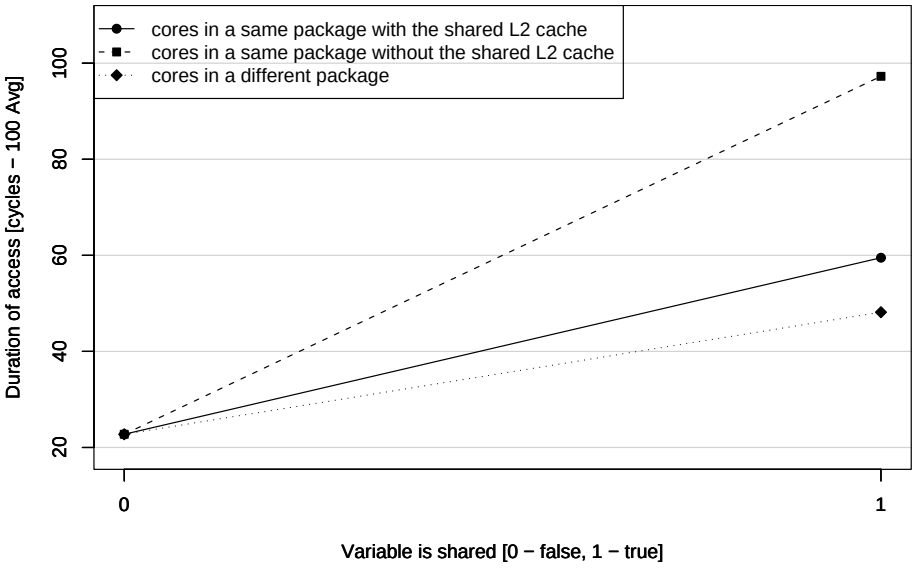


Figure 3.3: Time to write variable on the first core, the second core is reading

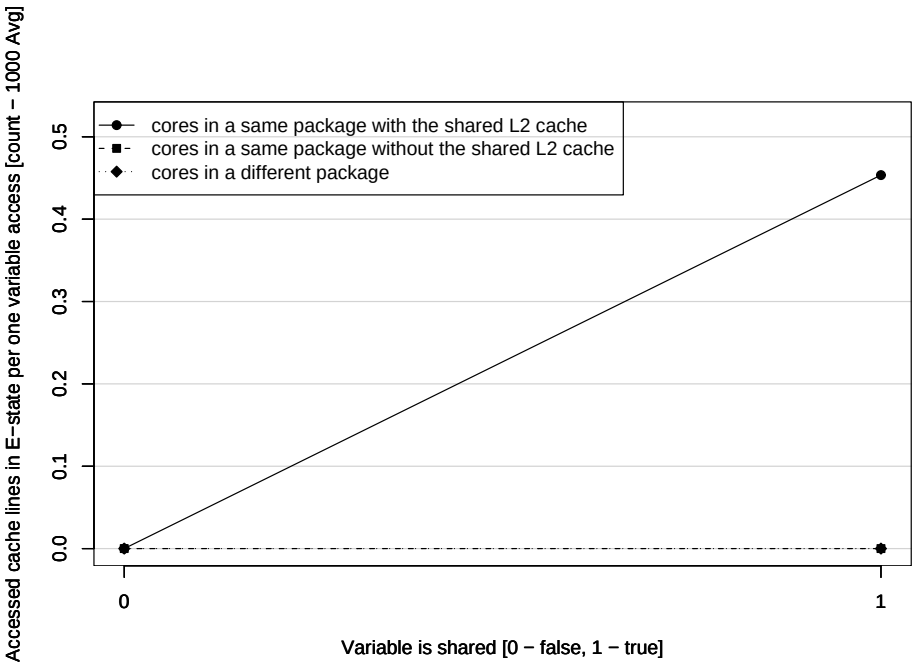


Figure 3.4: Number of accessed cache lines in Exclusive state per one variable access; both cores write

clocks write for cores is different packages. It looks like there is a different number of subsequent reads on one core and subsequent writes on the other core. The code used for reading and the code used for writing is different and this could happen, but we have nothing (including hardware counters) that supports this hypothesis.

Actually we have a proof that there is some subsequent access but we cannot tell that number of subsequent accesses is same for both cores. Figure 3.5 shows the number of HITM snoop responses per one variable access. This shows how many times (per one access) the other core steals shared data from our core. For cores with the shared L2 cache, this number is near one, showing us that the other core requests memory right after our core performs an access. This means that there is no subsequent access for the writing core. The problem is that we do not know how the reading core performs. We have a similar HIT counter for the reading core but for some reason, it appears not to give any proper results.

What we can also see from the results in Figure 3.5 is that other cores, which are further from each other, can perform more accesses before the other core requests the data. For CPUs in the same package with non-shared L2 cache, the data request from the other core comes after 10 subsequent accesses (value 0.1 in graph) and for CPUs in a different package, the number of subsequent accesses is 20 (value 0.05 in graph).

Write on both cores is presented on Figure 3.6. Load with both cores writing is greater (for all CPU locations) than with results on Figure 3.3, where the other core was only reading. A non-shared single-core write access costs 23 clocks. A shared access for cores with shared L2 cache costs 78 clocks, for cores with non-shared L2 cache it is 117 clocks and for cores in different packages it is 76 clocks. Here, the same situation with fast access to shared L2 cache as in Figure 3.3 also exists.

Open Issues The only issue here is with the HIT counter. We do not know if we are observing a bad counter or the counter is correct but our explanation of this effect is wrong.

Effect Summary Here we have simulated a scenario with a heavily accessed shared variable.

The average read access can grow from 4 clocks for simple non-shared access up to 60 clocks for shared access if the other core is writing. The average non-shared write costs 23 clocks and the shared write climes up to 117 clocks.

It also depends on the location of these two cores. Cores that are further apart from each other can do more accesses in a row before the other core requests

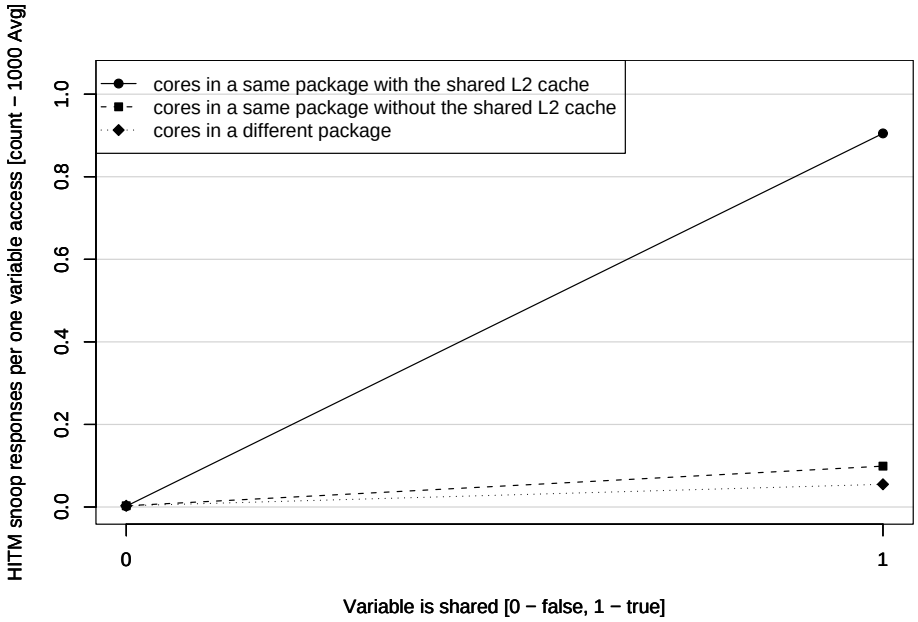


Figure 3.5: Number of HITM snoop responses per one variable access; we see core that writes, second core is reading

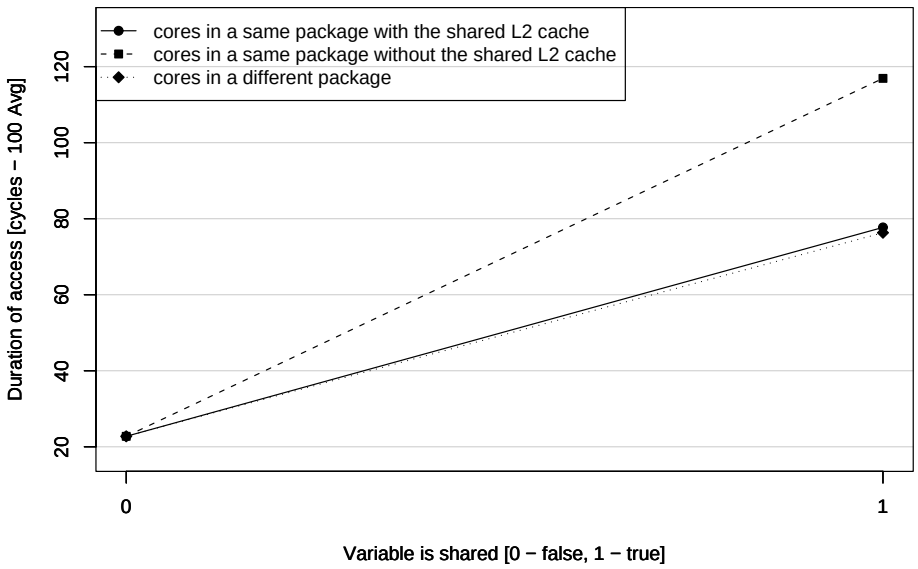


Figure 3.6: Time to write variable on the first core, the second core is also writing

the memory again. Also, cores that share the cache can take advantage of it and transfer the data faster.

When we have a shared variable which is often modified:

- the average read access can be 20 times higher than in the case of non-shared variable
- the write access can be on average 5 times slower than the non-shared access
- cores further apart from each other can do more accesses in a row before the other core requests the memory again

3.1.3 Producer - Consumer

Now we take a look at synchronization from a different point of view. In the previous experiment, we had one shared variable synchronized between two cores. Now imagine a situation where the first core produces some data and the second core uses (consumes) this data. This is also the case where the MESI protocol is used to transport data (maintain cache coherency). As we will show, there is number of ways how data can be transported from one core to another.

Producer is a part of the benchmark that allocates and warms up the data as presented in Listing 3.2. At the beginning, data are allocated using the random trail allocator (Listing B.3). There are two types of warm-up depending on in which state we want the data in a cache. In the first case, whole data are accessed and left in the cache if they fit. This state is called "clean data". In the other case, the data are fetched into the cache and a cache line is artificially changed to a modified state by simply zeroing and restoring an original value (modified data). For advance to the next address NEXT_ADDR macro (Listing B.1) is used. Now everything is prepared and consumer is notified. After consumer processes the data, producer takes care about deallocation.

Listing 3.2: Data creation (producer)

```
// allocate memory area
void * * mMemArea = MemInit::random(memSize, stepSize);

// warm-up
void * volatile * p = mMemArea;
while(*p != mMemArea) {

    if(write) {
        // tmp var for orignal value
        void * origVal = *p;
```

```
            // write there
            *p = origVal;

            NEXT_ADDR(p);
        }
        else {
            NEXT_ADDR(p);
        }
}
NO_OPTIM(p);

// indicates that data are prepared
atomic_set(consumeSyncVar, 1);
// wait for consummation
while(atomic_read(consumeSyncVar) != 0) ;
```

When the data are prepared, consumer processes the data in the way we can see in Listing 3.3. On the first line, consumer waits until producer allocates and initializes data. When the data are prepared, consumer simply reads a prepared trail and measures the time and PAPI counters. After the measurement, producer is notified that the memory block is processed.

Listing 3.3: Data processing (consumer)

```
// wait until something is produced
while(atomic_read(consumeSyncVar) == 0) ;

// get pointer to allocated memory block from producer
void ** memArea = Producer::getMemArea();
void ** p = memArea;

MEASUREMENT_START

// read whole memory prepared by producer
while(*p != memArea) {

    NEXT_ADDR(p);
}

MEASUREMENT_STOP

NO_OPTIM(p);

// let producer know that all is consumed
atomic_set(consumeSyncVar, 0);
```

A crucial parameter for this experiment is the producers and consumers position.

The position decides how exactly the data are transferred from one core to another. Next no less important parameter is memory size. If a memory block is small enough, it can reside in the L1 or L2 cache. Otherwise it is flushed back to memory and is requested from there. Note that step size in the random trail allocator (Listing B.3) is here also 64 bytes (cache line size). This is because the whole cache line is transferred at once and we don't need to access another location in this cache line.

Experiment: Producer - Consumer

Purpose Determine the effect of data sharing among different cores.

Measured Time to transfer data from the core that produces (code 3.2) data to the other core that consumes (code 3.3) the data.

Parameters produced data size: 16 KiB, 512 KiB, 8 MiB; produced data state: clean, modified; consumer location (relatively to producer): same package with the shared L2 cache, same package without the shared L2 cache, different package

Expected Results Data marked as clean are transferred directly from memory. This is because on our Intel Server, there is no other way how to transfer memory from one processor to another. An exception is the case where producer and consumer share the L2 cache and data are small enough to fit in. In this case consumer fetches the data right from the L2 cache.

When consumer accesses modified data, this data are first flushed from producers caches to main memory and after that, consumer fetches this data from memory. The exception with the shared L2 cache where consumer fetches modified data from the L2 cache is also here.

Measured Results Before we get to actual results, let us clarify what we will see in measured results. Here we measure the average over whole transported memory. With a bigger memory block, we need more memory accesses to transport whole data from one core to another, meaning that measured result (average) will be composed from more accesses. As was mentioned above, one transport unit is whole cache line (64 bytes) and times presented below are times to transfer cache line from one core to another.

As we can see on Figure 3.7, cores with the shared L2 cache share produced data and cost to fetch this data from L2 cache is about 23 clocks. Variants with no L2 shared cache have to fetch this data from memory with a cost of 135 clocks. As opposed to what we will see with modified data in Figure 3.8, in the shared L2 cache variant an access to data that are in L1 cache (size

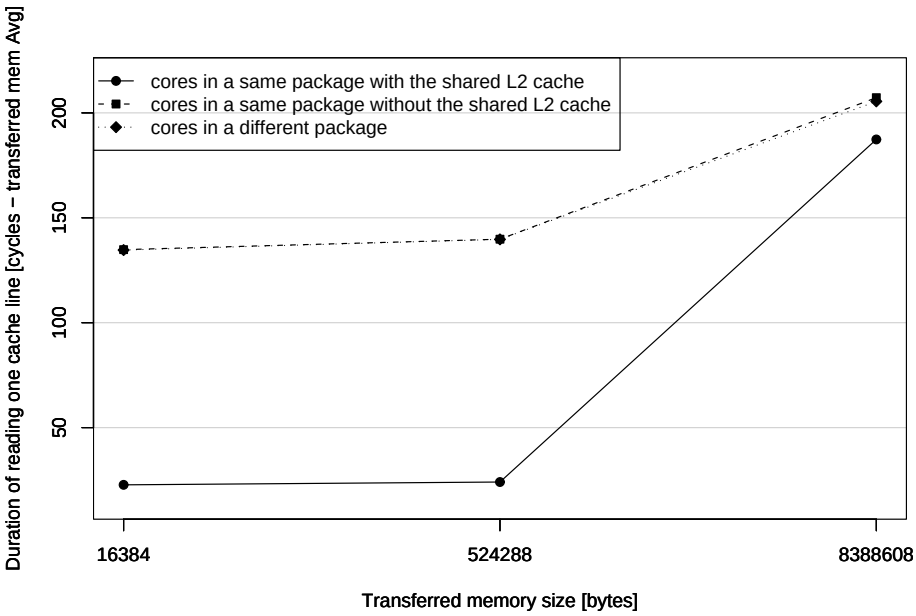


Figure 3.7: Time to transfer one cache line (clean data) between two cores

16 KiB) is also fast. According to Intel, the reason is that on our Intel Server the L1 cache is non-exclusive with the L2 cache and this means that data in L1 cache can be also present in L2 cache.

A rise at 8 megabytes for all variants is due to misses in data TLB where an additional access to the cache is necessary. Actually, address translation data are in main memory before page is accessed first time. The first look for the address translation data fetches the data from memory and following TLB misses for this page are satisfied from the cache. We count the average over the whole transferred memory so memory accesses and cache hits are mixed together and that creates the rise observed in results.

At 8 megabytes, the shared L2 cache variant differs from the two others. This is because some data still reside in the L2 cache and cache hits lower the average. That effect is also visible with modified data on Figure 3.8.

Scenario with modified data on Figure 3.8 differs from clean data 3.7 in two cases. The first difference can be observed by looking at the scenario named "core in same package with non-shared L2 cache". Time to transfer clean data is 135 clocks but time for modified data is only 71 clocks. As mentioned earlier modified data has to be written back to memory before they can be accessed by another core. We suppose that data addressed to memory controller are snooped (thanks to a shared memory bus) by the other core that needs them.

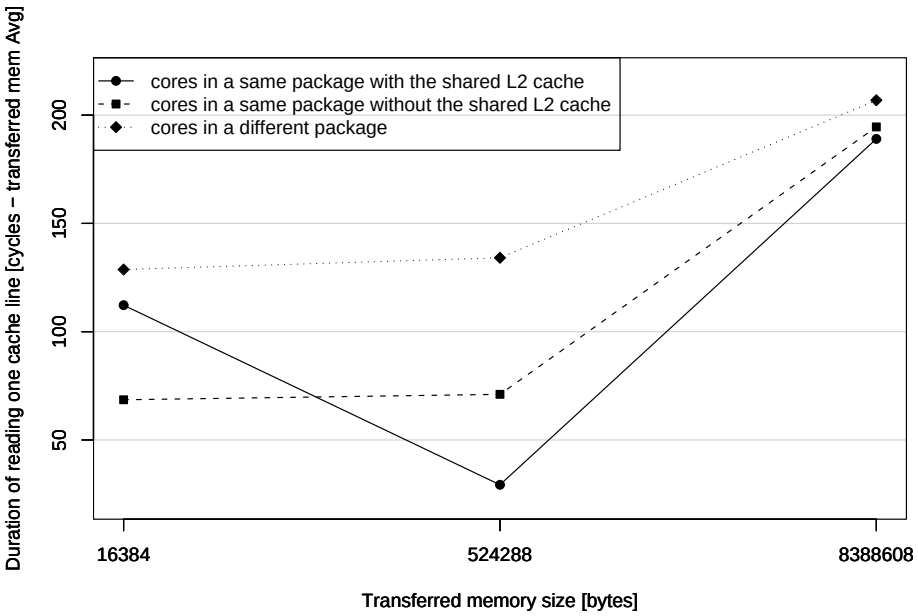


Figure 3.8: Time to transfer one cache line (modified data) between two cores

The core only has to wait for data to show up on the bus without sending a request to slow memory. Core in a different package than producer does not share the memory bus therefore it cannot use this optimization.

As said in Intel Knowledge base [7], modified data in the L1 cache are updated in the shared L2 cache where the other core can read them. We can see this case on Figure 3.8 (a shared L2 cache scenario with 16 KiB memory size block). The question is, why the time is 112 clocks if the data are in the L2 cache. One delay is introduced by the L1 to L2 update, but we do not believe that the update takes so long. Figure 3.9 shows the number of memory transfers. As we can see, for our scenario with 16 KiB memory block there is even more memory transfer than for scenarios with non-shared cache. With the L1 to L2 update, data are also transferred to memory. Only Intel knows the right reason.

Open Issues The first open issue is about the very long L1 to L2 data update (Figure 3.8) when modified data are transferred from the first L1 cache to another L1 cache (scenario with 16 KiB block). The L1 caches are located on cores sharing the L2 cache.

We do not know if another processor or core can read the data which are addressed to memory controller. This is our explanation for case where clean data transfer (Figure 3.7) takes longer time then modified data transfer (Figure

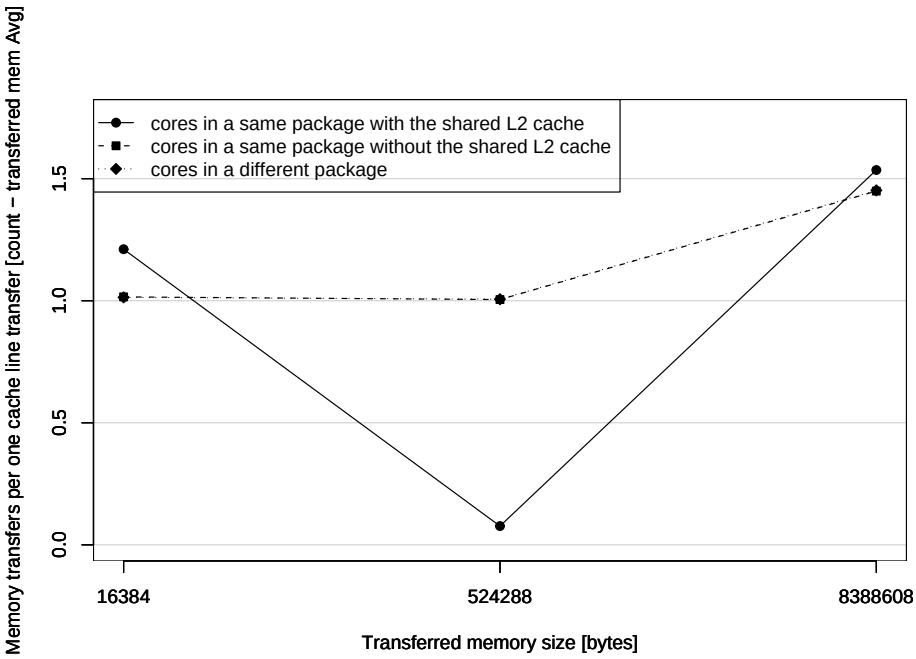


Figure 3.9: Number of memory transfers when consumer access modified data

3.8).

And one little thing remained. On Figure 3.7 and also on Figure 3.8 is visible a little rise between transfer from the L1 cache (16 KiB) and transfer from the L2 cache (512 KiB). We suppose that an eviction from the L1 cache in this case does not need to pass the L2 cache but the data are evicted directly into memory and that is why the eviction from the L1 cache is faster than the eviction from the L2 cache. Unfortunately, we do not have the reference that confirms that data can be evicted from the L1 cache directly into memory. So we left this as an open issue.

Effect Summary Here we tried to measure how costly it is to transfer one cache line from one core to another.

The clean data are fetched from memory and there is no transfer between two cores. The exception is the shared L2 cache where both cores can access the same data right there. Time for the cache line memory fetch is 135 clocks. Time for the cache line fetch from the L2 cache is only 23 clocks.

The modified data are transferred through memory bus. This scenario is faster than simple memory access and costs only 71 clocks. When the modified data are located in the L2 cache the scenario is same as for clean data. The modified data located only in the L1 cache are first updated (cache line is updated) in

the L2 cache and then are available to the second core. The L1 to L2 update needs additional write back to memory which rises the time to 112 clocks.

If we transfer data from one core to another:

- data transfer is relatively cheap for cores that share the L2 cache
- data transfer to the core that does not share with the source the L2 cache is 5 times slower then to the core which does
- modified data can be transported in some cases more than two times faster than clean data

3.1.4 Cache sharing

In the previous chapter, we were looking mainly on synchronization between caches in different cores. Now modern processors come with an improvement that should simplify and speed up data sharing among several cores. In current processors, two or more cores share highest level of caches. This can bring advantages when two cores owning the same cache share data. No MESI protocol is necessary to ensure synchronization and coherency. Also one core can use whole capacity of shared cache if other cores do not need it. In Producer - Consumer experiment, we could see some advantages that the shared cache brings. But the shared cache also has limitations and disadvantages. On our Intel Server, we saw limitations when we had data located in non-shared L1 cache. Data was transferred through memory and shared cache did not help[1]. Shared caches have an obvious problem that more cores have to compete for space in this cache. Another difficulty is that multiple cores share bandwidth to one shared cache. This could be a problem when cores heavily access cache or memory. In two following experiments, we will focus on bandwidth and space limitations in shared caches.

3.1.5 Shared cache - Bandwidth

As we will see in this experiment, bandwidth to the cache owned by more than one core is shared.

Memspeedmp visible on Listing 3.4 is one of our most used benchmarks and it is a base for other benchmarks that only slightly modifies it. Main purpose of this benchmark is to allocate memory block and access it as fast as it can.

[1] Here is advantage of current AMD processors. They are using System Request Interface (SRI), a special bus that speeds up the data transport between cores. Upcoming Intel Nehalem microarchitecture will have similar solution called Intel QuickPath Interconnect (QuickPath, QPI).

What is not presented in Listing 3.4 is a memory block allocation that is done using the random trail allocator (Listing B.3) or the linear trail allocator. The linear generator is used when we want to take advantage of a data prefetcher. The prefetcher generates higher loads but it can also introduce effects that are harder to explain. The random allocator eliminates the prefetcher and other effects that come up with the linear access. For this experiment, we will use the linear allocator.

The first four lines of the source code (Listing 3.4) are just preparation of a helper variable. Helper variable is connected with usage of multiple pointers. Linked pointer trail does not allow us to have more than one pending request into memory because we do not know the next address until the current one is fetched. But CPU supports multiple outstanding requests into memory and multi-pointer version can take advantage of this to create bigger load. We have also a version of memspeed called memspeedrg with a random address generator that should generate addresses independently of a data fetch but even with a simple generator the overhead is too big.

Next in code is read or write memory advancement performed for each of the pointers in one step. One step means that all pointers are moved before the first one is moved again. For advance to the next address NEXT_ADDR (Listing B.1) and NEXT_ADDR_WITH_WRITE (Listing B.2) macros are used.

Listing 3.4: Multi-pointer variant of memory load generator (memspeedmp)

```
// helper variable
void ** * p = new void ** [pointers];
for(int i = 0; i < pointers; ++i) {
      p[i] = memAreas[i];
}

if(write) { // write
      MEASURED_BLOCK_BEGIN {

            for(int i = 0; i < pointers; ++i) {
                  NEXT_ADDR_WITH_WRITE(p[i]);
            }

      } MEASURED_BLOCK_END
}
else { // read
      MEASURED_BLOCK_BEGIN {

            for(int i = 0; i < pointers; ++i) {
                  NEXT_ADDR(p[i]);
            }

      } MEASURED_BLOCK_END
```

```
}

NO_OPTIM_POINTERS(p, pointers);
```

In this experiment, one of the variant of memspeedmp is also used. This variant is called thrasher and an improvement is that it can control the intensity of an access. The code of the thrasher is almost the same as for memspeedmp except one thing. Before the pointers are moved to the next address, a variable (depends on parameter) number of NOP instructions is performed. This is done by NopFunc (Listing B.4). Adjustment for write can be seen on Listing 3.5, read is adjusted in the same way.

Listing 3.5: Part of thrashing benchmark (thrasher)

```
MEASURED_BLOCK_BEGIN {
    // do NOP instructions before memory access
    // frequency control mechanism
    callNopFunc(nopFunc);

    for(int i = 0; i < pointers; ++i) {

        NEXT_ADDR_WITH_WRITE(p[i]);
    }

} MEASURED_BLOCK_END
```

Memspeedmp runs on the first core and its purpose is to maintain the access into the shared cache and measure results. On the second core thrasher generates variable workload by inserting different amount of NOP instructions between cache accesses. The second parameter is memory block size. In one scenario, we want to hit the shared cache, in another we want to miss it. As we will see, this is an important difference.

Experiment: Shared cache - Bandwidth

Purpose Determine the latency to the cache owned by multiple cores.

Measured Latency to shared cache measured by memspeedmp (code 3.4) while thrasher (code 3.5) accesses this shared cache simultaneously.

Parameters memspeedmp memory block: 128 KiB; memspeedmp memory access: linear - read only; memspeedmp pointers: 1[2]; thrasher memory block:

[2] Here we measure only one pointer so we could use single-pointer memspeed. The reason is that we also measure more pointers but the results are not so interesting to include them here.

128K KiB, 8 MiB; thrasher memory access: linear - read only; thrasher pointers: 8; thrasher NOP instructions: 0, 1 - 512 Ki exponential step

Expected Results With increasing number of NOP instructions comes a higher latency between thrasher accesses. If there is some limited bandwidth to the shared cache, we should see it with the less number of NOPs.

Thrasher has two modes. In the first mode, thrasher hits the shared cache. In the second mode, thrasher misses the shared cache and data has to be loaded from memory. A memory fetch takes much more cycles than a hit access. So we expect that times with the hit access will be more frequent and we will see a higher number of cycles then in the scenario with the miss access.

Measured Results On Figure 3.10, we can see the overhead of a simultaneous access. We can observe first change at 32 NOPs then it lowers until 8192 NOPs and after this border it keeps constant. This indicates that until 32 NOPs, CPU can process instructions so fast that this NOP hole does not create any performance changes when observing number of L2 cache accesses. After 32 NOPs the artificial delay is in effect and the overhead keeps falling until 8192 NOPs and after than thrasher does not generate any measurable overhead.

An access without any interference of thrasher costs 17 clocks. This is the time to access the L2 cache plus the overhead of a multi-pointer support. When thrasher generates the maximum load, time rises to nearly 19 clocks. Two clocks is the overhead generated by thrasher that hits the shared cache simultaneously with memspeedmp.

On Figure 3.11 are results with thrasher missing the shared cache and memspeedmp hitting the shared cache. The curve has the same behavior as in the previous scenario but the numbers are different. We expected that this scenario will have a lower overhead than the one presented on Figure 3.10 but the results show that we were mistaken. The access costs 32 clocks with thrasher on second core with no NOPs. This is almost twice as much as the access time with no interference. We believe that the cache is actually much more loaded when CPU misses it than when it hits. Handling the cache hit only means to find a cache line in a cache but the cache miss is more complicated. On our Intel Server, when CPU misses the L2 cache, it has to send a request to memory to fetch that data. Now is good time to point out that thrasher uses eight pointers and keeps the L2 cache busy all the time.

We also tested the write access but we did not get any interesting results. The overhead was smaller than in the case of the read access and the complexity of the write access could play some role here.

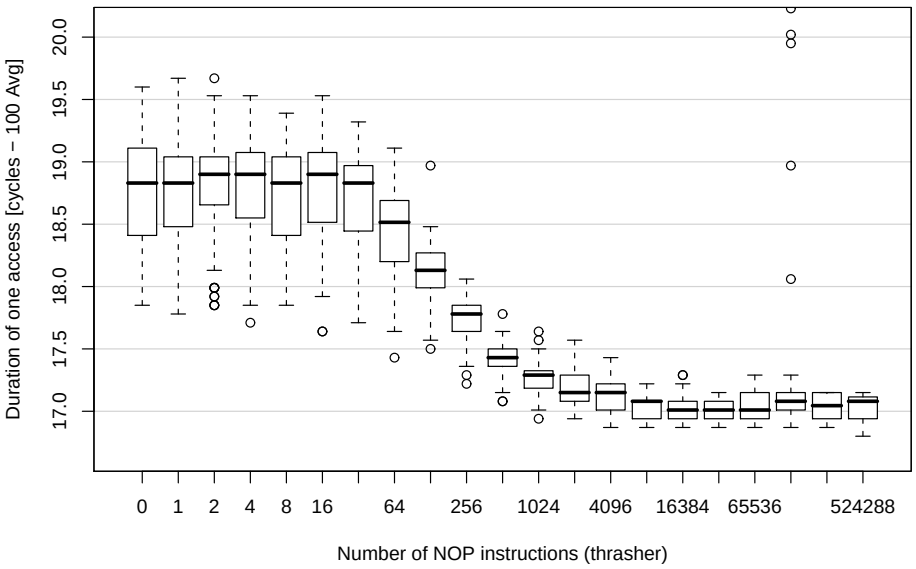


Figure 3.10: Time to access the shared L2 cache while thrasher is hitting the shared cache

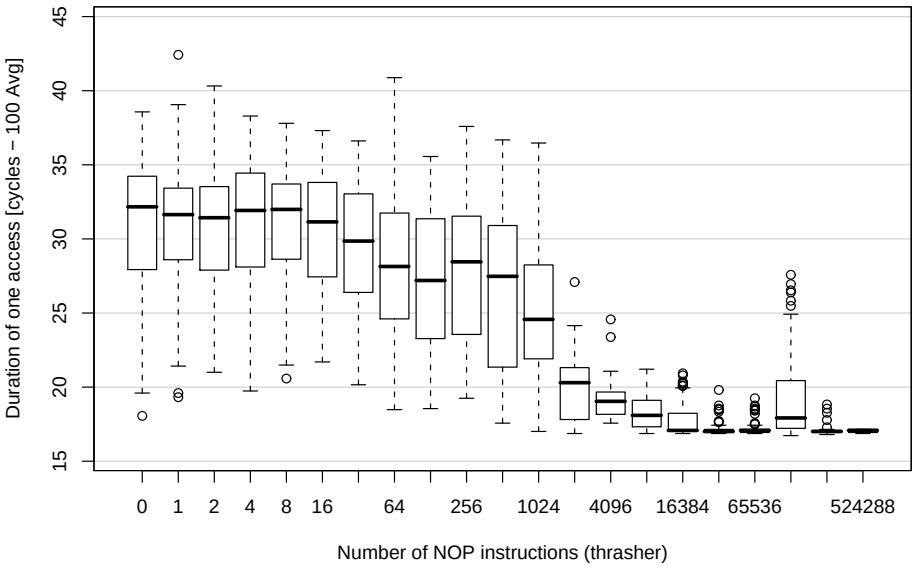


Figure 3.11: Time to access the shared L2 cache while thrasher is missing the shared cache

Open Issues We have no proof that the access missing the L2 cache on Figure 3.11 loads the memory more than the access hitting the L2 cache presented on Figure 3.10.

The rise at 128 KiB of Figure 3.11 is the measurement noise. We can adjust the framework to correct the measurement but these settings apply to all test. And that means rerun all test with the new, not well tested, settings.

Effect Summary This experiment confirms bandwidth limitations of the shared L2 cache.

If both cores are hitting the shared L2 cache then the overhead is relatively small and rises from 17 clocks for single-core access to 19 clocks for multi-core access. If one of the two cores starts missing the L2 cache, the overhead starts to grow and the core that is hitting the shared L2 cache has the access time about 32 clocks. But this is only true for cores with high number of accesses. If one of the two cores starts slowing down, the overhead disappears.

When two cores compete for the bandwidth to the shared L2 cache:

- if both cores are hitting the L2 cache the overhead is only about 12%
- if one of the two cores starts missing the L2 cache, the access time for core that is still hitting the L2 cache can be twice as high than in the case of the single-core access

3.1.6 Shared cache - Thrashing

Space in shared caches is limited and multiple cores have to compete for it. There are many factors that influence if previously fetched data still reside in cache or were already flushed back to memory.

If program uses too much memory, it simply cannot fit into a small cache. Access patterns are also important. Caches use special algorithms (eviction policy) to replace old cache lines for new ones. The problem is that eviction policy is not well documented and we can only guess how it exactly works.

The cache sharing introduces another level of complexity. We have more than one core with its own requirements. Also the intensity of memory access starts playing an important role here. Caches use LRU based eviction policy in each cache set. A core that accesses its data more often has greater probability that it finds the required data in the cache.

Creating a model that reflects all these variables is not easy. Let us assume for now that we have the model that describes cache behavior and we know how program

operates with memory. Program works with virtual memory but shared cache will be probably physically indexed. That means we cannot simply use a model directly with program behavior but we have to also include some statistics that determines a physical memory mapping[3].

In the following experiment we look at this problem from a different perspective. We will test a set of cases with different parameters and look how they perform.

Benchmark configuration is almost the same as in the previous experiment. It contains memspeed, which is one pointer variant of memspeedmp (Listing 3.4). The difference (Listing 3.6) is that the code for multiple pointers is missing and there is no additional overhead except the access itself. The thrasher on the second core is the same as presented in Listing 3.5.

Listing 3.6: Measurement part of memspeed (write)

```
MEASURED_BLOCK_BEGIN {

    // single pointer access (only difference in memspeedmp)
    NEXT_ADDR_WITH_WRITE(p);

} MEASURED_BLOCK_END
```

We will simulate an environment where two applications have their own workspace in a shared cache. The main focus is to change workspace sizes and the intensity of accesses and observe how the workload fits in the cache.

Both memspeedmp and thrasher uses random access as prepared by random trail allocator (Listing B.3). As mentioned before, the variable parameters are the memory block size for both memspeed and thrasher and the NOP delay for thrasher.

The difference from all other benchmarks is that here we focus on the number of evictions in the shared cache. We use this counter because from simple times we cannot directly see how much two measured cores compete for the shared cache. Write mode on thrasher is used so we can distinguish between cache lines that are used by thrasher and lines used by memspeed.

Experiment: Shared cache - Thrashing

Purpose Determine how cache sharing affects the cache space requirements.

Measured Number of shared cache misses measured by memspeed (code 3.6) while thrasher (code 3.5) accesses this shared cache simultaneously.

[3] This is true only for set associative caches.

Parameters memspeed memory block: 128 KiB - 8 MiB exponential step; memspeed memory access: random - read only; thrasher memory block: 128 KiB - 8 MiB exponential step; thrasher memory access: random - write only; thrasher pointers: 1; thrasher NOP instructions: 0, 1 - 512 Ki exponential step

Expected Results The testing set is relatively large so we will pick only two results that are interesting and also appear in Measured Results. The first result will include thrasher with small memory block size and the second will include thrasher with large memory block size.

The expectations are the same for both cases. For larger memspeed memory block sizes, thrasher competes with memspeed for space in the cache and we should see how memspeed evicts thrasher modified cache lines. The first few NOP holes do not affect results as we saw in Shared cache - Bandwidth 3.1.5. When the NOP hole is large enough, we should see slow fall of evicted lines down to zero. The zero value indicates that thrasher does not influence memspeed at all.

Measured Results First, we will look at scenario presenting thrasher with larger (4 MiB) memory block. The results can be seen on Figure 3.12. Our expectations were correct in all cases. The first memspeed memory block size that is affected by thrasher is 1 MiB but the effect can be barely seen. But for sizes equal and larger than 2 MiB, we can see slow fall after 64 NOP instructions. This is the threshold value where NOP hole takes effect and at 65 Ki NOPs is the second threshold value where thrasher has no effect on measured workload.

Memspeed with smaller memory block sizes is not affected by thrasher. First reason could be that memspeed and thrasher are small enough that both fit into the cache. Other option is that memspeed is small and fast enough and because of LRU eviction, thrasher cannot evict any of the memspeed lines. In each case, our workload stays in the cache so we will not investigate which of the following options is happening. One can find out this by looking at number of L2 cache misses done by thrasher.

Memspeed with 2 MiB memory block size evicts 14% of thrashers cache lines. For 4 MiB, this number is equal to 25% and for 8 MiB is over 30% of evicted modified lines. But we should realize that not each of these lines are reused if there is only memspeed running. This is especially true for memspeed with 8 MiB memory block where only small amount of cache lines is reused[4]. 30% of accessed cache lines is occupied by thrashers data but it does not mean that

[4] How the L2 cache is used is explained in another "experiment" called Eviction behavior in physically mapped cache

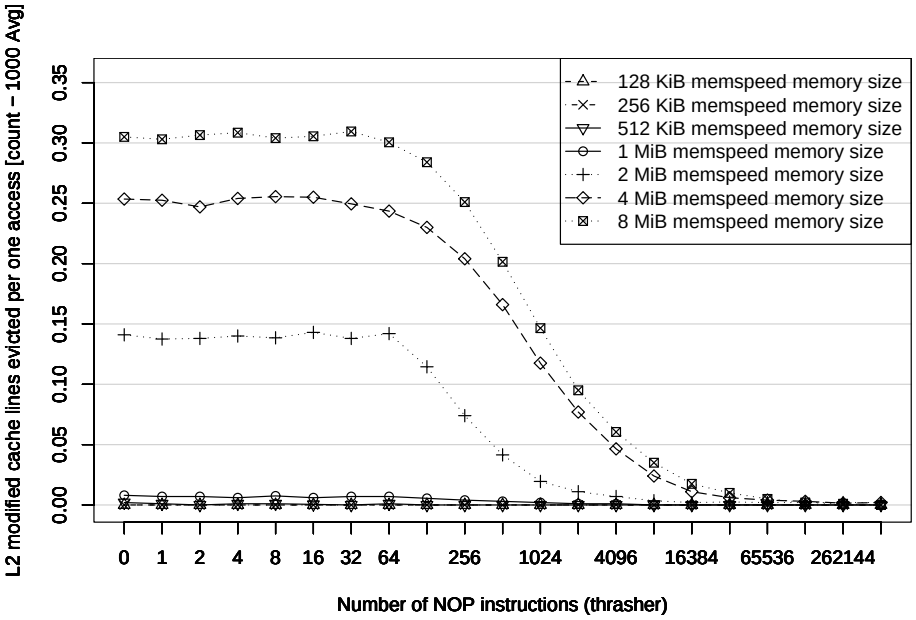


Figure 3.12: The number of L2 modified cache lines evicted - thrasher allocates 4MiB memory block

if it was not, it is used again by memspeed. It is the best what we know we can measure though. With simple measurement, we have no information if the cache line evicted by thrasher is reused again. It could be possible to compute (with some tolerance) the reusability of lines evicted by thrasher but that is out of the scope of this thesis.

On Figure 3.13 we can see similar scenario as the previous one but the thrasher memory block size is now only 1 MiB. Number of evicted cache lines with zero NOPs is now up from 6% to 8% for memspeed with memory block size larger than 1 MiB. But the interesting effect which we can see here starts after 64 NOPs. At this point, instead of the expected fall, the curve starts rising. Now we need to realize what is happening here. Thrasher is small and fits into the L2 cache. Until 64 NOPs thrasher is fast enough and thanks to LRU eviction policy is able to keep most of its data in the shared L2 cache. Memspeed has large working set and is not able to hit each set so fast as thrasher but thrasher starts slowing down. At this point thrasher is so slow that it losses the LRU eviction policy advantage and memspeed starts evicting more of thrashers cache lines. Fall after 512 NOPs indicates that thrasher does not have enough power to fill cache lines with its data.

Now we need to look a little bit forward at another experiment called Memory

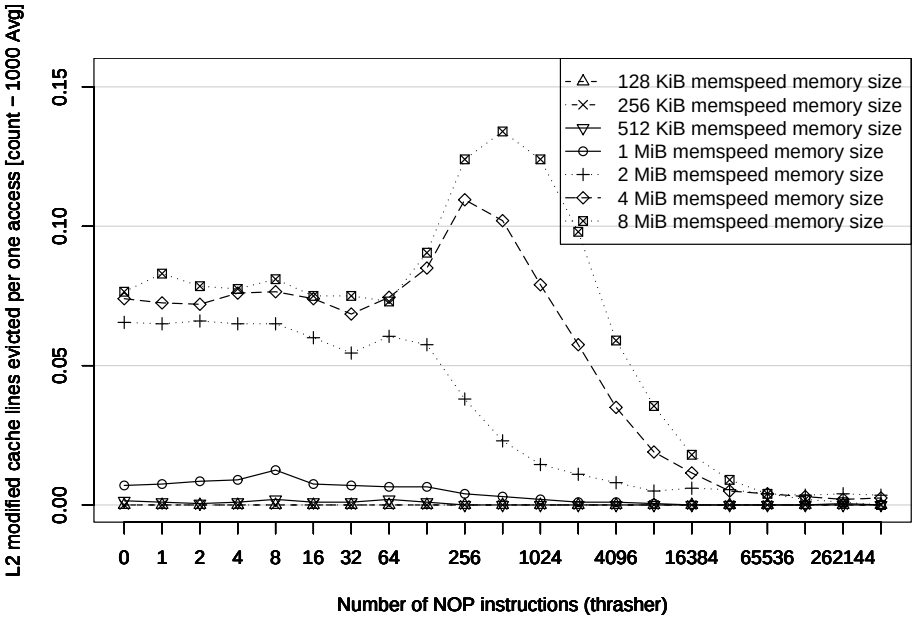


Figure 3.13: The number of L2 modified cache lines evicted - thrasher allocates 1MiB memory block

bandwidth - single-pointer. There are results on Figure 3.16 that are also useful here. Test is composed from two equivalent memspeeds that increase memory block size. One case covers cores that share the L2 cache. We can see that cache sharing with two workloads behaves like there is a cache with half size for each of them. Actually we think that the cache makes no difference between data from one core or another and same workloads divide the cache between themselves equally. This is one of the disagreements between our research and the paper of Ulrich Drepper [10].

Open Issues The experiment does not give any estimate about thrashers bad influence because we do not know which cache lines evicted by thrasher are reused again when memspeed is running alone.

Effect Summary Experiment determines how two cores compete for the space in the shared L2 cache.

Cores sharing the L2 cache start competing for the cache space when a sum of the allocated memory blocks is over 2 MiB. If the sizes of the memory blocks are twice as large as the capacity of the shared L2 cache, over 25% of accessed cache lines can be occupied by other core. But this does not mean that all 25% of cache lines would be used in the future.

The data access intensity is also the critical factor here. A core that does not access its data regularly is also evicted because of LRU eviction policy. The shared cache does not care how much of the total capacity each core occupies.

When two cores compete for the space in the shared L2 cache:

- a competition for the cache space starts when a sum of the allocated memory blocks is over 2 MiB
- a core with smaller size of data could be also evicted if it does not access the data fast enough
- the shared cache does not distinguish between data sources (different cores)

3.2 Memory

In previous chapter, we were talking about resource sharing inside CPU. System memory (memory) is another resource that is shared between multiple cores and even processors.

In cache experiments, we consider space and bandwidth requirements and we can do the same with memory. But space requirement in case of memory is somewhat different than with caches. It is not usually expected that the whole program fits into the cache but it is expected that it fits into memory. Size of caches is transparent for programs and most of them simply ignore it. When a system runs out of memory, every call for allocation fails. This is the reason why program cannot the size of memory. Operating system knows ways how to resolve this problem. For example it can kill processes with the highest demands or move some pages from memory to swap. But the first scenario is not transparent for running processes and the second one is a serious performance issue and used only as a last resort.

We will present series of experiments focused on bandwidth of the shared memory bus and memory controller. With respect to what was said above, we will skip space competition benchmarks.

3.2.1 Memory bus and memory controller

When we say memory it is not only memory module but also memory controller and memory bus. On our Intel Server we have four cores in one CPU package that share the same memory bus. Whole system is composed from two CPUs and each of them has its own memory bus. Buses are then connected to one memory

controller. Memory modules are hidden behind memory controller and therefore are not important for us because we cannot distinguish who cause the slowdown.

3.2.2 Eviction behavior in physically mapped cache

Let us start with complementary experiment that describes one effect that we will see in all our results. Until now, we have ignored that program works with virtual addresses but memory and last levels of caches are indexed physically. This results in a situation where the physically indexed cache can still hold some part of allocated data, even if the data size is greater then the capacity of this cache.

This experiment is only complementary so we will quickly describe it and show results. For this experiment, we use memspeed (Listing 3.6) on one core. Memory access is read-only random. The first part of the experiment has the memory blocks sizes from 16 KiB to 64 KiB and 8 KiB linear step is used. This configuration will show us the behavior on a border between the L1 and L2 cache. The second part uses memory blocks with sizes from 2 MiB to 8 MiB with 64 KiB linear step. Here we will see what happens when the memory block is greater than the L2 cache.

Results can be seen on Figure 3.14, be aware that the first and second part as described above is in one picture. The little jump we can see at the beginning is the border between the L1 and L2 cache. Data with the size of 32 KiB still fit into the L1 cache but this is no longer true for 40 KiB memory block. If data do not fit into the L1 cache, each fetched cache line flushes another cache line from the L1 cache back to the L2 cache. So if the data are requested again they have to be fetched from the L2 cache. But this is true only if all of them are true:

a) the L1 cache is virtually indexed meaning that if we allocate continuous block in virtual address space it will be also continuous in the L1 cache

b) The L1 cache uses LRU eviction policy or something very similar to this

c) whole memory block is accessed before it is accessed again[5].

All this combined together causes, that cache line is flushed back to the L2 cache before is accessed again and this applies to all cache lines from our memory block. We ignore the data that are used by our framework. Framework owns only small amount of data that are used frequently and all other data are flushed back to the L2 cache or memory in warm-up.

[5] This is a feature of random trail allocator.

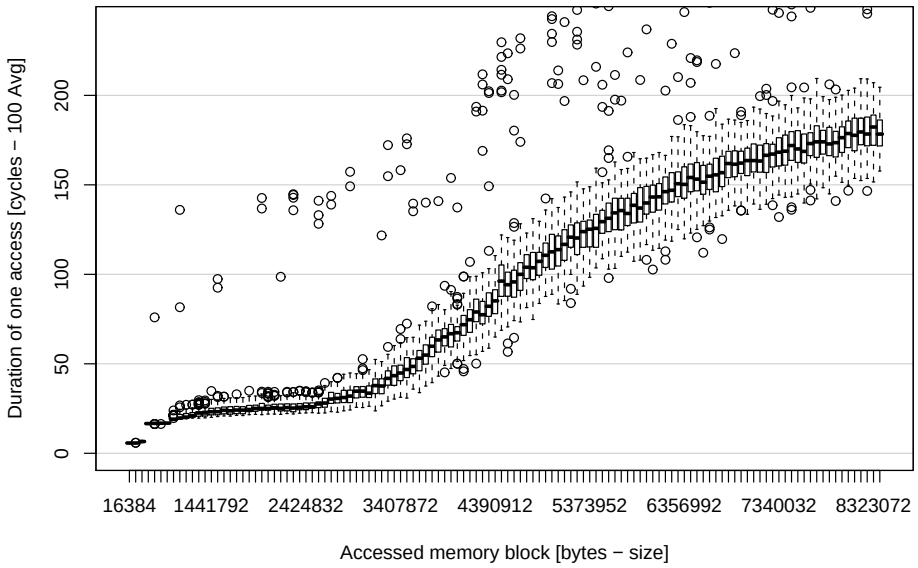


Figure 3.14: Eviction behavior in physically mapped cache

The second part of the measurement starts in the place where points appears right over the curve (little before the place where the second number starts on x axis) and ends with the end of the graph. Memory block with 2 MiB is the point where the data no longer "fit" into the L2 cache. Quotation marks used around fit is not coincidence. The reason why we see gradual rise as opposed to what we saw on the border of the L1 and L2 caches. Some of the cache lines are still in the L2 cache and some of them are not. To be sure we can check number of L2 cache misses on Figure 3.15. Here we can see that number of misses raises exactly the same as times.

System memory is physically indexed and memory blocks allocated by program are no longer consistent here. The basic building block in memory is one page and operating system decides where the pages are placed in memory and what physical addresses they rally have. The L2 cache is also physically indexed so here applies the same rules as in memory. This is a violation of the rule a) mentioned above and the data in the L2 cache are not in a one block but are located all over the cache. Now, when we uses greater memory blocks, the chance that more parts occupy the same cache set in the L2 cache grows. Some of the sets in the cache are fully occupied and some of them can be still half empty. In this situation, number of hits or misses depends on number of overfilled cache sets. And because the data size grows, also the number of overfilled cache sets grows. This means more fetches to memory and higher latencies.

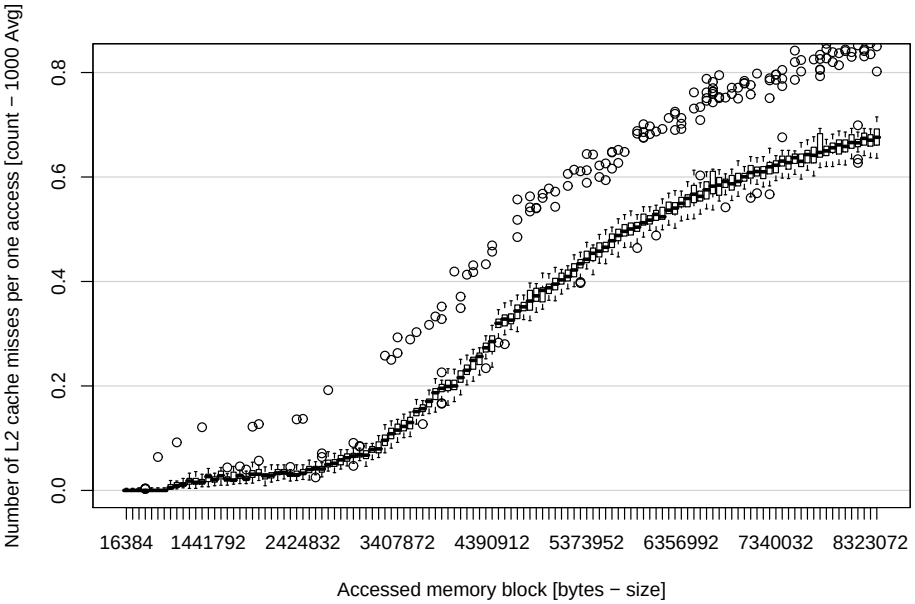


Figure 3.15: Number of L2 cache misses

3.2.3 Memory bandwidth - single-pointer

It is time for the first experiment that describes effect of sharing the memory bus and the memory controller. The experiment uses memspeed 3.6 on two cores and also on one core for results comparison. The location of these two benchmarks is changed in the same way as in Producer - Consumer experiment 3.1.3. That includes cores that share the L2 cache, cores that do not share the L2 cache but are located in the same package and cores that are located in different packages. Access is linear and random and for each of them we measure read and write scenario.

Experiment: Memory bandwidth - single-pointer

Purpose Determine the overhead caused by shared memory bus and memory controller.

Measured Times to access variable memory block using memspeed (code 3.6) running on two cores simultaneously.

Parameters memspeed memory block: 16 KiB - 2 GiB exponential step; memspeed memory access: linear and random - read and write

Expected Results The single-core variant should be the fastest one. It depends

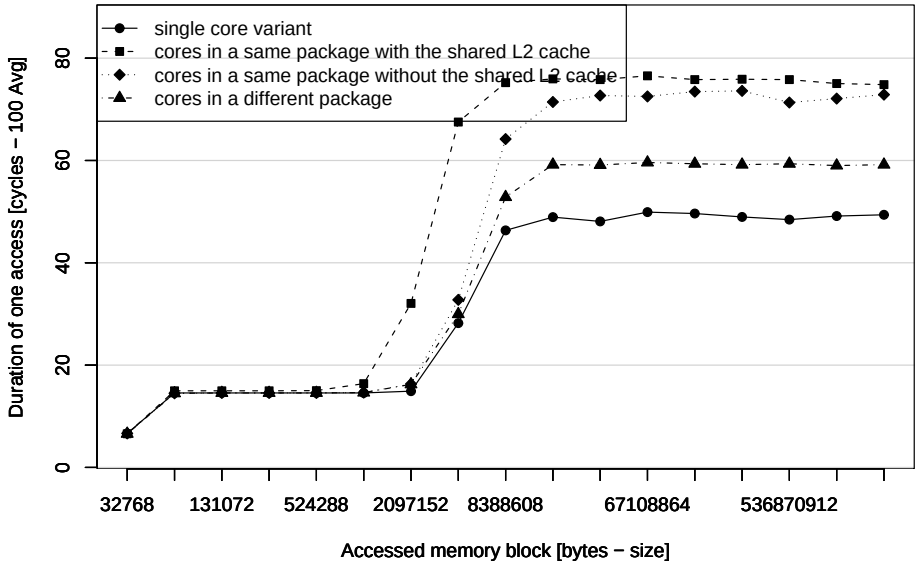


Figure 3.16: Duration of one read access - access pattern is linear

now how the shared component will behave. Let us expect that cores with the shared L2 cache will be the slowest ones. They share the L2 cache and they are on the same memory bus. Next should be cores with the non-shared L2 cache in the same package that shares the memory bus. Cores in different packages share only the memory controller so we expect the lowest times in the scenario with two cores.

The linear access can take advantage of a prefetcher so it should be faster than the random access.

Measured Results The read access with linear pattern is presented on Figure 3.16. The first rise right at the beginning and the second rise in the middle of the picture are the borders between the L1 and L2 cache, and the L2 cache and memory. This is exactly the same situation we described in previous chapter on Figure 3.14. We will see very similar curve on all of the following pictures so unless said otherwise, it is every time this same effect.

As we expected, single-core version is the fastest one with average of 49 clocks. There is also no surprise in other results. A different package scenario is the fastest two core setup and reaches almost 60 clocks. Time for two cores that do not share the L2 cache is 72 clocks. The slowest are cores with the shared L2 cache with 76 clocks.

Graph visible on Figure 3.17 shows a similar scenario as the previous one but the read access was replaced by the write access. The most interesting

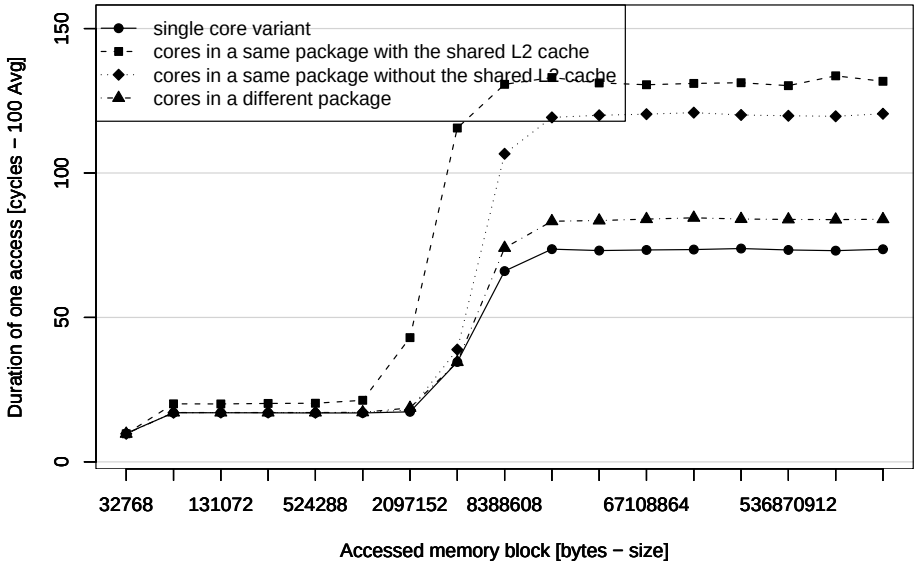


Figure 3.17: Duration of one write access - access pattern is linear

change is that times for some scenarios rise almost twice. The clocks are (from fastest to slowest) 73, 84, 120 and 131. Normally we do not compare the read and write access against each other because of different access algorithm but huge change like this indicates something. The reason for so high times is an eviction of a modified cache line from the L2 cache. All cache lines in the L2 cache are modified and have to be written back to memory and that is the overhead we see here.

On this graph, we can see an effect that we described before. The slowdown caused by a shared L2 cache access is visible between 64 KiB and 1 MiB and it is the same slowdown as described in Shared cache - Bandwidth experiment 3.1.5.

Now we take a look at random access. We can see the read access on Figure 3.18. In comparison with the linear access there is notable difference at the end of curves. As we can see on Figure 3.19, the rise is caused by a virtual to physical address translation. Address translation is normally done by a Translation lookaside buffer (TLB) but the data needed for translation are only cached here and are stored in memory. With greater memory blocks TLB cache cannot hold so many items and data are stored in normal caches too. Somewhere about 32 MiB the cache starts to be too small to hold both program and translation data and translations start to be satisfied from memory which is costly. The curve has the same behavior as the eviction from physically mapped cache 3.15 because the L2 cache handles the translation data in the

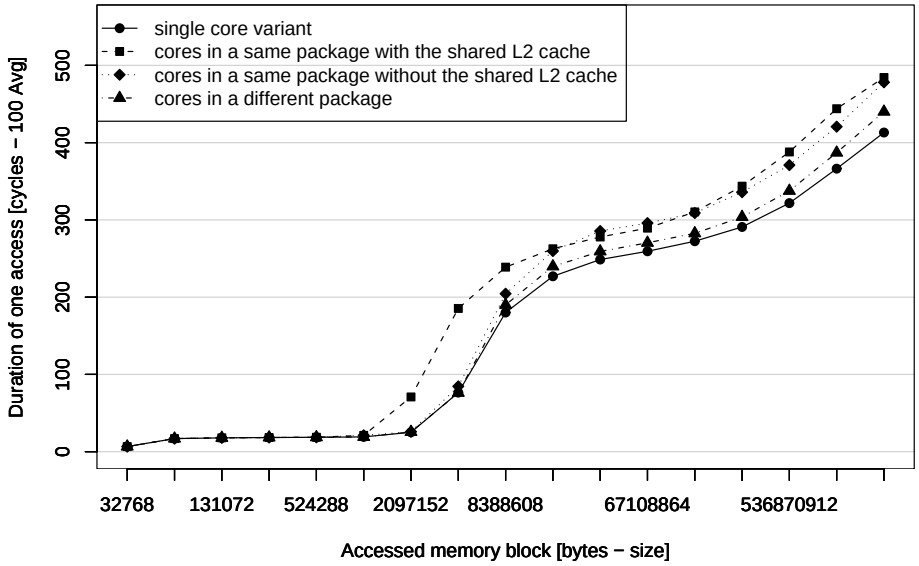


Figure 3.18: Duration of one read access - access pattern is random

same way as any other ordinary data.

Because the prefetcher cannot help with the random access, times are much higher than in the case of the linear access. Times for comparison are taken at 32 MiB where TLB translations do not influence the results that much. For fastest single-core setup is it 249 clocks. A scenario with cores in a different package is little bit slower with 259 clocks. An access for cores in the same package with the non-shared L2 cache costs 286 clocks and with the shared L2 cache costs 277 clocks.

As we can notice, around 32 MiB the shared L2 cache scenario is faster than the case where cores do not share the L2 cache. This effect will also appear in a setup with the write access and will be even more visible in a multi-pointer experiment 3.2.4 where it is explained.

The results for the random pattern with the write access are not included here because they are almost the same as in the case of the read access (Figure 3.18). In the scenario with the linear access, the times were almost doubled but not here. The random access does not saturate the memory bus so much and it has enough capacity to deal with an eviction of modified lines.

Open Issues None

Effect Summary An experiment focus is to determine the overhead of the shared memory bus and the memory controller.

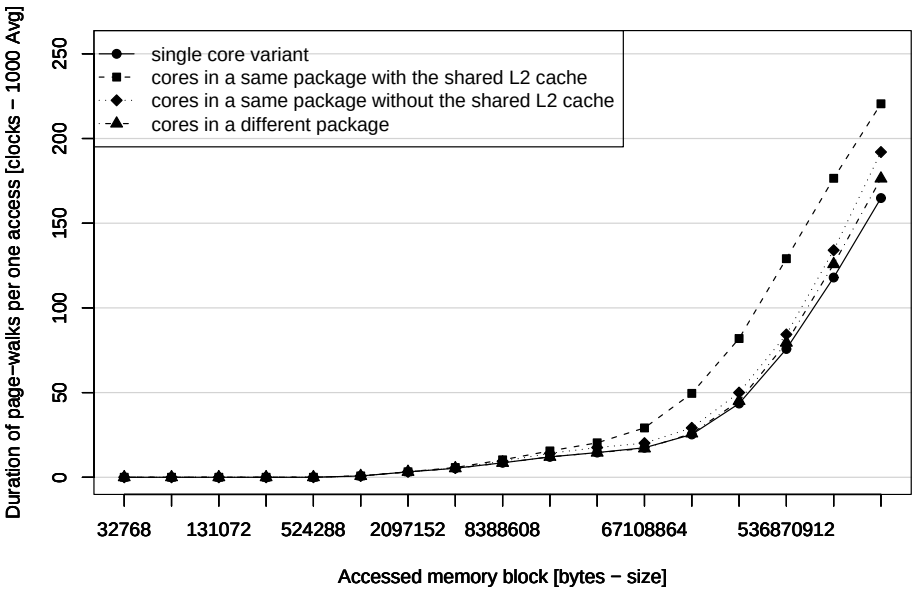


Figure 3.19: Number of cycles that CPU spends on page-walks

The read access with the linear pattern is fastest for cores that share only the memory controller. Overhead compared with the single access is 22%. Overhead for cores that share also the memory bus is now almost 47% and the slowest cores with the shared L2 cache have 55% overhead in comparison with the single-core access.

The write access with linear pattern has the same behavior as the read access but time for the worst scenario is almost twice as high.

The linear access pattern is faster than the random access pattern because of prefetcher. Also this is the reason why the random pattern does not load the shared components so much and the slowdown of the worst case is only 15%.

When two cores access the memory:

- with more shared components (like the memory bus or the shared L2 cache) an access slows down
- when using the linear access, the overhead can be almost 80% in comparison with the single-core access
- the reading linear access is almost two times faster than the writing linear access
- the linear access can be more than two times faster than the random access (compared worst cases on both sides)

- the slowdown of the worst random access is only 15% in comparison with the single-core access
- the reading random access has equivalent speed as the writing random access

3.2.4 Memory bandwidth - multi-pointer

As we already know, modern processor architectures are capable of handling multiple simultaneous requests into memory. That is the difference between this experiment and previous experiment with only one pointer. Multiple pointers generate bigger load and we try to discover how much useful it can be.

The settings are the same as for the previous experiment. The exception is benchmark where mespeedmp 3.4 is used instead of simple memspeed.

Experiment: Memory bandwidth - multi-pointer

Purpose Determine the overhead caused by shared memory bus and memory controller created by workload with multiple pointers.

Measured Times to access variable memory block using memspeedmp (code 3.4) running on two cores simultaneously.

Parameters memspeedmp memory block: 16 KiB - 2 GiB exponential step; memspeedmp memory access: linear and random - read and write; memspeedmp pointers: 1, 2, 4, 8

Expected Results Memory bandwidth - single-pointer 3.2.3 experiment is very similar to this, so expectations are the same.

Cores with more shared resources will be slower. Also linear access will be faster than random access.

Measured Results The resulting curves in graphs are similar to those that we saw in single-pointer experiment 3.2.3. So instead of graphs we will compare the single-pointer and multi-pointer results using two tables.

If we look back at memspeedmp code 3.4, we can see that all pointers are measured together but in the tables is a cost of access for only one pointer. As was said before, processor can handle more than one pending request into memory. Here we measure how long it takes to do all of the fetches from memory and then the average time for one pointer is computed. So what we actually show here is not the access latency but a throughput.

Table legend: L2 - cores with the shared L2 cache, SP - cores in a same package with the non-shared L2 cache, DP - cores in a different package

Table 3.1: Comparison between the single-pointer and multi-pointer linear access

	single-pointer (clocks)	multi-pointer (clocks)	speedup
L2 read	76	72	6%
L2 write	131	134	-2%
SP read	72	69	4%
SP write	120	122	-2%
DP read	60	53	13%
DP write	84	84	0%

Table 3.2: Comparison between the single-pointer and multi-pointer random access

	single-pointer (clocks)	multi-pointer (clocks)	speedup
L2 read	277	71	290%
L2 write	298	175	70%
SP read	286	85	236%
SP write	304	181	68%
DP read	259	64	304%
DP write	278	112	148%

Table notes:

- results for single-pointer and multi-pointer are both with two cores
- speedup represents how much multi-pointer is faster than single-pointer
- multi-pointer with linear access pattern uses four pointers and multi-pointer with random access pattern uses eight pointers, this will be explained later
- times for comparison are taken at 32 MiB where results are not affected by virtual to physical address translation
- multi-pointer times are averages over all pointers

As we can see from Table 3.1 and Table 3.2, multiple pointer throughput is bigger only in the random access scenario. Also the numbers indicate that the linear access can take an advantage of all available capacity. In multi-pointer scenario, the throughput for the write access is almost twice as high as for the read access and this applies to both random and linear access. The reason is that multi-pointer access saturates the memory bus and a dirty cache line eviction creates another load.

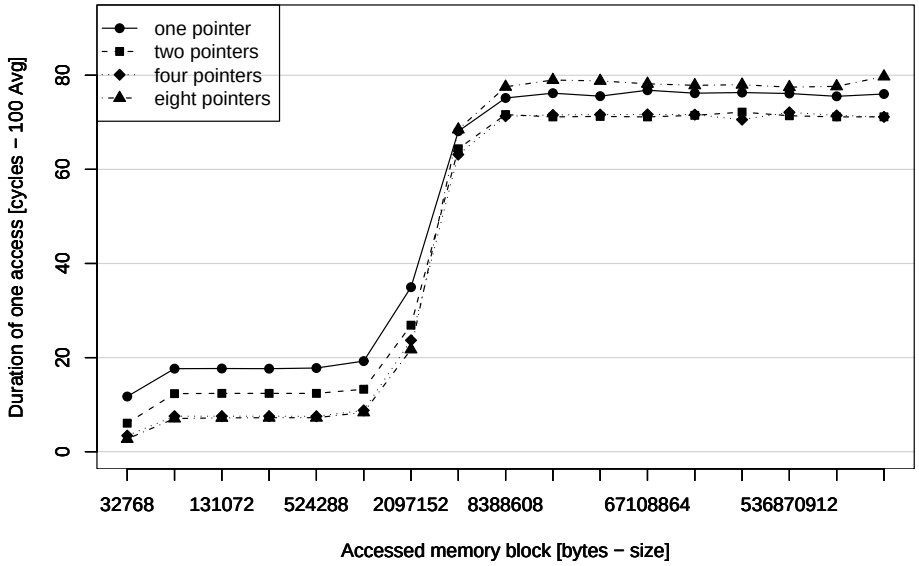


Figure 3.20: Multi-pointer access comparison - access pattern is linear

There are some limitations on how many pointers we can use. CPU cannot process unlimited number of requests into memory and also memory bus and memory controller have their limits. In this experiment, we get the highest throughput with four pointers in linear access and with eight pointers in random access.

On Figure 3.20, we can see comparison of different number of pointers using the linear access. The number of pointers with highest throughput is four or eight but only for caches. When we look how it performs with memory access, the situation is different. The access that uses eight pointers is equivalent to the access that uses only one pointer and accesses with two and four pointers are little bit faster here. So the winner in both scenarios is the access with four pointers that gives the maximum throughput.

Figure 3.21 shows the pointers comparison for the random access. The scenario with eight pointers generates the highest throughput.

Little disadvantage of this experiment is that we did the testing only for powers of two and it is possible that different number of pointers not tested here gives even better results.

There is still one more interesting result I want to explain here. As we can see on Figure 3.22 the scenario with cores that shares the L2 cache is faster than cores without the shared L2 cache. If you can remember, random access in a previous experiment has the same behavior. The effect is surprisingly caused

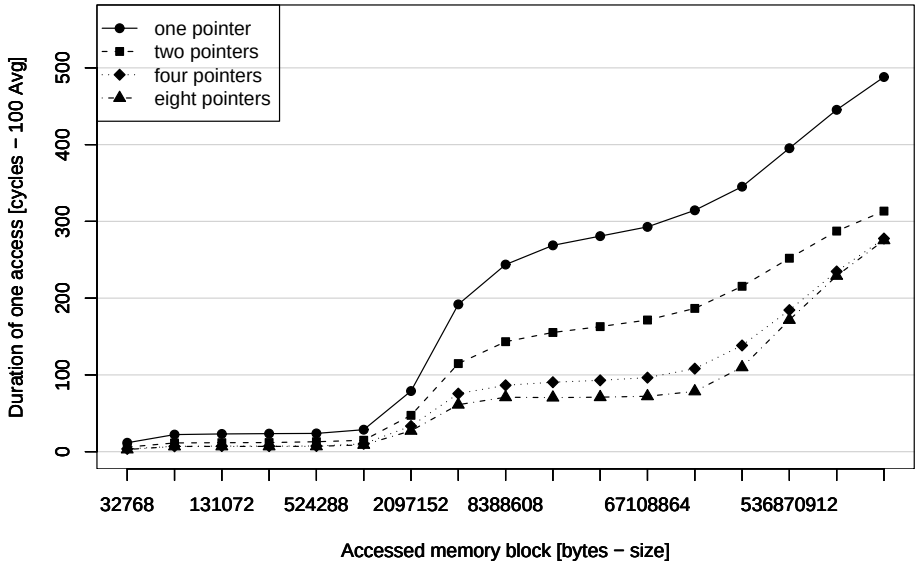


Figure 3.21: Multi-pointer access comparison - access pattern is random

by the shared L2 cache. On tested Intel Server, there is a mechanism called Adjacent Cache Line Prefetch. When the processor reads the cache line, it also fetches the adjacent cache line. But when the shared L2 cache is too busy, it turns off this optimization. Now the memory bus is not so overloaded and can process requests faster. The turn-off is visible on figure 3.23 that shows the number of prefetched cache lines.

Open Issues In this experiment we have tested scenarios with 1, 2, 4 and 8 pointers only. But different number pointers maybe give even better results.

Effect Summary This experiment uses multiple pointers to stress the memory bus and the memory controller.

The usage of multiple pointers makes sense only with random access pattern. The multi-pointer support does not have almost any advantage with linear access pattern.

The number of simultaneously accessing pointers with the highest throughput is eight for random access and four for linear access.

An intensive access into the shared L2 cache can disable prefetching and improve the throughput.

When two cores access the memory with high intensity using multiple pointers:

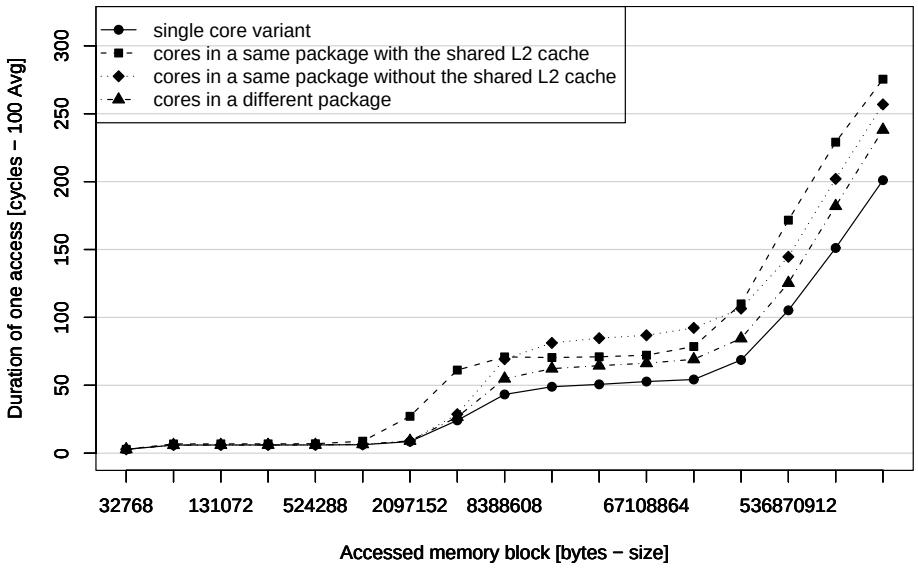


Figure 3.22: The average reading access time computed from an eight pointer access - access pattern is random

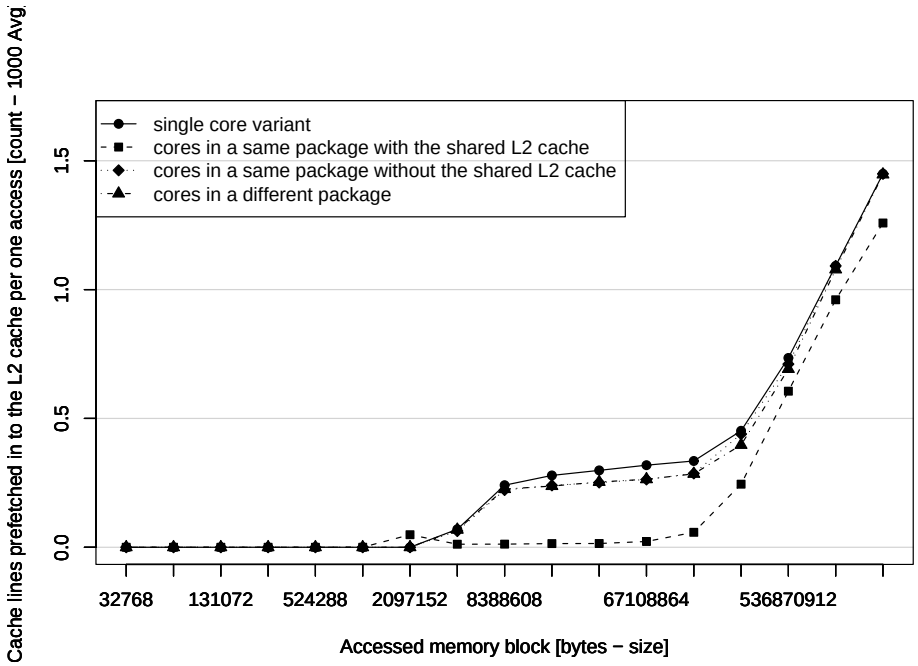


Figure 3.23: Number of cache lines prefetched to the L2 cache

- multiple pointers in random access can improve the throughput more than 300%
- an intensive access into the shared L2 cache can disable prefetching and improve the throughput

3.2.5 Memory Saturation

In previous two experiments, we have determined the overhead of two running cores sharing one memory bus or one memory controller. We know now how big delay we should expect if we run the setup with two cores. But is the memory bus and the memory controller stressed at maximum or can we expect that three or four cores can process more memory at given time?

To answer these questions, we need to do series of tests with different sets of cores. For two cores we had three different scenarios. With more than two cores, the number of possible settings grows rapidly. And it is not only about running these tests but also tuning them and finally explaining (or try to explain) all effects that we encounter.

We manage to run two tests with two and four cores in different packages (none of them shares the L2 cache). Four cores in this setup process at given time more memory than two cores only but we computed that two cores are able to stress the memory at 90% as compared to four cores workload. This means that only two cores can saturate the memory pretty well.

But these are only the preliminary results and a lot of work needs to be done to get some sensible numbers.

3.3 Hard disk drive

Benchmarking hard disk drive (HDD) is in many ways different than what we saw so far. When we tested CPU or memory we had "direct" access to tested hardware but this is not true with HDD. Operating system wraps the HDD with many other layers such as I/O schedulers, file systems, user space libraries and others. Some of these layers are easier to bypass, some of them are not.

But there is another level of complexity. Server based SCSI HDD with 15K rotation speed behaves differently than 7.5K SATA desktop HDD. Also, modern HDDs called Solid State Disks (SSD) are based on totally different technology with different characteristics. HDDs can be assembled into Redundant Array of Inexpensive Disks

(RAID), which merges more HDDs in one virtual disk with some improved attributes like throughput or redundancy.

Our idea was to measure average times to read or write a predefined block of data from multiple sources. We have done multiple measurements with two or more cores and also tried combinations of reading and writing workloads. In almost every case, we were not able to explain the results that we get. Each of the named level adds some sort of complexity and nondeterminism and all packed together creates effects that are very hard to explain.

Now we believe that measuring times is a wrong way and better results can be achieved with measuring number of I/O operations per unit of time. This is the recommended way for future development.

Chapter 4

Conclusion

On the implementation side, the thesis has developed an experiment framework called RIP, which facilitates collecting precise CPU clock and hardware counter values for a variety of workloads. The framework is able to automatically vary the experiment parameters, easily combine a variety of workloads running on different cores, all done declaratively through a single configuration file. The framework also provides the necessary synchronization between the measured workloads and automatically collects and stores the results after the measurement is done. The collected results are plotted and basic statistics are calculated automatically.

The framework is easily extensible and relatively well tested. It was successfully used in the Q-ImPrESS Project Deliverable D3.3 [8, page 51] and also in the experiments with the Finis Terrae computing cluster run by the Centro de Supercomputing de Galicia in Spain.

Even though the framework itself is relatively large, an even more important result of the thesis is the design and implementation (and sometimes redesign and reimplementation), fine tuning and explaining of the experiments that identify and quantify the influence of resource sharing on software performance. We have design the experiments that cover processor caches, memory and hard disk drive sharing scenarios.

Among the more surprising results, the ping-pong experiment has revealed that sharing a frequently accessed variable between multiple cores can be really expensive. The producer-consumer experiment has underscored the importance of properly choosing the cores used to run processes or threads that share data. While it is expected that a memory bus has a bandwidth limit, we have also demonstrated that a similar limit exists for the shared caches. When two processes share the L2 cache, the performance penalty does not depend only on their space requirements,

but also on the overall data access frequency. Finally, we show how the fact that the processors can handle multiple pending requests to memory improves throughput, especially in situations where prefetching does not help because of the random nature of the access pattern.

Although these results are not directly useful to a programmer, they can provide very important information for operating system or compiler designers, as well as in any other low level implementation activities. The thesis results can also help improve the current cache models.

Finally, we should point out that we provide not only the full analysis of the experiment results, but also everything necessary to reproduce the results, or obtain similar results on different computing platforms.

4.1 Future Work

The thesis has demonstrated a strong potential for pointing out interesting future work directions. On the more specific topics related to the particular experiments, the future work directions include:

- The memory saturation issues can be investigated in still more depth. We have carried out all the experimental work for two core experiments, but work on workloads with more than two cores is still at its beginning.
- The hard drive sharing experiments need a different approach than the one adopted in the thesis. It appears that measuring time with hard drive operations does not provide results that would be clear enough for analysis. The experiments should be redesigned from scratch, since measuring the number of I/O operations seems to be a better way how to test HDD sharing effects.
- The framework and most of the tests are ready to be run on a different platform, however, modern platforms are increasingly using a NUMA memory design and our framework does not yet handle sensible memory allocation in such an environment.
- Another technology worth testing is hyper-threading, which duplicates only some parts of the processor package or processor core, but shares most execution units. We expect that impact on performance because of this sharing will be high.

Another future work direction rests with the fact that all the tests presented in the thesis used artificial workloads. This is good when we want to isolate or emphasize

a particular effect for more clear explanation, it but it does not tell us much about performance in real applications. Resource sharing in real applications is therefore also a topic for future work.

Bibliography

[1] Intel® 64 and ia-32 architectures, optimization reference manual http://www.intel.com/design/processor/manuals/248966.pdf.

[2] Intel® 64 and ia-32 architectures, software developer's manual, volume 2b: Instruction set reference, n-z http://download.intel.com/design/processor/manuals/253667.pdf.

[3] Papi - homepage, http://icl.cs.utk.edu/papi/.

[4] perfctr - homepage, http://user.it.uu.se/~mikpe/linux/perfctr/.

[5] R - homepage, http://www.r-project.org/.

[6] Rib framework - svn, https://aiya.ms.mff.cuni.cz/svn/rib/trunk/.

[7] Software techniques for shared-cache multi-core systems, http://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems.

[8] Vlastimil Babka, Lubomír Bulej, Martin Děcký, Johan Kraft, Peter Libič, Lukáš Marek, Cristina Seceleanu, and Petr Tůma. Project deliverable d3.3 resource usage modeling (version 641), 2008.

[9] Vlatimil Babka. Influence of resource sharing on performance. Master's thesis, Faculty of Mathematics and Physics, Charles University in Prague, 2007.

[10] Ulrich Drepper. What every programmer should know about memory (version 1.0), http://people.redhat.com/drepper/cpumemory.pdf, 2007.

Appendix A

Memory content cache

With the authors permission this whole chapter is taken from Project Deliverable D3.3 [8, page 54-56].

For better cache understending is also recommended to read What Every Programmer Should Know About Memory [10] written by Ulrich Drepper.

Memory content cache is a shared resource that provides fast access to a subset of data or instructions in the main memory for all components running on the same processor core or the same processor.

Whenever a component reads data from memory, the processor searches the memory caches for the data. If the data is present, it is fetched from the cache rather than directly from memory, speeding up the read. The cache is said to hit. If the data is not present, it is fetched directly from memory and copied into the cache. The cache is said to miss.

What happens when a component writes data to memory depends on whether the cache is a write-back cache or a write-through cache. With a write-back cache, the processor stores the data into the cache, speeding up the write. With a write-through cache, the processor stores the data directly in memory.

Since instructions and data tend to occupy unrelated addresses, separate caches can be used for instructions and data.

An access to a cache is different from an access to memory in the way data is looked up. A memory entry holds only data – there are as many memory entries as there are different addresses and given an address, the entry holding the data at that address is selected directly. A cache entry holds both data and address – there are fewer cache entries than there are different addresses and given an address, the entry holding the data at that address is found by searching multiple entries.

Since circuits that search for an address are more complex and therefore slower than circuits that select based on an address, a compromise is between the two is often im-

plemented in the form of a limited associativity cache. A limited associativity cache is organized in sets of entries. Given an address, the limited associativity cache first selects a single set that can hold the data at that address, and then searches only this set for the data at that address. The cache is said to have as many ways as there are entries in a set, a fully associative cache can be viewed as having an equal number of sets and entries.

Both virtual and physical addresses can be used when searching for data. Since the virtual address is available sooner than the physical address, faster caches are more likely to be indexed using virtual addresses, while slower caches are more likely to be indexed using physical addresses.

As a general rule, larger caches tend to be slower and smaller caches tend to be faster. The conflict between size and speed leads to the construction of memory architectures with multiple levels of caches. The cache levels are searched from the smallest and fastest to the largest and slowest, the higher level cache only accessed when the lower level cache misses, and the memory only accessed when the last level cache misses. The cache levels are numbered in the search order.

The separation into caches for instructions and data, as well as cache sharing, may vary with the cache level. Typically, there is a separate L1 instruction cache and L1 data cache for each processor core. In contrast, the L2 cache tends to be unified, storing both instructions and data, and shared by multiple processor cores. The same goes for the L3 cache, if present.

With multiple levels of caches, an inclusion policy defines how multiple levels share data. In strictly inclusive caches, the data present in a lower level must always be present in the higher level. In strictly exclusive caches, the data present in one level must never be present in another level. A mostly inclusive cache stores the data in all levels that have missed when handling a miss, but the data can later be evicted from higher levels while staying in lower levels.

Rather than operating with individual bytes, caches handle data in fixed size blocks, called cache lines – a typical cache line is 64 bytes long and aligned at 64 bytes boundary. Since a cache line cannot be filled partially, a write of less than an entire cache line requires a read of the cache line. Only entire cache lines are transferred between cache levels and between cache and memory.

The bus transaction that transfers a cache line from memory takes several bus cycles, each cycle transferring part of the cache line. Rather than waiting for the entire cache line to be transferred, the accessed data is delivered as soon as it arrives, with the rest of the cache line filled afterwards. With a memory subsystem implementation that transfers the cache line linearly, this would mean that the access latency depends on the position of the accessed data within the cache line. To avoid the dependency, memory subsystem implementations can transfer the cache line starting with the accessed data, employing what is called a critical word first policy.

Coherency between multiple caches is enforced using various coherency protocols. A com-

mon choice is the MESI protocol, in which each cache tracks the state of each cache line by snooping the activity of the other caches. A cache that reads a line is notified by the other caches whether it is the only cache holding the particular line. A cache that writes a line notifies the other caches that it must be the only cache holding the particular line. A cache that holds a modified line flushes the line on access from other caches.

Appendix B

Basic code constructs

Here are the Listings with basic constructs used in benchmarks. Note that macros are listed without "\" at the end of each line.

NEXT_ADDR macro is presented on Listing B.1. It is only simple memory advancement.

Listing B.1: NEXT_ADDR macro

```
#define NEXT_ADDR(p)
(p) = (void * *)(*(p))
```

On Listing B.2 is shown similar macro as the previous one but before the pointer walk, the current value of the pointer is modified. The next to the last line writes into the pointer same number but with negated last bit. The last line advances the pointer and the AND guarantees that zero is in the lowest bit.

Listing B.2: NEXT_ADDR_WITH_WRITE macro

```
#define NEXT_ADDR_WITH_WRITE(p)
/* 0 in lowest bit and 1 in every other place */
const uintptr_t PTR_ALIGN_MASK = -2;
const uintptr_t PTR_WRITE_MASK = 1;

uintptr_t readp = (uintptr_t)(*(p));
*(p) = (void *)(readp ^ PTR_WRITE_MASK);
(p) = (void * *)(readp & PTR_ALIGN_MASK)
```

The random trail allocator gets all possible pointers and shuffles them. Then it gets one pointer after another and writes them at the current location. Then use the newly written address for jump and then it gets another pointer. The last pointer finishes the circle.

Listing B.3: MemInit::random memory allocator

```
char * memArea = (char *) mmapAlloc(memSize);
```

```
// number of pointers in memSize
size_t pointersCount = memSize / stepSize;

// field of pointers
void ** pointers = new void *[pointersCount];

// get list of all available mem places
for (size_t i = 0; i < pointersCount; ++i) {
     pointers[i] = memArea + i * stepSize;
}

// this shuffles data with random access iterator - arrays too
std::random_shuffle(pointers, pointers + pointersCount);

// get first from list...
void ** p = (void **)pointers[0];

// int i = 1 ...start from the second pointer (first is used)
for (size_t i = 1; i < pointersCount; ++i) {
     *p = pointers[i];
     p = (void **)(*p);
}

// set last pointer to point to the beginning
*p = pointers[0];
```

The linear trail allocator is very similar to the random trail allocator (Listing B.3) but the trail is not shuffled.

The NOP function is a little bit tricky. We cannot have NOP function with unlimited size because of the limited size of the L1 instruction cache. So we have two NOP functions. The first function has the maximum size that is allowed in a configuration file and is called fullNopF. The second function has a variable size and is called endNopF.

Example: We need 74 NOPs and in a configuration file the fullNopF function has the value 16. We create our endNopF like this: endNopF = neededNOPs modulo fullNopFLength. So endNopF is in this case 10. Now we call (as on Listing B.4) 4 times fullNopF and one time endNopF and we have exactly 74 NOPs.

Listing B.4: NopFunc call

```
for(size_t i = 0; i < nf.fnfLoops; ++i) {
     nf.fullNopF();
}
nf.endNopF();
```