UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Study and Benchmarking of Modern Computing Architectures

Master in Innovation and Research in Informatics (MIRI)

High Performance Computing (HPC)

May, 2017

Cristian Morales Pérez
(cristian.morales@bsc.es)

Director: David Vicente Dorca (david.vicente@bsc.es)
Co-director: Jorge Rodríguez Rey (jorge.rodriguez@bsc.es)
Tutor: Daniel Jiménez González (djimenez@ac.upc.edu)
Departament d'Arquitectura de Computadors (DAC)

Universitat Politècnica de Catalunya (UPC) – BarcelonaTech

Facultat d'informàtica de Barcelona (FIB)

Acknowledgments

I would like to thank my thesis directors David Vicente and Jorge Rodríguez for the opportunity offered to work on this project and for all the comments that guided me on the right direction. Thanks also to the thesis tutor Daniel Jiménez for all his support during the project.

To Carlos, thank you for all the ideas to improve this project. In general, thanks to all my colleagues at Operations Team for making an enjoyable working environment.

Special thanks to Esther for being always there, providing continuous encouragement and useful tips.

Abstract

Modern computing architectures change rapidly and exhibit high levels of complexity and heterogeneity. For example, the Barcelona Supercomputing Center (BSC) will have one general purpose cluster and three smaller clusters with Emerging Technologies like Many Core architecture from Intel, Power from IBM and ARMv8 from Fujitsu. These are technologies currently being developed to accelerate the arrival of the new generation of pre-exascale supercomputers. We have analysed these architectures in order to gain knowledge and experience to get the best performance possible from them running HPC applications that exploit the strengths of each architecture.

Additionally, we have run a benchmark suite on each available computer architecture with the purpose of characterize them. In order to complement the benchmark results, we have decided to analyse the execution of each benchmark with a performance analysis tool. There already exist several performance analysis tools which cover different performance areas like hardware events counting, performance simulation, traces, etc. However, these tools are dependent of being ported to modern computing architectures and they normally have different usages for different architectures.

We propose a solution based on the perf_event interface, which is included on the Linux kernel, that allows us to analyse the benchmark execution on the different modern computing architectures and it allows users easily to get performance information of their applications. The information that we provide with our solution are a sampling trace of hardware events, a profiling report and monitoring information of CPU and memory. Our solution enhances the performance results of other tools with extra features that help a better understanding of the bottlenecks of applications and/or architectures, and so, benchmarking. In addition, our proposal can be used in any system with Linux.

Content

List of Tables

List of Figures

1 Introduction

Nowadays, modern computing architectures change rapidly and exhibit high levels of complexity and heterogeneity. Further, the high-performance computing systems are usually composed of different computer architectures clusters with a variety of advantages and disadvantages for each of them. One of the challenges to reach the exascale computing is to be able to deal with the heterogeneity of the systems[1], [2]. Additionally, developing compilers that can boost productivity while producing efficient, optimized code for these rapidly evolving targets is a difficult challenge[3].

When a high-performance computing system is being setting-up, the system architects of these systems have to configure and tune it to obtain the best possible performance and take benefit of the strength of the computing architecture of the system. During the process of set-up of the high-performance computing systems, the system architects use benchmarks to check if the performance results are as expected for these architectures. Normally, we evaluate the high-performance computers by their performance on the High Performance LINPACK Benchmark (HPL), but due to the kind of algorithm that composes HPL, it strongly favours computers with very high floating-point performance and only adequate streaming memory systems[4]. So, executing a suite of benchmarks with different workloads as well as HPL will complement the performance result evaluation and it can provide a first view of which kind of application workloads will run better on this system with a specific architecture.

Aside from the final results reported by executing a suite of benchmarks, a general overview of the execution analysed by different performance tools can help to understand why the benchmarks results reported are better, worse or as expected. There are a lot of performance tools to analyse and evaluate the performance of any software. Also there are a lot of metric to measure with these tools, as hardware counters, CPU load, memory consumption, tracing, etc.

Normally, this type of performance analysis tools are used to analyse the behaviour of the application that runs on the high-performance computing

systems. The performance analysis tools are commonly used when an application is being developed, ported to a specific architecture or when the application does not have the expected performance. To sum up, when a cluster with a new computing architecture is on production, both system architects and users usually need tools to analyse their executions in detail.

1.1 Motivation

We are part of the Operations Team on the Barcelona Supercomputing Center (BSC), specifically we are on the User Support and Application Consultant Team. The Support team is responsible for providing user support to researchers using the BSC HPC-Resources, compiling, porting and debugging applications on each system, assisting the users to the efficient use of supercomputing resources solving issues related with performance, parallelization, optimization and scalability. Another task of the Support team is execute different benchmarks and applications when a cluster is being set up by the system administrators to test and evaluate the system and check that the performance results reported are similar to the expected. Normally we use the High Performance LINPACK (HPL) Benchmark, the High Performance Conjugate Gradients (HPCG) Benchmark and the Stream Benchmark.

The new general purpose supercomputer that will have the BSC in the next months is the MareNostrum 4, based on Intel Xeon E5V5 architecture with a performance capacity of 11 Petaflop/s[5]. To complement the general purpose supercomputer, BSC will provide a group of different clusters, known as "Emerging Technologies Clusters", with different kind of modern computing architectures to provide to the users the possibility of testing the architectures that will have the big clusters in the future. These are technologies currently being developed in the US and Japan to accelerate the arrival of the new generation of pre-exascale supercomputers. These clusters will serve users' needs and, in turn, will allow the centre to test and analyse the performance of the most recent developments in the field of supercomputing.

One of these type of clusters is based on Intel Many Integrated Core Architecture with Knights Landing (KNL) processors provided by Fujitsu. In a future it will be updated with a Knights Hill (KNH) processor provided by Lenovo. They are the same processors that will be inside the Theta and Aurora supercomputers purchased by the US Department of Energy for the Argonne National Laboratory[6]. Its computing power will be in excess of 0.5 Petaflop/s. Another cluster is based on Power architecture from IBM with Power8 processors with NVIDIA accelerators connected by NVLink technology. In a future it will be updated with IBM Power9 processors and newer NVIDIA GPUs, which are the same components that IBM and NVIDIA will use for the Summit and Sierra supercomputers that the US Department of Energy has commissioned for the Oak Ridge and Lawrence Livermore National Laboratories[7]. Its computing power will be over 1.5 Petaflop/s. Finally, a third cluster will be formed of 64 bit ARMv8 processors that Fujitsu will provide in a prototype machine, using state-of-the-art technologies from the Japanese Post-K supercomputer[8]. This cluster's computing power will also be over 0.5 Petaflop/s.

So as we commented, in the next years the BSC will have a group of four clusters, with different kind of modern computer architectures. In order to carry out our job correctly, we need to know and understand the specific characteristics of each computing architecture of the HPC systems that we provide support. To achieve this knowledge, we need to analyse in detail the computing architectures and test them exhaustively executing benchmarks and typical HPC applications. Thus, when a user contacts us with an issue, we have to have enough experience with the architectures to understand and solve the issue. For example, if a user reports low performance on his application or a failure execution it can be caused by the characteristics of the architecture where the application was running.

Furthermore, if we analyse the applications that users run on the clusters that we provide support we will be able to recommend the best architecture to execute their codes. Analysing their applications we can detect the bottlenecks of the application and which kind of performance behaviour has the execution. So, with this knowledge we will be able to complement the architecture analysis

with the application analysis. There are a lot of tools to analyse applications, like profilers, tracing tools, etc. But normally they are specialized on one area of the performance analysis and the tools that cover a large part of the different performance analysis areas need to be ported by the developers for each new computing architecture[9]. So, as we want to analyse in a simple way any application on new computing architectures, we have decided to develop a user-friendly performance analysis tool that covers different performance analysis areas and can be used on any system with Linux and not having to wait.

When the Marenostrum 3 was on production, one thing that we provided to the BSC users of the machine were the plots of the CPU load and Memory consumption aggregated per node for each finished job. This is very useful when the users are running codes that they are developing or debugging to try to understand a bit better why their applications runs better or worse. The two main problems were that only the users of BSC (and only for the jobs submitted with the BSC account) could access to these plots and that we had only provided for one machine, Marenostrum 3. So, a good point can be to provide to all users the possibility of track their application performance on any of the different clusters that BSC will have.

1.2 Objectives

Due to our needs and the scenario we can define the main objectives of the project on the next points:

- Analyse the computer architectures that BSC will have in the future, to provide to the users the best possible support. The architectures are Intel Many Integrated Core Architecture, Power8 with Power Architecture from IBM and ARMv8 by Fujitsu.

- As soon as we have a cluster with one of these architectures available, run benchmarks with different workloads on the system to evaluate if our previous analysis is on the right way. The benchmarks used are HPL, HPCG and Stream.

- Develop a performance analysis tool to analyse the executions of the previous benchmarks and the user's HPC applications. The required features of the performance analysis tool will be sampling of hardware counters, monitoring of the CPU load and memory consumption and backtrace to do a profiling of the program. Using the different features of our application we can see how the benchmark runs on each architecture and not only analyses the architecture with the final results of the benchmarks. Our approach is that it should be easy to use for any user and the code must be portable for any new computer architecture. Additionally, prepare the performance analysis tool to provide it to the users of the different HPC systems that we give support to trace their own applications. It will help to the users to have a general overview of their applications and help them in the process of optimize or ports their codes to the new computer architectures.

1.3 Document Structure

The rest of the document follows the next structure. Chapter 2 reviews the most relevant related work. Chapter 3 details the tools, architectures and benchmarks used in this project. Chapter 4 explains the design of our proposal and also provides details of the implementation. Chapter 5: composes it by the architecture analysis. After that, Chapter 6 consists of an evaluation of our work. Finally, Chapter 7 summarizes the work done and provides some concluding remarks, and future work proposals are presented in Chapter 8.

2 Related Work

This section comments some of the already existing approaches to characterize architectures, to characterize applications and others performance analysis tools that are used with similar purposes of the tool that we have developed.

2.1 Architecture characterization

On High-Performance Computing there are other projects that try to characterize architectures executing benchmarks codes and analysing the results. For example the BlackJack Bench is used to compile consciously of the architecture where the application will be executed or the HPCC (High-Performance Computing Challenge), a Benchmark suite used to complement the Top500 list results.

2.1.1 BlackjackBench

BlackjackBench [3] is a suite of portable benchmarks that automate system characterization and statistical analysis techniques for interpreting the results, similar to our objectives related with the architecture characterization. It is part of DARPA's AACE project, Architecture-Aware Compiler Environments that automatically characterizes the hardware and optimizes the application codes accordingly. The BlackjackBench discovers the effective sizes and speeds of the hardware environment rather than the often unattainable peak values. It characterizes the memory hierarchy, including cache sharing and NUMA characteristics of the system, properties of the processing cores affecting instruction execution speed, and the length of the OS scheduler time slot. It shows how they all could potentially interfere with each other and how established classification and statistical analysis techniques reduce experimental noise and aids automatic interpretation of results.

They have a similar idea to ours of characterize with a benchmark suite, but in their case focusing on memory characteristics of the architecture. It could be a good start point to our project, but it seems that the BlackJack project was abandoned, we have not found more information. Additionally, the source code is removed from their web page and the section of BlackjackBench is not visible on the ICL page, so we need to access directly to the URL.

2.1.2 HPC Challenge

The HPCChallenge [10] suite of benchmarks examine the performance of HPC architectures using kernels with memory access patterns more challenging than those of the High Performance Linpack (HPL) benchmark used in the Top500 list. The HPC Challenge benchmark consists at this time of 7 benchmarks: HPL, STREAM, RandomAccess, PTRANS, FFTE, DGEMM and b_eff Latency/Bandwidth.

We take the idea of a suite of benchmarks of HPCC, but we want to extend with other Benchmarks as HPCG or remove some of them. Moreover, HPCC executes all the benchmarks at the same time and our idea is to do the executions separately. Another difference is that we want to analyse with our performance analysis tool each benchmark to understand the performance results.

2.2 Application characterization

On the application part, there is also an interest to characterize applications for understanding the performance aspects of the applications. In this case, we have tried two licensed software, Performance Reports from Allinea and HPC Performance Characterization Analysis from Intel Vtune.

2.2.1 Allinea Performance Reports

The "Allinea Performance Reports" feature is included on the debugger produced by Allinea Software, Allinea DDT[11]. The main problem to use it in our project is that we need to have a portable code and Open Source, and Allinea DDT is a proprietary commercial software. Furthermore, it is dependent of the system and for some of them is not ported. They provide an effective way to characterize and understand the performance of HPC application executions with one single-page HTML report. On the report, there are detailed different performances areas, CPU consumption, communication (MPI), IO and Memory. We can see on the Figure 1 a part of the Allinea report analysing a HPL Benchmark execution.

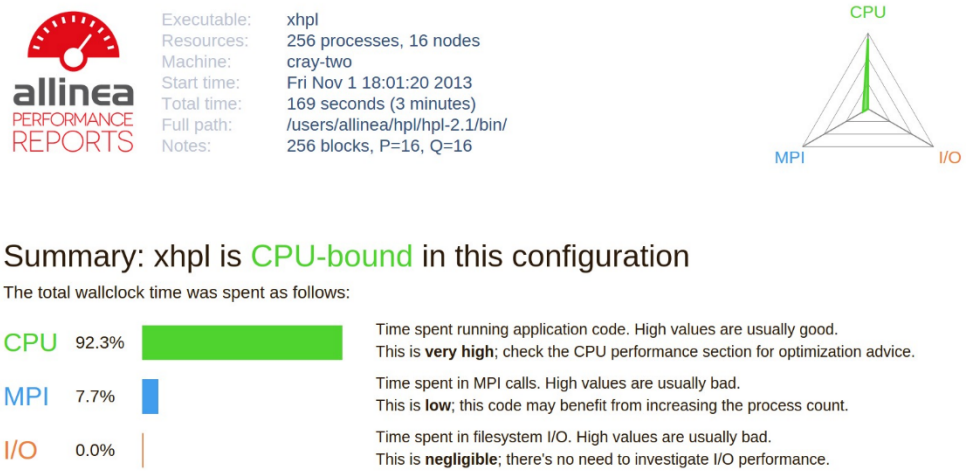

Figure 1: Allinea Report Example

2.2.2 Intel Vtune: HPC Performance Characterization Analysis

One feature of the performance profiler developed by Intel (Vtune) is the HPC Performance Characterization Analysis[12]. It helps to identify how effectively a compute-intensive application uses CPU, memory, and floating-

point operation hardware resources. In this case, it is a powerful tool and useful to achieve our project objectives, but it is also a proprietary commercial software (the complete version) and it only works completely for Intel Architectures.

2.3 Tools

As we have discarded the previous solutions to analyse the benchmark and application executions, we need to develop our own performance tool to obtain information of the profiling, the hardware events and the CPU and memory consumption. There are a lot of application that can satisfy our requirements, but only a few of them are close of satisfy all of them.

For example, in the area of profiling, one of the most known profilers is Gprof[13], but it only provides a profiling analysis and we must recompile our applications with the "-p" or "-pg" compiler flags to use it. Also, we have Oprofile[14] that reports the hardware events and the profiling. But, it only reports a final result of the Hardware events of whole execution, not a sampling trace. Furthermore, to read the same hardware counters on different architectures with Oprofile, we have to specify different event names for each architecture. Moreover, Oprofile must be ported by the developers to the most modern computing architectures.

Other option to read the performance counter hardware found in most major microprocessors is PAPI[15], but we have to instrument the code to use PAPI directly. One option is to use a third party application that uses PAPI, for example, PAPI_EX measures the entire run of an application using PAPI[16]. Another applications that uses PAPI are Extrae and Folding, developed by Tools group at BSC[17], [18]. But the problem that PAP has is the same that we have for Oprofile, we should wait some time meanwhile the tool is ported to the most modern computing architectures.

So, if one inconvenient is the porting of the performance analysis tool to the new computer architectures, we should focus on applications that comes with

the Linux Kernel, as “perf”. Perf is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command line interface[19]. Perf can instrument CPU performance counters, tracepoints, lightweight profiling, etc. And it is frequently updated and enhanced. Furthermore, Perf is based on the perf_events interface exported by recent versions of the Linux kernel and it is the interface that we will use on the developing of our own performance tool[20].

3 Methodology

This section covers the tools used on the project, the different tools that we have used to evaluate and validate our results, the computing architectures analysed during the project, and an overview of the benchmarks used.

3.1 Tools

3.1.1 perf_event interface

As we have commented the core of our solution is the perf_event interface, and specifically the perf_event_open system call, in order to analyse any execution doing sampling. A call to perf_event_open creates a file descriptor that allows measuring performance information. Each file descriptor corresponds to events that are being measured. The perf_event interface is included on the Linux kernel by default.

3.1.2 Post-processing

We have chosen Python to do the post processing when the execution of the traced application has finished. We have some experience with this language and it is very readable. Also, it has a variety of modules that provides features that make easier the post-processing task. For example, to process the traces we have used NumPy and to plot the traces we have used Matplotlib.

NumPy is a package for scientific computing with Python[21]. One feature that provides NumPy is an easy, but powerful way to manage with N-dimensional array objects. Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms and it combines perfectly with NumPy arrays to plot them[22].

3.2 Validation Tools

In order to ensure that our measurements are right and our tool works as expected, we have used other verified tools commented on the state-of-art part. As we commented, Perf is a profiler tool based on the perf_events interface, the same that we have used to develop our performance analysis tool, so to ensure us that we are using correctly the interface we have compared our results with the Perf results.

On our tool, we are performing an approximation of a profiling analysis, so to evaluate our precision on this type of analysis we will compare our approximation with the results of Gprof and Oprofile. Gprof is a profiling program which collects and arranges statistics on applications and OProfile is a performance analysis tool capable of monitoring native hardware events occurring in all parts of a running system, from the kernel (including modules and interrupt handlers) to shared libraries to binaries. With Oprofile we can evaluate also the hardware event counting.

3.3 Architectures

The approach of this section is to analyse the different architectures that will have the BSC in a near future and where we will execute a benchmark suite using our performance analysis tool. Also, we will describe the specification of the machines with these computing architectures that we have used for this project.

3.3.1 Intel General Purpose – Haswell

We have decided to analyse a general purpose to have it as a reference to compare it with the architectures from the Emerging Technologies Clusters. Another reason why we have chosen to analyse the Haswell architecture is to compare the results in the future with MareNostrum 4 with the new Intel architecture.

The Haswell microarchitecture builds on the successes of the Sandy Bridge and Ivy Bridge microarchitectures. The basic pipeline functionality of the Haswell

microarchitecture is depicted in Figure 2. Some of the innovative feature of the Haswell architecture are support for Intel Advanced Vector Extensions 2 (Intel AVX2) and FMA, support for Intel Transactional Synchronization Extensions, each core can dispatch up to 8 micro-ops per cycle, improved L1D and L2 cache bandwidth, four arithmetic logical units (ALUs), support for optional fourth level cache, etc.

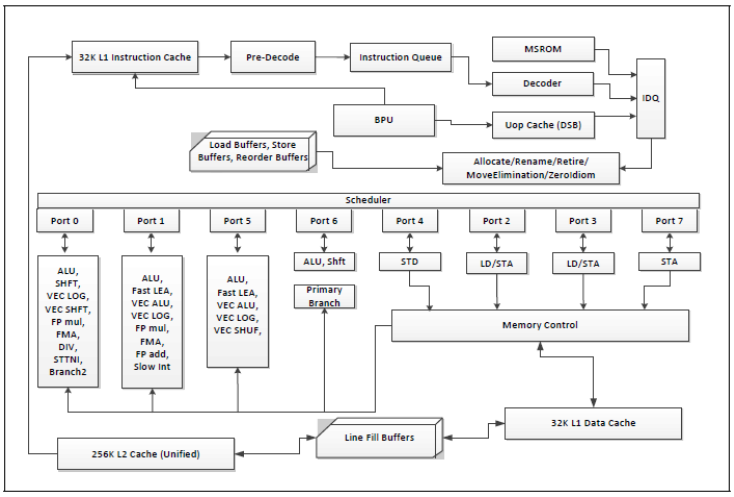

Figure 2: Intel Haswell Architecture [23]

The machine that we have used to analyse an Intel General Purpose architecture is Minotauro. MinoTauro is a heterogeneous cluster with 2 configurations. The servers that we have used on this projects are 39 bullx R421-E4 servers, each has the specification described on the Table 1.

CPU	2 Intel Xeon E5–2630 v3 (Haswell) 8-core processors, (each core at 2.4 GHz, and with 20 MB L3 cache)
Accelerators	2 K80 NVIDIA GPU Card
Memory	128 GB of Main memory, distributed in 8 DIMMs of 16 GB – DDR4 @ 2133 MHz - ECC SDRAM
Disk	120 GB SSD (Solid State Disk) as local storage
Network	1 PCIe 3.0 x8 8GT/s, Mellanox ConnectX ®–3FDR 56 Gbit and 4 Gigabit Ethernet ports.
OS	Red Hat Enterprise Linux Server release 6.7 (Kernel 2.6.32)

Table 1: Minotauro specification

We have not done an intensive analysis of the Haswell because we have only used it to have a reference architecture. Moreover, it is not a modern computing architecture, the CPU of Minotauro was released on the third quarter of 2014 and the Haswell partition of Minotauro was installed on December 2015. The most attractive part of the Haswell nodes of Minotauro are the 2 GPUs K80 per node, but we will not analyse NVIDIA accelerators on this project.

3.3.2 Intel Many Core - Knights Landing

The Knights Landing (KNL) processor differs from the usual Intel processor due to its very high core count and the hardware threading architecture. It represents an approach where a large number of simple cores are employed in large number as opposed to larger more sophisticated cores in smaller number. The idea is that a higher fraction of the transistors could be used for arithmetic operations[24].

The Knights Landing processor architecture is composed of up to 72 Silvermont-based cores running at 1.3~1.4 GHz. The cores are organized into tiles, each tile comprising two cores, and each core having two AVX-512 vector processing units as we can see on the Figure 3. Each tile has 1MB of L2 cache, shared by the two cores, for a total of 36MB of L2 cache across the chip. The tiles are connected in a 2D mesh topology. The cores are 14nm versions of Silvermont with claims by Intel that the out-of-order performance is vastly improved. Each core is out-of-order and is multithreaded, supporting 4 SMT thread. Being backward compatible with Intel Xeon products, KNL supports all previous ISA extensions: SSE, AVX and AVX2. On top of them, KNL adds support for the AVX-512 ISA, which will also be supported by upcoming CPU architectures such as Intel Skylake.

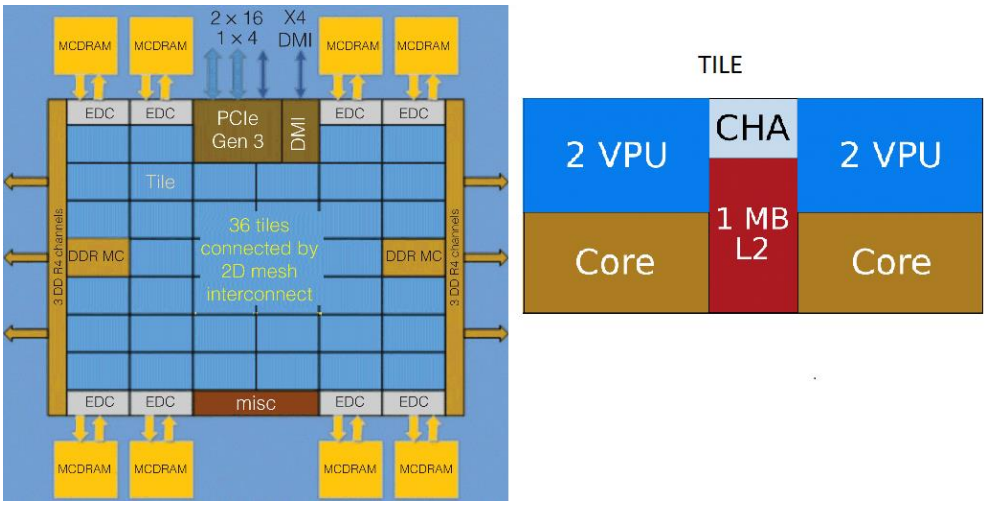

Figure 3: Knights Landing architecture [25]

The memory subsystem is composed of 16GB of high-speed stacked memory accessed by 8 high-speed memory controllers, as well as up to 384GB of DDR4 (96GB in our case) accessed by 2 3-channel memory controllers. While raw floating point and integer performance is an important aspect of the Knights Landing design, the achievable memory bandwidth is perhaps of the same or even greater importance. As it can be seen in Figure 4, the KNL memory can work in three different modes. These are determined by the BIOS at POST time and thus require a reboot to switch between them.

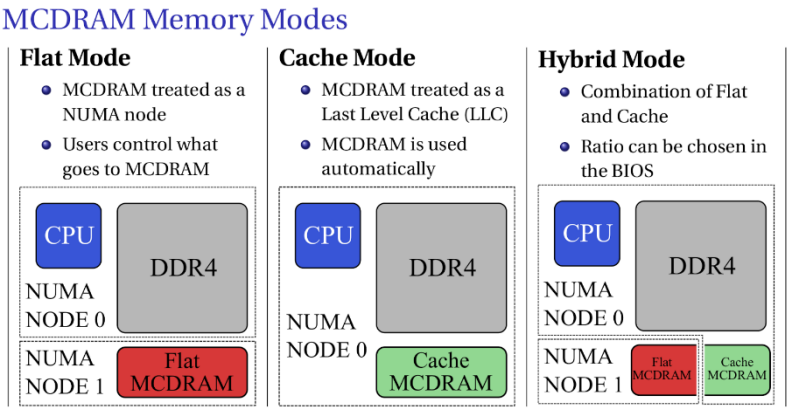

Figure 4: MCDRAM Memory Modes

As noted before, in KNL each of its cores has a L1 cache, pairs of cores are organized into tiles with a slice of the L2 cache symmetrically shared between

the two cores, and the L2 caches are connected to each other with a mesh. In the mesh, each vertical and horizontal link is a bidirectional ring. To manage this complexity and set the optimal mode of operation for any given computational application, the programmer has access to cache clustering modes. The Knights Landing interconnecting mesh operates in one of this clustering modes: all-to-all, quadrant, and sub-NUMA. These modes are selected at boot-time, and Intel does not provide any way to modify this setting without restarting the system. The performance varies with the different modes and the suggested setting from Intel for daily work is Quadrant, which we have used on the whole project.

To analyse this architecture, we have used the CTE-KNL that is based on Intel Xeon Phi Knights Landing processors, a Linux Operating System and an Intel OPA interconnection. On the Table 2 we can observe the configuration of the 16 nodes of the cluster.

Processor	1 Intel(R) Xeon Phi(TM) CPU 7230 @ 1.30GHz 64-core processor
Memory (DDR4)	96 GB main memory distributed in 6x 16GB DDR4 @ 1200 MHz
High Bandwidth Memory	16 GB high bandwidth memory distributed in 8x 2GB MCDRAM @ 7200 Mhz
Disk	120 GB SSD as local storage
Network	100 Gbits/s Omni-Path interface (GPFS via Ethernet 1 GBit)
OS	SUSE Linux Enterprise Server 12 SP2 (Kernel 4.4.21)

Table 2: CTE-KNL Cluster

3.3.3 IBM - Power8

One of the Emerging Technologies Clusters that has been setting up is a machine provided by IBM based on the Power8 architecture with NVLink (codenamed “Minsky”). The Power8 processor is the latest RISC (Reduced Instruction Set Computer) microprocessor from IBM. It is fabricated using the company’s 22-nm Silicon on Insulator (SOI) technology. Going against this industry trend, the Power8 processor relies on a much improved core. Figure 5

shows the processor architecture and the core architecture of a Power8 processor[26].

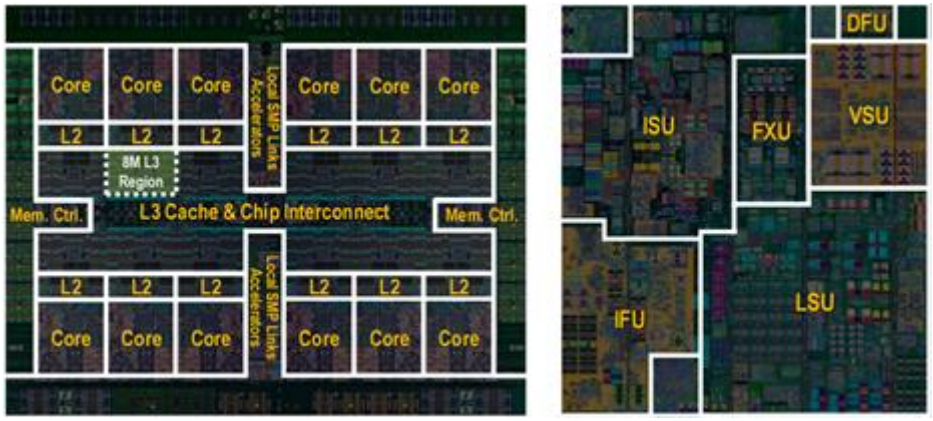

Figure 5: Left: Processor architecture. Right: Core architecture [27]

The core consists primarily of the following six units: instruction fetch unit (IFU), instruction sequencing unit (ISU), load-store unit (LSU), fixed-point unit (FXU), vector and scalar unit (VSU) and decimal floating point unit (DFU).

In addition, the Power8 processor design is also targeted for big data, analytics, and cloud application environments. Big data applications typically have a larger memory footprint and working set than traditional commercial applications. Correspondingly, compared to the Power7 core, the Power8 core has a L1 data cache that is twice as large, has twice as many ports from that data cache for higher read/write throughput, and has four times as many entries in its TLB (Translation Lookaside Buffer). As mentioned, the L2 and L3 caches in the Power8 processor are also twice the size of the corresponding Power7 processor caches, on a per core basis. Furthermore, thanks to CAPI (Coherent Accelerator Processor Interface) Accelerators, it enables heterogeneous compute (GPU, FPGA, etc.) that it is common used on big data systems. CAPI, is a high-speed processor expansion bus standard, initially designed to be layered on top of PCI Express, for directly connecting CPUs to external accelerators like GPUs, ASICs, FPGAs or fast storage. It offers direct memory access connectivity between devices of different instruction set architectures with low latency and high speed.

Also, the SMT8 of the Power8 architecture helps to hide the memory latency with many threads per core.

The next step of IBM after release the Power8 architecture was presenting a new version of its Power8 chip featuring NVIDIA's NVLink interconnect (used to be called Power8+). NVLink is an energy-efficient, high-bandwidth path between the GPU and the CPU at data rates of at least 80 gigabytes per second, or at least 5 times that of the current PCIe Gen3 x16, delivering faster application performance. NVLink is the node integration interconnect for both the Summit and Sierra pre-exascale supercomputers commissioned by the U.S. Department of Energy, enabling NVIDIA GPUs and CPUs such as IBM Power to access each other's memory quickly and seamlessly. In addition to speeding CPU-to-GPU communications for systems with an NVLink CPU connection, NVLink can have significant performance benefit for GPU-to-GPU (peer-to-peer) communications as well. Our performance tool is not prepared to analyse CUDA application, so we could not evaluate this Power8 feature.

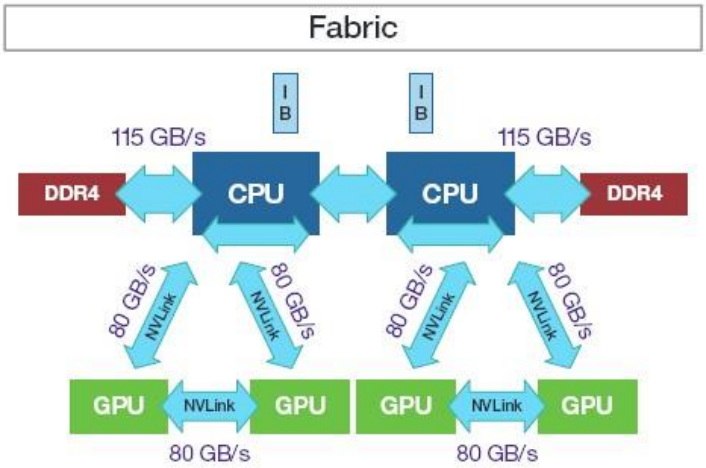

Figure 6: NVLink [28]

As the Emerging Technologies Cluster with Power8 with NVLink was not available when this project has been done, we have done our tests on other BSC cluster with Power8 architecture, named bscpower8. This cluster has four Power8 nodes, two nodes are the model 8247-42L and the other two are model 8335-GTA. We have used the model 8247-42L, on the Table 3 we can observe the model specification.

Processor	4 sockets Power8 6C @3,4GHz (8 threads each core)
Memory (DDR4)	16x 64GB CDIMM @ 1600 MHz
Disk	4x 300GB 15Krpm SAS
Accelerator	2x NVIDIA Tesla K40m
Network	Ethernet
OS	Ubuntu 16.04.1 LTS (Kernel 4.4.0)

Table 3 - Power8 Cluster

3.3.4 Fujitsu - ARMv8

The last cluster to be installed of the first wave of CTE on BSC will be the cluster based on ARM provided by Fujitsu. Few years remain for that, but we can prepare us a bit to understand better the ARMv8 architecture.

Nowadays, the only information about the future cluster is that it will be a small cluster based on the same technologies of the Post-K supercomputer from Fujitsu. This Post-K system represents the fourth generation of Fujitsu's "K" supercomputing line, which up until now was based on SPARC64 processors. Riken and Fujitsu are working together to provide a successor to the K computer with application R&D teams using co-design approach. They will develop their own CPU adopting ARMv8 with Scalar Vector Extension Instruction Set Architecture. The main goals of Fujitsu and Riken for this supercomputer is to have high application performance with good power efficiency and to provide a good usability and better accessibility for the users[29].

In order to have a prevision of what we can expect from Fujitsu and Riken, the achievements of the K supercomputers are being right now on the first place of the HPCG list and of the Graph500 list, being the winner of some categories of the HPCC Challenge awards and being on the 7th place of the Top500 list of November 2016 (it was the 1st when the supercomputer was set up, 6 years ago). We can expect a Many Core architecture for the ARMv8 processor of the Post-K supercomputer. They are working on a Scalable Many Core concept on the FX100 computer, the previous step to Post-K Supercomputer. As we do not have already the cluster, we should have done the tests on other current machines with ARMv8 architecture. The main problem with this machines is that they are

not manufactured by Fujitsu, because Fujitsu is developing a custom CPU core adopting ARMv8 architecture adding a lot optimization, including a custom software stack and compilers. So we have decided to do not perform any benchmark on current ARMv8 machines, because the results cannot be comparable with the results that we will perform when the Fujitsu cluster arrives.

3.4 Benchmarks

In order to analyse the capacity of the systems we have used the High Performance Linpack (HPL) Benchmark, the High Performance Conjugate Gradients (HPCG) Benchmark and the Stream Benchmark. HPL was introduced by Jack Dongarra and the benchmark measures the amount of time it takes to factor and solve a random dense system of linear equations (Ax=b), and converts that time into a performance rate (Floating Point operations per second). The HPL benchmark is the most widely recognized metric for ranking high performance computing systems[30].

The input of the HPL benchmark is specified on a file, where we define the problem size, block size, problem distribution, etc. In order to find out the best performance of the system, the largest problem size fitting in memory is what we should aim for. The amount of memory used by HPL is essentially the size of the coefficient matrix. In the case of the block size, HPL uses it for the data distribution as well as for the computational granularity.

However, HPL is increasingly unreliable as a true measure of system performance for a growing collection of important science and engineering applications. HPL is a simple program that factors and solves a large dense system of linear equations using Gaussian Elimination with partial pivoting. The dominant calculations in this algorithm are dense matrix-matrix multiplication and related kernels. This kind of algorithm strongly favours computers with very high floating-point computation rates and adequate streaming memory systems. That is why Jack Dongarra introduced the HPCG benchmark.

HPCG is composed of computations and data access patterns more commonly found in applications. The HPCG benchmark implementation is based on a 3-dimensional regular 27-point discretization of an elliptic partial differential equation. The implementation calls a 3D domain to fill a 3D virtual process grid for all the available MPI ranks. HPCG uses the preconditioned conjugate gradient method to solve the intermediate systems of equations and incorporates a local and symmetric Gauss-Seidel preconditioning step that requires a triangular forward solve and a backward solve. HPCG implements matrix multiplication locally, with an initial halo exchange between neighbouring processes. Using HPCG we strive for a better correlation to real scientific application performance and expect to drive computer system design and implementation in directions that will better impact performance improvement[31]. The HPCG input parameter are specified on a file, but in this case they can be passed to the program as an argument. There are only two variables, the input size and the running time of the benchmark.

Also, in order to characterize an architecture depending on the performance of the memory we have used the STREAM benchmark[32]. The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for four simple vector kernels. We can observe this vector kernel on the Table 4. The vector size is defined on compilation time with an environment variable. Each of the four tests adds independent information to the results: "copy" measures transfer rates in the absence of arithmetic, "scale" adds a simple arithmetic operation, "sum" adds a third operand to allow multiple load/store ports on vector machines to be tested and "triad" allows fused multiply/add operations. Of all the vector kernels Triad is the most complex scenario and is highly relevant to HPC.

NAME	KERNEL
COPY	a(i) = b(i)
SCALE	a(i) = q*b(i)
SUM	a(i) = b(i) + c(i)
TRIAD	a(i) = b(i) + q*c(i)

Table 4: STREAM Kernels

4 Development

In this section we will explain the development of our performance analysis tool. The tool can be divided into four parts: the sampling of hardware counters, the sampling of the stack trace, the monitoring and the post processing. The three first parts are on the same application and they are written on C, instead the Post-Processing is written in Python. Also, we have analysed an example application to see how our performance analysis tool works.

4.1 Perf-tool Development

The core of our performance analysis tool is based on the perf_events interface that provides the perf_event_open call to measure performance information. With this function we can measure any available event of any process running, specifying the PID of the process. So, the first step on our tool is run a new process to get the PID and stay the child waiting until the tool initializes all the structures and methods.

```
process.pid = fork();
   if (process.pid == 0) { // child
      raise(SIGSTOP);
```

Figure 7: Snippet - Init process

Aside from the PID of the analysed process, we have to specify on the perf_event_open function some attributes to define the behaviour of the performance monitoring of perf_event, for example the sampling frequency, the extra information that we want on our samples and the most important, which hardware event we want to count. Furthermore, one of this options is the read method for the hardware event, counting or sampling. A counting event is one that is used for counting the aggregate number of events that occur. In general, counting event results are gathered with a read call. A sampling event periodically writes measurements to a ring-buffer that can then be accessed via

mmap and can obtain more information from the samples, as the stack trace or the CPU where the process is running. So, the best option is the sampling, but has some incompatibilities with other features of the perf_event interface, like inherit the hardware events counting of the child process. The inherit option is interesting because we can analyse easily the parent and the child hardware events aggregated on a single file descriptor. But, with the sampling mode, we can detect when the analysed process creates a child and its TID (thread identifier). So, with the TID we can call again to the perf_event_open specifying the TID on the PID field and start counting the hardware events of the created child as a new process. Finally, another option that we use by default and it is incompatible with the inherit mode is grouping events. It allows to create a group of events to measure multiple events simultaneously with a single file descriptor. On the Table 5 we can see the incompatibilities between the perf_event options.

Inherit	Read mode	Group events	Detect threads creation
NO	MMAP	YES	YES
YES	READ	NO	NO
NO	READ	YES	NO

Table 5: Incompatibilities between perf_event options

The hardware events that we have chosen to count by default are: cycles, instructions, last level cache accesses, last level cache misses, branch instructions and branch miss predictions. We have chosen these events because one requirement for our tool is that could be used on any architecture and these events are the generalized hardware events provided by the kernel in most of the modern architectures. Additionally, we can use optional events like, context switches, CPU migrations, L1 cache loads and L1 cache loads misses. Most CPUs support events that are not covered by the generalized events. So, as an optional feature, we have added that the user can add any hardware event to be sampled. The libpfm4 library can be used to translate from the name in the architectural manuals to the raw hexadecimal value perf_event_open expects on the event field[33]. With this option we can extract more specific performance information, for example we can get the cycles where the execution is stalled due to L1 data cache misses. Our first idea was to add by default a metric that measure the floating point operation executed using specific hardware events, but since the

Haswell architecture, Intel does not provide the proper hardware events to count the Floating Point Operations executed[34].

With the process created and the attributes defined, we can call the perf_event_open function to get the file descriptor to read directly it or create the ring-buffer. The next step is to define how we count the events periodically. Events can be set to notify when a threshold is crossed, indicating an overflow. The overflow events can be captured via signal handler, by enabling I/O signalling on the file descriptor with the F_SETOWN and F_SETSIG operations in "fcntl". With the F_SETSIG operation we will trigger a SIGIO periodically depending on the frequency specified on the attributes and with the F_SETOWN we will specify that the tool get the signals. But, before doing this, we have used the "sigaction" system call to change the action taken by a process on receipt of a SIGIO signal. The action that we will set is the function that we have developed to read the samples from the buffer or to read the hardware event directly on the file descriptor. Furthermore, we initialize other parts of the code that we will comment on the next sections.

Now we have all initialized and ready to analyse the process, so we send a SIGCONT signal to the process of the analysed application that we had stopped on the beginning of the tool execution. The child process of the application was stopped just before of doing an "execv" to the application that we want to analyse. Just when the application does the "execv", the hardware events start counting and we will start to receive signals on our tool to read the values.

The function that is executed periodically to read the events has two modes, read directly from the file descriptor or read from the buffer created with mmap. If we read directly, we use the system call "read" on the file descriptor returned by the perf_event_open call and we obtain the values on a specific format. This format is defined on the attributes when we call to the perf_event_open. In our case the format is always the same and it is the struct of the Figure 8.

```
struct read_format {
      unsigned long nr;       /* The number of events */
      unsigned long time_running;
      struct {
         unsigned long value;/* The value of the event */
      } values[nr];
};
```

Figure 8: Snippet - Read format struct

On the struct "values" that it is inside the struct "read_format", we have the values of the different events grouped which we have initialized previously. Also, we have the total time that the event was running.

On the other case, when we read the values from the buffer created with mmap is more complex. First of all, on the tool initialization we have to create the buffer calling to the mmap system call with the file descriptor and the size. The mmap size should be 1+2^n pages, where the first page is a metadata page that contains various bits of information such as where the ring-buffer head is. The mmap will return a pointer to the buffer.

```
buf = mmap(NULL,mmap_size, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
```

Figure 9: Snippet - Ring-Buffer initialization

Every time that the perf_event interface introduces an event sample on the buffer, the event overflow will occur and the function that reads the sample will be executed. Furthermore, on this buffer there are other kinds of samples that are not only the event counting, for example when a fork event happens on the execution, or a process exit. There are more, but we will use only this three samples. Each sample has a header to differentiate with a type identifier, the size of the sample and additional information about the sample.

The hardware event sample that we will read from the buffer has more information than when we read directly from the file descriptor. This extra information is configurable, in our case we read always the same fields. This extra fields are, the PID and TID of the process, a timestamp obtained via "local_clock", on which CPU is being executed the process, and the current call chain. Finally, we have a field with the same format that we have on the "read mode" with the event counting.

```
struct sample_format {
   struct perf_event_header header;
   unsigned int pid, tid;
   unsigned long time;
   unsigned int cpu, res;
   struct read_format v;
   unsigned long    nr;
   unsigned long    ips[nr];
}
```

Figure 10: Snippet - Sample Format struct

When we read a fork or and exit sample, we read the PID, TID, parent PID and parent TID of the process that has created or exited and the timestamp when it happens. With this information, when a child process is created we can do the same process of initialization and start reading samples from the child. And stop counting when we read a process exit sample.

Finally, when the execution of the analysed application finishes, the tools wakes up from a "waitpid" call and start storing all the information on the file system. All the data from an execution is stored in the same folder. This folder is created at the tool initialization stage and the name is specified on the tool options. For each process of the execution the tool creates a file for the event samples and the monitoring information, another file with the samples of the stack trace with a header of the mapping of the execution and another file with a final report with all samples summed for each event. All the samples (events with monitoring and stacktrace) are stored ordered with the timestamp when the sample was recorded. Also, the final report includes derived metrics from the hardware events as can be miss prediction ratio, IPC, CPI or cache misses ratio.

4.1.1 Call stack

As we have commented, a feature that provides the perf_event_open is that for each event sample that stores on the buffer includes a "callchain" of the specific moment when the event sample is recorded. The "callchain" is composed of multiple 64-bit instruction pointers (addresses) of the active subroutines at a certain point in the time during the execution of the analysed program. So, with these instruction pointers we can do a profiling on the post processing if we know which function it belongs to. The conversion of instruction pointer to function names will be done on the post-processing. To process it when the analysed application has finished, we store on a file each call chain with the timestamp when it has recorded. Moreover, the conversion of instruction pointers to function names of the shared libraries is not directly. So, we store the mapped memory regions of the analysed process by accessing to the /proc/<PID>/maps file which contain this information.

4.1.2 Monitoring

As we are executing a function periodically while the tool is running, we can take advantage of this and read from the Linux virtual file system /proc some values to calculate the CPU load consumption and memory consumption of each process. On the /proc folder we can find a lot of information about hardware details of the system and from any process that is running on the moment when we access[35].

In the case of the calculation of the CPU load, we have used two files to extract the information that we need: /proc/uptime and /proc/<PID>/stat where <PID> is the process ID of the process that we want to monitor. From the /proc/uptime file we need the first value of the file which contains the uptime of the system in seconds. For the /proc/<PID>/stat file we need different values described in the Table 6.

Position	name	Description
14	utime	Amount of time that this process has been scheduled in user mode, measured in clock ticks
15	stime	Amount of time that this process has been scheduled in kernel mode, measured in clock ticks
16	cutime	Amount of time that this process's waited-for children have been scheduled in user mode, measured in clock ticks
17	cstime	Amount of time that this process's waited-for children have been scheduled in kernel mode, measured in clock ticks
22	starttime	The time the process started after system boot.

Table 6: Need values from /proc/<PID>/stat for the CPU load calculation

As the values are in clock tick we need to calculate the system's Hertz (number of ticks per second) using the sysconf function passing as parameter the variable _SC_CLK_TCK. Finally, to get the CPU load, we calculate the total time of the application adding the utime, stime, cutime and cstime values, and dividing the result by the system's Hertz we have the total time in seconds of the application running. Then, with the difference between the uptime and division of the starttime and the system's Hertz we get the seconds since the application starts. Dividing the total time of the application running by the time since the application starts we have the CPU load. In a clear way:

(utime + stime + cutime + cstime) / hertz) / (uptime - (starttime / hertz)

In the case of the memory consumption, we want to calculate the virtual memory size and the real memory used. To obtain this two numbers we have the next two values from /proc/<PID>/stat:

Position	name	Description
23	vsize	Virtual memory size in bytes
24	rss	Resident Set Size: number of pages the process has in real memory. This is just the pages which count toward text, data, or stack space. This does not include pages which have not been demand-loaded in, or which are swapped out.

Table 7: Memory values

For the virtual memory we have the value directly, but for the real memory we need to multiply the rss value for the page size of our system. We can check easily the page size calling the function sysconf with "_SC_PAGESIZE" as a parameter.

4.2 Post-Processing

When the execution of the analysed application has finished, all the data is stored on the file system and it is ready to post-process all of this data to extract the valuable information. We have developed a tool to do the post-processing operations.

One function of the post-processing is to convert all the addresses of the stack-trace reported during all the execution to function names and obtain a profile similar to the provided by tools like Gprof or Oprofile with the most consuming functions of the code. We have used the function of UNIX "addr2line" that translates addresses into function names. We calculate the most consuming functions counting how many samples are on each function. We can observe an example of the profile report on the Figure 11. It is the profiling of a HPL Benchmark running on Minotauro. We can observe how most of the time is spent running a "dgemm" function of the Math Kernel Library from Intel.

Function Name	#samples	%samples
mkl_blas_avx2_dgemm_kernel_0	39308	82.7728526606
mkl_blas_avx2_dgemm_dcopy_down12_ea	1276	2.68693802775
poll_all_fboxes	1218	2.56480448104
MPID_nem_mpich_blocking_recv	713	1.50140032429
HPL_lmul	563	1.18553770347
mkl_blas_avx2_dgemm_dcopy_down4_ea	458	0.964433868896

Figure 11: Profiling Example - HPL

As for each sample of the stack trace we have the timestamp when it was taken, we have implemented an extra feature to generate a profile of specific

parts of the execution. We can determine a specific moment to get the function on this moment or specify two times of the execution and get the profile of this portion of the execution. When we see a part of the execution with a low performance on the hardware event plots (low IPC, high cache misses ratio, etc), with this feature we can detect easily the functions of the code that are generating this downgrade on the performance.

Another function that is done on the post processing part is the plot generation with the hardware event values and the monitoring. In order to parse all the traces of our tool, we have used NumPy arrays to process the trace files. We generate two figures, one focused on the CPU and memory consumption plots of the execution and the other one with the hardware event sampling plots. In the Figure 12 we can see the monitoring report of an MPI execution of the HPCG Benchmark with 16 process on the same node. On the top-left plot, we can see the CPU load of each process over the execution time. On the top-right and bottom-left plot we can see the virtual memory consumption and the Resilient set size memory over the time of each process. Finally, we have the bottom-right plot, where we can observe on which CPU had been running each process during the execution. On the four plots, each line is a MPI process. We can see a similar CPU load on all processes, close to the maximum (1.0). Also, we can observe a similar memory consumption for all processes, close to 4GB ($4x10^9\ bytes$) per process for the RSS and close to 4,5GB ($4{,}5x10^9\ bytes$) per process for the virtual memory.

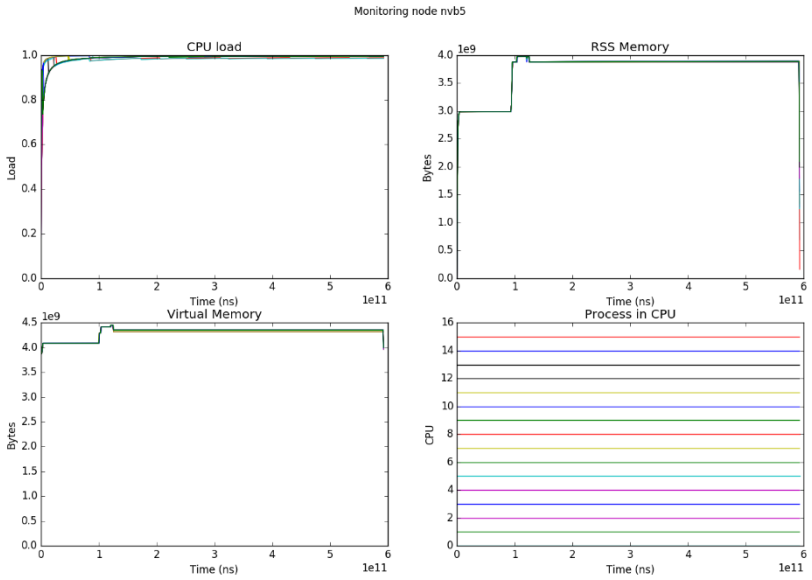

Figure 12: Monitoring report example

If the analysed application is multi-mode, we generate a figure with the monitoring values aggregating the processes values of each node. On the Figure 13 we can see an example of an analysis of an execution of HPCG Benchmark on Minotauro with 16 process per node on 16 nodes. We can observe that the CPU load of each node is close to 16. Also, the memory consumption per node is approximately 60GB, below the 128GB available on the node.

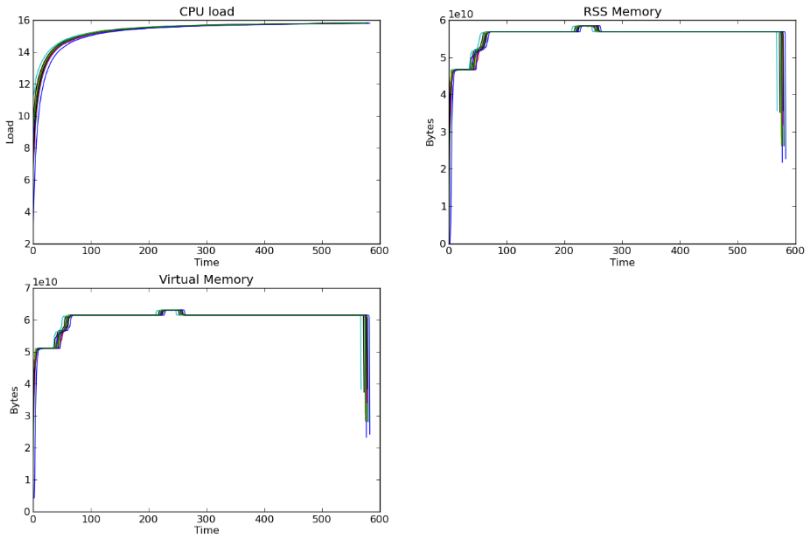

Figure 13: Monitoring report grouped by node example

Finally, we generate a hardware event figure for each process. On the hardware events figure (Figure 14), we can see four plots of the same HPCG execution, but in this case is a report of only one MPI process. The sample frequency was of 1 second. On the top-left plot we can observe the cycles and instruction values of the samples over the execution time. Additionally, we have generated the IPC metric deriving from the other two values. On the top-right plot we can see the last level cache accesses and misses values of each sample over the execution time. On the bottom-left plot we can see the branch instructions and branch miss prediction values of each sample over the execution time. On the bottom-right plot we can see the L1 cache accesses and misses values of each sample over the execution time. For this last three plots we have generated the miss ratio metric deriving from the other values. The scale for each derived metric (IPC and miss ratios) is the scale that we can find on the right axis of each plot. For example, the performance information that we can extract from the Figure 14 is that the each second (1 sample) occurs 2,4 x 10^9 cycles (2,4GHz) and the IPC is usually 1. Moreover, we can observe a strange behaviour when the HPCG has been running for 100 seconds ($1x10^{11}$ ns), probably due to an initialization of the benchmark.

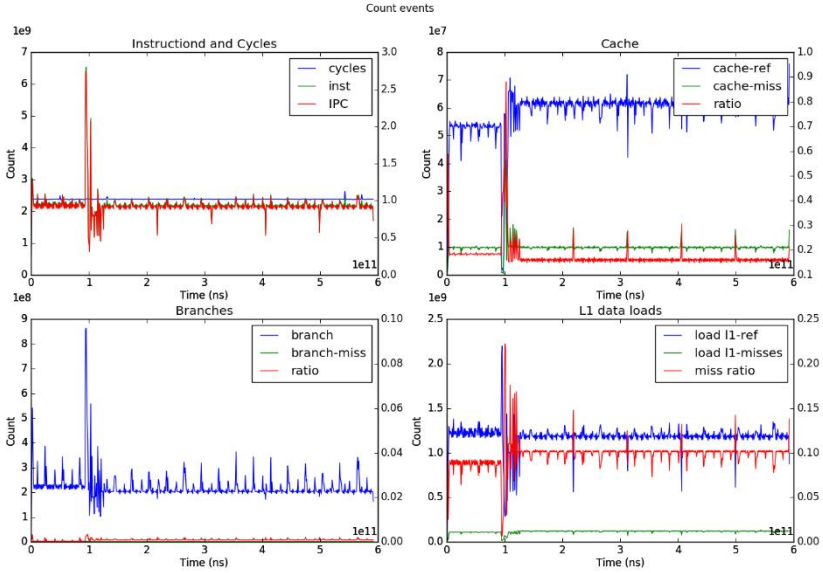

Figure 14: Hardware Events Example

4.3 Analysis Methodology: Case of study

To understand how our performance analysis tool can be used, we have created a very basic example to see how can detect a performance problem in our code. The example is the execution of three functions looping over the same matrix. The first one initializes a matrix row by row with a random number. The second multiplies by two each value of the matrix but now we loop through columns. The last function multiplies by two each value again but looping through rows. Figure 15 shows the three C functions.

```
int a[N][N];

void init_for(){
      int i, j;
      for (j = 0;j<N;++j)
      for(i = 0; i<N; ++i)
            a[j][i] = rand();
}


void wrong_for(){
      int i, j;
      for (i = 0;i<N;++i)
            for(j = 0; j<N; ++j)
                  a[j][i] = a[j][i] *2;

}

void right_for(){
     int i, j;
      for (j = 0;j<N;++j)
            for(i = 0; i<N; ++i)
                  a[j][i] = a[j][i] *2;
}

void main(){
      init_for();
      wrong_for();
      right_for();
}
```

Figure 15: Snippet – Example

We ran the program analysing with our performance tool, specifying a sample frequency of 10 samples per second. We generate the Hardware events plots and we can identify on the Figure 16 different behaviours with different

levels of performance. First we have the initialization of the array, so we can see how the number of branches is higher due to the "rand" function call. Also, we can see that it takes 15 seconds approximately. Then, we have the function that loops by columns. On this part we can see how it takes longer, approximately 25 seconds, and the instructions per cycle are lower. We can find the reason of this performance downgrade on the top-right plot and bottom-right plot. There are a lot of misses on the L1 cache, and in consequence there are a lot of accesses to the last level cache. This is because the method that C uses for storing multidimensional arrays in memory is row-major order, so, consecutive elements of a row reside next to each other. So, this loop does not exploit the spatial locality. Finally, we can see the third part, which it takes 4 seconds approximately. It is doing the same work than the previous function, but in this case we can see how the IPC is higher and that there are a lot of hits on the L1 cache and less accesses to the last level cache because this loop is exploiting the spatial locality.

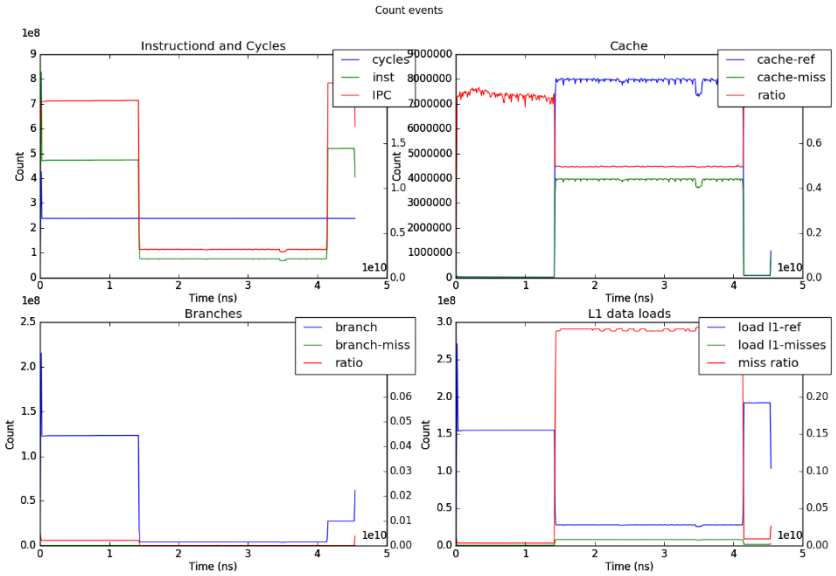

Figure 16: Hardware events - Example Code

Additionally, we can observe on the Table 8 the profile generated on the post processing to see the most consuming parts. Additionally, we have timing functions to the code to compare results. We can see how the "wrong_for" is the most consuming part and how much accurate is our profile.

Function	Samples	% Samples	Timing (s)	% Time
init_for	23	5.02%	14.14	31.40%
__random	96	20.96%		
wrong_for	271	59.17%	26.97	59.90%
right_for	40	8.733%	3.91	8.69%

Table 8: Profiling - Example Code

Additionally, to ensure us which function is being executed on each behaviour of the plot, we can use the partial profiling feature of the post processing. For example, if we specify from the second 20 to the second 40 of the execution, we can see how the 100% of the time is on the "wrong_for" function.

Finally, we can see the monitoring plot (Figure 17), but in this case it does not report us any relevant information. We can see that there is one process on the CPU 1 that is consuming normally the maximum CPU load. On the memory plots we can see that it is always consuming 4GB of virtual memory (the array is a square matrix with 32768 rows and columns). On the Resident Set Size plot we can see how the memory consumption grows progressively when we initialize the array.

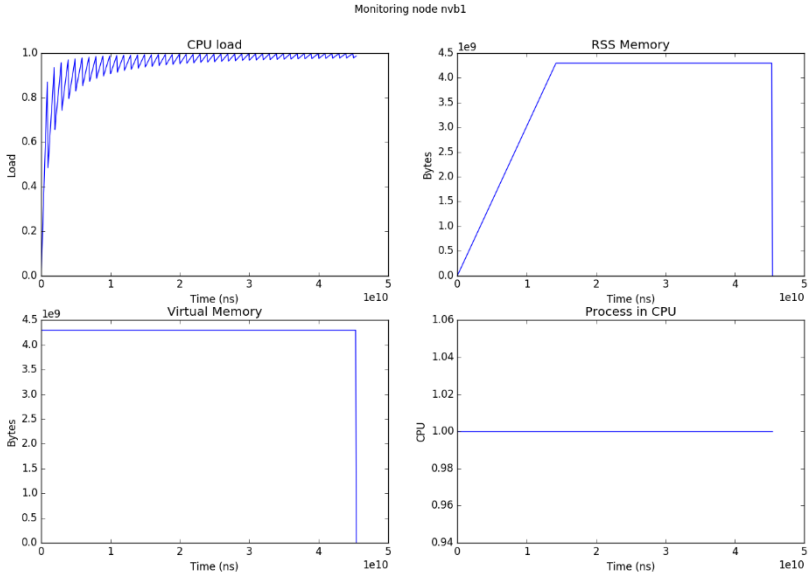

Figure 17: Monitoring - Example Code

5 Benchmarking Analysis: Architectures

The approach of this section is to analyse different computer architectures running the HPL, HPCG and Stream benchmark and analysing the executions with the performance analysis tool developed during the project.

The software used for build the benchmarks, along with their versions, can be seen in Table 9. On the next sections we have specified which compilers we have used to build each benchmark on each architecture.

Software	Version
GNU C/Fortran compilers	6.2.0
Intel C/Fortran compilers	2017.1
Intel MPI	2017.1
Open MPI	2.0.1
MKL	2017.1
ESSL	5.3.2

Table 9: Used software and their versions

5.1 Intel General Purpose – Haswell

First of all, we started running the benchmarks on a general purpose architecture like Haswell from Intel. As we have commented, to analyse this architecture we have used the Minotauro cluster from BSC. We will use the performance values of Minotauro as a reference to compare them with the next architectures.

To build the HPL Benchmark, we have used a Makefile example provided on the benchmark tarball that is thought to use it with Intel Compilers and Math Kernel Library (MKL) from Intel. We ran the benchmark on a single node with a pure MPI execution. On the input file we have specified the problem size with a value of 58368 and the block size with a value of 256. In the case of the problem distribution we have chosen a PxQ of 2x8. This input values are prepared to use

32GB of memory per node and 16 MPI process. In order to find this values, we have done executions with different combination of options.

The HPL benchmark performance result on a single node of Minotauro is 463.2 GFLOPS. We can see on the Figure 18 that it uses 2GB of memory per process and that the processes are always on the same CPU.

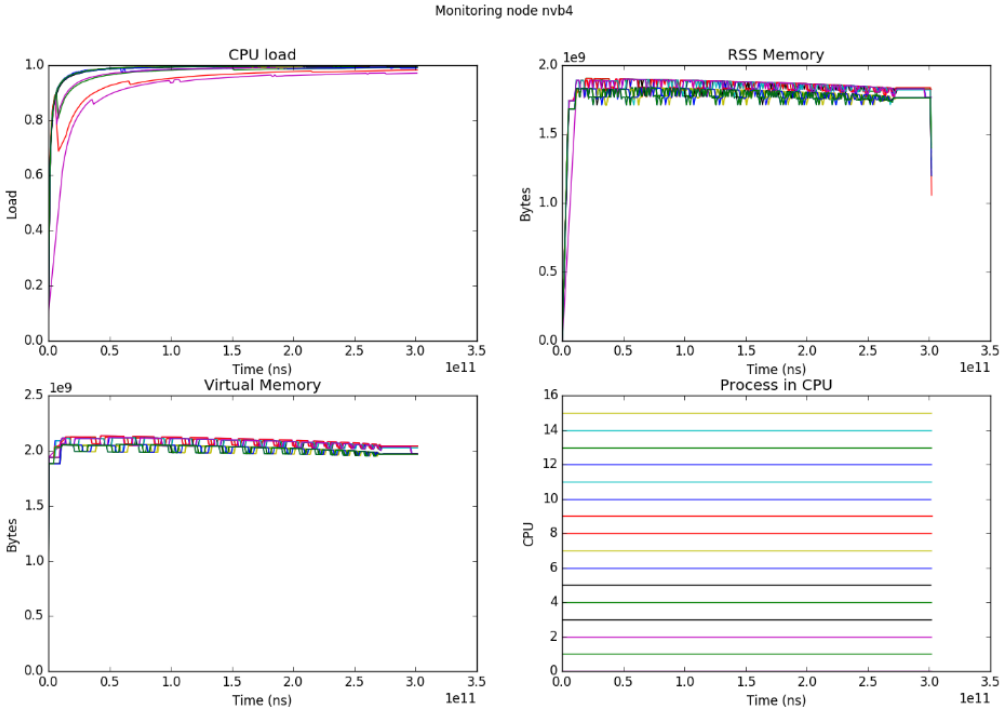

Figure 18: Monitoring HPL Minotauro

Furthermore, we can see on the Figure 19 the events sampling plots with a frequency of 1 sample per second of one MPI process of the HPL execution. If we observe the top-right plot we see that the IPC value is usually 3. Also, if we observe the cycles we detect that the CPU frequency is constant on the whole execution. Observing the cache plots we can see each second there are $4x10^7$ accesses to the last level cache and $3x10^9$ accesses to the first level cache. The observed high IPC and the high number of accesses per second is due to that the HPL Benchmark gives a good correction of peak performance of a system.

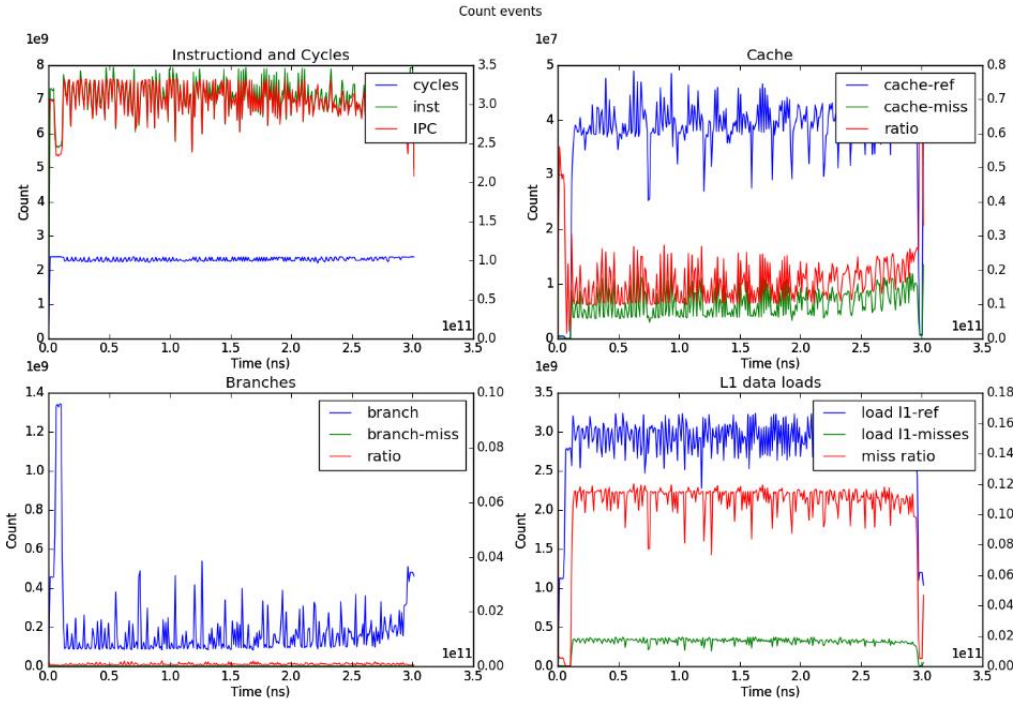

Figure 19: Hardware Events HPL Minotauro

In the case of the HPCG Benchmark, we have used again a Makefile provided on the tarball of the benchmark to build it. As on the HPL benchmark, this Makefile is thought to use it with Intel Compilers. We run the benchmark on a single node with a pure MPI execution and a sample frequency of 1 samples per second. In this case the problem size is 160 and the running time is 300 seconds. The HPCG performance result on a single node of Minotauro is 8.18 GLFOPs and we can see the monitoring plot (Figure 20) and the event sampling plot (Figure 21) of one MPI process. We can not see any strange behaviour, the processes are pinned to the cores, consuming 3,5GB of memory per process and with maximum CPU load. On the hardware events figure we can see a very uniform behaviour of the execution. But if we compare the performance results with the HPL, we can see now that the IPC value is usually 0,5. Moreover, the number of accesses to the last level cache per second is similar, but now the miss ratio goes from 0,2 to 0,5 approximately. This is because HPCG has irregular accesses to memory to simulate access patterns from real HPC workloads.

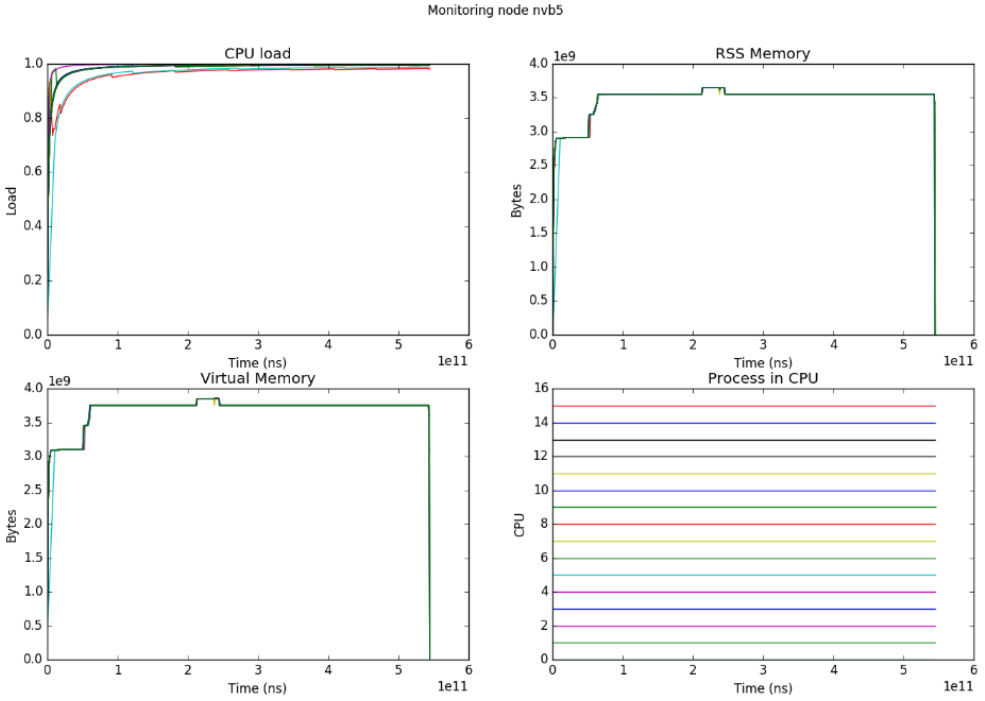

Figure 20: Monitoring HPCG on Minotauro

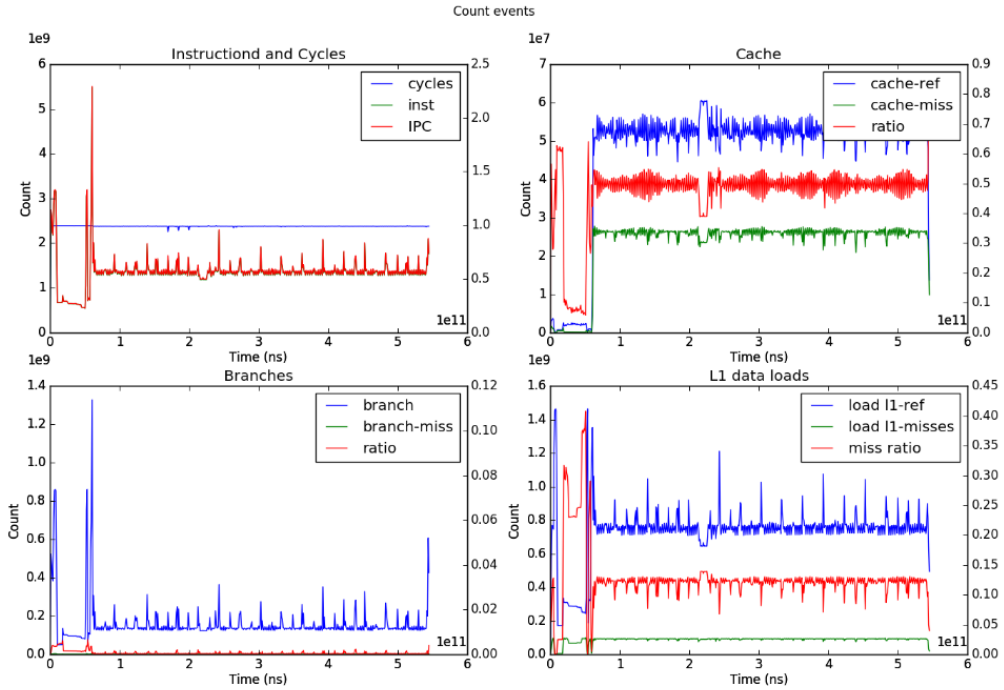

Figure 21: Hardware Events HPCG on Minotauro

However, for the HPCG, we have compiled an Intel optimized version that reports a performance of 12.9062 GFLOPs with the same input parameters. On the Figure 22 we can see how the Instructions per cycle are much better than on the normal version. Also, we can see a better usage of the cache hierarchy. We can observe on the L1 data loads plot almost the twice as of accesses than in the normal version and a similar number of access to the last level cache, but with a lower miss-ratio.

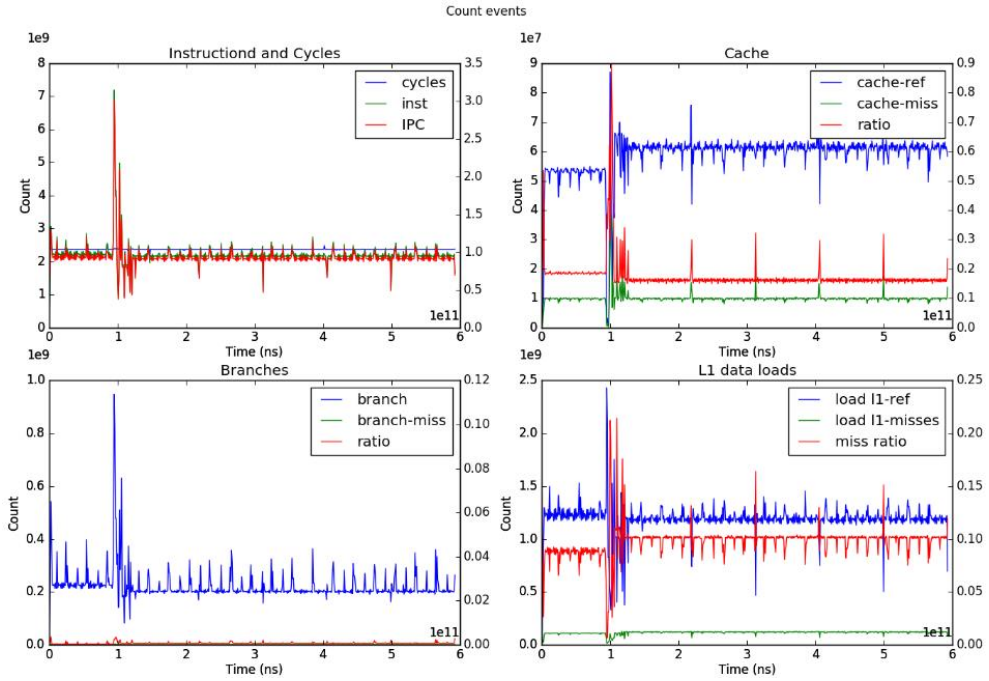

Figure 22: Hardware Events HPCG Optimized by Intel on Minotauro

For the stream benchmark, we have compiled the code directly with the Intel Compilers, and set the array size of the problem to 10^9 elements. In total, we will require 22,4 GB of memory for the 16 MPI process (1.4GB per process) on one node. In this case, due to the execution time of this benchmark is much sorter we have used a sample frequency of 20 samples per second to have better precision. Once we have executed the benchmark and we can see the results on the Table 10. Each kernel will be executed 10 times.

Function	Best Rate MB/s	Avg time (s)	Min time (s)	Max time (s)
Copy:	82325.7	0.196901	0.194350	0.216972
Scale:	65512.3	0.248414	0.244229	0.270567
Add:	73359.3	0.327479	0.327157	0.328164
Triad:	73262.0	0.328710	0.327591	0.335341

Table 10: Stream Benchmark Results on Minotauro

On the Figure 23 we can observe clearly 10 similar shapes that corresponds to the 10 executions that the benchmark of the kernels have done. If we focus on one of these shapes, we can observe how there are different L1 cache access patterns that corresponds to the different kernels.

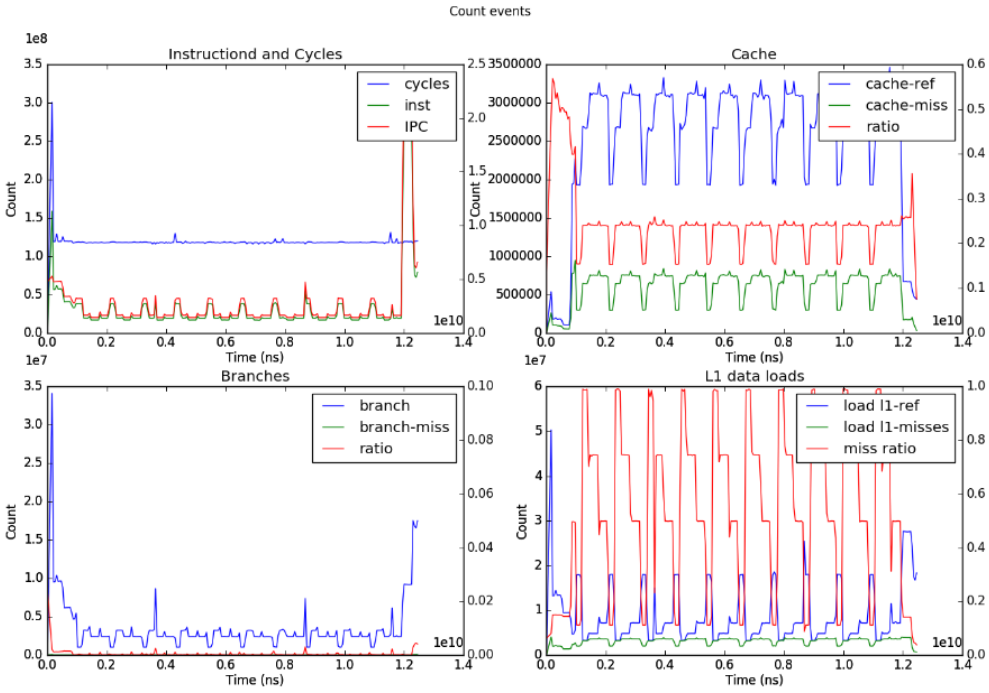

Figure 23: Hardware Events Stream on Minotauro

5.2 Intel Many Core - Knights Landing

We continue running the benchmarks on an Intel Many Core architecture like Knight Landing. As we have commented, to use this architecture we have used the CTE-KNL cluster from BSC. One limitation of the KNL architecture is that we can only record three hardware events at the same time. As our tool is prepared to read any number of events, we ran multiple executions of the benchmarks in

order to read all the events that we want to plot. The plotting function is also prepared to merge different event traces of different executions. Moreover, on the KNL the event that counts the L1 data loads accesses is not available.

As we have done for the Haswell architecture, we have used a Makefile example provided by the benchmark tarball to build the HPL with Intel Compilers and Math Kernel Library (MKL) from Intel. We ran the benchmark on a single node with a pure MPI execution. On the input file we have specified the problem size with a value of 58368 and the block size with a value of 256. In the case of the problem distribution we have chosen a PxQ of 4x16. This input values are prepared to use 32GB of memory per node and 64 MPI process. We have used a sample frequency of 1 sample per second.

The HPL benchmark performance result on a single node of CTE-KNL is 1752,44 GFLOPS and we can see the monitoring plot on the Figure 24. We can see that it uses 2GB of memory per process and that the processes are always on the same CPU. Furthermore, we can see on the Figure 25 the event sampling on one MPI process of the execution.

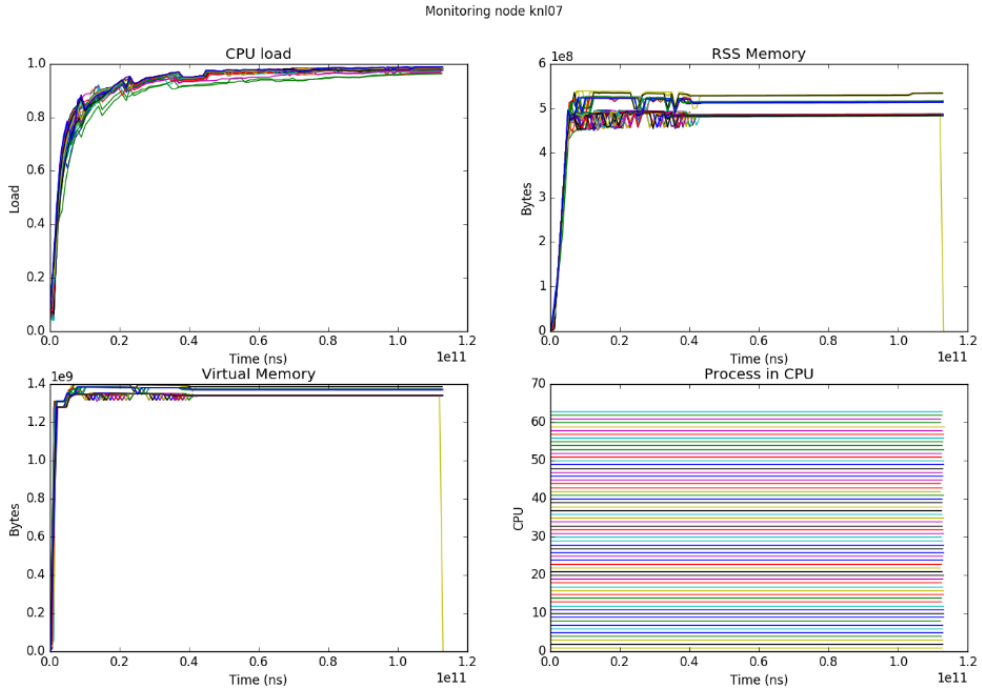

Figure 24: Monitoring HPL on KNL

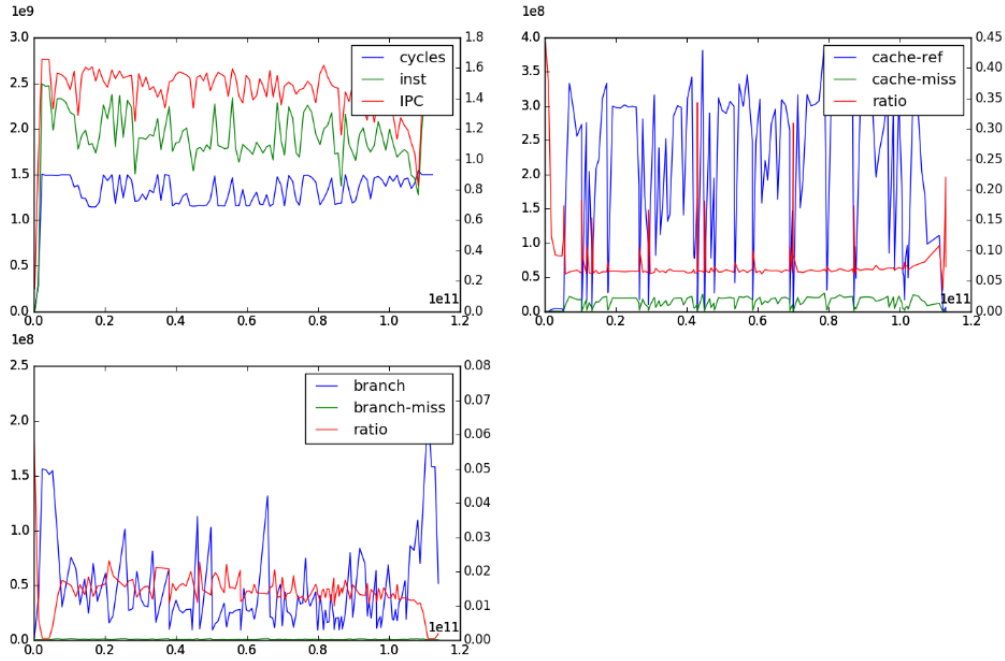

Figure 25: Hardware Events HPL on KNL

On the plot of the Instructions and cycles of the Figure 25 we can see a variability on the cycles, so we are observing that the CPU frequency is not constant. This is because the CPU clock is different when it is executing AVX instructions. It can drop from 1,3GHz (or 1,5GHz with turboboost) to 1,1~1,2GHz[36]. Moreover, we can observe an increment of branch prediction misses if we compare with the Haswell architecture. This may be because it has a worse predictor and/or the GLIBC library version that we have on the system, has a bug on the Silvermont architecture that usually misses the prediction when the jump is large. This bug is fixed on the 2.23 version of GLIBC [37]. Normally, the larger branches happens when a function of the shared libraries are used, for example the MKL functions.

For the HPCG, we have used the Intel optimized version directly because the result of executing the normal version was too low for this architecture (18,43 GFLOPs). On this case, the best configuration to run the HPCG on KNL is to run 2 MPI process with 32 OpenMP threads for each process. The inputs used to run HPCG on KNL are a problem size of 192 and 180 seconds running. With this configuration, each MPI process will consume 7GB of memory, so in total 14GB,

less than the 16GB of the MCDRAM. We choose this configuration in order to use only the MCDRAM as memory in flat mode. We have compiled with Intel Compilers too. The performance is of 45.58 GFLOPS. On the monitoring figure (Figure 26) we can see how the CPU load has two different behaviours. The first part is all the initialization, where the CPU load goes down. The second part is the benchmark execution, where we can see how the CPU load grows, but does not achieve the maximum load (32). Also, we can see how this second parts takes 180 seconds. On the Figure 27 we can see the same two parts of the execution. We can observe how it takes approximately 200 seconds to initialize the problem. If we compare the execution of the optimized HPCG on Haswell and KNL we can observe that on KNL we have more accesses to the last level cache but with a very low miss ratio. This low miss ratio is due to that the MCDRAM is configured as cache and becomes the last level cache and it has 16 GB. Moreover, despite of the execution on KNL has less frequency and less IPC per core compared with the execution on Haswell, it has more accesses to the last level cache per core. With this performance information we can affirm that the good performance HPCG results on KNL is due to the High-Bandwidth Memory.

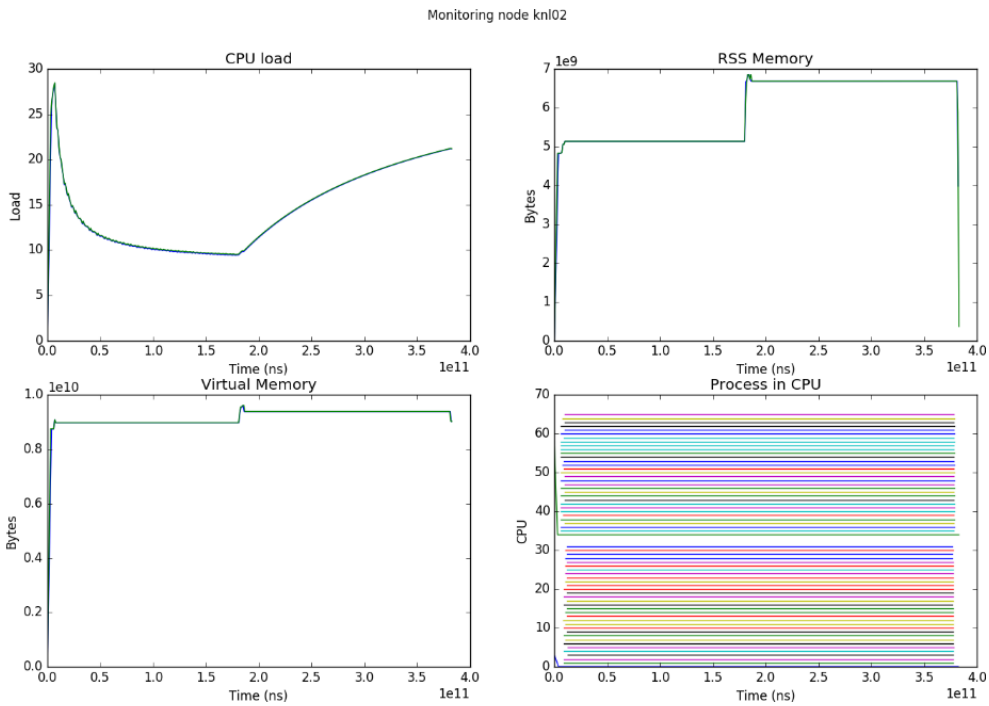

Figure 26: Monitoring HPCG Optimized by Intel on KNL

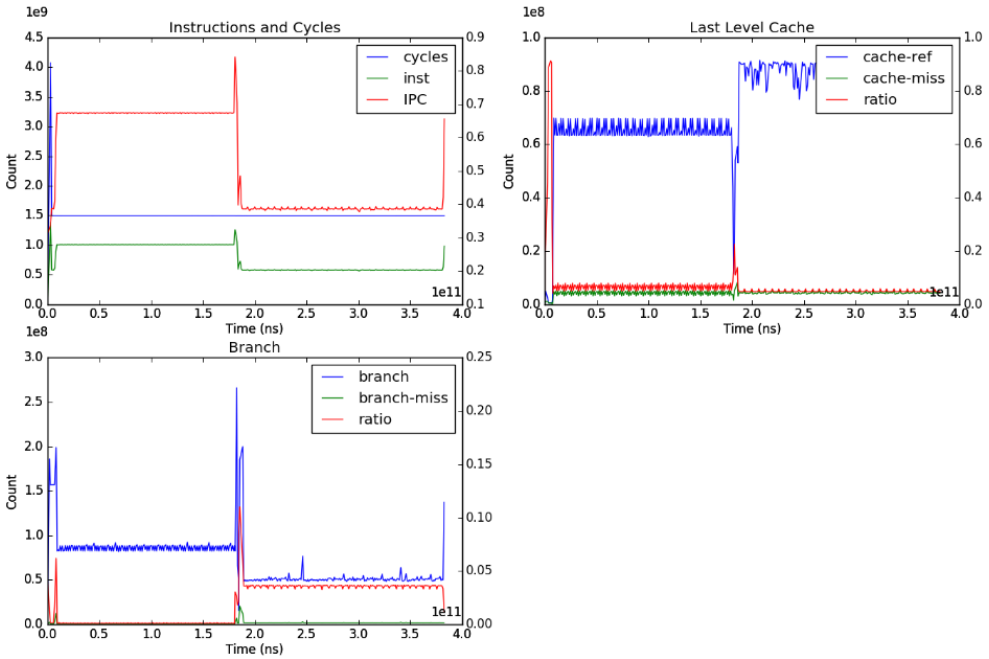

Figure 27: Hardware Events HPCG Optimized by Intel on KNL

For the stream execution we have done two different tests, the first one running on the MCDRAM of the KNL and the second one running on the DDR4 memory. We have used a sample frequency of 100 samples per second. To run it on the MCDRAM we have executed the benchmark with the command "numactl -p 1". The results for the MCDRAM can be found on the Table 11. Also, we have the hardware events and monitoring figures (Figure 28 and Figure 29). On the CPU plots of the Figure 28 we can see that due to the Hyper-threading the tool detects up to 256 cores but we are using only the 64 physical cores without exploiting the hyper-threading to be coherent with the tests done previously. Moreover, we can observe that on the initialization of the benchmark and on the final part there are a lot of CPU migrations. But during the execution of the kernels the processes are usually using the same CPU.

For the stream benchmark, we have compiled the code with the Intel Compilers, and we have set the array size of the problem to $6x10^8$ elements. In total, we will require 13,4 GB of memory for the 64 MPI process (214.6 MiB per process) on one node. Also, each kernel will be executed 10 times again.

Function	Best Rate MB/s	Avg time(s)	Min time(s)	Max time(s)
Copy:	314258.5	0.030611	0.030548	0.030714
Scale:	326839.5	0.029497	0.029372	0.030030
Add:	343363.5	0.042046	0.041938	0.042456
Triad:	344703.9	0.041846	0.041775	0.042011

Table 11: Stream results on KNL – MCDRAM

As it happened with the HPCG performance, we can observe on the Table 11 a big difference between the Stream performance on Haswell and KNL. If we focus on the last level cache plot, we can observe that on KNL usually there are $8x10^7$ accesses per second and on the Haswell $6x10^7$ accesses per second. The main difference is that these values are per core, and we have 64 cores on KNL and 16 on Haswell.

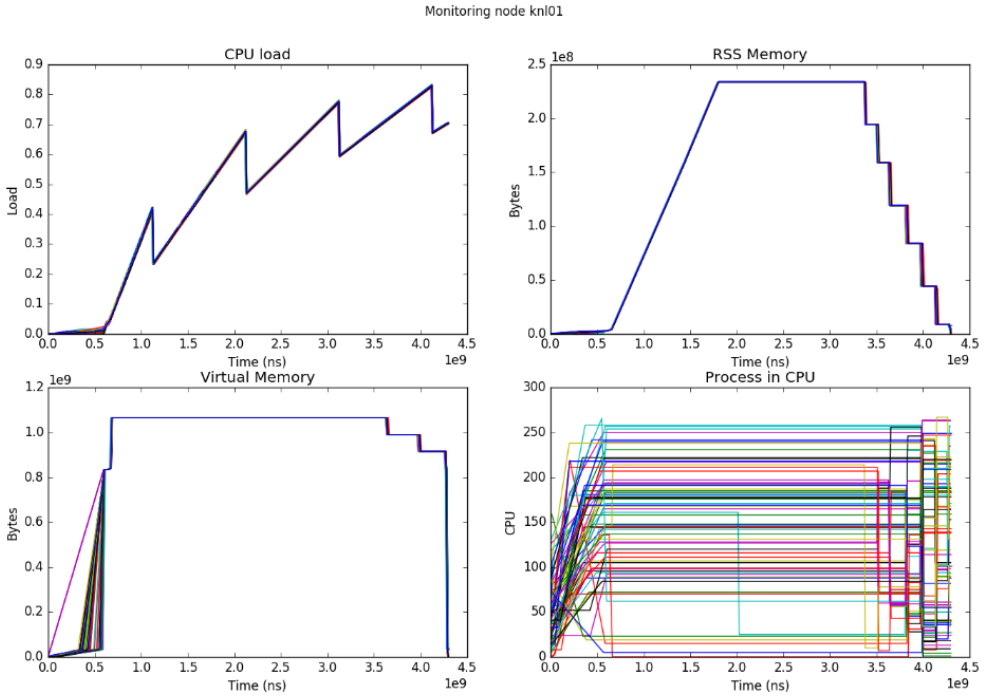

Figure 28: Monitoring Stream on KNL – MCDRAM

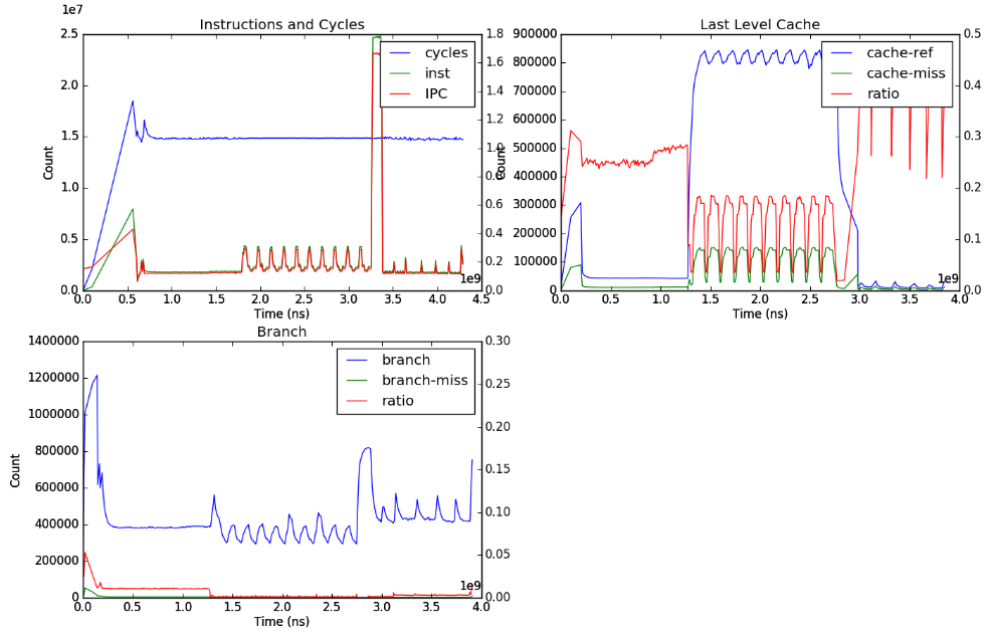

Figure 29: Hardware Events Stream on KNL – MCDRAM

The results of the DDR4 of the Table 12 compared with the MCDRAM are really bad. We can see that when we execute the benchmark on the MCDRAM (Figure 29) we have 5 more times of cache accesses per second than when we execute the benchmark on the DDR4 (Figure 30). If we observe the bandwidth reported by the benchmark on both execution, we can see that it is 5 times faster running on the MCDRAM than on the DDR4. As we can expect due to the type of benchmark, the performance of this benchmark is totally influenced by the memory used.

Function	Best Rate MB/s	Avg time(s)	Min time(s)	Max time(s)
Copy:	57298.1	0.167743	0.167545	0.167868
Scale:	57579.1	0.167011	0.166727	0.167155
Add:	65279.5	0.220841	0.220590	0.220976
Triad:	65059.2	0.221480	0.221337	0.221834

Table 12: Stream results on KNL - DDR4

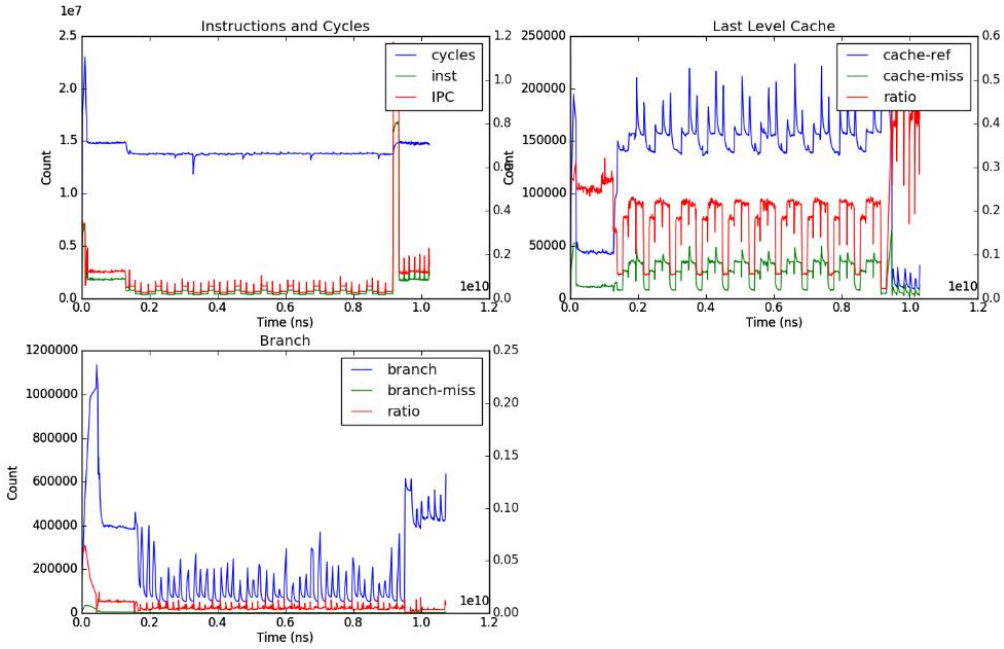

Figure 30: Hardware Events Stream on KNL - DDR4

5.3 IBM - Power8

Finally, we have run the benchmarks on a Power8 architecture from IBM. As we have commented, to use this architecture we have used bscpower8 cluster from BSC. As it happens on KNL, the Power8 architecture has a limitation of four hardware events at the same time. So, we have do the same that we have done for KNL previously running different execution of the benchmarks to generate the plots. Additionally, we have used a sample frequency of 100 samples per second in all executions.

To build the HPL Benchmark for Power8 we have used GCC, OpenMPI and the ESSL libraries from IBM. We have built our own Makefile to use these compilers and libraries. Also, we have added some compiler flags to optimize the code to Power8 architectures. We ran the benchmark on a single node with a pure MPI execution. On the input file we have specified the problem size with a value of 16384 and the block size with a value of 128. In the case of the problem distribution we have chosen a PxQ of 4x6. The HPL benchmark performance on a single node of bscpower8 is 518,41 GFLOPS. We can see on the Figure 31 the event sampling on one MPI process of the execution.

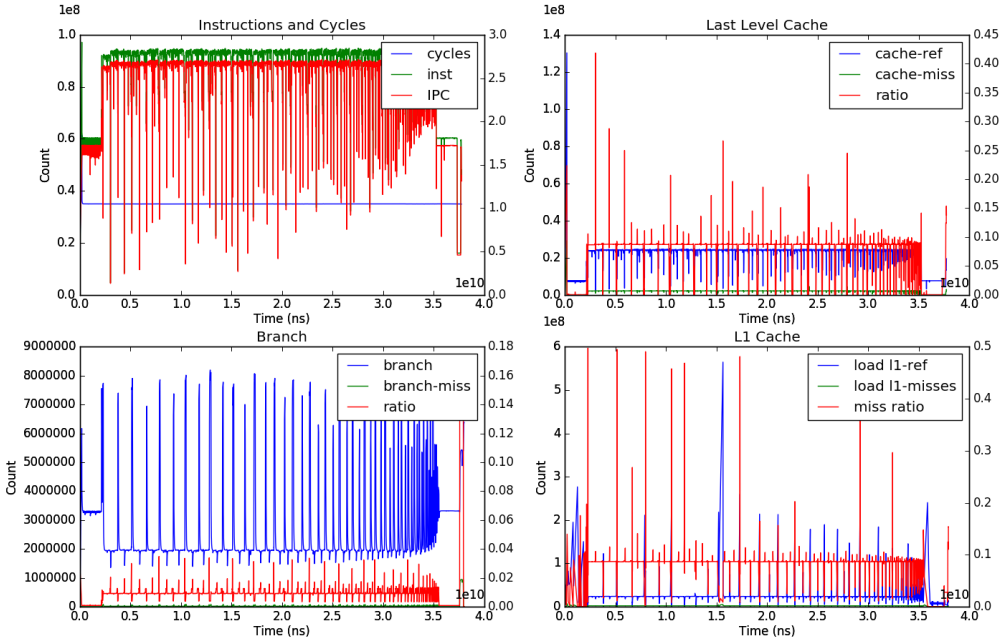

Figure 31: Hardware Events HPL on Power8

On the top-left plot of the Figure 31 we can observe that usually it has 2,6 instructions per cycle (IPC), a bit less than the 3 IPC of Haswell. But, the Power8 has a higher clock frequency. In the case of last level cache events, we can observe a lower miss ratio. However, on the L1 cache plot, we can observe similar number of accesses per second (between $2x10^9$ and $3x10^9$) and similar miss ratio (0,10).

Moreover, we can observe that the hardware events counting of the samples are more variable for the Power8 architecture. Despite of this variability on some samples, we can get a general overview of the execution, for example, on the Figure 32 we have the hardware events plots of three execution of the HPL Benchmark with three different block sizes. The HPL benchmark has the option of execute the benchmark with different input options and combining each of them on the same execution. In this case, it is only three different block sizes (4, 16 and 128). We can differ easily on the four plots the three tests with different performance. We can observe that when it is using the block size of 128 can execute more instructions per cycle, has a lower miss branch predictor and cache miss ratio and can do more cache accesses per second.

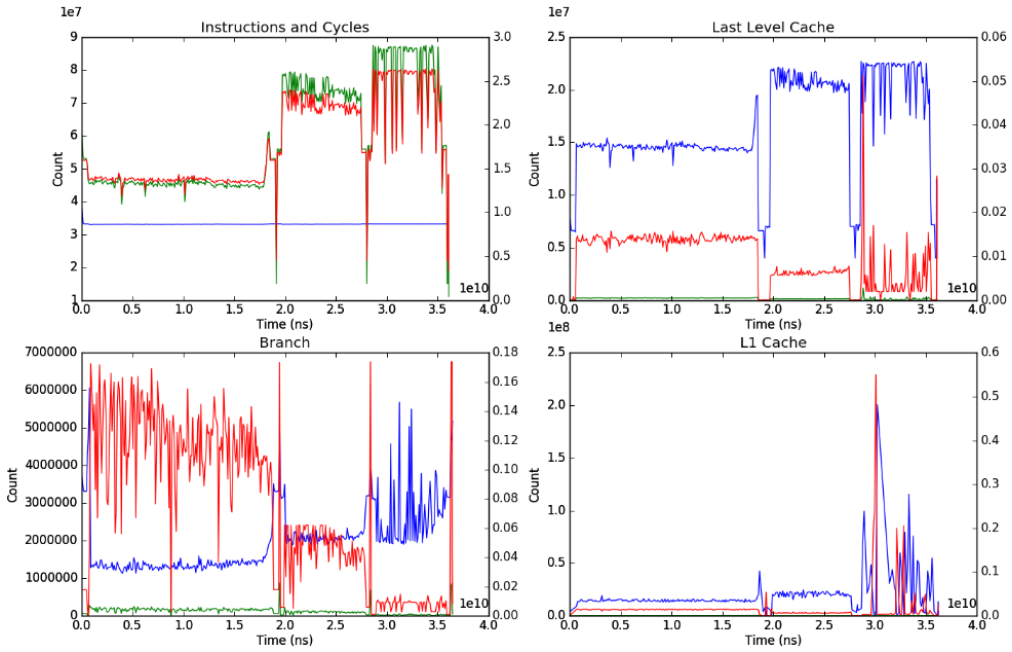

Figure 32: HPL Power8 with different block sizes

For the HPCG, we have built the benchmark with GCC and OpenMPI. On this case, we have used 16 MPI and a problem size of 192. The HPCG performance with this configuration is of 11.58 GFLOPS. As we have seen on the HPL execution, the IPC is similar with the Haswell execution of HPCG (without optimization), but with higher clock frequency. On the L1 events plot we can observe that it is doing $1x10^9$ accesses to the L1 for second, a bit more than the $8x10^8$ accesses per second on KNL. But, the main difference is the low miss ratio that we can see on the L1 cache (0.04 approximately), compared with the L1 miss ratio of the execution on Haswell (0.12 approximately), but as we saw on the HPL execution, with the same number of accesses per second. It can be a caused by having a L1 cache with the twice as much capacity than Haswell, 64kB and 32kB respectively. Due to these differences, the performance of the HPCG on Power8 has a speedup of 1.41 over the Haswell performance with the same number of MPI processes.

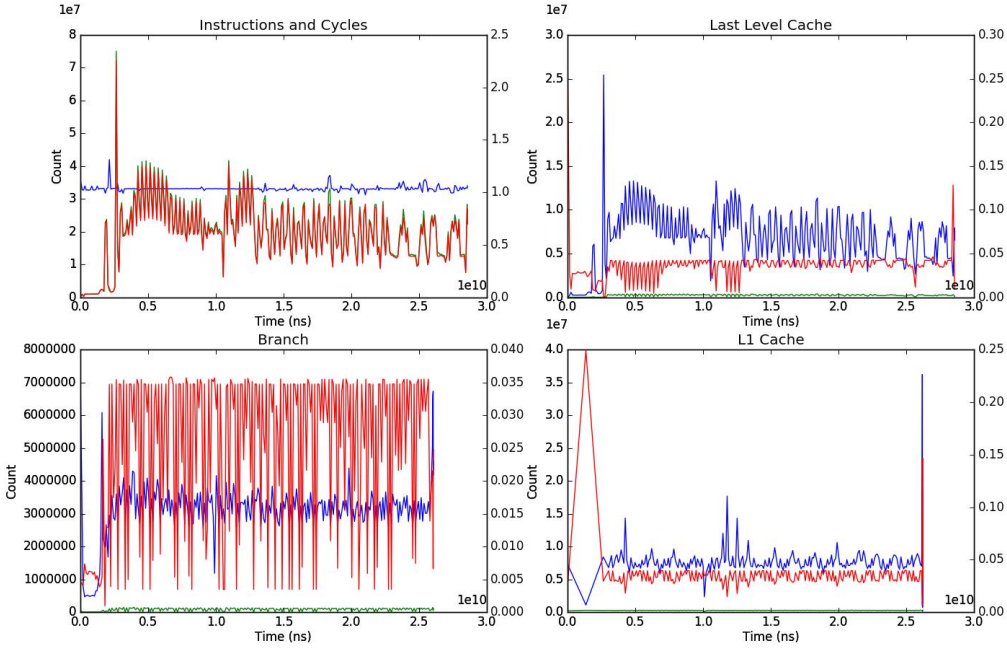

Figure 33: Hardware Events HPCG on Power8

For the stream benchmark, we have compiled the code directly with the GCC and OpenMPI, and setting the array size of the problem to $6x10^8$ elements. In total, we will require 13,4 GB of memory for the 12 MPI process (1.1 GB) on one node. Each kernel will be executed 10 times. We have executed the benchmark and we observe on the Table 13 that the Stream Performance on Power8 are very good if we compare them with the Haswell performance.

Function	Best Rate MB/s	Avg time(s)	Min time(s)	Max time(s)
Copy:	244678.8	0.039564	0.039235	0.040211
Scale:	246141.6	0.042610	0.039002	0.046685
Add:	252025.8	0.058956	0.057137	0.060591
Triad:	254133.9	0.057367	0.056663	0.062122

Table 13: Stream Results on Power8

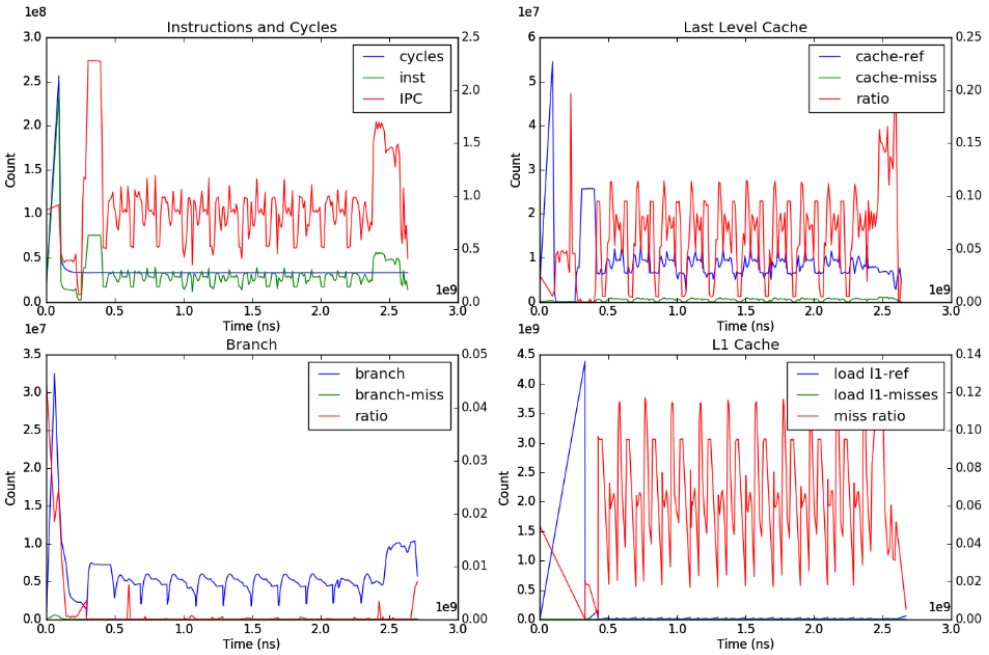

Figure 34: Hardware Events Stream on Power8

Normally, the results of the Stream Benchmark and the HPCG maintain a relation between them, but in the case of the Power8 architecture we can see how the Stream Benchmark has an expected performance, but for the HPCG it is close to the performance of the Haswell architecture. This difference can be caused because the HPCG is more dependent of having an optimized code for the architecture than the Stream Benchmark. For example, Stream Benchmark is much simpler than HPCG Benchmark. To sum up, with our tests we have observed that the Power8 architecture provides powerful cores with a memory hierarchy that provides high bandwidth and good performance on memory oriented workloads. On Power8, the L1 and L2 cache sizes are 64kB and 512kB per core, exactly twice as large as Haswell caches. But, in the case of the L3, Haswell has a 20MB cache for all the chip and Power8 has a L3 cache with 8MB for each core of the chip, in the case of the model that we have used 48MB per chip. Additionally, Power8 has a L4 cache with 16MB per DIMM, and the node that we have used has 16 DIMMs.

6 Validation of the proposal

This section is intended to evaluate the performance impact that our tool produces over the execution of the analysed application and to compare the results that we report with other performance tools to validate the results of our solution.

6.1 Overhead application

In order to measure the overhead of our tool, we have run a HPL Benchmark while sampling it with our performance analysis tool. Then we have compared the benchmark performance with an execution without analysing it with our tool. We have done the same executions with different number of samples per second and the maximum number of processor per node and then we have compared the performance of the HPL with the performance of running it without our tool by analysing the execution. On the Table 14 we can observe the average performance of executing 10 times for each tests and on the Figure 35 we can observe the overhead produced with different levels of sample frequency. As we can observe, our solution is low-overhead and can be lost in the noise (1-3%).

Level of Analysis (samples per second)	**Performance (GFLOPs)**	**Overhead (%)**
Without tool	451.1	-
1	448.3	0,63
10	448.2	0.66
100	446.4	1,06
1000	439.3	2,68

Table 14: Overhead perf. Tool

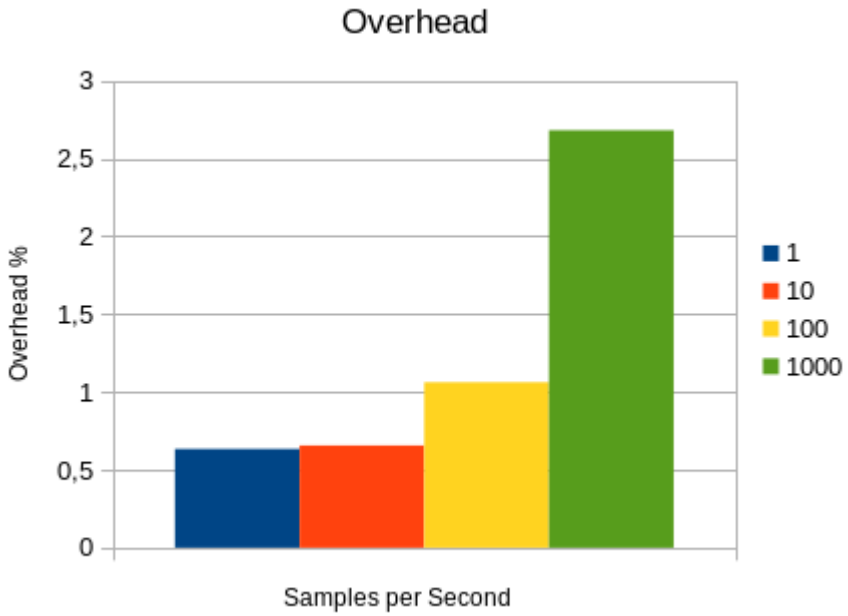

Figure 35: Overhead perf. Tool

The Table 15 illustrates the overhead introduced by the performance analysis tools on a HPL Benchmark performance result on Minotauro. For each test we have done 10 executions and we have calculated the average and the standard deviation. We can observe that the overhead of all tools is inappreciable and can be lost in the noise of the system. Additionally, we can observe that the tool that introduces less overhead is "perf". The overhead difference with our solution is due to our solution sample more data in order to have information of each point of the execution.

	Performance (GFLOPs)	**Standard Deviation**	**Overhead (%)**
Without Analysis	451,1	6,59	-
Our Solution	448,2	8,9	0,64
Perf	449,2	6,56	0,42
Gprof	447,7	8,2	0,75
Oprofile	447,8	7,88	0,73

Table 15: Overhead Comparison with other tools

6.2 Result differences with other tools

To validate the results that our performance analysis reports we have compared them with the results of other tools. In the case of the hardware events counting we have compared the results with the perf tool to ensure us that we are using correctly the perf_event interface. Also, we have compared them with the Oprofile results. On the Table 16 we can observe the hardware events counting for each tool of a HPCG execution on Minotauro. As we can see, the results of the three different tools are very similar.

	Our tool	**perf**	**Oprofile**
cycles	678.088.822.214	695.124.326.419	698.745.504.595
instructions	2.092.577.299.017	2.148.622.593.192	2.161.905.149.375
cache-ref	10.994.852.062	10.675.153.799	11.016.063.965
cache-miss	1.633.766.532	1.545.402.980	1.606.328.199
branch-ref	49.745.953.913	49.454.037.247	53.193.852.421
branch-miss	45.802.575	44.920.571	58.625.481
L1 loads	842.024.996.230	866.042.918.423	858.264.868.688
L1 loads misses	92.760.952.682	94.609.364.029	95.143.365.865

Table 16: Comparison performance tools

Also, we have compared the result of the profile that our tool had reported of a HPCG execution with 16 MPI process on a single node of Minotauro with the profile that reports Gprof. We have analysed it with our tool with a sample frequency of 10 samples per second. The results of our tool is an aggregation of the profile of the 16 MPI process.

Function Name	**Our tool - #samples**	**Our tool - %**	**Gprof - Seconds**	**Gprof - %**
ComputeSYMGS_ref	35211	55.09	221.87	58.98
ComputeSPMV_ref	17124	26.79	107.56	28.59
SetupHalo_ref	5283	8.26	32.52	8.65
MPID_nem_mpich_blocking_recv	1287	2.01	--	--
ComputeWAXPBY_ref	955	1.49	6.35	1.69
poll_active_fboxes	849	1.32	--	--
poll_all_fboxes	701	1.09	--	--
ComputeDotProduct_ref	426	0.66	2.59	0.69

Table 17: Comparison profiling

We can see that the percentages of the most consuming functions of the HPCG code of both tools are similar. Also, if we want to compare the seconds of Gprof with our tool, we can approximate the time consumption of each function with our tool multiplying the number of samples by the period of the samples, in this case 100ms. Finally, we should divide this number by the number of MPI process, in this case 16. For example, for the "SetupHalo_ref" function doing 5283 samples * 0.1 seconds / 16 process = 33.01 seconds. Also in this case the results are similar on both tools.

Finally, we can see how our profile includes on the profile functions of shared libraries of the code that did not appear on the Gprof profile. This is because to get a gprof profile, we should compile the code with a profiler flag ("-p" or "-pg"), and normally this shared libraries are not compiled with this flag.

7 Conclusions

In this work, we have analysed the main computer architectures of the near future BSC clusters. The analysis has helped us to gain a deep knowledge and experience on this architectures by understanding how we can exploit the strengths of them and, but not less important, we have discovered some problems and weakness that we should avoid.

The proposed performance analysis tool can be easily used and allows to analyse any application on any modern computer. The current proposal performs hardware event samples and also helps to debug and develop HPC codes thanks to the profiling and monitoring features. The presented performance results have helped us to understand the several applications and architecture performances. All of this, and due to the usefulness of the analysis done thanks to our proposed tool, the Operations Team of the BSC has decided to provide it to the final users. Indeed, this tool is flexible and has been developed with the objective of continuous evolution depending of future needs.

We have evaluated the performance impact of our performance analysis tool over the analysed application and the results show a competitive overhead with the direct use state-of-the-art solution. Additionally, we have compared our results with other tools and we have obtained similar results providing additional information of any point of the execution with a more usable solution for the final user.

8 Future work

This project has been the beginning of a bigger task developed on the Support Team of BSC. It is a long term project, and we have defined the next steps to continue with the development and research.

Data intensive supercomputer applications are increasingly important for HPC workloads. Current benchmarks and performance metrics do not provide useful information on the suitability of supercomputing systems for data intensive applications. For example, one of the strength of the Power8 according to IBM is the performance on data intensive workloads or the Knight Mill oriented to Machine Learning workloads. It would be interesting to add to the benchmark suite data intensive benchmark, like Graph500 Benchmark.

Furthermore, we have used machines that have NVIDIA accelerator like Minotauro with K80 GPUS and the Power8 machines. NVIDIA provides "nvprof", a command-line profiler and tracer for NVIDIA CUDA applications that provides a summary of GPU and CPU activity, trace of GPU and CPU activity and event collection. So, one of the next steps could be analyse the possibility to adapt our tool to provide an easy way to use nvprof and add to our report the data reported by nvprof.

Finally, to improve the usability on the analysis stage of our performance tool, it could be interesting to adapt our traces to the Paraver format. Using Paraver as a GUI for our performance tool will allow users to do more functions with the traces exploiting the interactivity of having a GUI. It would be helpful for users to have a final summary report on PDF or HTML to get a more readable format of the performance results. Additionally, we will continue working on the performance analysis tool to reduce the overhead on the execution and optimizing the post processing part.

9 References

[1] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, Xuebin Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Zhong Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. van der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The International Exascale Software Project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, 2011.

[2] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, M. Richards, and A. Snavely, "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," *Gov. Procure.*, vol. TR-2008-13, p. 278, 2008.

[3] A. Danalis, P. Luszczek, G. Marin, and J. S. Vetter, "BlackjackBench: Hardware Characterization with Portable Micro-Benchmarks and Automatic Analysis of Results."

[4] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," *Sandia Report, SAND2013-4744*, vol. 312, 2013.

[5] "MareNostrum 4 supercomputer to be 12 times more powerful than MareNostrum 3 | BSC-CNS," 2016. [Online]. Available: https://www.bsc.es/news/bsc-news/marenostrum-4-supercomputer-be-12-times-more-powerful-marenostrum-3.

[6] "Aurora Fact Sheet."

[7] "Collaboration of Oak Ridge, Argonne, and Livermore (CORAL)."

[8] "Post-K Supercomputer Overview."

[9] "Linux support performance counters." [Online]. Available: http://web.eece.maine.edu/~vweaver/projects/perf_events/support.html.

[10] P. Luszczek, J. Dongarra, D. Koester, R. Rabensiefner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi, "Introduction to the HPC Challenge Benchmark Suite, Lawrence Berkeley National Laboratory, Paper LBNL-57493, March 2005," *Pap. LBNL-57493*, p. 12, 2005.

[11] "Characterization of HPC codes and problems | Allinea." [Online]. Available: https://www.allinea.com/products/characterization-hpc-codes-and-problems.

[12] "HPC Performance Characterization Analysis | Intel® Software." [Online]. Available: https://software.intel.com/en-us/node/605669.

[13] J. Fenlason and R. Stallman, "GNU gprof," pp. 1–48, 2000.

[14] J. Levon and P. Elie, "Oprofile: A system profiler for linux." 2004.

[15] S. Browne, "A Portable Programming Interface for Performance Evaluation on Modern Processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, 2000.
[16] P. J. Mucci, "PapiEx-execute arbitrary application and measure hardware performance counters with PAPI." 2007.
[17] "Extrae | BSC-Tools." [Online]. Available: https://tools.bsc.es/extrae.
[18] "Folding : detail performance evolution," pp. 5–8.
[19] "perf(1) - Linux manual page." [Online]. Available: http://man7.org/linux/man-pages/man1/perf.1.html.
[20] "perf_event_open(2) - Linux manual page." [Online]. Available: http://man7.org/linux/man-pages/man2/perf_event_open.2.html.
[21] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: A structure for efficient numerical computation," *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 22–30, Mar. 2011.
[22] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 99–104, 2007.
[23] V. Codreanu, J. Hertzer, C. Morales, J. Rodriguez, O. Widar Saastad, M. Stachon, and V. Weinberg, "Best Practice Guide Haswell/Broadwell," 2017.
[24] V. Codreanu, J. Rodríguez, and O. Widar Saastad, "Best Practice Guide - Knights Landing," 2017.
[25] A. Sodani, "Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor," in *2015 IEEE Hot Chips 27 Symposium, HCS 2015*, 2016.
[26] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler, "IBM POWER8 processor core microarchitecture," *IBM J. Res. Dev.*, vol. 59, no. 1, p. 2:1-2:21, 2015.
[27] J. Stuecheli, "Power8," in *Hot Chips*, 2013, vol. 25, p. 2013.
[28] "Whitepaper NVIDIA ® NVLink TM High-Speed Interconnect: Application Performance," 2014.
[29] T. Shimizu, "FUJITSU HPC and Post-K Development of the Post-K Supercomputer," 2016.
[30] J. J. Dongarra, P. Luszczek, and A. Petite, "The LINPACK benchmark: Past, present and future," *Concurr. Comput. Pract. Exp.*, vol. 15, no. 9, pp. 803–820, Aug. 2003.
[31] J. Dongarra, M. A. Heroux, and P. Luszczek January, "HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems *."
[32] J. D. McCalpin, "The STREAM Benchmark," 2005.
[33] "libpfm4 documentation." [Online]. Available: http://perfmon2.sourceforge.net/manv4/libpfm.html.
[34] "Counting floating point operations in MATLAB." [Online]. Available: https://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:SandyFlops#Counting_Floating_Point_Operations_on_Intel_Haswell.
[35] Linux Programmer's Manual, "proc(5) - Linux manual page." [Online]. Available: http://man7.org/linux/man-pages/man5/proc.5.html.
[36] G. Lento and I. Sse, "Optimizing Performance with Intel® Advanced Vector

Extensions Intel® Advanced Vector Extensions 2 instructions can provide significant performance increases. Intel® Advanced Vector Extensions Performance Intel® AVX," 2014.
[37] "glibc 2.23: changelog." [Online]. Available: https://abi-laboratory.pro/tracker/changelog/glibc/2.23/log.html.