Towards Modeling and Simulation of Exascale Computing Platforms

Luka Stanisic

Master of Science in Informatics at Grenoble
Master Mathématiques Informatique - spécialité Informatique
option<Parallel, Distributed and Embedded Systems>

Towards Modeling and Simulation of Exascale Computing Platforms

Luka Stanisic

<22.06.2012>

Research project performed at <LIG laboratory>

Under the supervision of:
<Arnaud Legrand, CNRS researcher>
<Jean-François Méhaut, Professor Polytech'Grenoble>

Defended before a jury composed of:
[Prof] <Martin Heusse>
[Prof] <Florence Maraninchi>
[Prof] <Nadia Brauner>
[Prof] <Vincent Danjean>

Abstract

Future super-computer platforms will be facing big challenges due to the enormous power consumption. One possible solution to this problem would be to develop HPC systems from today's energy-efficient hardware solutions used in embedded and mobile devices like ARM. However, ARM chips have never been used in HPC programing before, leading to a number of significant challenges. Therefore, we experimented with ARM processors and compared their performance with the architectures that are known better, in this case last generations of Intel processors. Due to the memory bottleneck of most scientific applications, understanding the performance of CPU caches in this context is crucial, thus this research was investigating the processor performance depending on memory hierarchy. We present not only differences and complexity of these two architectures, but also how changing seemingly innocuous aspects of an experimental setup can cause completely distinctive behavior. Additionally, we demonstrate very clean and systematic methodology, which aid us in achieving good performance estimations.

Résumé

L'économie d'énergie est en passe de devenir un des principaux défits pour le renouvellement des super calculateurs. Les futurs système HPC pourrait trouver solution en se développant à partir de composants efficacent et basse consommation ARM aujourd'hui présent dans les systèmes embarqués ou mobile. Cependant, ces composants n'ont encore jamais été utilisé en programation HPC. C'est pourquoi, on se proposent ici d'évaluer expérimentalement des processeurs ARM et de comparer leur performance avec des architectures largement déployées telles que les dernière génération de processeur Intel. En raison de l'importance des performances mémoire pour les applications scientifiques, comprendre la performance des mémoires cache au sein des CPUs est crucial. Notre étude portera donc sur la performance processeur par rapport à la hierarchie mémoire. Nous présentons non seulement les différences et la complexité de ces architectures, mais aussi comment le changement d'un aspect innofencif en apparence peut causer des comportement totalement distincts. En outre, nous montrons une méthodologie simple et systématique permettant de réaliser de bonnes estimations de performances.

Acknowledgements

First and foremost, I want to thank my supervisor Arnaud Legrand for his support, guidance and encouragement. Also express my gratitude to Brice Videau and Jean-François Méhaut for their valuable advises. Last, but certainly not least, I would like to thank all other members of MESCAL, MOAIS and NANOSIM teams for the wonderful atmosphere and pleasant company.

Contents

— 1 —
Introduction

There is an ever-growing need for computing power in scientific applications (weather forecast, molecular modeling, simulation for engineering/finance/biology etc.). One CPU being insufficient for such high requirements, supercomputers were introduced. These machines typically comprise thousands of CPUs interconnected through high-speed networks and equipped with accelerators like GPUs. Today's fastest supercomputer is "K computer" in Japan with 88,128 2.0 GHz 8-core processors and it is reaching peak performance of around 10 petaflops (10^{16} floating-point operations per second). If the previous trends continue, it is expected that 1 exaflops (10^{18} floating-point operations per second) will be reached in 2020. Nevertheless, with the technology we currently have at our disposal, it will not be an easy task.

In the last decades, computing power of individual CPUs mainly improved thanks to the frequency increase, miniaturization, and hardware optimizations (cache hierarchy and aggressive cache policies, out-of-order execution, branch prediction, speculative execution, etc.). Frequency increase and hardware optimizations are now facing hard limits and incur unacceptable power consumption. It is proved that power consumption grows more than quadratically with the growth of frequency. Additionally, speculative execution performs many useless operations and although they seem free in terms of scheduling on the CPU resources, in terms of power utilization they waste a lot of energy. Supercomputers and data centers typically consume as much electricity as a small city and the price for powering them for a few years is the same as the initial price of the hardware that it is composed of. To achieve exascale, the power efficiency of individual CPUs will have to be reduced by a factor of 30.

Two main approaches are envisioned. The first one attempts to improve the power consumption of current high-performance hardware. The second approach is to build on existing low-power CPUs commonly used in embedded systems and try to improve their performance. My internship was done in the context of the second approach as a part of European Mont-Blanc project [1]. Montblanc project aims in developing scalable and power efficient HPC(High Performance Computing) platform based on low-power

1. Mont-Blanc project: www.montblanc-project.eu

ARM technology. ARM (Advanced RISC Machine and, before that, the Acorn RISC Machine) processors are particularly designed for portable devices as they have very low electric power consumption. Nowadays, these CPUs are running on almost all mobile phones and personal digital assistants.

However, such CPUs have very different characteristics and behavior from the ones typically used in HPC. It is based on a fact that such processors are much simpler than classical Intel CPUs regarding hardware optimizations. Furthermore, they have much smaller CPU caches with very little memory hierarchy. Keeping in mind that memory accesses are the bottleneck of most scientific application, understanding the performance of caches in this context is thus crucial.

The primary goal of this internship was to investigate behavior of the caches on ARM processors and compare it with the architectures that are known better, in this case last generations of Intel processors.

These systems are particularly complex and require systematic analysis. Apart from getting the actual results, we wanted to do the experiments in a clean, coherent, well-organized and reproducible way. Related work in this area is not very well documented. Since the code is not available and many important information about the system configuration are not stated, it is often impossible to reproduce the experiments.

The secondary goal of this internship was thus to use a systematic and sound experimentation methodology that would allow to comprehend and control the impact of all parameters of the system and to reproduce the experiments.

The rest of this document is organized as follows: Chapter 2 reviews related work. Chapter 3 explains our experimental methodology. Chapter 4 shows and discusses experimental results. Finally, Chapter 5 concludes with a summary of results and future research directions.

— 2 —
Related Work

In this chapter we will first present differences between ARM and Intel microarchitectures, with the emphasis on CPU caches. Then, possible performance prediction methods will be introduced, along with their advantages and drawback. Finally, we will review the state of the art regarding methodology, which served us as a reference towards developing our own workflow that will be discussed later in the Chapter 3.

2.1 CPU Caches on ARM and Intel Architectures

Even though they had the same starting point, the evolution of CPU speed and memory speed took a different pace. Today, memory is in order of magnitude (sometimes even worse) slower, thus it is usually the bottleneck of the most applications. Computer manufacturers tried to deal with this problem by developing memory hierarchy, more precisely CPU caches. Multiple level of CPU caches (most commonly referred as L1, L2, L3 caches) are basically small, faster memories that are closer to CPU. L1, the smallest but fastest cache, usually belongs to a single processor core (it is private). L2, which is further from CPU thus bigger and slower, can be either private or shared between cores depending on the specifics of the architecture. L3, present only in latest processors, is respectively even bigger and slower than its predecessors and most commonly it is shared. All these levels of caches have in common that they try to exploit the temporal and spatial locality of the data. Temporal locality is the characteristics that data, that is accessed now, is very likely to be accessed again in near future. Spatial locality is the characteristics that data, that is adjacent in memory to the data that is accessed now, is very likely to be accessed in near future. By storing the most frequently accessed data, caches decrease the average memory access time of the program. Caches on two processors microarchitectures differ from each other mainly by the number of cache levels and their size, but also by the internal organization (set-associativity, cache line size, etc.).

Since they were designed for distinct purposes, there are numerous differences between ARM and Intel architectures. For instance, ARM Cortex A9, used in this intern-

ship, have only dual-core 1GHz CPUs comparing to the 4 cores (8 with hyperthreading) Intel i7 Sandy Bridge CPUs running on frequencies of 3.4 GHz.

Another rather obvious distinction is the memory hierarchy which can be clearly seen in Figure 2.1 ARM has two levels of caches where L1 cache is a 32KB Instruction and 32KB Data 4-way associative cache with 64 byte line size and shared L2 cache is a 512KB 4-way associative cache with 64 byte line size. Intel has three levels of caches where L1 cache is a 32KB Instruction and 32KB Data 8-way associative cache with 64 byte line size, private L2 cache is a 256KB 8-way associative cache with 64 byte line size and shared L3 cache is 8MB 16-way associative. Higher processor speed and bigger cache size provide much better performance to the Intel CPUs, but for the significant price in terms of energy.

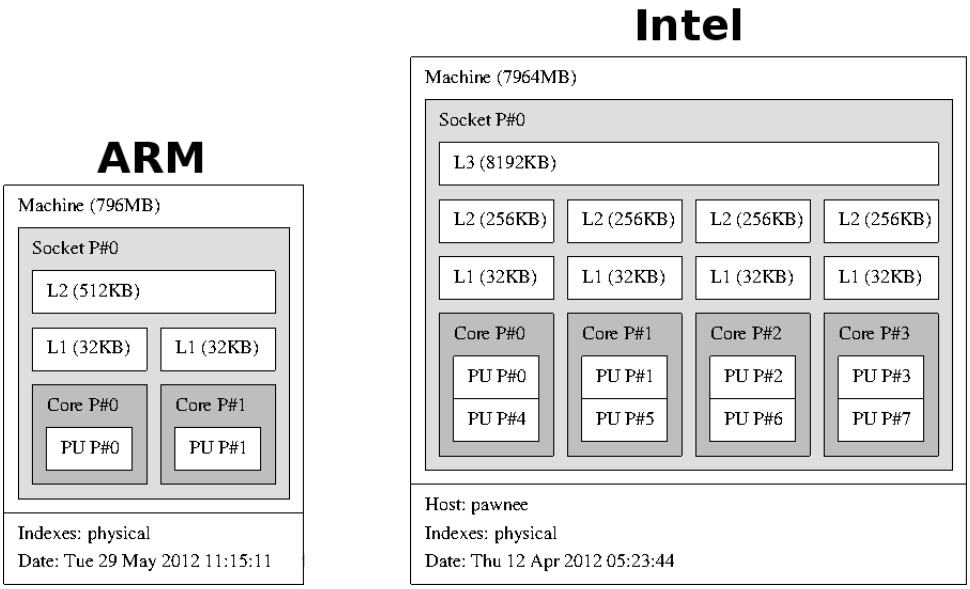

Figure 2.1: ARM and Intel microarchitectures.

Additionally, there are other differences that cannot be seen from only looking into the processors specifications. ARM belongs to RISC (Reduced Instruction Set Computing) families of processors, thus it has much simpler instruction set. In contrast, Intel's CISC (Complex Instruction Set Computing) x86 supports extremely large and complicated instruction set (for reverse-compatibility reason). Significant die space (and energy) of x86 is spent on translation system, whose role is to transfer all these instructions into elementary microcode. Furthermore, ARM CPUs have much shorter pipeline, which means that in cases of branch misprediction, they will have to flush less. Hence, the overall power consumption per instruction is much lower. Also, Intel processors have much more complicated and aggressive optimization techniques (speculative execution, branch prediction, simultaneous multithreading etc.) than ARM, although these difference are beginning to fade.

Despite the fact that they had opposite starting points, ARM insisting on power efficiency and Intel on performance, these two architectures are evolving towards each other. HPC is just one of the examples how they are crossing into each others territory. Recent ARM processors started introducing advanced hardware optimizations, while new generations of Intel processors tend to lower the number of pipeline stages and try to lower the power consumption. Nevertheless, as shown before, there are still severe distinctions between these two architectures and accordingly their performances differ.

2.2 Performance Prediction on HPC

This work in general is a step towards predicting performance on HPC platforms, evaluating the time an HPC code would take to execute on hypothetical machines (clusters). There are several known methods to performs this and we will briefly review their advantages and drawbacks:

2.2.1 Instruction Level Approaches

Simulation

Simulations, especially cycle-accurate simulations, have been heavily used in past. They are relatively easy to build and allegedly they provide very precise results. Nevertheless, their problem is that they take up to 1 million times[4] more than the original runtime of the application. For that reason, cycle-accurate simulators are typically used only to measure few seconds of the program execution. This arises the accuracy question of such predictions, since they are based on only isolated part of the whole application[10]. Additionally, these simulators are often proprietary and uniquely designed for particular architectures which limits their usage even more.

Emulation

Emulation is another approach, very similar to simulation. In this case, there is a program that creates an extra layer between an existing computer platform (host platform) and the platform to be reproduced (target platform). Then, the host machine is running through this layer the desired code, planned to be executed on the target platform. The measured performance of host machine represent a good estimation of the possible target behavior. Although emulation is often faster and scales better, it still holds the same limitations and weaknesses as the simulation.

2.2.2 Macroscopic Approach

There is alternative technique, that was the starting point of this research. In this "macroscopic" approach, one is trying to characterize the code as a whole with numbers that can later be related to platform characteristics to evaluate performances. This idea was for example used in the PMAC framework[8] depicted in Figure 2.2. Framework for performance modeling and prediction is faster than cycle-accurate simulation and more informative than simple benchmarking of the application. The idea behind it is very intuitive and well structured.

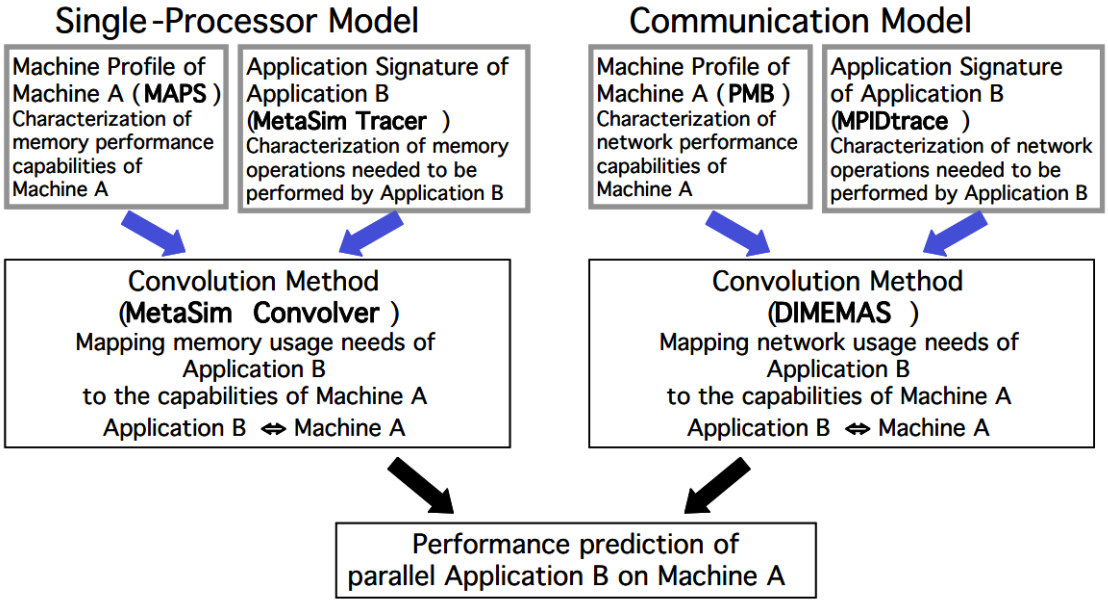

Figure 2.2: Components of PMAC Framework.

First a model of a single processor is built by relating application signature to the machine capacities. Having in mind that memory is the bottleneck of most applications (as explained in the previous chapter), this model is mainly based on the memory hierarchy of the measured machine.

Essentially, the application code is divided into blocks, which represent the repetitive parts of the program. One can calculate overall run-time of the application by multiplying the execution time of each block with the number of times it is executed and finally doing summation of all these multiples. The execution time of each block is estimated based on the characteristics of the whole block (number of memory accesses, floating point instructions, cache misses etc.). The example of block characterization[8] is displayed on Figure 2.3.

After that, these block measurements are combined with the machine characteristics to estimate behavior of the single processor. One of the obvious weaknesses of the

Attribute	Value
Basic block number	1021
Function name	Calc
Source line number	112
Number of memory references	1.E10
Number of floating-point operations	1.E8
Spatial locality score	0.95
Temporal locality score	0.85
L1 cache hit rate for target system	98.5%
L2 cache hit rate for target system	99.7%

Figure 2.3: Sample of block signature.

authors work is that although they plead for this alternative approach, the code characterization is done using cache simulations. These simulations have the same shortcomings as any other simulations.

Second, the same principle is used for modeling communication: Performance capabilities of the network cards and network in overall, are combined with the number, order and types of network operations of the code.

At the end these two models are merged into the final performance prediction of the application on the specified machines.

In this internship, we were mostly interested in single core performance depending on memory hierarchy. In terms of related work, it would be machine profiling of the single processor (the top-left box of the Figure). To characterize the processors, authors propose to use simple "for loop" kernel and measure the overall time it takes to execute it. This simple kernel is the root of the memory benchmark MAPS (and the later, upgraded version MultiMAPS) that were derived from STREAM[5] benchmark.

Pseudocode of the kernel look as following:

```
MultiMAPS( size, stride, nloops)
   allocate buffer size;
   time1;
   for(i=1: nloops)
      access elements in buffer by stride;
   time2;
   bandwidth=accesses/time;
   deallocate buffer;
```

Essentially, this algorithm measures the time to execute nested for loops in which it makes memory accesses to an array. Finally, it computes the memory bandwidth as the total number of accesses divided by the time it took to execute all of them. There are two key dimensions from which access pattern depends:

1) Memory size of the array.
2) Stride of access (by which step the array will be traversed).

In fact, these two parameters should represent the temporal and spatial locality characteristics of the memory.

One of the typical results provided by MultiMAPS can be seen in Figure 2.4.

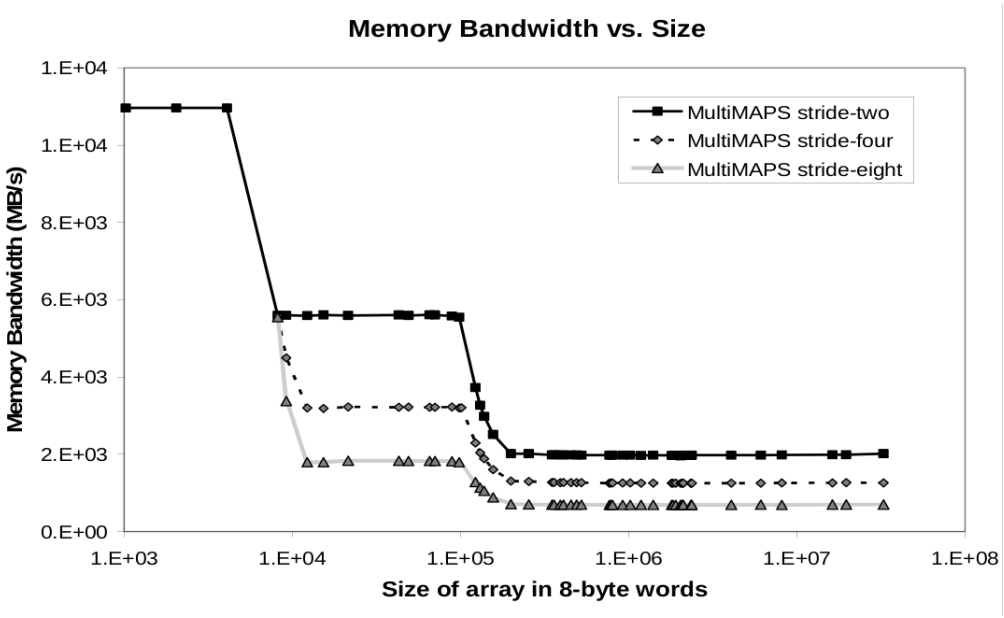

Figure 2.4: Sample MultiMAPS output for Opteron.

This figure shows the results only for strides 2, 4 and 8 on Opteron machine (2.8GHz with 2 level of caches where L1 cache is a 64KB 2-way associative cache and L2 cache is a 1MB 16-way associative cache). Plateaus in Figure 2.4 are directly correspond to the size of L1 cache, L2 cache and main memory, respectively. One can also notice that the strides have no impact while all the accesses are done inside L1 cache, but that they play important role when the array size increases, lowering the bandwidth almost by the factor of 2.

The results of this work seemed very comprehensive and useful. Nevertheless, when we tried to reproduce them, we encountered numerous difficulties.

The plot on Figure 2.4 presents only the average values of several runs. Data aggregation and statistics are done on the fly for practical and intrusiveness reasons. As we will show later in our experiments, it's dangerous to reason solely on aggregated data. Furthermore, although the previous example looks very simple, these are complex systems with potentially a lot of parameters that may influence the final measurement. By looking at their code, one could not skip the impression that many configuration parameters are tailored for a specific platform the experiments were run on.

In fact, our first results were more like the one shown in Figure 2.5, with a lot of noise which resembles very little to the one we expected, something similar to Figure 2.4. Although only 3 repetitions have been performed for this experiment, there is already sufficient variability for each of the strides(1,2,4). One of the reasons for this unstable behavior might lie in the fact that the machine the measurements are performed is rather old, but it still does not explain the average value curves for different strides constantly crossing each others paths.

That is why we needed to do measurements in a very clean, coherent and systematic way. For this kind of experiments, with a lot of hidden details and unknown parameters impact, methodology plays a crucial role on the road to understand the final results.

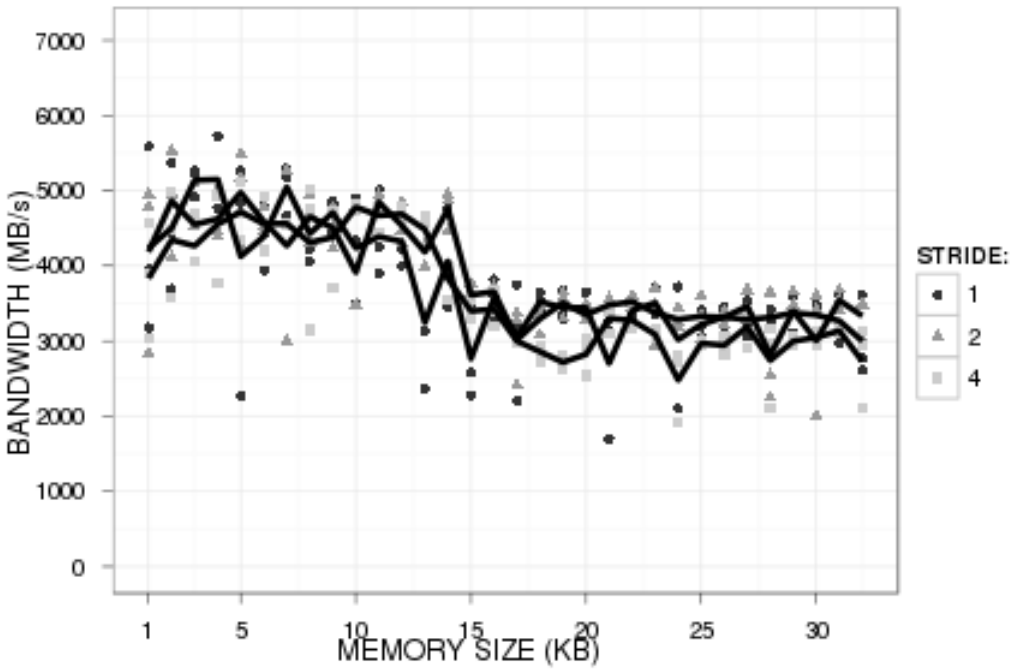

Figure 2.5: Sample of our first experiments on old Intel machine: a lot of noise, with average values of strides crossing each others trajectory; 3 repetitions; buffer size 1KB-32KB; strides 1,2,4.

2.3 Methodology

PMAC MultiMAPS benchmark is extremely complex, with the code that was not very well documented and many system configuration details missing. Additionally, many parameters were obviously tuned for specific architectures and applications used in their experiments, without any explanation. It is impossible to get a good performances out of the box in HPC systems. This context leads to reproducibility issues and control of the experiments.

Another fundamental problem is that this benchmark was built on a set of assumptions, a *model*. Authors make some first order measurements, measuring the elapsed time, in a very specific way and then derive values as a second order measurements, bandwidths. PMaC work is rather old, at least compared to the speed at which computer architectures evolve, and originally not suited for NUMA, multicore architectures. Models and assumptions behind their experiments may not be valid anymore on new machines.

These are the reasons why we wanted to start all over again from scratch. We wanted to be sure that everything is checked and not rely on some code based on tacit assumptions we do not completely understand. We planned to be exhaustive and not to miss any influential parameters. As it can be seen in[7], often if we do not pay attention to all details, measurement bias can lead as to incorrect conclusions.

Phd. thesis from Christine Jacqmot[2] served us as a sample of how the methodology of a complex systems experimentation should be done. Although that thesis focuses on another topic (distributed systems), the fundamental ideas behind its methodology approach is very well structured. We also followed the instructions from recognized books in the area of performance evaluation[6][3] to make sure we have the valid experiment design, execution workflow and finally analysis.

— 3 —

Towards Systematic Performance Evaluation of Cache Hierarchies

We started from the simplest example from[9]: the memory intensive kernel described above. In the beginning we focused solely on direct Kernel parameters CYCLES, STRIDE and SIZE. Even with only these parameters in mind our results were far from what we expected. In order to assure reproducibility and gain better performance we were forced to add other influential characteristics. Nevertheless, all the additional specifications were always fixed during one measurement, i.e. for a single experiment, only the SIZE and STRIDE were changing values, all other criteria were constants. We ended up with 5 additional groups of features, but due to the lack of space, we will present only a short summary for each of these groups:

1) Kernel:

SIZE and STRIDE were the original parameters form PMaC and as already described, they were intended to dictate the behavior of this simple kernel. CYCLES is the number of loops it will be performed. For a small number, the results could be imprecise since syscall that is measuring time on processors, although it takes only around 60 nanoseconds, can still introduce some bias. In contrast, taking very large number for the CYCLES would cause experiments to last very long time, with practically no benefit comparing to the balanced intermediate option. Nevertheless, all the "medium", well suited values (that can be different for different range of SIZE), produce the similar results. From now on, we will assume that the value of CYCLES was always well adjusted to the experiment setup.

2) Memory allocation:

Before executing the kernel, memory chunk (from now referred as buffer) needs to be reserved and filled with random values. Several memory allocation techniques were examined. Also, we tried different types of variables from which the buffer is made of.

These are int (32 bits), long long int (64 bits) along with vectorized instructions m128i (128 bits) and only for Intel Sandy Bridge m256i (256 bits). This is very important because these types occupy different size in memory, thus the final results may severely differ.

3) Compilation:

For the compilation on both ARM and Intel we used gcc compiler. We tried all optimization flags, but mostly focused on -O2 and -O3 as they we generating best and sometimes distinct results. Since our kernel is practically one for loop, we tried to manually unroll it and see if that influences the performance.

4) Operating system:

Experiments on Intel were performed on UNIX (debian distribution) operating systems, while on ARM it was UNIX-like OS called Linaro [1]. It is important to state that our measurement kernel is single-threaded, so the differences between number of cores on different architectures play no role. Nevertheless, we ensured that the whole execution was pinned to one core (although OS would do it himself in most cases). We also tried different scheduling priorities and CPU frequency governments, as we suspected, based on previous experience and literature, that this may have an impact on final results. Finally, we made it certain that we are alone on the machine, in order to be sure that our results are not product of some external process.

5) Architecture:

As described in chapter 2.1, there are severe differences between ARM and Intel processors. Therefore, it is not surprising that the behavior on them even with the same input parameters are very distinctive. Apart from the already mentioned microarchitectures, we have conducted experiments on other machines as well:

For ARM: NVIDIA Tegra 3, quad-core ARM Cortex A9 1.4GHz, 4 cores with two levels of caches from Tibidabo cluster in Barcelona.

For Intel: Intel Core microarchitecture, Xeon Processor LV 5148 CPU 2.33GHz, 2 cores with two levels of caches and Intel, Nehalem microarchitecture, Xeon Processor E5520 CPU 2.26GHz, 4 cores (8 with hyperthreading) with three levels of caches both from GRID5000 [2]

Although these microarchitectures had their own characteristics (especially “Nehalem“), in general the differences between them and their relatives where much smaller than the differences between families of processors. Additionally, we observed some very unique behavior for individual microprocessors, for specific experiment setups. Since internal details of the microprocessors are not publicly available, it is impossible

1. Linaro: www.linaro.org
2. Grid5000: www.grid5000.fr

to fully understand these anomalies and we had to settle with only our observations and assumptions on what were the causes. Again due to the lack of space, in this report we will only demonstrate the results from “Snowball“ and “Sandy Bridge“ as we strongly believe that they are good representatives and sufficient to illustrate the goal of this internship.

6) Design of experiments:

This is completely independent, i.e. outside the main code. It is performed at the beginning and later linked to the data analysis. The important point in the experiment design is sequence order of the chosen SIZE. Sequence order may have an influence (warm up, prefetch) and one can be protected from this with various techniques: reboot all system before measurement, wait a long time between measurements, randomize to get rid of bias etc.. We chose randomization, since it is the fastest and we expected it to be sufficient for our case. We were also checking the influence of this choice carefully during our experimentation period. Furthermore, we made enough repetitions in order to be certain in our results. On contrary, in related work authors mostly did only few repetitions of their experiments, which maybe disabled them from seeing some anomalies. Although these techniques may seem obvious and negligible, we claim that they aided us in finding some intriguing phenomenon during our experimentation period.

One of the key points in our methodology was to ensure reproducibility, to be certain that our results truly demonstrate general behavior of the processor. In order to do so, we were logging the SVN version of source code. Additionally, we were recompiling the code for each single experiment.

Information about all these parameters is captured while traversing throw our workflow. It is recorded at every level and passed to the next level in comments of output file. At the end, our output files along with the measurements results contain a large collection of extra information about the experiment setup. A sample of one of these files can be found in the Appendix.

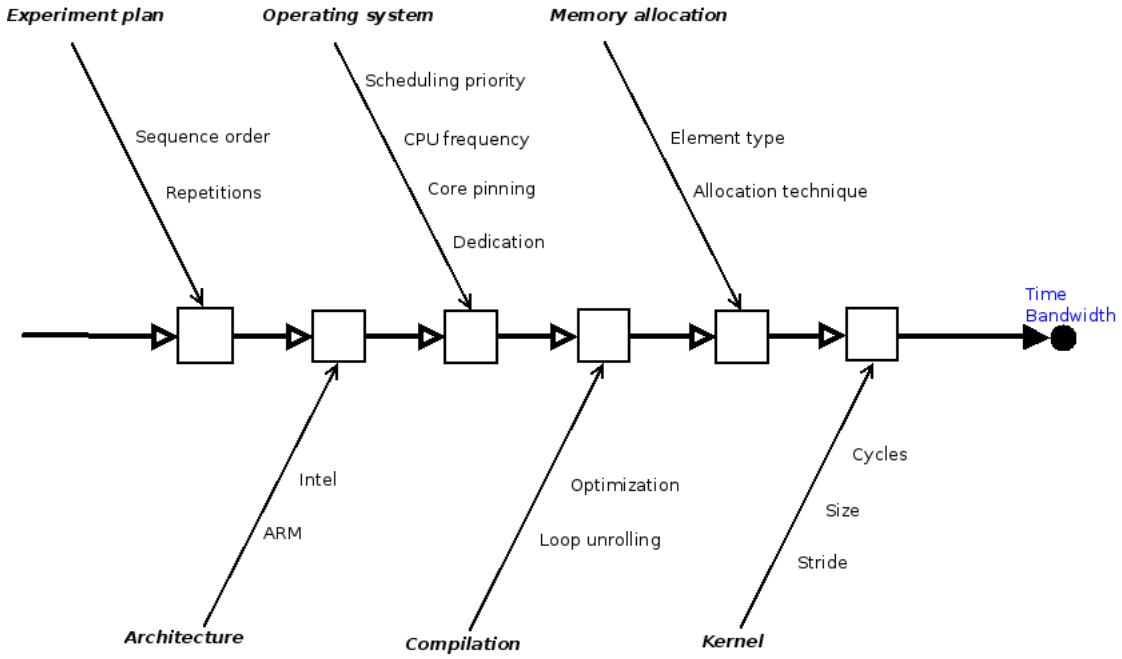

Figure 3.1: Cause-and-effect diagram of our workflow.

Figure 3.1 illustrates our experiment workflow, with all important parameters sorted in described groups. We use modified "cause-and-effect" (fishbone) diagram[6] to display steps in experiment planning. We will demonstrate in chapter 4 how some of this parameters can drasticlly change the final results.

It is important to state that we do not perform any statistical analysis until this point. We do not do any aggregation, we keep all information and delay the analysis to the end in order to spot the outliers and strange behavior, instead of losing them. Lastly, we use R [3] and Sweave to build automatically reports and analysis. R is an open source programming language and software environment for statistical computing and graphics. It supports data manipulation and transformations, as well as sophisticated graphical displays, which we intensely needed for the analysis of our gather data. Sweave is a function that enables integration of R code into LaTeX documents. The purpose is to create dynamic reports, which can be updated automatically if data or analysis change. In other words Sweave allows us to have both data analysis written in R and explanations in classic LaTeX in the same file and at the end build final .pdf report directly.

All our code and data are available to anyone interested in reviewing the results or doing the similar experiments on other platforms. R and Sweave used in the analysis are both freely available online. We strongly believe this, inspired by the open science trend, is the best way to accelerate research and to allow more rigorous peer review.

3. R: www.r-project.org

— 4 —

Experimental Results: Taxonomy of Unexpected Behaviors

During this master thesis project, we studied the influence of various parameters grouped in a previously described way. It will be demonstrated in this chapter how some of these parameters not only affect the absolute values of the final output, but can also cause completely distinctive behavior.

All observations throughout this work are gathered with the help of our well-structured methodology, that enabled us to recognize and understand often unexpected results.

4.1 Influence of Stride Parameter

We started this internship by trying to reproduce the results obtained by MultiMAPS algorithm for small buffer memory size shown on Figure 2.4. One can notice that for small buffer size that fits into L1 cache, all three strides generate the same bandwidths. This is due to the fact that once fetched into the cache, all memory accesses are performed inside L1, regardless of the adopted pattern. Additionally, even when looped back (since inner loop is executed *cycles* time) the whole buffer is still inside the L1, therefore there will be no cache misses.

In case when the buffer size is larger than L1 cache, accessing the element that is outside L1 will result in cache miss and the data will be fetched from L2. Since L2 caches are much slower than L1, it will require longer time to execute the for loop, therefore the final bandwidth value will be smaller. This explains the drop of the curve on L1 cache size value. Regarding the different stride performance, it is the easiest to illustrate it through *stride=1* and *stride=2* comparison. For stride=1 kernel accesses all the elements of buffer one by one, then loops back and repeats this *cycles* times. For stride=2 kernel accesses every second element, as it traverses through buffer by two, then loops back and do it again *cycles**2 times. In general, both strides will have approximately the same number of cache misses inside one inner loop run, but *stride=2* will execute this loop twice as many times. Therefore, due to the spatial locality of the

data and prefetching mechanism, *stride=1* will in overall cause less cache misses, hence the performance will be better.

Additionally, from the Figure 2.4, one can conclude that increasing stride values accordingly decreases the bandwidths in a very regular way. By closely looking into bandwidth values, one can observe that multiplying strides by two lowers the performance by approximately the same factor. This is consistent with the previous explanation about the stride influence on bigger buffer size measurements.

This simple benchmark was allegedly dependent only on *stride* and buffer *size*, although when we tried to reproduce it we encountered the opposite. There are numerous parameters that have severe influence on measurements and we will discuss some of them in the following sections. Here these parameters are already carefully manually tuned, so we can concentrate solely on *stride* and buffer *size*. First, we will show how it is possible to achieve the results that do resemble to the ones from Figure 2.4. After that, we will demonstrate the existence of many pitfalls regarding *stride* selection, on both ARM and Intel processors.

4.1.1 Stride Influence in General

Figure 4.1 displays the experiment for strides 1,2,4,8,16,32 on Intel Sandy Bridge processor. 42 repetitions have been performed, for the buffer size values ranging from 1KB to 100KB and the results have been represented with the boxplots. If looking only on strides 1,2,4, one can recognize the behavior similar to what we expected, based on Figure 2.4. There are two clean plateaus and sharp drop of performance between them. Different strides produce same bandwidths inside L1 cache, but different bandwidths for larger buffer size. Nevertheless, bigger strides already introduce some differences. The bandwidths for bigger strides, for small buffer size that is inside L1 cache, are lower than for the strides 1,2,4. Additionally, the difference in performance for larger buffer size between strides start to fade after *stride=8*. Furthermore, anticipated drop of performance by the factor two is not present even for the initial strides.

We remind that these results were obtained after carefully choosing environment setup. Additionally, strides used in this experiment are all (apart from starting point *stride=1*) multiples of 2. For the different stride selection, results are much more irregular.

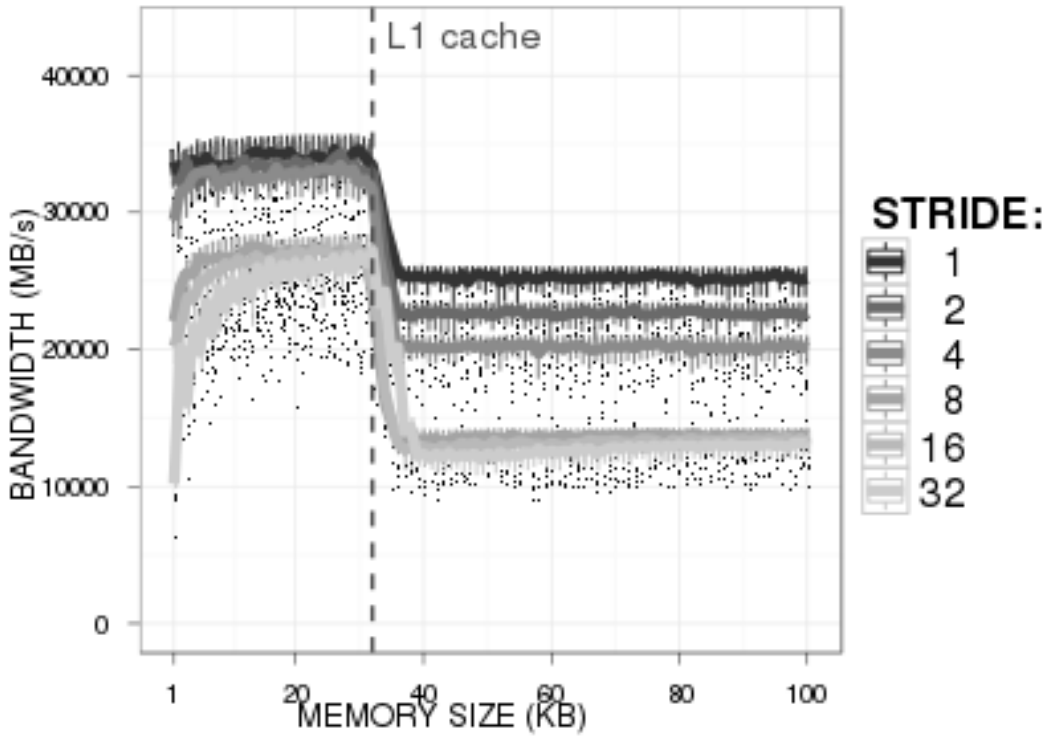

Figure 4.1: Optimistic results on Intel Sandy Bridge: 2 plateaus with a sharp drop between them; lower performance for larger strides for buffer size values bigger than L1 cache; 42 repetitions; buffer size 1KB-100KB; boxplots with average values line and dotted outliers; strides 1,2,4,8,16,32.

4.1.2 Unexpected Behavior Caused by Different Strides

Anomalies on Intel architecture

Figure 4.2 shows the results from the same Intel Sandy Bridge processor, with the same experiment setup previously described only difference being that this measurements were performed for the strides 32,64. On this plot, one can still observe the drop of performance on L1 cache size, along with the plateaus for the bigger buffer size. Nevertheless, the behavior for smaller buffer size values, the ones inside L1 cache, is completely different. Instead of clean plateaus, there is high variability that almost looks like a measurement noise. As a matter of fact, these deviations are not random at all, since the additional experiments with the same input parameters proved that these atypical patterns are completely reproducible.

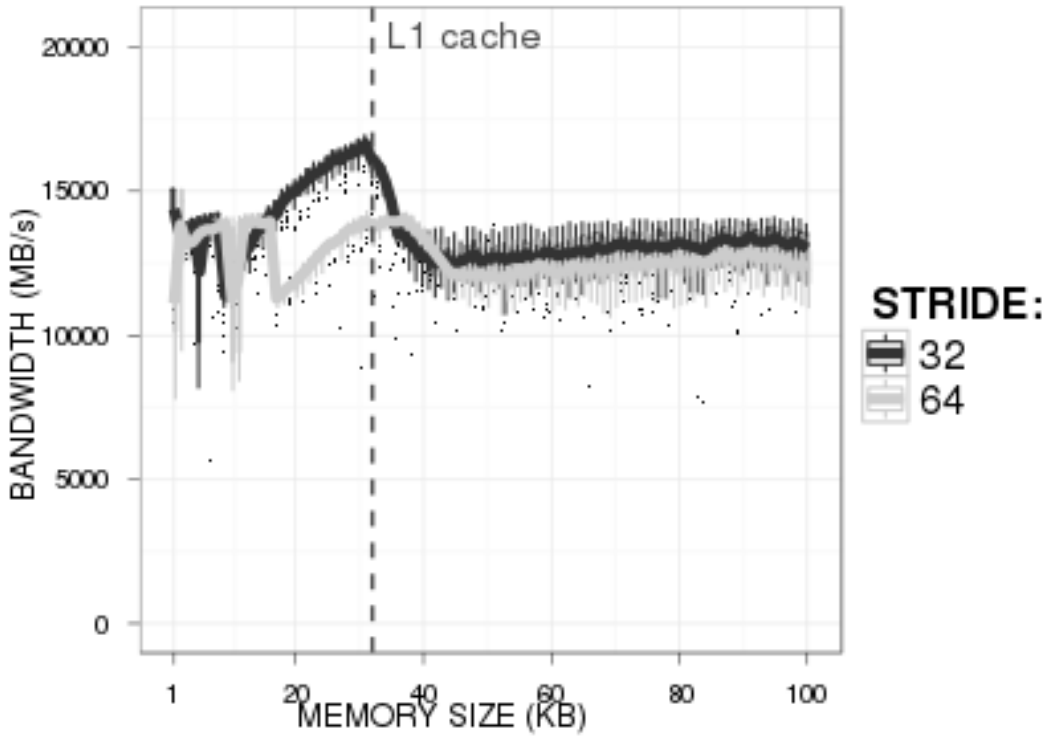

Figure 4.2: Atypical behavior inside L1 cache on Intel Sandy Bridge: unexpected, reproducible pattern for both strides for small buffer size. 42 repetitions; buffer size 1KB-100KB; boxplots with average values line and dotted outliers; strides 1,2,4,8,16,32.

Anomalies on ARM architecture

Figure 4.3 depicts the results from ARM Snowball processor, for strides 8,10,12,14,16. Buffer size ranges from 1KB to 50KB, with the boxplots representing 42 repetitions for each stride and buffer size. Left plot is showing only strides 8,16. Although there is a sufficient noise for buffer size values around L1 cache size (that will be explained in section 4.3), one can still observe the drop of average performance (represented by the solid line) caused by the buffer size getting to big to fit L1 cache. However, on right plot that displays all strides (8,10,12,14,16), one can see that intermediate stride values (10,12,14) do not induce performance drop, as the bandwidth values are uniform even when the buffer size goes larger than L1 cache size.

Explanation

One could expect slightly different performance when changing the *stride* parameter in for loop, but certainly not completely unique patterns. Without the complete access

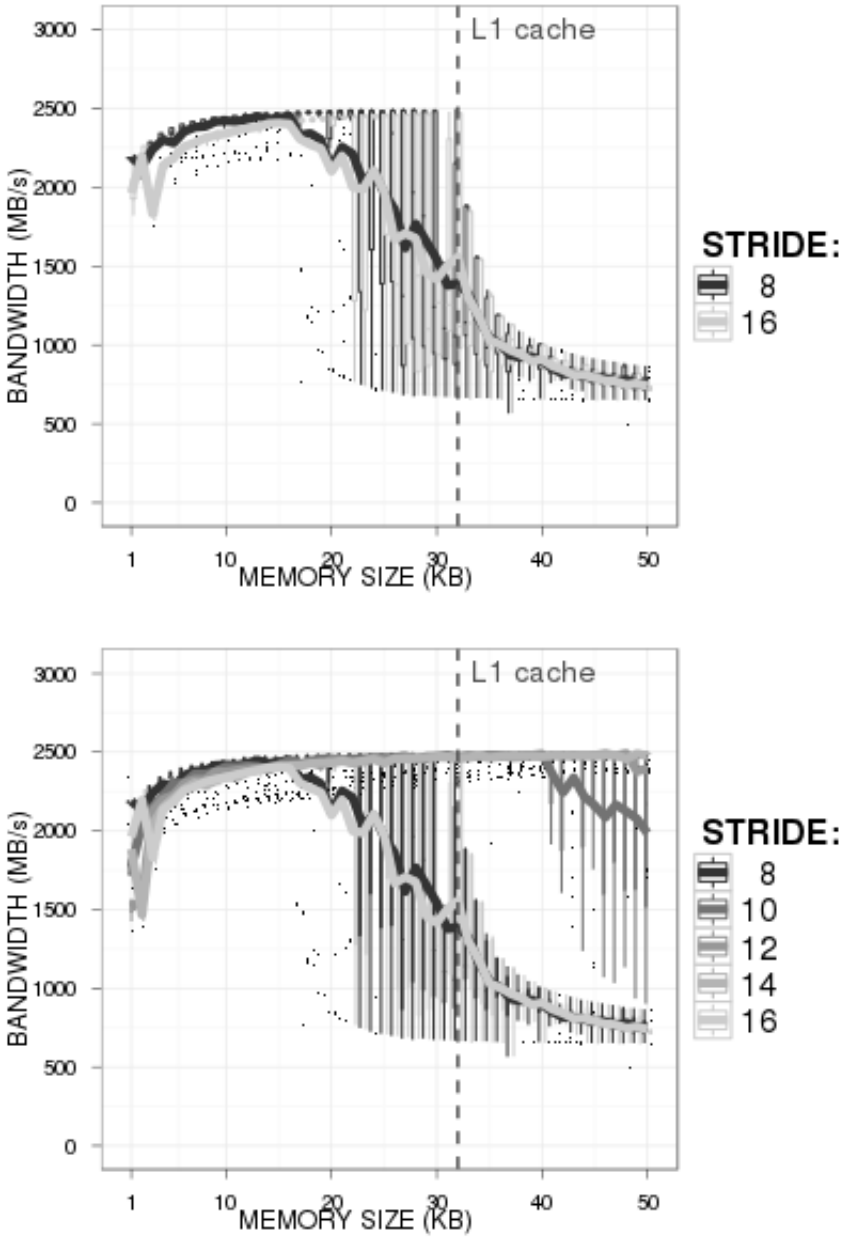

Figure 4.3: No drop on L1 cache size for strides 10,12,14 on ARM Snowball: strides 10,12,14 counterintuitively have better performance than stride 8; 42 repetitions; buffer size 1KB-50KB; boxplots with average values line and dotted outliers; strides 8,10,12,14,16.

to the hardware implementation details, it is hard find the satisfactory explanation for this unexpected behavior.

We did not go into deep investigation of what is causing described anomalies, as it was not our primary objective. The goal of this internship was not to find and explain all odd behaviors that can occur in measurements using our simple kernel. We only wanted to explore which parameters have the significant influence. These experiments are a good illustration of how very simple programs (just like our kernel), from which we expect regular behavior, can produce surprising results.

4.2 Noticable Behavior Change with Large Buffer Size

In previous section we presented measurements for buffer memory size that is small enough to fit into L1 cache or it is slightly exceeding it. The results for larger memory sizes, the ones that reach the limits of L2 cache and go beyond it, have their own characteristics.

In the related work (Figure 2.4), one could observe the sharp drop when the buffer size exceeds the L2 cache size. This is caused by the fact that the buffer cannot fit into L2 cache memory, therefore our simple kernel will have more L2 cache misses. Data will be fetched from the main memory (or L3 cache if it exists) and it takes more time (cycles) to do it, hence the overall time of execution of our kernel will be longer. Thought by our previous experience, we expected to have to do a lot of parameters tuning in order to get results similar to the ones on Figuree 2.4, but that we will reach them at the end. On contrary, once again we obtained unforeseen results.

The experiments were performed on ARM Snowball and Intel Sandy Bridge processors whose full memory hierarchy along with the other architectural differences can be found in chapter 2.1 . It is important to point out here that the L2 cache size is different for each of them, ARM having 512KB and Intel 256KB L2 cache size. Although ARM appears to be superior, which is counter-intuitive, we remind that Intel Sandy Bridge has three level of caches, with the 8MB shared L3 cache, while ARM has only two levels. Another substantial distinction between L2 on these architectures is that on ARM it is shared between two cores, while on Intel it is private. Nevertheless, we ensured that this last characteristics do not make the influence on our results, since no external process (except the system ones from operating system) was running on the second core on ARM during the whole experimentation period.

4.2.1 Smooth Performance Drop

Figure 4.4 represents the results for larger buffer memory size from both ARM Snowball (left plot) and Intel Sandy Bridge (right plot) processors. The strides are 1,2,4 but only the average values are shown on both plots. The observed behavior is analogous

on two machines: The performance decrease “smoothly“, there is neither clean plateaus neither sharp drop on L2 cache size. In additional experiments we proved that changing the element type and loop unrolling only changed the absolute values of the bandwidths, but the shape of the curves stayed the same.

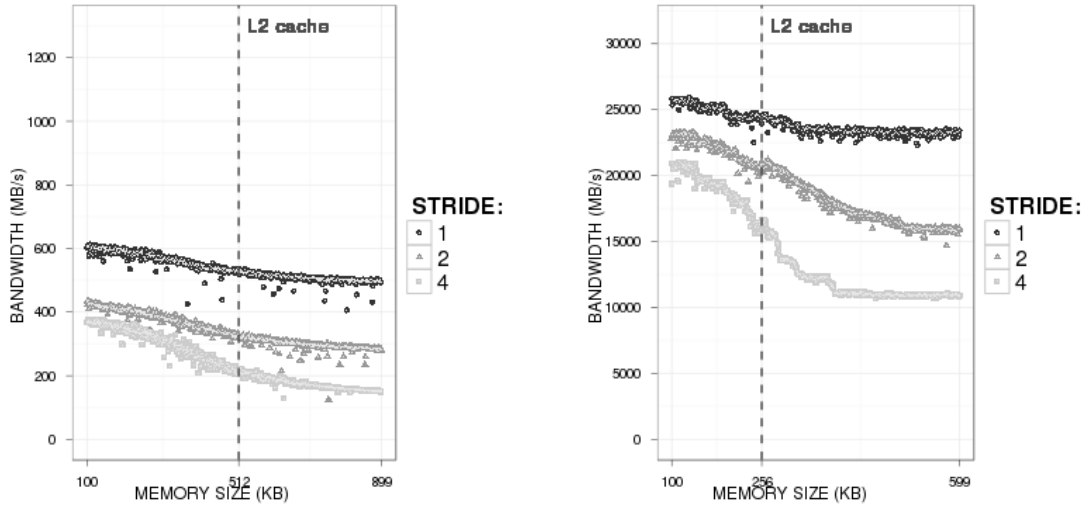

Figure 4.4: Smooth performance drop on ARM Snowball (left) and Intel Sandy Bridge (right): large buffer size; presenting only average values; strides 1,2,4.

There could be various causes for different behavior of L1 and L2 cache. Primary, all the L1 cache characteristics have the influence on L2 results as well, since most of the memory accesses even for large buffer memory size are still inside L1 cache. Respectively, for buffer memory size that exceeds the L2 cache size and goes into main memory (ARM) or L3 (Intel), there is an influence of both L1 and L2 cache. Secondary, the internal organization of L1 and L2 caches are not the same, with possibly different cache eviction and prefetching techniques which have severe influence on performance.

4.2.2 Performance Variability

Another interesting phenomenon we observed on larger buffer size experiments is regarding measurement noise. The average values do not vary from one measurement to another, but the confidence intervals of our results are bigger comparing to the small buffer size experiments. Although there is not a lot of outliers, the overall noise of experiments is not negligible any more. The one desiring to model the caches could not solely rely on average values, since this variation is significant enough to severely influence the final performance prediction.

Figure 4.5 on the left plot displays the same measurement on ARM Snowball as the Figure 4.4, but with the all values obtained from the experiment. Strides 1,2,4 can

still clearly be seen on the plot, but with much more noise. On the right plot of the Figure 4.5, there is a measurement from the same machine (ARM Snowball), but for the smaller values of buffer size. This plot also represents the strides 1,2,4 although it cannot be seen in the first part of the graph, while the buffer fits into L1 cache. It is due to the fact that the stride has no influence for the small buffer memory size, as it was explained in section 4.1. By comparing two graphs, one can recognize how for the smaller buffer size all the single measurement are very closely grouped together (plot on the right), while for the larger buffer size there is a substantial noise (plot on the left).

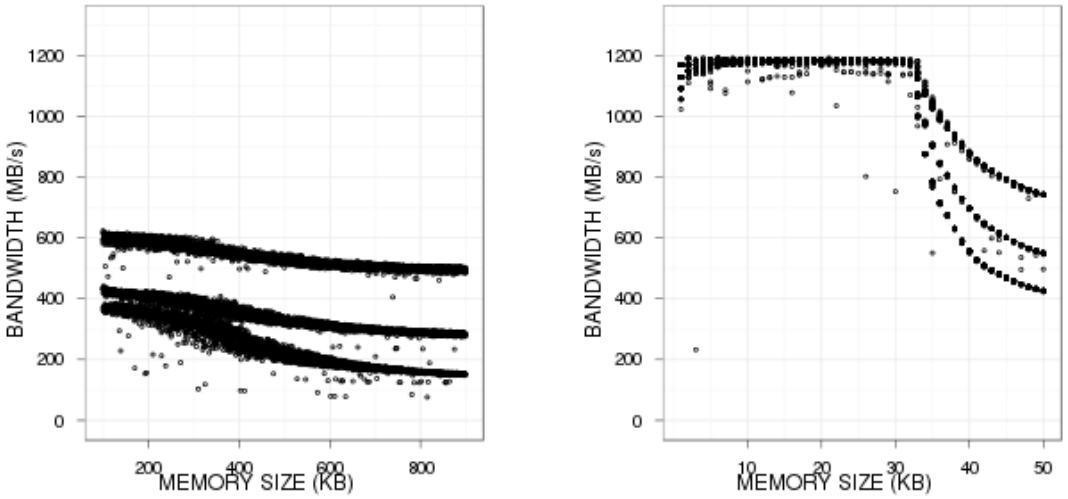

Figure 4.5: High performance variability on ARM Snowball: comparing large buffer size (left) with smaller buffer size (right); presenting all measurement values; strides 1,2,4.

Additionally, Figure 4.5 clearly shows how the drop on L1 cache size is much sharper than the one on L2, as already explained in subsection 4.2.1.

4.3 Influence of Allocation Strategy

Experiments on Snowball show that even though there was very little noise in our measurements, from one run to another we were getting very different results. The environment setup and input parameters were completely the same for all experiments. Figure 4.6 shows results of 4 consecutive experiments on Snowball, for stride that is equal to 1. Other strides we have tried in our experiments demonstrated similar behavior, which is the reason they are not displayed on the plots. 42 repetitions for each buffer memory size (on each plot) are represented by the boxplots. One can observe that there is very little variability in each experiment, but that the performance drop occurs on different places. Extreme values of buffer memory size were always exhibiting the same behavior, but the middle part (around L1 cache size) was unpredictable. This applies not only for these 4, but to all experiments we have done on ARM, since we encountered the same issue on Tegra 3 quad-core ARM Cortex A9 1.4GHz from Tibidabo cluster in Barcelona. On contrary, this issue was never present in any Intel machine we have used.

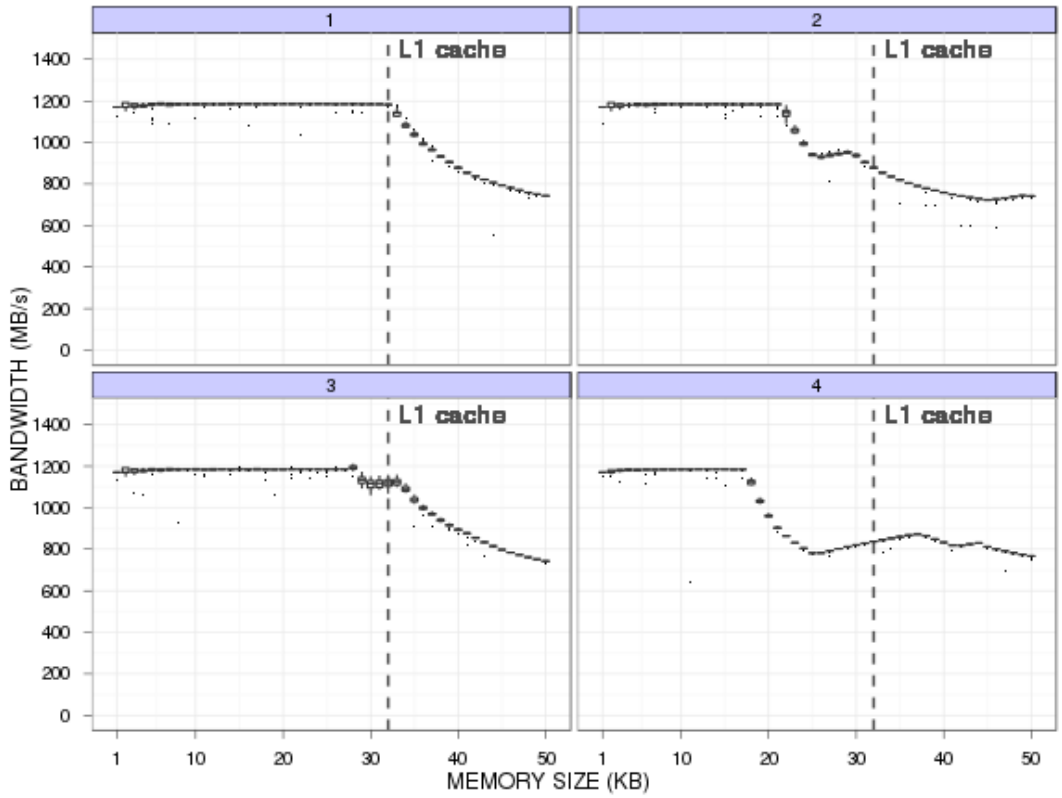

Figure 4.6: Reproducibility issue on ARM Snowball: 4 consecutive experiments with identical input parameters behaving differently; 42 repetitions for each buffer size depicted by boxplots show no noise for each single experiment; only showing stride=1.

After much efforts, we finally found the source of this surprising phenomenon. It comes from the way OS on ARM allocates physical memory pages. In some cases, nonconsecutive pages in physical memory for buffer memory size around 32KB (the size of L1 cache) are allocated, which causes much more cache misses, hence the drop of overall performance. Furthermore, during one experiment run, OS is likely to reuse the same pages, as we do malloc/free repeatedly for each buffer size. Hence, buffer starts from the same physical memory location for each buffer memory size during one experiment, which explains why there is no noise in the results.

To corroborate our claims, we tried different memory allocation technique. In the initial kernel, we were using individual malloc function call for each memory size and repetition to allocate buffer, after that executing for loop and finally freeing the memory. In the alternative version, we were doing only one memory allocation at the beginning of our program for all the measurements. We would allocate one big memory block (e.g. 2 MB), much bigger than our maximum buffer memory size (in these experiments 50KB). After that, for each memory size and repetition, we would randomly choose the starting point inside our big memory, i.e. we would perform memory accesses in for loop on randomly chosen small part inside our allocated big memory block. This way, we planned to experience the influence of different physical memory pages during the one experiment run and not always use the same.

Figure 4.7 supports our assumptions. Plot shows all single measurements, 42 for each memory size when using alternative memory allocation technique. Solid line is displaying the average values for each buffer size. This curve is behaving according to our first expectations from these measurements, with possibly only slightly less sharp angle of the performance drop due to the L1 cache size. Regarding variability on this figure, although at first sight it resembles to the random noise, if inspecting more closely one can recognize some regular patterns which represent the same behaviors as the examples from Figure 4.6.

If we have used second allocation technique from the start and concentrated only on maximum values of bandwidths, we would observe very regular behavior with a clean drop on L1 cache size. This way we would miss this physical memory allocation phenomenon, which eventually in modeling process could lead us to incorrect predictions. The same applies to the case of using only average values, as they would hide this high variation of results for the measurements around L1 cache size. This is a good example of how our methodology and decision to keep all data without any aggregation proved beneficial. It provided us with the opportunity to notice anomalies in our experiments which with the approach from related work would have been missed.

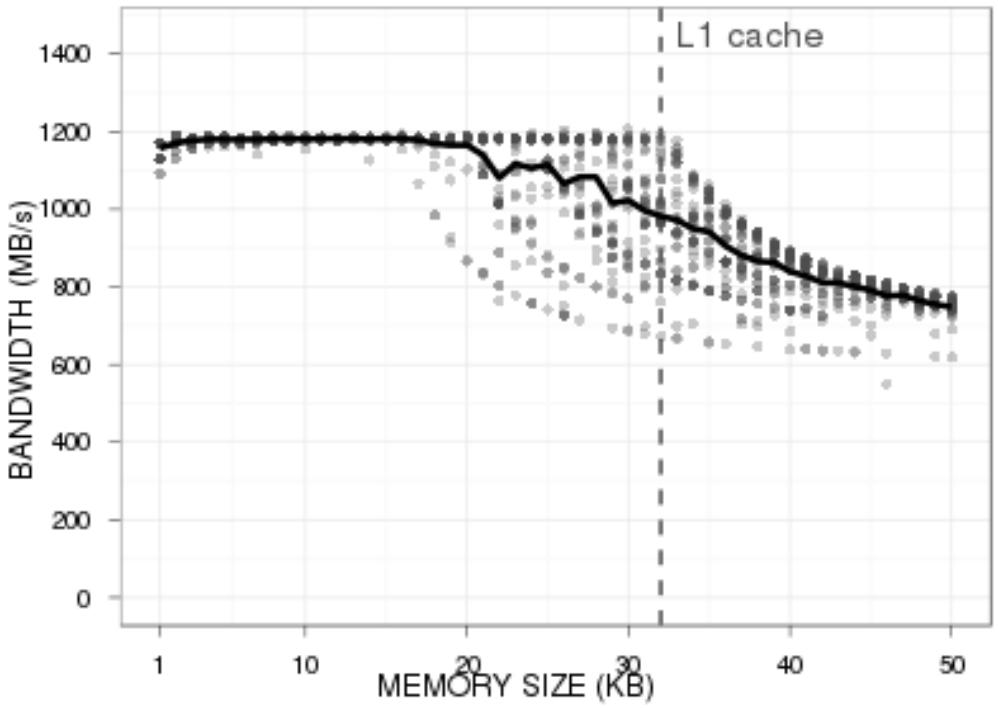

Figure 4.7: Explanation for strange results on ARM Snowball: using alternative memory allocation technique; all 42 repetitions represented by dots; solid line for average values; only showing stride=1.

4.4 Influence of Code Optimizations

In previous experiments we were mostly interested in trends of our measurements when buffer memory size is growing, i.e. the “shape“ of the average values curve. We presumed that the absolute value of the bandwidth depend mostly on the processor and memory bus frequencies. These parameters are always well known for each machine, therefore including them in the equation for the final model should not be hard. Nevertheless, there are some optimization techniques that can have severe influence on the bandwidth values (and surprisingly on the “shape of the curve“ as well) and they will be discussed in this section.

If we inspect more precisely the implementation of our code, the heart of the kernel (for loop) looks as following:

```
for(j=0;j<buffersize;j+=STRIDE)
  {
   sum+=buffer[j];
  }
```

Since the values inside buffer are completely random, we are not interested in the final result of the variable *sum*. Its sole purpose is to ensure desired memory accesses will be indeed performed and not discarded by compiler as dead code. Since simple addition, as the one performed here, takes sufficiently less CPU cycles than the memory access of the element inside the buffer, we can unreservedly claim that the memory speed and memory hierarchy is completely dictating the execution time of such code.

Evidently, any optimizations to this for loop would improve the execution time and consequently the bandwidth values would increase. We will demonstrate two techniques to do so, one changing the type of the buffer and the other regarding loop unrolling.

4.4.1 Changing the Size of Element

First optimization is concerning the type of the element our buffer is composed of. In the initial version of the code, buffer is a large array of integers. Integer size in C language is 32b (bits) (4 Bytes), from which follows that if the whole buffer memory size is 1KB (which is the minimal value we start our experiments) the buffer is composed of 1024B/4B=256 integers. If we change the buffer type and take for example *long long int* which is 64b (8B), for the same array size 1KB we would have two times less elements in the buffer (1024B/8B=128). The loop iterator is traversing whole buffer memory size element by element (multiplied by stride), hence for the *long long int* (64b) case we will have two times less iterations than for the *int* (32b). Since the final bandwidths are computed as the total number of accesses divided by the execution time of the loop, doubling element size should double the bandwidths as well.

This is a good example of how data level parallelism can improve the performance. Since increasing CPU frequency has reached its limits due to the high power consumption, hardware manufacturers are trying to use this feature along with increasing the number of cores and the size of caches in order to build faster machines. That is the reason why new generations of processors introduce hardware support for vectorized instruction. The main idea behind vectorized instructions is to pack the standard variables into larger vectors and then instead of performing the operation on each element do it on the whole vector. This is known as SIMD (Single Instruction Multiple Data). For example in our case, if one needs to do the sum of 256 (32b) integers, it would be much more efficient to do the sum of 32 (256b) vectors, each composed of 8 integers. Of course, special instructions need to be added to the processor instruction set to unable performing these kind of operations. Another drawback is that programers who want to benefit from this feature need to change their habitual approach. They need to use specific libraries and adjust their programing style to them.

In our experiments on Intel Sandy Bridge, we used SSE3 hardware extension for 128b vectors composed of two 64b *long long int*. For the 256b variables, we encountered a problem. Although AVX hardware extension supports vectors composed of integers (and long long integers), it does not support their addition. This feature will only be introduced in new Intel processors, still not available on the market. That is why we

were obliged to use vectors composed of *double*s, whose addition is supported by AVX. We are aware of the fact that comparing integer and double addition can be dangerous, since they do not use the same processor pipeline stages. Even though memory accesses are still the bottleneck of our simple program, we performed experiments and especially the comparisons extremely cautiously. It is important to point out that the goal of this internship was not to achieve maximum performance of our simple algorithm, but rather to investigate all possible influential parameters and build a model based on them.

On ARM Snowball we used "NEON", Advanced SIMD extension that is a combining 64b and 128b single instruction multiple data (SIMD) instruction set. Once again we encountered an issue that resembled to the one with the AVX on Intel. Particularly, The ARM Cortex A9 processors (family Snowball belongs to) do have the support for 128-bit vectors, but they can execute instructions with just 64b at a time. In other words our code cannot fully benefit from this optimization.

4.4.2 Loop Unrolling

Second optimization involves loop unrolling. Loop unrolling is a code transformation technique that tries to minimize branch penalty by unwinding the code inside the loop in order to lower the program's execution time. The price is paid in binary size of the code that increases drastically (space-time tradeoff). Nowadays, optimized compilers often perform unrolling automatically or upon request. In our experiments we have done it manually.

The code of our unrolled, optimized loop is displayed here:

```
for(j=0;j<buffersize;j+=STRIDE*8)
 {
  sum+=buffer[j];
  sum+=buffer[j+STRIDE];
  sum+=buffer[j+2*STRIDE];
  sum+=buffer[j+3*STRIDE];
  sum+=buffer[j+4*STRIDE];
  sum+=buffer[j+5*STRIDE];
  sum+=buffer[j+6*STRIDE];
  sum+=buffer[j+7*STRIDE];
 }
```

It is important to indicate that we have done only experiments with unrolling by the factor of 8, as we learned from the experts that the optimal number is often around this value. We do not claim that we achieved maximum performance with this number, but neither that was our objective. Our intention was only to examine whether this optimization has an influence on bandwidths and whether it is significant. Additionally, for the better comparison reasons, we wanted to do the unrolling by the same factor on all our

processors with all configuration setups. In fact, optimal factor for each architecture, even for the simple kernel as this one, probably differs.

4.4.3 Evaluation on Intel

Intel Sandy Bridge experiments are displayed on Figure 4.8. Only *stride=1* measurement are presented, as other ones behave similarly. The results from 42 repetitions for each buffer memory size are represented by the boxplots, which due to very high precision (very little noise) almost resemble to the line. Plots in the left column depict the results measured with the initial kernel (no loop unrolling), while on the right are the ones with 8 times unrolled loop. Rows are showing the results depending on the element type of the buffer, more precisely the size of the element (32b, 64b, 128b, 256b).

As it can be observed, increasing element type from 32b *int* to 64b *long long int* practically doubles the bandwidths. This trend, only a bit mitigated, continues with the vectorized instructions as values increase even more. Loop unrolling also has a positive effect, as the bandwidths increase in all cases except one. For the 256b vectorized instructions with loop unrolling, in the place where we expected the highest values, the actual results are extremely low. As it was not the primary goal of our work and we were constrained by the time, we did not fully investigate the reasons behind this anomaly. We assume that it is connected to the fact that the 256b vectors are composed of *double*s and not *int*s, but we have nothing to corroborate that hypothesis. For that reason, we will exclude the results from 256b element type measurements from the further analysis, although we think it was important to mention them, since they are a good example of how unexpected results for this kind of measurements can be.

Another very important phenomenon demonstrated by these plots comes from the "shape of the curves". One can observe that difference between the buffer memory size that is fitting L1 cache and the bigger ones is becoming more noticeable as the absolute values of bandwidth rise. Even more, for the 32b element type there is no drop at all when buffer size is getting out of the cache size. The cause for this behavior comes from the fact that we are not using the full capacity of our processor. When we add loop unrolling and increase element size, we are going towards the true boundaries of the processor, hence the limitations of the hardware are becoming more evident.

This implies that there could be some other deviations hidden each time we are not reaching the full potential of our machine. One can recognize that only with these two factors (element type and loop unrolling), we have achieved very distinctive results. In the context of modeling caches, this fact can become very troublesome.

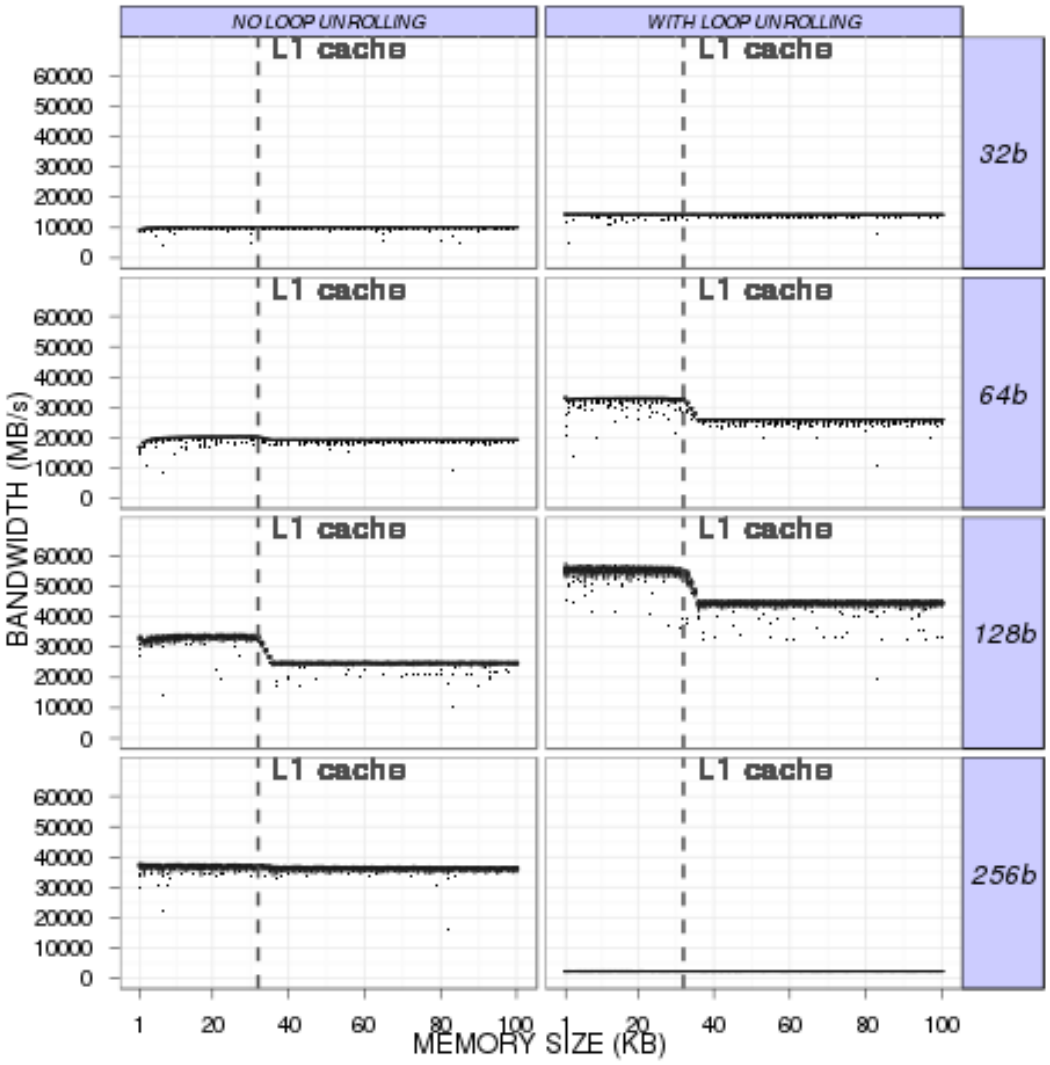

Figure 4.8: Influence of code optimizations on Intel Sandy Bridge: adding loop unrolling and changing buffer element size can increase performance and change curve pattern; 42 repetitions; buffer size 1KB-100KB; boxplots with dotted outliers; only showing stride=1.

4.4.4 Evaluation on ARM

Concerning the results from ARM Snowball, they are depicted on the Figure 4.9. Once again, since there is very little noise in these experiments, boxplots representing 42 repetitions of each buffer size, almost resemble to the line. The plots are organized analogous to the Intel results depicted on previous figure. The only difference is that on Figure 4.9 there is no plots for 256b element type, since such hardware support does not

yet exist for ARM processors. The fact that the drop of performance occurs on different places before L1 cache size is already described in section 4.3 and we will ignore it now, since it is the artifact of another parameter. We are now concerned only about the bandwidth value depending on the optimizations used.

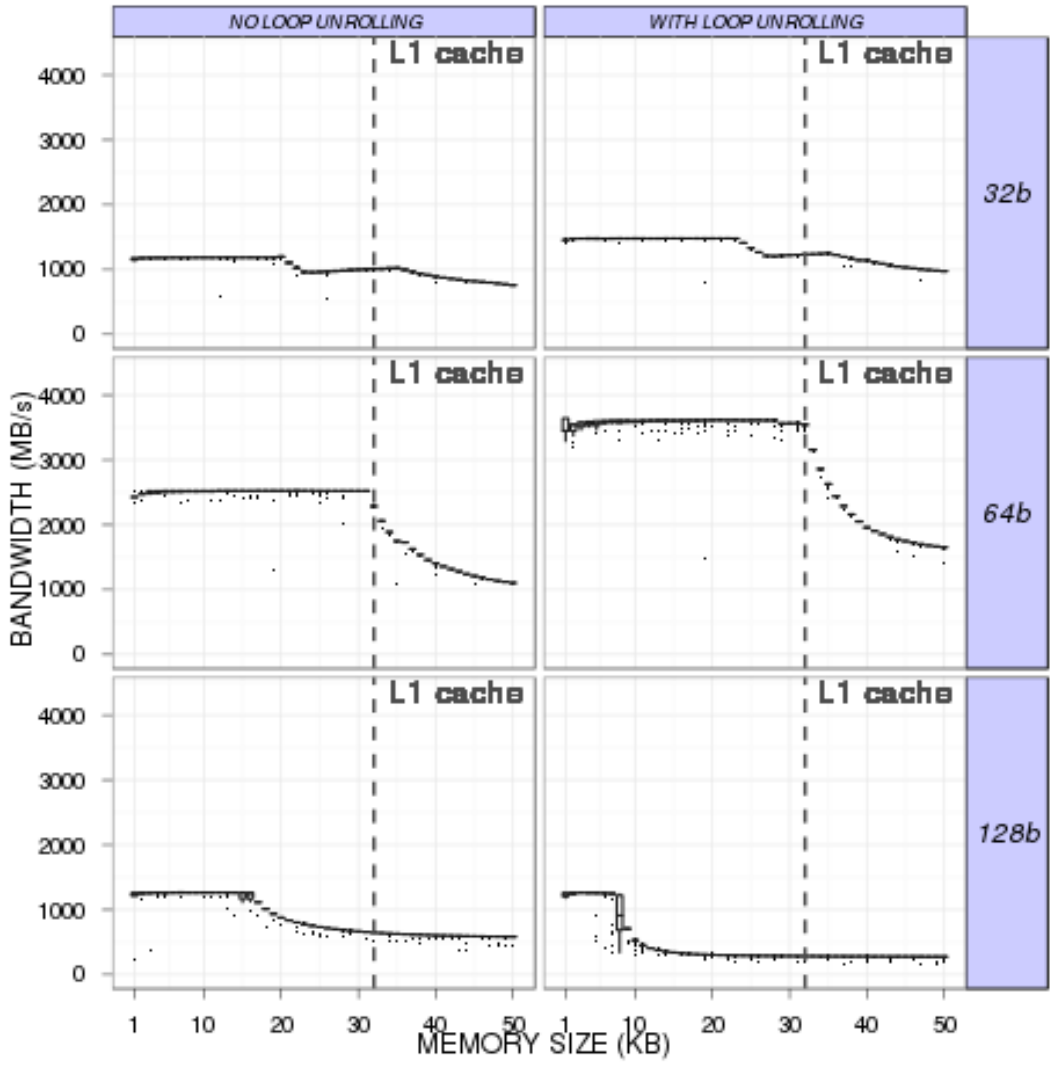

Figure 4.9: Influence of code optimizations on ARM Snowball: adding loop unrolling and changing buffer element size can increases performance and change curve pattern; 42 repetitions; buffer size 1KB-50KB; boxplots with dotted outliers; only showing stride=1.

Similar to Intel results, changing element type from 32b *int* to 64b *long long int* doubles the bandwidths. Additionally, using loop unrolling on both element types also

increases the bandwidth drastically.

Although the fact that 128b element size does not increase performance may at first appear as unexpected, it is actually due to the ARM Cortex A9 characteristics. These ARM processors do have the SIMD extension for 128b vectors, but they can execute instructions with only 64b per cycle. Additionally, loop unrolling has no influence on 128b vectorized instructions.

It is also important to notice that on ARM, contrary to the Intel, the drop caused by buffer memory size getting bigger than L1 cache is present in all cases, even when no optimization technique is used.

Finally, we can observe that, as expected, distinctive microarchitectures behave differently when some optimizations are applied to the code executed on them. First characteristics are absolute values of bandwidth they can achieve, which come primary from the differences in processor and memory frequencies. Second, more interesting, is the trend (shape) of the curve, i.e. changes optimizations make on drop at L1 cache size.

4.5 Influence of Compiler Optimization Option

Another way to optimize the performance is regarding the compilation options. As our code is written in pure C language, we used "gcc compiler" on both ARM and Intel machines. Various feature on gcc compiler have the unique goals when optimizing code, so it is not a surprise that using different options produces distinctive final results.

Although we have tried all levels, the best and most interesting results came from last two optimization flags. Gcc -O2 optimization enables all supported optimizations within the given architecture that do not involve a space-speed trade-off. On contrary, the third (-O3) and highest level enables even more optimizations by putting emphasis on speed over size. This includes optimizations enabled at -O2 and rename-register. The optimization inline-functions are also enabled here, which can increase performance but also can drastically increase the size of the object, depending upon the functions that are inlined. Although -O3 can produce fast code, the increase in the size of the image can have adverse effects on its speed. For example, if the size of the image exceeds the size of the available instruction cache, severe performance penalties can be observed. Therefore, in some cases it may be better simply to compile at -O2 to increase the chances that the image fits in the instruction cache.

We point out once again that we were recompiling our code for each measurement. Together with the optimization level, we were also logging the report of the compiler in order to capture any possible anomalies.

Figure 4.10 depicts results from ARM Snowball for two described compiling options. For each memory size there are 42 measurements, represented by small dots on both plots and solid lines connect the maximums. As all stride values we have tested behave similarly, we here present only the results for *stride=1*.

The noise around L1 cache size is already explained in section 4.3 and we will concentrate now only on maximum bandwidths these experiments have scored.

For the smaller buffer memory size, gcc=-O2 option (plot on the right) is not achieving the same bandwidths as gcc=-O3 (plot on the left). It can be observed that gcc=-O2 is reaching the boundaries in some cases (peaks for some memory sizes just before L1 cache size), but the gcc=-O3 option is securing these bandwidths during the whole period inside L1 cache. To conclude, the gcc=-O3 optimization is producing substantial speed up for our simple kernel. Nevertheless, it would be unreasonable to assume that this compiling option is better for other kernels, even the very similar ones. This stresses even more our claim that there are many setup configuration parameters that can drastically influence final performance prediction.

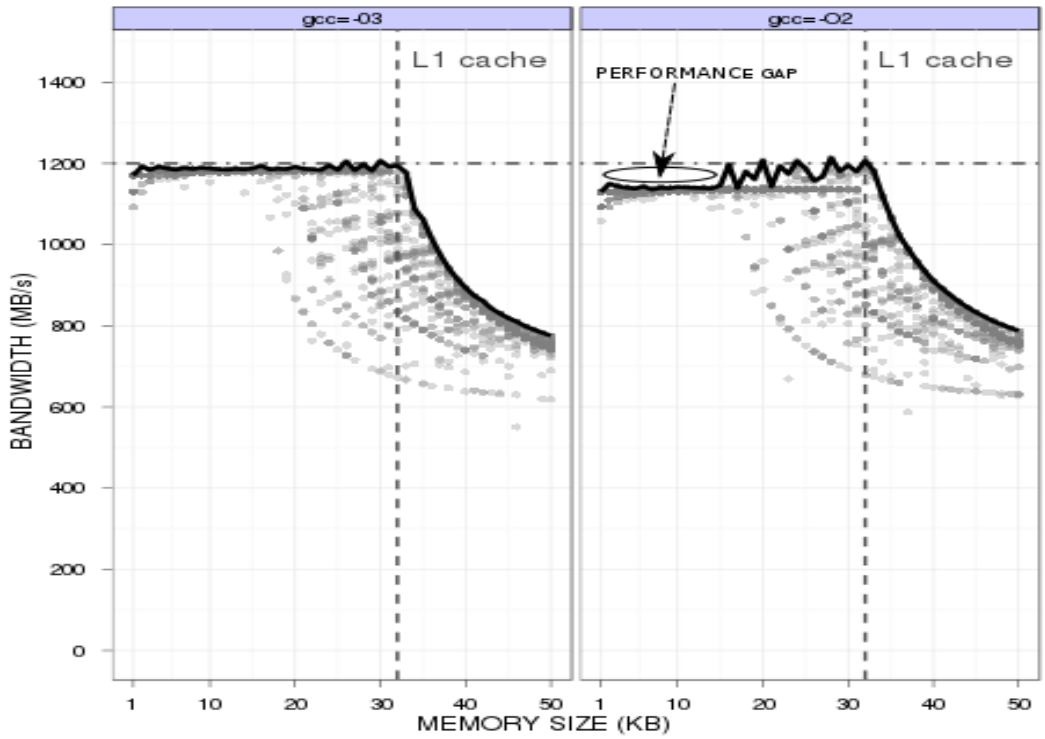

Figure 4.10: Different compilation optimization on ARM Snowball: -O3 providing better performance than -O2 for small buffer size; all 42 repetitions for buffer size 1KB-50KB are presented; solid line for maximum values; only showing stride=1.

Even though the results from Figure 4.10 are coming from ARM Snowball machine, we have observed the same behavior on Intel architectures. It is important to state that when other optimization techniques are enabled (loop unrolling and different element type), the differences between gcc levels gradually fade. It applies to both ARM and Intel microarchitectures. This demonstrates how particular parameters may or may not have the influence, depending on the rest of the configuration. It makes potential modeling of the machine behavior even harder.

4.6 Influence of OS Scheduling Policy

Although our kernel is very simple, single-threaded and we were alone on the machine while performing experiments, inevitably some external influence still existed. Primary, there was an operating system with its own processes running on processor in parallel. To minimize the influence, we removed all the unnecessary OS services, leaving only the elemental ones. Additionally, the core on which our program is executed was strictly pinned, in order to avoid potential changing of the core during the run-time.

Another important parameter is OS scheduler, since the way it is giving access to system resources directly affects the execution time of the code. Experiments with different scheduling policies were performed and we will demonstrate how in some cases on ARM architectures these can produce distinctive results.

The ARM Snowball measurements previous displayed were all obtained for the default scheduling priority. When executing the same experiments with "nice" priority, we observed equivalent results. Nevertheless, with real-time priority we encountered a peculiar anomaly.

Figure 4.11 shows the results of the experiment on ARM Snowball with the real-time priority. 42 repetitions for each buffer size ranging from 1KB to 50KB has been performed with *stride=1* (other strides behave similarly). All single measurements have been displayed on both plots as they represent the same experiment. Left plot is organized the same way as all the other plots previously presented, while right plot has a different x-axis.

Variations for bigger buffer memory size are explained in section 4.3 and we will ignore them in this analysis. We will focus solely on the presence of the two modes of execution.

First mode, the one with the higher bandwidth values is similar to the results we have obtained with other scheduling priorities.

Second mode, absent in previous experiments, has the bandwidth values that are almost 6 times lower. From the solid line that represents the average values (on the left plot), one can also conclude that this second mode is present in between 20-25% of the time.

This observation is even more clear when looking on the graph on the right side of the Figure 4.11. This plot is displaying the same data as the experiment on the left, with as usual y-axis representing the bandwidth values, but with x-axis changed, now representing the sequence order. Sequence order is the order in which our single measurements inside one experiment are conducted. As the order of using different buffer memory size values is completely randomized, approximately the same number of second mode execution is present for all buffer memory size values.

What is even more important to notice from the right plot is that the whole second mode occurred as "one big block", i.e. throughout one period of time during the whole experiment execution. This is very significant, as it is pointing to the fact that the second mode is almost certainly caused by some external process that was running in parallel

on the same core during that period of time. We were convinced even more in this assumption when we did additional experiments with equal setup parameters as they provided similar results.

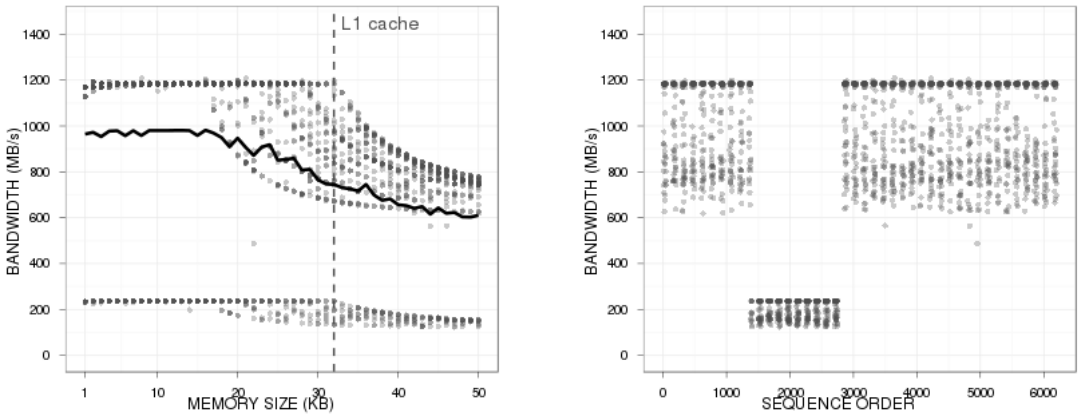

Figure 4.11: Real-time priority on ARM Snowball: 2 modes of execution; same data file on Bandwidth vs. Size plot (left) and Bandwidth vs. Sequence order plot (right); all 42 repetitions for each buffer size 1KB-50KB are presented; solid line for average values (left plot); only showing stride=1.

This anomaly could have probably been avoided if we had performed our experiments on a “bare metal“, i.e. without operating system. There would be no external influence and additionally the final bandwidths would be even higher. Nevertheless, the purpose of this internship was not to achieve the best performance but to evaluate it in a realistic environment. The future HPC code is going to run on the machines that will possess similar OS, not on “bare metal“. We are interested in investigating cache behavior in an environment that simulates the ones that will be loaded on future processors.

One can again recognize that our choice do postpone the aggregation of data along with logging all the relevant information during the experimentation process, proved to be crucial. For example, solely looking at average value would only show as that real-time priority is slower, but it would completely hide the relevant information about the existence of the two modes of execution. Finally, although this anomaly was exclusive to ARM architectures, there is no guarantee that it may not occur on some Intel microarchitectures we have not tested or on the ones with a different OS setup.

— 5 —
Conclusion and Future Work

Conclusion

This work has been done in the context of future HPC systems that will certainly be composed from large number of multicore processor. Due to the power consumption limitations, it is very likely that this future platforms will be built from more energy-efficient processor, such as ARM. Nevertheless, ARM processors have never been used so far on such a large scale and their behavior when executing HPC code is unknown.

Performance prediction, required to carefully plan large computer platforms, commonly rely on accurate processor models. However, previous models from the architectures used in such platforms might not be appropriate representation of ARM characteristics. Thus, this research is an important step towards better understanding of ARM architecture in the described context. Results obtained during this internship met some of our expectations and on the other hand showed some surprising phenomenons, but they all provide immense contribution for future use of ARM processor in HPC programing.

Numerous experiments regarding CPU caches on several different processor microarchitectures (from ARM and Intel families) have been performed. Although simplistic kernel was used, the results intensively varied depending on environment setup and optimizations. One can conclude from our experiments that predicting cache behavior is very difficult and that there are many parameters with great influence on the final results. Additionally, specific architectures have their own characteristic features (e.g. physical address issue on ARM), which makes any prediction even harder.

However good performance and estimations are still possible, but it requires a lot of careful tuning of all the parameters. One could question the motives for doing so, as it might not represent the realistic environment in which the future applications will be executed.

Apart from actual results, we have presented very clean, coherent and well-structured methodology that can be used in performance evaluation, not only in HPC area. We decompose our workflow into several steps and traverse through them carefully, logging all relevant information, as numerous parameters on every level can cause distinctive behavior. Later, we demonstrate how this systematic approach enabled us to distinguish anomalies that would have been missed otherwise.

This whole research has been done in the open science trend. In contrast to the PMaC[9][8], this work with all its source code, data and experiment analysis is publicly available. Additionally, our smooth, intuitive methodology makes it easy for everyone to reproduce the experiments and contribute to this research topic.

Future Work

We decided to restrict the scope of our research to thorough study of the simplistic kernel. Nevertheless, it would be interesting to investigate slightly modified code, adding functions that put more pressure on CPU. Even then, the program could still be memory bounded, but it might not be generating same bandwidth patterns.

Next natural step would be to try a multi-threaded version of our code. Since data would be shared among the threads, the influence that they would have on each other would dictate the overall performance.

Another direction would be to try running multiple instances of this simple kernel on the same machine. This way threads would be sharing processor resources, but not using the same data, thus the behavior would be different from the one with multi-threaded version of the program.

Loop unrolling has proved beneficial to the overall performance of our program. We have tried to do the unrolling only 8 times, as this number was suggested to us by more experienced researchers. Nevertheless, we suspect that the optimal value might not exactly be this one and that it could be different depending on the processor microarchitecture (possibly some additional parameters as well). Perhaps this optimal value would not only induce higher bandwidths, but also reveal some behavior not yet present in our results.

Another interesting feature, in the context of Mont-Blanc project and energy consumption, would be to measure actual power efficiency of our program. Specialized equipment have been developed for this purpose for ARM, USB device that connects directly to Snowball board and then captures to power consumption. Unfortunately, we did not have the opportunity to use this device while performing our experiments.

Finally, my personal ambitions are to continue with this work through a Phd. thesis. The goal would be to build precise models of processors depending on their cache hierarchy, but in an advanced context, investigating multi-threaded kernels on multicore machines. Later, these models could be used by frameworks like SimGRID [1] in order to predict performance of future computer platforms.

Although this would be a difficult path, filled with pitfalls similar to the ones encountered during this internship, I think that after this invaluable experience, I am well prepared for the task.

1. SimGRID: simgrid.gforge.inria.fr

Appendix

Sample of a Shortened Output File

```
##############################################
# Fri Jun  8 15:37:35 CEST 2012
##############################################
# EXECUTION PRIORITY:
# FILE NAME:   data/Snow2Data0.dat
##############################################
# VERSION:
# Linux version 3.1.0-1-amd64 (Debian 3.1.8-2) (ben@decadent.org.uk)
    (gcc version 4.6.2 (Debian 4.6.2-11) ) #1 SMP Tue Jan 10 05:01:58
    UTC 2012
##############################################
# CPU INFO:
# processor : 0
# vendor_id : GenuineIntel
# cpu family   : 6
# model    : 42
# model name   : Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
# stepping   : 7
# cpu MHz    : 1600.000
# cache size   : 8192 KB
# physical id : 0
# siblings   : 8
# core id    : 0
# cpu cores : 4
# apicid     : 0
# initial apicid   : 0
# fpu    : yes
# fpu_exception : yes
# cpuid level : 13
# wp     : yes
# flags    : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
    cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
    syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good
    nopl xtopology nonstop_tsc aperfmperf pni pclmulqdq dtes64 monitor
     ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm sse4_1 sse4_2 x2apic
    popcnt aes xsave avx lahf_lm ida arat epb xsaveopt pln pts dts
    tpr_shadow vnmi flexpriority ept vpid
# bogomips   : 6784.71
# clflush size   : 64
# cache_alignment : 64
# address sizes : 36 bits physical, 48 bits virtual
# power management:
#
```

```
//Here file is shortened, as the processor possesses 8 identical
    logical cores.
##############################################
# SVN REVISION:
# Path: Snow2
# URL: svn+ssh://stanisic_luka@scm.gforge.inria.fr/svnroot/memo/
    people/lstanisic/Snow2
# Repository Root: svn+ssh://stanisic_luka@scm.gforge.inria.fr/
    svnroot/memo
# Repository UUID: 71a86571-7a23-0410-9aa8-cd970210d03d
# Revision: 14129
# Node Kind: directory
# Last Changed Author: stanisic_luka
# Last Changed Rev: 14126
# Last Changed Date: 2012-06-07 21:16:21 +0200 (Thu, 07 Jun 2012)
#
##############################################
# COMPILATION:
# gcc -I$HOME/include -L$HOME/lib/ -g -w -O3 -DVERBOSE -DTYPE="int"
    -DVECTOR="0" -msse3 -mavx -D_GNU_SOURCE -c -o kernel.o kernel.c
# gcc -I$HOME/include -L$HOME/lib/ -g -w -O3 -DVERBOSE -DTYPE="int"
    -DVECTOR="0" -msse3 -mavx -D_GNU_SOURCE kernel.o -lm -lrt -o
    kernel
# rm kernel.o
##############################################
# #PARAMETERS FOR INPUT GENERATOR:
# 42 =SEED            //Seed for random number generator
# 1 =N_ARRAYS    //Number of arrays
# 2 =N_REP     //Number of repetitions of the whole measerement for 1
    stride
# 1024 =N_CYCLES    //Number of loops for 1 N_REP, for small values
    of memory should be 32768
# 1 =MIN_MEM  //Starting array size, the smallest size; When MODE=4
    these values are ignored
# 50 =MAX_MEM  //Ending array size, the biggest size; When MODE=4
    these values are ignored
# 1024 =TYPE_MEM //The size of one block in memory, if 1024=>1KB
# 1 =MIN_STRIDE //Starting stride size, the smallest size
# 4 =MAX_STRIDE //Ending stride size, the biggest size
# 3 =MODE   //Mode for producing input file: 1=REGULAR; 2=INVERT; 3=
    RANDOM; 123=ALL 3 MODES; 4=FIXED MEMORY; 5=FIXED STRIDES
# #THESE PARAMETERS ARE USED ONLY IN SPECIAL CASES (MODE=4 || MODE=5)
    :
# 1 2 4 8 16 32 64 128 256 512 1024 2048 4096 0//For FIXED MEMORY
    MODE: Values of the MEM
# 1 13 17 3 9          0//For FIXED STRIDES MODE: Values of the STRIDE
#
# !PARAMETERS FOR KERNEL:
# 3 =GCC     //GCC optimization 0=default; 2=-02; 3=-03
```

```
# 4 =CORE    //Core on which the program will be executed. 99 is for
    not specified.
# 1 =ALLOCTYPE  //Type of the buffer: 1=int; 2=long long int; 3=
    vectorized instructions
# 42  =SEED            //Seed for random number generator
# 1 =EXEMODE  //Execution mode: 1=STANDARD; 2=LOOP UNROLLING
# 1024  =ACCESSES_SQR //Number of accesses for EXEMODE=3, still not
    stable to use
# 1 =ALLOCMODE  //Allocation mode of memory: 1=MALLOC; 2=STATIC FOR
    BIGGEST MEMORY SIZE; 3=ALLOCATING WITHOUT FREEING; 4=ONE BIG FOR
    RANDOM ACCESS
# 2100  =STATIC_SIZE  //If ALLOCMODE=2 or ALLOCMODE=4, this is the
    size of statically allocated memory in TYPE_MEM (number of blocks)
    , 2100 default value for ALLOCMODE=4
# 1024  =TYPE_MEM //The size in bytes of one block in memory for
    statical memory allocation for ALLOCMODE=2 or ALLOCMODE=4, 1024=1
    KB

################################################
# MEASUREMENTS=
 NUM     STRIDE    SIZE    ADDRESS       START_TIME        END_TIME       TIME
      BANDWIDTH     MODE  EXEMODE
   0   1     19456     0x20d9490   1339162656.524169   1339162656.528627
       0.004458   4261.978909     3     1
   1   1     47104     0x20d9490   1339162656.528887   1339162656.539663
       0.010776   4268.692675     3     1
   2   1     7168    0x20d9490   1339162656.539708   1339162656.541354
       0.001646   4252.400632     3     1
   3   1     43008     0x20d9490   1339162656.541551   1339162656.546983
       0.005432   7732.017123     3     1
   4   1     11264     0x20d9490   1339162656.547010   1339162656.548162
       0.001152   9552.425448     3     1
   5   1     1024    0x20d9490   1339162656.548169   1339162656.548277
       0.000108   9244.023739     3     1
   6   1     22528     0x20d9490   1339162656.548322   1339162656.550482
       0.002159   10187.633047   3     1
   7   1     3072    0x20d9490   1339162656.550491   1339162656.550794
       0.000303   9907.562442     3     1
   8   1     36864     0x20d9490   1339162656.550865   1339162656.554464
       0.003598   10004.821768   3     1
   9   1     23552     0x20d9490   1339162656.554511   1339162656.556828
       0.002316   9929.959248     3     1
  10     1     40960     0x20d9490   1339162656.556908   1339162656.560923
       0.004014   9964.523804     3     1
//Here file is shortened, as the actual data is too big.
# ID of this measurement = 71061504; Total time: 0.825522 sec
```

Bibliography

[1] S. Browne, J Dongarra, N. Garner, K. London, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14:189–204, 2000.

[2] Christine Jacqmot. *Load Management in Distributed Computing Systems: Towards Adaptive Strategies*. PhD thesis, Universite catholique de Louvain, 1996.

[3] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1 edition, April 1991.

[4] Jack L. Lo, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, and Dean M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15:322–354, 1997.

[5] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec 1995.

[6] Douglas C. Montgomery. *Design and Analysis of Experiments, Student Solutions Manual*. Wiley, August 2005.

[7] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44(3):265–276, March 2009.

[8] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A framework for performance modeling and prediction. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[9] Mustafa M Tikir, Laura Carrington, Erich Strohmaier, and Allan Snavely. A genetic algorithms approach to modeling the performance of memory-bound com-

putations. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 47:1–47:12, New York, NY, USA, 2007. ACM.

[10] Vincent M. Weaver and Sally A. Mckee. Are cycle accurate simulations a waste of time?