Adaptive and Application Dependant Runtime Guided Hardware Reconfiguration for the IBM POWER7

MASTER THESIS

UNIVERSITAT POLITÈCNICA DE CATALUNYA - BARCELONATECH

Director:
Marc Casas Guix
Autor:
David Prat Robles
Codirector:
Miquel Moretó Planas
Ponent:
Mateo Valero Cortés

MASTER EN INVESTIGACIÓ I INNOVACIÓ EN INFORMÀTICA

DEPARTAMENT D'ARQUITECTURA DE COMPUTADORS

FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)

September 4, 2014

Acknowledgements

I would like to express my deep gratitude to both my master thesis supervisors. Dr. Miquel Moretó and Dr. Marc Casas for the useful comments, remarks and engagement through the learning process of this master thesis.
Special thanks are given to all RoMoL team and its collaborators. They are all my workmates. We often help to each other and discuss different ideas to solve engineering problems.
I would also like to thank all my classmates and friends from the MIRI master at the UPC with whom I have shared two years of my life in an enriching adventure that has given me plenty of opportunities to learn at university lectures and to work in companies such as the BSC, Telefónica I+D and Sony Europe; I was very lucky to meet with outstanding people in these places.
Last but not least important, I owe more than thanks to my family members which include my parents, my lovely grandmother and my brother and sister. Without their support, it would have been impossible for me to finish my college and graduate education seamlessly.

Abstract

Hardware data prefetcher engines have been proposed over the years, which has been translated in their inclusion in contemporary general purpose server-class microprocessors. Nonetheless, microprocessors' hardware prefetcher engines don't include any automatic hardware control able to tune their operation while system is up. Instead, they are usually enabled in default configuration when system boots up.

Including a complete autonomous control at hardware level would mean an increase in terms of microprocessor complexity that nowadays technology is unable to deliver to market. This lacking architectural feature causes systems to operate with prefetcher in fixed configuration, which in many cases, is a source of detriment to performance and energy consumption. For example: prefetching a great amount of data in each memory request may involve bringing unnecessary data that not only wastes power by overloading memory bus but also pollutes cache memory hierarchy potentially causing false sharing and saturating their consistency protocol, which can become a serious thread in multicore environments not to mention in manycores.

This master thesis contains a piece of software that solves the discussed problem. The proposed solution involves using a runtime as a bridge between prefetcher registers represented in the operating system and user applications, which indeed are the ones that shape the system workload. This approach provides, in practice, platform portability as only runtime recompilation is needed when moving from machines with different instruction set. Furthermore, no changes in the operating system are needed at all; this can be very convenient in environments such as supercomputers and massive clusters. Finally, it goes without saying that users don't need to take any action rather than to launch their applications as they normally do. The intelligence included in the runtime will take charge of finding out the best prefetcher configuration depending on applications' demands.

In this work, the adaptive prefetcher runtime has been deployed on the IBM POWER7

microprocessor. Presenting a multicore and multithreaded architecture, the mentioned processor is an excellent candidate for shared-resources developments. Moreover, given its richly tunable features such as a hardware prefetcher or capability to configure its simultaneous multithreading way number, combined with a responsive managing middleware in the backdrop, the IBM POWER7 can become a leading piece of hardware as far as hardware adaptivity is concerned.

The final appliance has been set to run applications written under OmpSs programming model. Consciously programmed applications presented outstanding results, these ones showed speed-ups of up to 300% by choosing the right prefetcher configuration. Although available HPC OmpSs applications presented milder benefits, performance improvements have reached around 15% to 50% of gain, depending on applications' characteristics. The development of this appliance has brought interesting remarks that will be expanded in detail through the document. For example, by extracting hardware counters data and being able to match them to OmpSs tasks, a powerful tool that can help to understand machines' issues has been presented. Moreover, being able to plot different metrics from microprocessor, such as used memory bandwidth or data cache misses, has deepen understanding of prefetcher's side effects over the system. This has provided a valuable knowledge to improve the autonomous mechanism that makes possible an automatic and transparent adaptability.

Contents

Chapter 1

Introduction

1.1 Thesis motivation

This master thesis is the fruition of my master in Innovation and Research in Informatics at "Facultat d'Informàtica de Barcelona" with the specialization of High Performance Computing. The thesis is done at Barcelona Supercomputing Center within a project in collaboration with IBM Thomas J. Watson Research Center. The topic of this master thesis is expected to become my main interest in my PhD thesis at BSC keeping the collaboration with IBM. Given that I like hardware systems, I decided to chose this topic about real-time hardware adaptive reconfiguration. It certainly looks like a promising topic as hardware, as an architecture, is a static element to which software is always struggling to obtain its maximum performance. The more the hardware can adapt itself to software's workload, the better power-performance we will be able to get from our whole system. Contemporary microprocessors will require the mentioned adaptability increasingly as their architectures become heterogeneous and scale their cores and threads. Given this evolution in the architecture, microprocessors' shared resources such as last level caches, instruction decoding priorities, interconnection networks between cores and, in the case of this work, hardware prefetcher engine management of each hardware thread, raise a challenge to computer architecture. Orchestrating so many different shared resources to harmonize microprocessor's performance requires complex algorithms that, if they had to be synthesized in hardware, would make commercialization of these processors infeasible with current transistor technologies. For this reason, operating systems and runtime

libraries are keystone; they are the interface between hardware and software and their thorough understanding will unravel what are the best choices so as to make possible a responsive interaction between hardware and software in which a balance in the trade-off between overhead and ease of usage is found.

1.2 State of the art

Prefetching data at hardware level is one of the most powerful techniques to mitigate the so-called memory-gap problem, which is becoming bigger and indeed the main bottleneck in current processors. This problem is ralated to the difference in performance between the processor and the memory access time. Nowadays, there are lots of server-class microprocessors that include data prefetch engines so as to deliver acceptable results when dealing with memory-intense workloads. Moreover, these prefetchers usually include some parameters to tune its functionality; for example: the depth of each prefech action, what stride size do data have, disable prefetching, etc. However, in commercial systems, hardware prefetcher is not commonly an automatic feature, being configured at systeem boot-time to a default value passing the responsibility of its configuration to the programmer. The issue about this approach is that neither operating systems nor software vendors deal with the problem to find an optimal prefetch configuration in any program execution. Because of this, most of the times systems fail to prefetch data in an effective way degrading performance due to a misuse of memory bus as well as cache pollution.
There have been many works that have dealt with data prefetch [1, 25, 34]. First attempts were based on sequential prefetchers. Based on the fact that many applications access data following a sequential pattern, this approach suggest to prefetch memory blocks sequentially. Despite being effective in these cases, this solution is not able to yield performance when the application doesn't follow a sequential data access pattern. Due to this, further research in prefetchers was done to try to capture the non-sequential nature of those applications. Prefetch techniques aimed to deal with pointer-based applications have been studied in [15, 33, 41]. Solihin et al. made use of an user-level memory thread to do prefetching, getting in the applications with irregular accesses significant speedups [42]. Joseph and Grundwald worked on Markov-based prefetchers in [13]. Although most of these works about prefetching have not been put into practise with real processors, limit studies and prefetch analytical models have been shown in [17, 35].

A further step in data prefetching is to consider the interaction between threads that takes place in the CMP processors. Ebrahimi et al. [27] and Lee et al. [31] study the effect of thread-interaction on prefetch and design prefetch systems that improve throughput and/or fairness. Liu and Sohilin present a study about the impact prefetching has and bandwidth partitioning in CMPs [28].
Although there are many sutudies about data prefetching on top of simulators, there are very few works that make use of real processors. For instance: Wu and Maronosi characterize the prefetcher of an Intel Nehalem processor and provide a straightforward algorithm that can control dynamically the activation and deactivation of the prefetcher. Nevertheless, their work only contemplates intra-application cache interference obviating actual system performance. Liao, et al. build a machine learning model that dynamically modifies the prefetch configuration of the machines in a data center (based on Intel Core2 processors) [19]. Their work also bases its approach on turning on and off the prefetcher. Beyond enabling and disabling the prefetcher, there are other kind of works targeted to control thread execution rate. For example, playing with fetch policies within a SMT processor is studied in [18, 11]. They aim to increase throughput and/or provide quality of service (QoS). In the same line, the work of Boneti et al. [4] studies the usage of the dynamic hardware priorities in IBM POWER5 processor aiming to yield performance from resource balancing and prioritization. Qureshi and Patt [32] study how to improve throughput through solving the problem of partitioning the last-level cache for multiple applications. Moreto et al. [30] show a similar solution based on achieving QoS for multiple applications running at the same time.

1.3 Project objectives

The aim of this project is to develop adaptive resource management systems for the improvement of the power-performance metrics associated with the current and future IBM POWER-series microprocessors. Given their simultaneous multithreaded architecture, these microprocessors are increasingly demanding resource-sharing management and because of the problem's complexity, an accessible solution is to develop a low latency piece of software that can be dealing continuously with the problem at runtime. The proposed mechanisms will be a backdrop that, to begin with, will be used by OmpSs programming model [5], [6], [9] implemented with Nanos++ runtime system [10], [8] and Mercurium

source-to-source compiler [7].

This master thesis puts the scope into hardware prefetcher engines, which represent a well-known and powerful performance enhancement technique. The IBM POWER7 has 32 hardware threads, each of them has an independently configurable hardware prefetcher [29]. Nevertheless, their configuration is not automatic and is set to a default value when the system boots up. The target in this master thesis is to construct a responsive piece of software that can harmonize constantly such an amount of prefetcher operations form different threads, configuring them independently, according to a variable workload generated by user applications.

Chapter 2

Proposal of the solution

2.1 Methodology

Resource sharing is and will become an even bigger issue as microprocessor architecture goes multicore and heterogeneous. In this thesis, a proposal to solve this issue is presented and implemented. We propose a solution, which manages the hardware prefethcer, based on a run-time library because of its benefits; it is a neat and portable way to add hardware adaptivity that is independent of the underlying machine. Moreover, as it will be shown, it doesn't require any additional action as far as the user is concerned. In this case, the run-time library chosen is Nanos++ RTL because is largely developed and it has good support. We will use OmpSs programming model due to its capacity to squeeze parallelism of the system underlying hardware thanks to its capability to control data dependency of an application at run-time.

Given that we want to do an optimization on the processor shared resources using processor's configuration registers, we need some way to analyse the processor's performance in order to choose its best configuration. Thanks to transistor shrinking, each new processor generation comes with a wider range of performance counters. These are special registers dedicated to count processor events such as how many instructions of a type a processor has executed or how many memory cache misses a processor has had during a lapse of time. These counters actually are the most accurate way to measure performance and their usage overhead is the lowest possible due to their inclusion in hardware.

To sum up, the basic idea of the proposed solution is to use performance counters so

as to measure processor's performance at run-time aiming to be executing with the best possible prefetcher configuration during the maximum lapse of time in each phase of the application.

2.2 OmpSs programming model

OmpSs is a programming model developed by the BSC aimed to extend OpenMP with new directives to support asynchronous parallelism and heterogeneity. That is to say, OmpSs is able to execute regions of code in parallel even if they are not explicitly declared as parallel with respect to other regions. This is because these regions of code declare their potential parallelism with respect to data they use instead of being just a loop iteration with some variable dependencies, which is the case of OpenMP. Regarding heterogeneity, OmpSs provides support for GPUs as well as other accelerator-like processors giving to the programmer the possibility to include OpenCL and CUDA kernels.
As above mentioned, OpenMP has a fork-join model. Nevertheless, OmpSs defines a thread-pool model where all usable threads exist at the outset of the execution [14]. One of these threads is designated master thread, being responsible of the user code execution whereas the remaining threads hold on ready to undertake the execution of some regions of code. This includes the possibility that worker threads can become master threads if there is a nested directive so as to fork more parallelism. In this case, the following action is to ask for more worker threads to the thread-pool. In OmpSs there is no need to make use of the parallel directive explicitly, (such in OpenMP); that is because all threads from the thread-pool are created from the very beginning of the execution and they can be requested in any nested task construct.

In OmpSs, potential parallel regions of code are defined as tasks; these tasks are defined by the programmer and with them their data dependencies, which will become the rules at execution time for their potential parallelism. OmpSs task constructor defines the clauses **in** (standing for input), **out** (standing for ouput) and **inout** (standing for input/output). These clauses are indeed what allows such an asynchronous parallelism, providing what data a task is waiting for and signalling its readiness. Suffice it to say that it is the programmer responsibility whether a task really uses that data in the specified way or not.

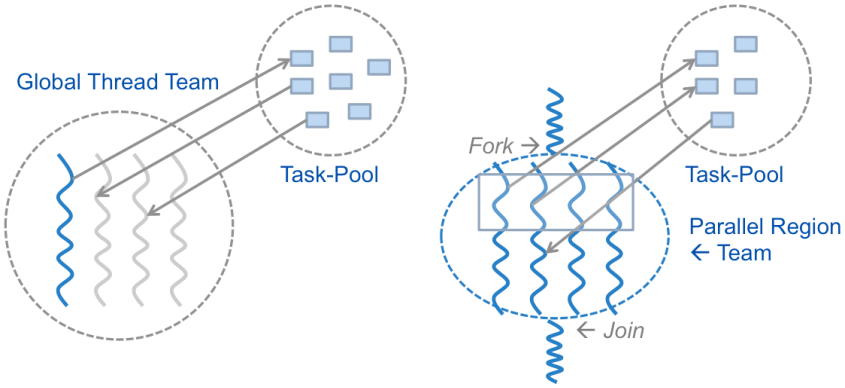

Figure 2.1: OmpSs execution model vs. OpenMP execution model. Image borrowed from [38]

When a new task is created its input and output dependencies are tested with respect to the other existing tasks. In case dependencies, either RaW, WaW or WaR, are found, that new task is taken as a successor of those matched related tasks. As it will be shown; a runtime library will construct a dependency graph based on these task dependency relations.

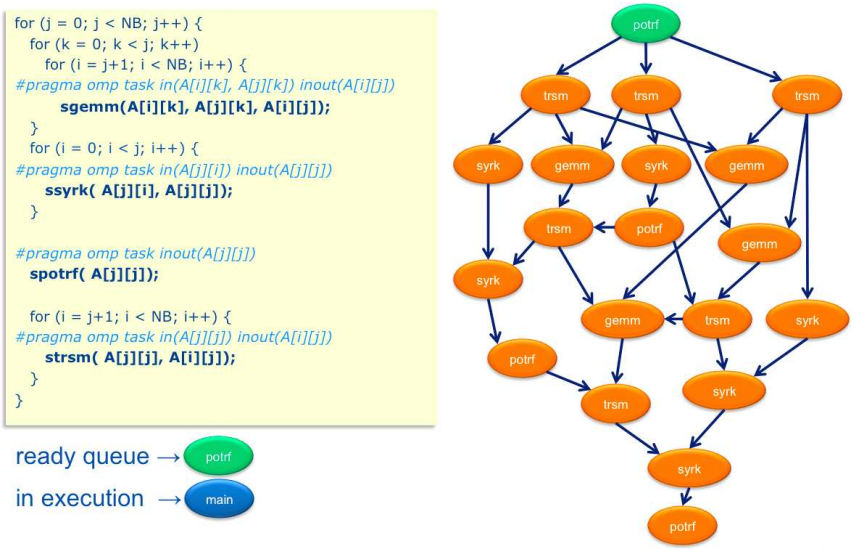

Figure 2.2: Cholesky factorization example using OmpSs. Image borrowed from [38]

In picture 2.2 an example of how OmpSs works is shown. The example represents a Choleksy factorization. The algorithm uses different kinds of functions, in this case they

are all preceded by an OmpSs directive called pragma in which the programmer specifies what input and ouptut dependencies does that task have. As the code follows its execution, the master thread requests for worker threads to perform encountered tasks. However, before a task is executed by a worker, it needs to have its dependencies ready to consume and here is when the runtime library dependency graph comes into play. Figure 2.2 shows what the graph would look like if master thread was able to traverse all application code, (and therefore all tasks had been executed), before all dependencies were satisfied. In practise, the graph depth is increased as workers finish their assigned tasks.

Apart from supporting native C/C++ values, OmpSs is able to keep track of array region dependencies. Array sections allow to refer to multiple elements of an array or pointer data in one expression. OmpSs allows to specify regions in two different ways: the first one has the form of variable[lowerbound : upperbound] where as the expression suggests, the lower and upper bounds are given explicitly by the programmer. The second form is variable[lower ; size], here instead of telling what is the upper bound, this one is inferred from the lower bound plus the size of the region. Let it suffice to say that the last element of the region is: lower bound + size - 1.

OmpSs not only offers ease of use to the programmer but also higher performance. Furthermore, it is very simple to translate a code to OmpSs, whether it is a serial or a parallel code such as pthreads. As a matter of fact, some tests made with Parsec benchmarks showed a significant LOC (lines of code) reduction while keeping performance. That Parsec benchmark porting also included a performance comparison in which, in most of the cases, OmpSs showed a better scaling making a notorious difference when dealing with high thread numbers.

2.3 Understanding Nanos++ runtime library

Nanos++ is a runtime library (RTL) targeted to give support to parallel environments. Its main purpose is to be in charge of solving the dependencies mentioned in section 2.2 in OmpSs programming model. A part from this, it also implements modules allowing support for OpenMP and Chapel. The runtime implements OmpSs tasks with user-level threads when possible; for example: x86, x86.64, ia64, ppc32 and ppc64 are supported.

It is straightforward to use it; the programmer only needs to compile their source, usually a parallel code using OmpSs pragmas, linking Nanos++ RTL to that executable. The result is that Nanos++ will take charge of program data dependencies building the graph dependencies shown in section 2.2.

Nanos++ is mainly developed in C++ and can expanded and installed using autotools. Its design is shown in figure 2.3. Nanos++ core is composed of three modules: slicers, dependencies and instrumentation or tracer. OmpSs tasks are represented in Nanos++ by workdescriptors, which contain meta information of a task. A slicedWD is a type of workdescriptor that can be divided into smaller workdescriptors; it is used for example when the code has a parallel directive referencing a loop. In figure 2.3, the component PE means processing element, that is the internal representation for each processing element a given execution has available, (depending on for example: used programming language). Processing elements such as microprocessors, GPUs or other kinds of accelerators are represented by Nanos++ storing useful information like the NUMA node in which that processing element resides, a unique identifier, etc.

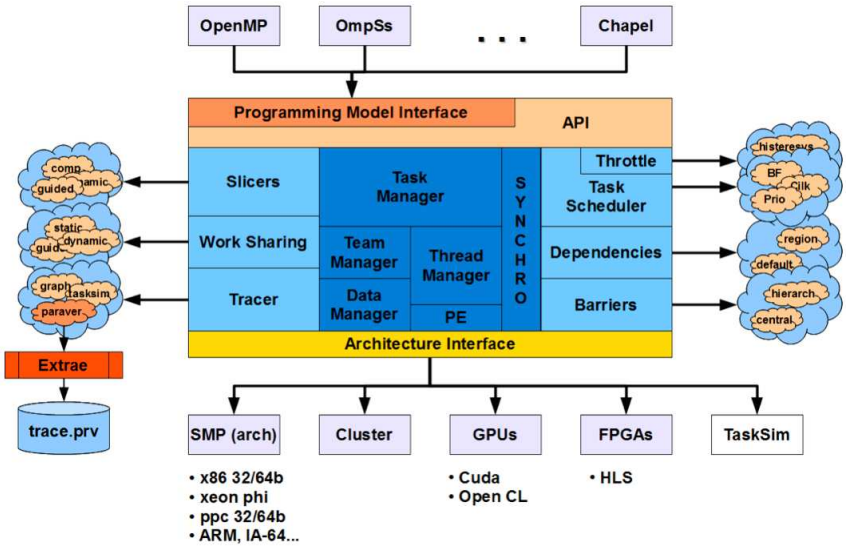

Figure 2.3: Nanos++ overview. Image borrowed from [37]

Nanos++ structure is mainly an idle loop in which threads spin while they are not executing user code. Each time a master thread needs to assign work to a worker thread a

workdescriptor is created for that task using a thread also represented in Nanos++ RTL. A Nanos++ thead may change its workdescriptor execution at any time depending on factors such as different execution phases or pre-emptions.

Figure 2.4 shows an example of how Nanos++ consumes tasks from the tasks dependency graph as they get free of dependencies. As the graph is constructed and tasks are free of dependencies, they are queued in the ready queue waiting for processing elements to be free. In the example, a breath first search algorithm is used to explore the task dependencies graph; the result is that by executing red tasks with priority according to the algorithm, all green tasks are soon free of dependencies enabling full parallelism very soon and therefore yielding much more work in comparison with other programming models. The figure shows a quad-core CPU in which the second red task is being executed in the first processor and some of green tasks are also being executed; it is very important to notice that the first red task has already finished allowing more tasks to be executed as it is the one that serializes a part of the code.

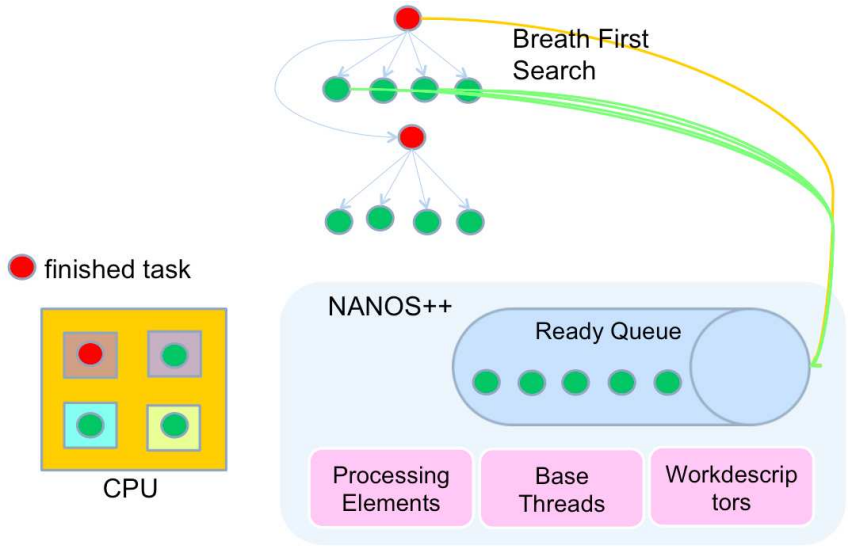

Figure 2.4: Nanos++ operation

The runtime allows the user to set parameters for an execution. Table 2.1 shows some commonly used ones.

Any Nanos++ package can be compiled in four versions: performance, debug, instrumentation and instrumentation-debug. The fist version is the one used to get the most out

Option	Description
–pes	set number of processing elements
–threads	set number of threads
–cores-per-soket	set number of cores per socket
–binding-start	set initial cpu for binding (binding required)
–binding-stride	set binding stride (binding required)
–disable-binding, –no-disable-binding	disables/enables thread binding (enabled by default)
–architecture	defines default architecture. Where arch. can be smp, gpu, opencl or cluster

Table 2.1: Nanos++ runtime application execution options

of Nanos++ performance as its code doesn't make any unnecessary call. Debug version is able to show into more detail execution traces to debug an application. Finally, instrumentation version enables support to use BSC instrumentation and simulation tools such as Paraver and Dimemas.

2.4 Mercurium source-to-source compiler

Mercurium is a source-to-source compiler that currently supports C and C++ languages. Mercurium is commonly used with Nanos++ runtime library (RTL) as it provides support for OmpSs and OpenMP programming models. Nevertheless, because of its extensibility it also has been implemented to support other programming models and compiler transformations such as Cell Superscalar, Distributed Shared Memory, Software Transactional Memory and the ACNOTES project. This extensibility is possible thanks to the use of plugins that are written in C++ and dynamically loaded by the compiler depending on what configuration does the programmer chose. One feature of Mercurium source-to-source compiler is that code transformations are implemented in terms of source code, avoiding any need of modifying or knowing any compiler's internal syntactic representation.

Mercurium compiler has stable support to compile the most used programming languages. Table 2.2 shows current supported languages.

The way to tell the compiler the code to compile has OpenMP or OmpSs directives is shown in table 2.3

Mercurium	Language
mcc	C
mcxx	C++
mnvcc	CUDA and C
mnvcxx	CUDA and C++
mfc	Fortran

Table 2.2: Mercurium compilation options for back-end compilation choice

Flag	Description
–ompss	OmpSs
–nanox	OpenMP

Table 2.3: Mercurium compilation options for programming model choice

A wide range of compilation options related to Nanos++ run-time library is available in Mercurium. Table 2.4 shows some important ones. Spins option deserves special attention because is actually a parameter to tune Nanos++. It tells the thread associated to a workdescriptor how many times does it have to iterate in the idle loop before it is put to sleep.

Option	Description
–threads=n	Defines the number of threads used by the process
–disable-yield	Disables thread yielding to OS
–spins=n	Number of spins before sleep when idle
–sleeps=n	Number of sleeps before yielding
–sleep-time=nsecs	Sleep time before spinning again
–disable-binding=nsecs	Do not bind threads to CPUs
–binding-start=nsecs	Initial CPU where start binding
–binding-stride=nsecs	Stride between bound CPUs

Table 2.4: Mercurium compilation options for Nanos++

2.5 Designing components' interaction

Now that the problematic and system's purpose have been described as well as all the underlying software to support the solution, we are ready to take a look to the interactions between each piece of the system.
Let's see how Nanos++ environment works:

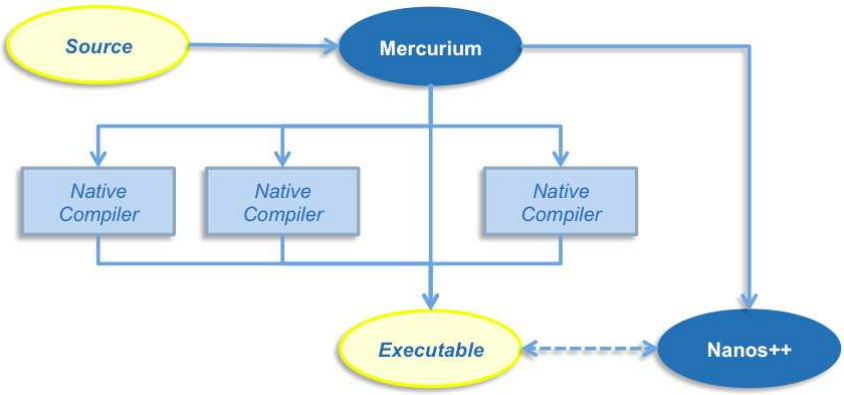

Figure 2.5: OmpSs implementation. Image borrowed from [38]

Figure 2.5 shows how source code is transformed to an executable. Given a file source code, Mercurium compiler parses the file finding its language extension so as to generate and intermediate code that can be processed by a back-end compiler such as GCC or G++, transforming OmpSs directives to run-time calls. Once the executable is obtained, its execution is tied to Nanos++ run-time library which can keep tack of data dependencies apart from other things like creating/executing tasks, synchronization or memory consistency.

A straightforward explanation of how Nanos++ runtime can interact with PAPI library, (a library that allows to extract performance counters in a straightforward way), is showed in figure 2.6. As it has been commented, the user application source containing OmpSs pragmas is compiled with Mercurium source-to-source compiler which uses a back-end compiler such as GCC, G++, etc. depending on what language the user used to write the application. Once the executable is obtained, this one keeps interaction with the runtime system during its execution so as to the later one can control what tasks can be executed every moment.

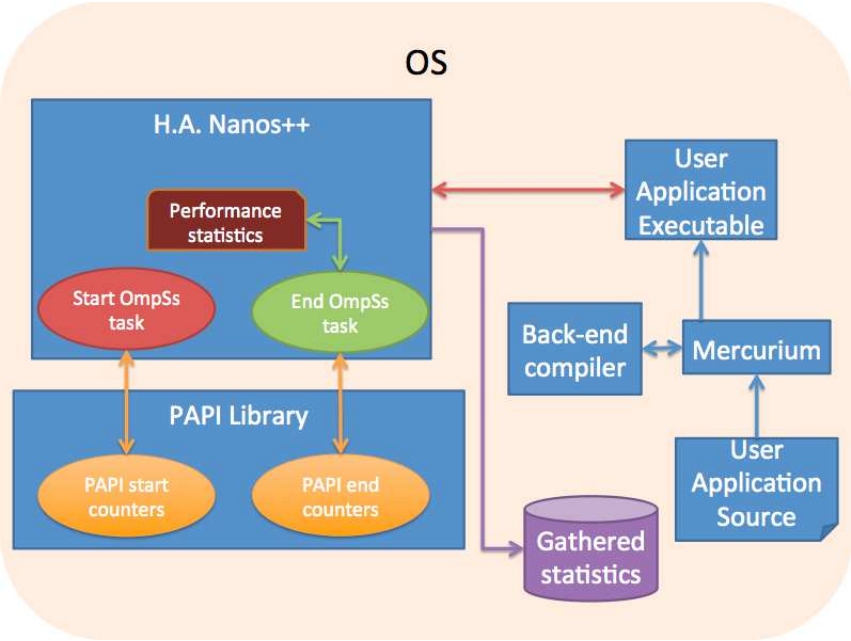

Figure 2.6: Components' Interaction

Simplifying how the mechanism works: with the application in execution, each time the runtime creates a workdescriptor to represent an OmpSs task, a PAPI call to start hardware counters is called taking into account that immediately after the PAPI call only task code should be executed to avoid losing precision in the measurement. With this in mind and also not wanting to interpose any additional library between the runtime and the user application nor modifying or adding any of the usual steps the user has to do to compile the source code, the point is to find the best place in the runtime to put the PAPI calls; this will be explained in section section 4.2. Basically, once the task ends up the workdescriptor, it executes a predefined function, so a good place to save those hardware counters is there. When the whole user application finishes, the runtime executes another predefined function, so here is the moment when our version of the runtime can call to additional functions to gather statistics about each kind of task in each SMT thread for as many performance counters as we are interested in. Those statistics are bulked in text files in the same directory from where the execution was taking place.

Chapter 3

Experimental setups

In this chapter, machines used through this project are explained in detail with special attention to their microprocessors. There is also a section dedicated to introduce all benchmark and application suites used during the experiments. As it will be shown, these will compose a wide range of codes ported to OmpSs programming model, including applications from many different areas.

3.1 IBM POWER7

IBM POWER7 is a superscalar symmetric multiprocessor based on Power Architecture. It was released in 2010 being the successor of the IBM POWER6. One key feature of POWER7 is that it has support for global shared memory space for POWER7 clusters; enabling to the programmers to program a cluster without using message passing. The POWER7 can be manufactured including 4, 6 or 8 physical cores per microchip, each core is capable of four-way simultaneous multithreading, what makes a maximum of 32 hardware threads per chip. Each core has its private data and instruction data caches, a private L2 cache and a local L3 region that can be shared between all cores. One of the main differences between the POWER6 and the POWER7 is that the latter one executes instructions out-of-order instead of in-order. Although the POWER7 decreased the maximum frequency with respect to POWER6, from 5.0 GHz to 4.25, each core ends up having higher performance that one of POWER6, while having up to 4 times the number

of cores.
For those applications that require fastest sequential performance rather than high parallelism, POWER7 includes a special TurboCore mode that can turn off half of the cores from an eight-core processor. This way four cores have access the two memory controllers and L3 cache at higher clock speeds. TurboCore also can be used to reduce software costs in half for applications that are licensed per core. This is a very appealing feature as contemporary architectures are increasingly including more cores per die.
IBM POWER7 chip has 567 mm^2 and is fabricated in IBM's 45 nm Silicon-on-Insulator (SOI) CMOS technology with 11 layers of low-k copper wiring. A key innovation introduced at the 45 nm technology node is the deep trench capacitors technology used for the embedded DRAM L3 and for on-chip voltage supply decoupling. The functionally equivalent chip transistor count would have been over 2.7B against the current 1.2B if the L3 had been implemented with a conventional 6 transistor SRAM cell [40].

Figure 3.1 shows POWER7 building blocks; the chip scheme is composed of independent modular blocks to separate unused cores from supply voltages. For example: each Cache/Core Partition (CCP) chiplet has its own set of separated supply voltages for logic, SRAM and DRAM as well as an analog voltage for the Digital Phase-Locked Loop (DPLL). Additionally, each core including their associated L2 and L3 arrays, can operate at a different, dynamically adjustable frequency to balance system performance as well as power consumption. Other important modules include the local and remote SMP interconnect partitions and the memory interfaces. A central interconnect that operates at a fixed frequency gathers each core/cache complex, the 2 integrated memory controllers, the off-chip local and global SMP links and the I/O controllers using asynchronous connections.

POWER7 processor cores implement aggressive out-of-order (OoO) instruction execution to achieve high efficiency by using their available execution paths. Processor core includes mainly seven separately integrated units. Figure 3.2 shows these parts. The flow of instructions starts with the instruction fetch unit (IFU) as they are transferred from the L2 into a 32 KB instruction cache in packets of up to 32 instructions. Then the decoder loads instructions in groups of 6 instructions; each branch encountered is predicted in the IFU logics. Once the instruction sequencing unit (ISU) has the instructions, it puts them into into multiple issue queues while they are waiting for the resources to be avail-

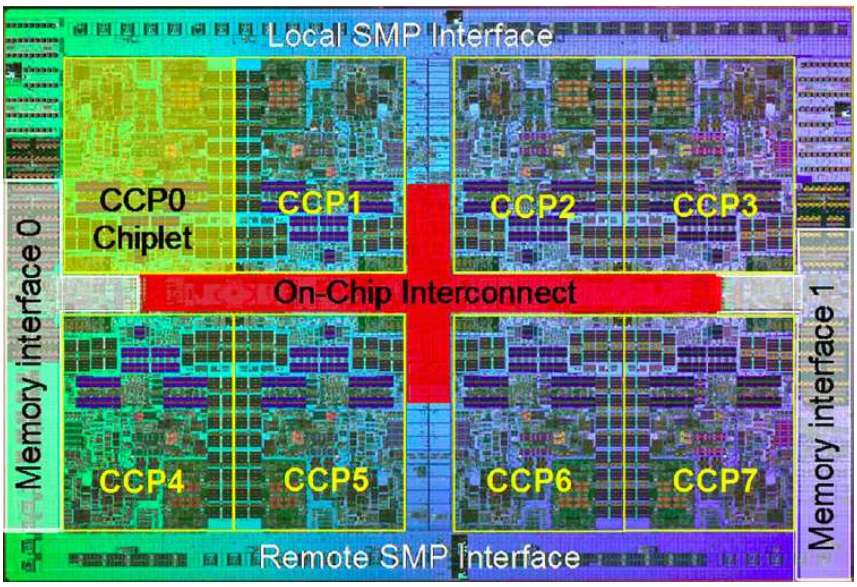

Figure 3.1: IBM's eight core POWER7 processor. Image borrowed from [40]

able. Instructions to be executed in the fixed point unit (FXU), load store unit (LSU), decimal floating point unit (DFU) and instructions targeted for the merged vector scalar unit (VSU) + floating point unit (FPU), are kept in two unified queues. These queues are divided into two 24-entry halves. Apart from this, branches are kept in a 12-entry branch issue queue. When a branch can be resolved it is passed down to the branch execution unit (BRU) also located in the IFU. If there is a misprediction, the instruction fetch is redirected to the right target of the branch.

The IBM's POWER7 execution pipeline is shown in figure 3.3. The scheme is a simplification of the real one so as to remark POWER7's basic ideas about its operation. Most modern processors can dispatch groups of machine instructions each cycle, likewise the POWER7 has been designed bearing that in mind. Basically, instructions within a group can execute out of order. However, all group operations need to finish before a group can mark as completed, that is to say, retired.
Apart from providing an abstraction of POWER7's pipeline, figure 3.3 is also a decomposition of how the processor is divided in parts that can be monitored in terms of CPI (cycles per instruction) performance. The processor features a CPI stack that can be referred as CPI stall analysis. This can be very useful if a fine grained study of the IPC

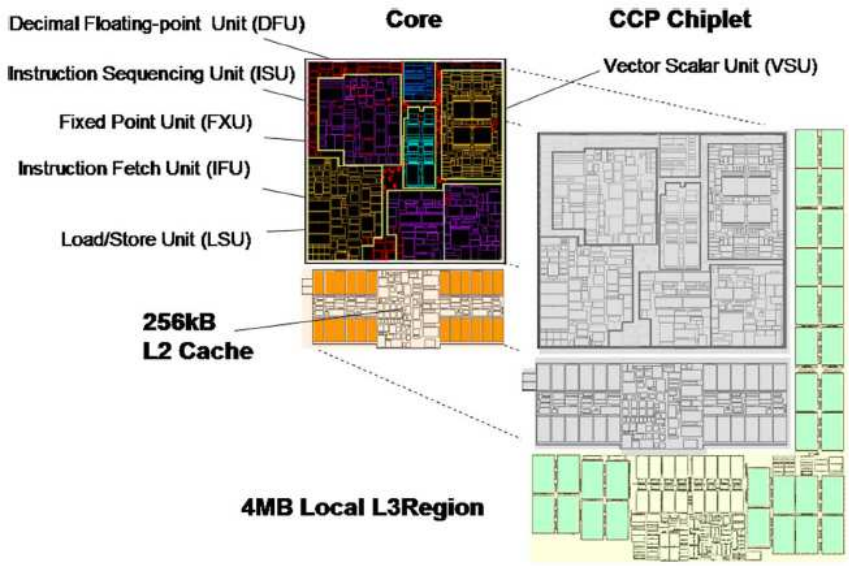

Figure 3.2: IBM's POWER7 core and its L3 region. Image borrowed from [40]

(instruction per cycle) is needed to analyse application's performance. Nonetheless, this work won't go into that detail in this issue as it is priority to analyse the prefetcher's behaviour.

POWER7 has these specifications: [36]

- 45 nm SOI process, 567 mm2
- 1.2 billion transistors
- 3.0 4.25 GHz clock speed
- max 4 chips per quad-chip module
- 4, 6 or 8 cores per chip
- 4 SMT threads per core (available in AIX 6.1 TL05 (releases in April 2010) and above)
- 12 execution units per core:
  - 2 fixed-point units

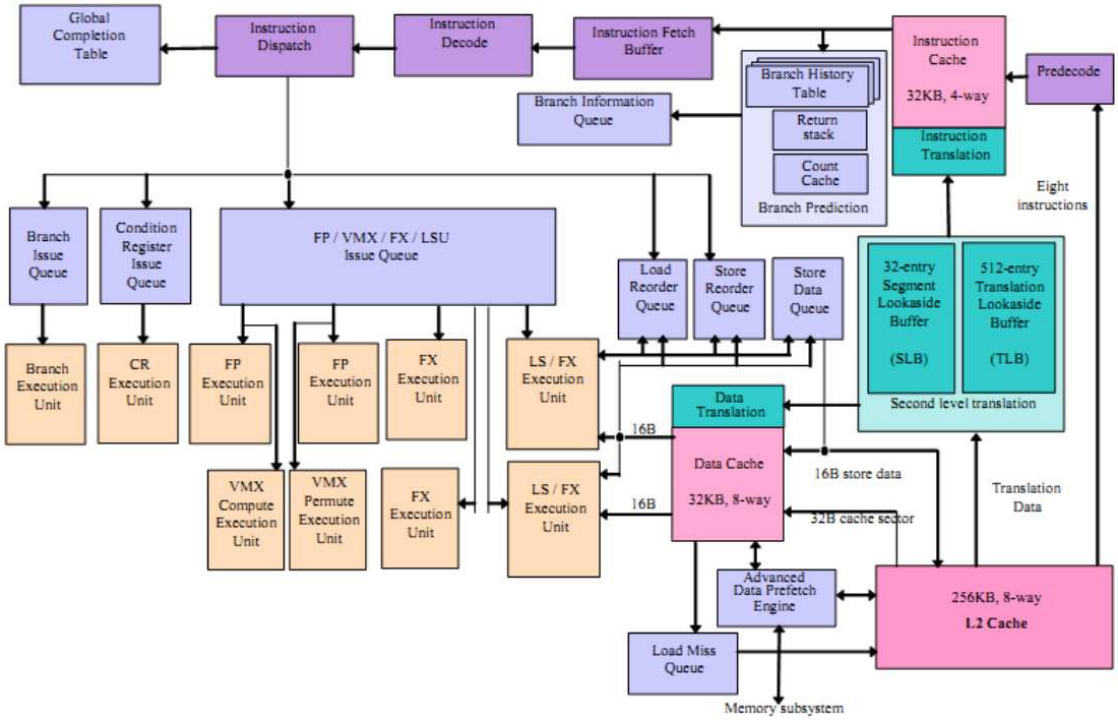

Figure 3.3: Simplified execution pipeline of IBM's POWER7. Image borrowed from [16]

  - 2 load/store units
  - 4 double-precision floating-point units
  - 1 vector unit supporting VSX
  - 1 decimal floating-point unit
  - 1 branch unit
  - 1 condition register unit

- 32+32 kB L1 instruction and data cache (per core)[14]
- 256 kB L2 cache (per core)
- 4 MB L3 cache per core with maximum up to 32MB supported. The cache is implemented in eDRAM, which does not require as many transistors per cell as a standard SRAM[5] so it allows for a larger cache while using the same area as SRAM.

The POWER7 processor has a set of twelve execution units; this gives the following theoretical performance figures (based on a 4.14 GHz 8 core implementation):

- max 99.36 GFLOPS per core
- max 794.88 GFLOPS per chip

4 64-bit SIMD units per core, and a 128-bit SIMD VMX unit per core, can do 12 Multiply-Adds per cycle, giving 24 FP ops per cycle. At 4.14 GHz, that gives 4.14 billion * 24 = 99.36 GFLOPS, and with 8 cores, that increases the figure to 794.88 GFLOPS.

3.1.1 IBM POWER7 reconfigurability

This section is dedicated to the POWER7 microarchiterctural components that provide it with adaptability paying special attention to the hardware prefetcher which is indeed the component this work is based on. The POWER7 includes mainly two hardware components that can be configured from the operating system. They are in charge of: prefetcher operation and SMT management.
Regarding the prefetcher operation, multiple configurations can be set. For example the user can enable and disable it. Furthermore, a sort of options to tune its depth and way to operate can be selected. This is done by setting values to a processor control register called data stream control register or DSCR. The layout of this register is one that makes use of bits 58 to 63 out of its 64 bits [21].
Starting by the less significant bit, the purpose of each set of bits is the following one:

- bit 58, called load stream disable or LSD; disables hardware detection and initiation of load streams
- bit 59, called stride-N stream enable or SNSE; enables hardware detection and initiation of load and store streams that have a stride greater than a single cache block. Such load streams are detected only when LSD is also zero and SSE is one.
- bit 60, called store stream enable or SSE; enables hardware detection and initiation of store streams.

- bits 61 to 63, called default prefetcher depth or DPFD; supplies the prefetch depth for hardware-defined streams for which a depth of zero is specified or for which DCBT/DCBTST with TH1010 is not used in their description. Their values configure the depth as follows:
  - 0: default depth
  - 1: disable prefetching
  - 2: shallowest
  - 3: shallow
  - 4: medium
  - 5: deep
  - 6: deeper
  - 7: deepest

The contents of the DSCR have an impact on how processor handles hardware-detected and software-defined data streams. Any user set up of DSCR causes all active and nascent data streams to cease to exist. It also has to be taken into account that their set up requires OS privileges because the way it was included in the architecture needed it to be compatible with the application binary interface.
Regarding LSD, SNSE and SSE fields, their values don't affect the streams specified by using the dcbt and dcbtst instructions.

Given the scope of this work, SMT management possibilities in POWER7 also deserve attention. In POWER7 there are basically two aspects tunable of SMT: number of SMT threads allowed per core and their priorities with respect to other threads in the same core.
SMT thread number can be changed to run without SMT, 2 SMT threads and 4 SMT threads. For example: Linux distributions come with the ppc64_cpu command that allows the user to easily change SMT mode. If we wanted to run with 4 SMT threads the command to type would be ppc64_cpu –smt=4.
With respect to SMT priorities, there are different ways to deal with them: by modifying the SMT priority directly using the PPR register, through the usage of special no-ops and

using the AIX thread_set_smt_priority system call [20].
POWER7 priorities can be set to low, medium-low or medium. In POWER7+ an additional option, very-low, was added.

3.1.2 IBM BladeCenter PS701

In this work the system used has been an IBM BladeCenter PS701; which basically is a blade containing one socket with an 8 cores IBM POWER7 running at 3.0 GHz. Figure 3.4 presents a photo of the used blade.

Figure 3.4: IBM's BladeCenter PS701. Image borrowed from [22]

IBM's BladeCenter PS701 has a single socket containing an IBM POWER7 processor running at 3GHz. Although chapter 3 explains IBM's POWER7 architecture into detail, here a quick view of PS701 node is done; Figure 3.5 shows IBM BladeCenter PS701 main components and their connections. Among all its components, an interesting one in the background of this work, is the SMP connector. This one is used to scale effectively the number of processors used by a single shared memory user application. The POWER7 processor uses a combination of local and global SMP links with high coherency bandwidth and makes use of the dual-scope broadcast coherence protocol. The SMP designpoint is able to scale up to 32 sockets with IBM POWER7 processors. If we take them

into account when talking about memory hierarchy, SMP links would be the next level after a processor last level cache (LLC), which in this case, is the L3 cache.

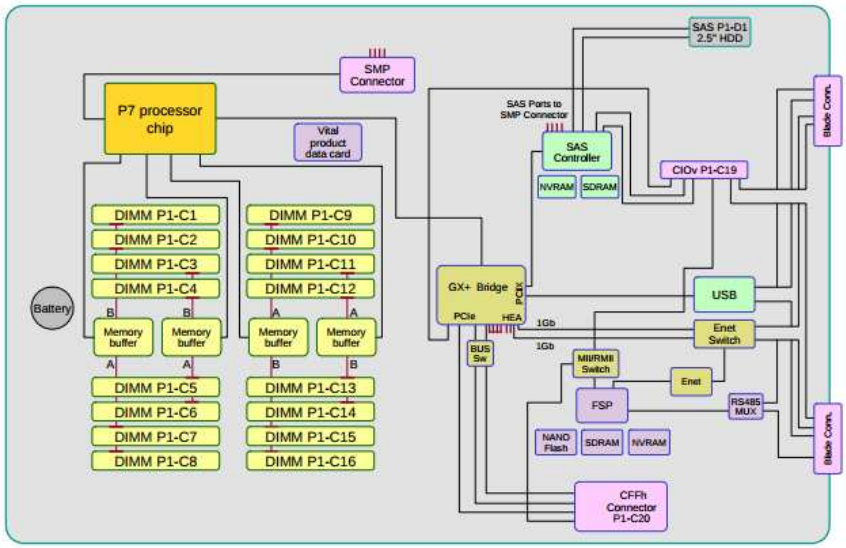

Figure 3.5: IBM BladeCenter PS701 logic data flow view. Image borrowed from [39]

Going on with the node memory subsystem, figure 3.6 shows how it is implemented in the IBM BladeCenter PS701. The POWER7 chip includes two memory controllers with four channels for each memory controller. However, the implementation on the PS701 uses a single memory controller of the processor. Each memory buffer chip connects to four memory DIMMs, 16 in total per processor chip. Regarding error correction, the bus transferring data between the memory and the processor is provided with a CRC error detection that not only uses operation retry mechanism but also has the ability to retune bus parameters dynamically when a fault occurs. Furthermore, the memory bus has spare capacity to substitute a spare data bit-line for which is determined to be faulty. The four advanced memory buffer chips help to increase performance acting as read/write buffers; they are on the system planar and support four DIMMs each.

3.2 Intel Sandy Bridge-EP E5-2670

Intel Sandy Bridge-EP E5-2670 was launched in 2012, some of its main attractiveness are support for quad-channel rather than triple-channel memory support, 1.600MHZ rather

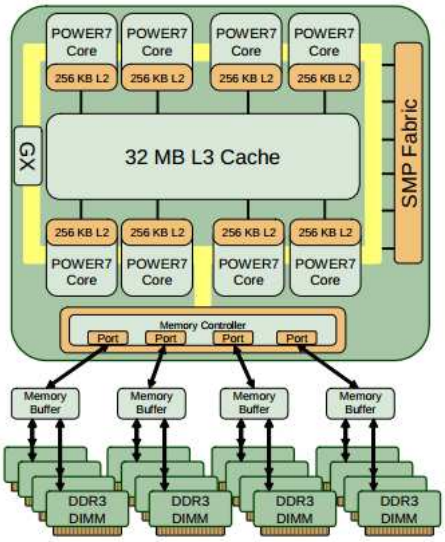

Figure 3.6: IBM BladeCenter PS701 memory subsystem. Image borrowed from [39]

than 1.300MHz memory support, a native PCI-E 3.0 controller, which was actually the first CPU to do so, and three rather than two QPI links. Figure 3.7 shows Sandy Bridge microprocessor architecture layout. Special attention is deserved to LLC design, which is shared among the 8 processors having a bidirectional ring to interconnect all its portions.

Intel Sandy Bridge-EP E5-2670 has these main specifications: [12]

- 32 nm lithography
- 2.6 to 3.0 GHz clock speed
- 8 cores per chip
- 2 SMT threads per core
- 115W of thermal design power
- L1 data of 8x32 KBytes and L1 instructions of 8x32 KBytes
- 8x256 KB L2 cache
- 20 MB LLC cache
- 3200MHz QPI link speed

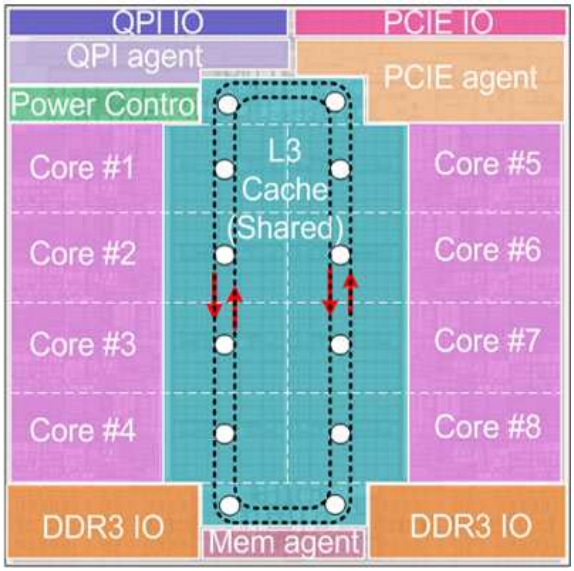

Figure 3.7: Intel's eight core Sandy Bridge processor. Image borrowed from [2]

3.2.1 MareNostrum III

MareNostrum III Supercomputer's main component in which benchmark results will depend more is its processors as all benchmarks will use at most one CPU. However let's now take a look to supercomputer's specifications in general by looking at figure 3.8. To begin with, MareNostrum III uses Intel Sandy Bridge-EP E5-2670 2.6GHz/1600 20M 8-core 115W processors. It is important to take into account that 2-way SMT processor feature is turned off in MareNostrum III because of performance degradation when not tuning applications code. As it has been mentioned, in the experiments only one core will be used, so in this case, available memory RAM per core will be raised to 4 GB as two sockets in a node share all RAM memory. Regarding the network, there's no need to take it into account in these experiments.

Each computation node of a MareNostrum's III rack has the architecture described in figure 3.9.

3.3 Application suites

This section explains all application suites used during the project. They are all written in OmpSs programming model.

Compute	Cores/chip	8
	Chip/node	2
	Cores/node	16
	Nodes	3028
	Total cores	48448
Performance	Freq.	2,6
	Gflops/core	20,8
	Gflops/node	332,8
	Total Tflops	1000,0
Memory	GB/core (GB)	2
	GB/node (GB)	32
	Total (TB)	96,89
Network	Topology	Non- blocking Fat Tree
	Latency (µs)	0,7
	Bandwidth (Gb/s)	40
Storage	(TB)	2000
Consumption	(KW)	1080

Figure 3.8: MareNostrum III Supercomputer specifications. Image borrowed from [3]

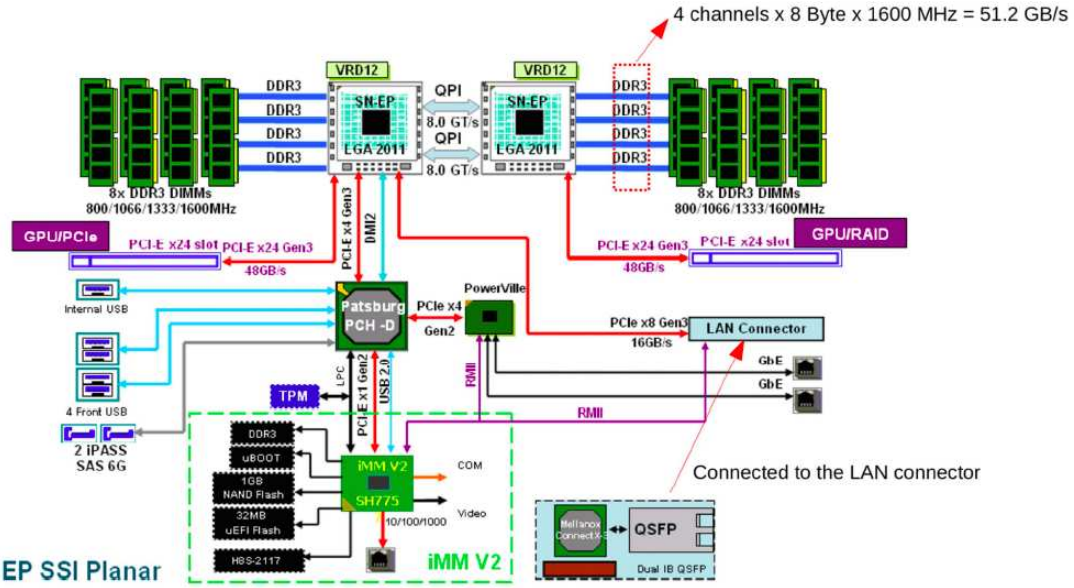

Figure 3.9: MareNostrum III Supercomputer computing node. Image borrowed from [3]

3.3.1 Parsec benchmark buite

Parsec benchmark suite takes its name from Princeton application repository for shared-memory computers. This benchmark suite is composed of multithreaded programs and,

in this work, it has been the first benchmark suite put to the test because is a recent and updated suite that well fits with the multicore architectures of the IBM POWER7 and Intel's Sandy Bridge-EP E5-2670. This suite has a sort of very different benchmarks representative of nowadays most used applications. They appear not only in computer science but also for example in entertainment.

Used Parsec benchamarks in this work include:

- Blackscholes: is used in computational finance. It prices a portfolio of options with the Black-Scholes PDE.
- Swaptions: a computation finance application that prices a portfolio of swaptions with the Heath-Jarrow-Morton framework.
- Fluidanimate: this benchmark is a computer animation application that simulates the underlying physics of fluid motion for realtime animation purposes with SPH algorithm.
- Bodytrack: tracks a markerless human body being a computer vision application.
- Freqmine: a data mining application that identifies frequently occurring patterns in a transaction database.
- Dedup: detects and eliminates redundancy in a data stream with a next-generation technique called deduplication. Is an enterprise storage kernel.
- Canneal: this algorithm minimizes the routing cost of a chip design with cache-aware simulated annealing. It is an electronic design automation (EADA) kernel.
- Ferret: search engine that finds a set of images similar to a query image by analysing their contents. It is a server application for content-based similarity search of feature-rich data.

3.3.2 HCA applications repository

HCA Applications are an application repository created for the Mont-Blanc project, which is under an European Unions Seventh Framework Programme for research. They were

mainly created to execute well known computational kernels and applications in highly parallel supercomputers based on ARM microprocessors featuring low power GPUs.

Kernels included in HCA applications are the following ones:

- Jacobi: It is a numerical linear algebra method translated to a stencil code kernel, which computes solutions for a system of linear equations diagonally dominant.
- Kmeans: is a popular algorithm for cluster analysis in data mining. k-means clustering goal is to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean.
- Knn: or k-nearest neighbours, is a machine learning non-parametric method used for classification and regression.

HPC Applications included in HCA applications repository are the following ones:

- Heat: is an implementation of an iterative solver for heat distribution. The application lets the user to choose between three solvers: Jacobi, Gauss-Seidel and Red-Black.
- Specfem3D: this application simulates a 3D seismic wave propagation in any region of the Earth based on the spectral-element method.

3.3.3 BSC applications repository

The BSC applications repository was set up as collaboration between different BSC departments to share real HPC applications ported to OmpSs programming model.

Used applications include:

- NPB-BT-MZ: ported from NAS parallel benchmarks. MZ designates that this benchmark has different versions that exploit different levels of parallelism. The used one makes uneven-size zones within a problem class and it increases the number of zones as problem class grows.
- Hydro: solves compressible Euler equations of hydrodynamics. Hydro is based on a finite volume numerical method using a second order Godunov scheme for Euler equations.

3.3.4 Custom developed applications

Given that Parsec benchmarks were not giving the expected results regarding changing the hardware prefetcher configurations. Custom developed applications written in OmpSs programming model also have been a task in this work. Nevertheless, reasons for their creation will be explained later in this document.

Custom applications developed include:

- Dotproduct: an OmpSs implementation of a dot product algorithm. This created benchmark was specifically created to test the hardware prefetcher in a very friendly environment. This is because dot product makes consecutive read accesses to two vectors. Given this algorithm's nature, the hardware prefetcher should be able to detect two read data streams during most part of the application execution.

- Martix multiplication: another interesting algorithm that access consecutive memory positions is matrix multiplication. Provided that our implementation accesses to the first matrix and the written matrix by rows and the second matrix by columns, the prefetcher should by able to at least find a read data stream for the first matrix, a write stream for the written matrix, when the prefetcher detects store data streams and, if the prefetcher is configured to detect accesses with strides, it could detect a read data stream in the second matrix.

Chapter 4

Probing the processor performance

4.1 Performance counters

As has already been commented, some way to analyse the processor performance is needed. Nowadays processors are increasingly becoming more capable to allocate microarchitectural components. Thanks to this, each newer generation provides a wider range of hardware counters; that is to say, the programmer can monitor more kinds of machine instructions and more of them at the same time. For example: the total instructions executed, how many L1 cache misses those instructions had, how many of them were floating point operations, etc. The benefits from using hardware counters against software instrumentation are meaningful. Low overhead in the instrumentation as well as no need to modify source code. On the other hand, drawbacks are their platform dependence and their restrictions when the programmer wants to monitor several events at the same time. To tackle these problems there are libraries that provide platform independence through an API; these libraries have can add support to future processors through updates.

In this work, we have chosen PAPI PAPI, (Performance Application Program Interface), library to do the processor metric extraction [26]. It has been our choice because of its wide portability, its support as well as its ease of use.

4.2 Adding instrumentation to Nanos++ runtime library

As it has been mentioned in section section 1.3 and section 2.1, the objective of this research is to be running most of the time with the best prefetcher configuration so as to gain performance and save power consumption. The first step to do this, is to be able to gather valuable information of the execution in terms of performance metrics. A good first step is to start by achieving maximum performance, so the highest level metric that can be used for this purpose is instructions per cycle. Using instructions per cycle (IPC), each processor core performance can be monitored. The more IPC a core has during a lapse of time, the better the performance. One thing that we are expecting to see in general, is a representative gain in the LLC hit ratio got by the smart use of the prefetcher.

4.2.1 Design and implementation

Having chosen the IPC as the performance metric to compare different prefetcher configurations, the point is to get that metric in each core. In the IBM POWER7 each SMT thread is represented as a core for the OS, so in this sense there is no big issue in telling the runtime we want to store performance metrics for each SMT thread as if it was a complete physical core. Because this work is built on top of Nanos++ RTL using OmpSs programming model, we want to get the most out of OmpSs, so each OmpSs task executed is monitored using PAPI calls and once ended, its IPC among other important information and metrics are stored in runtime fast data structures such as hash tables. At the time the program execution ends up, the runtime traverses those data structures summarizing each SMT thread performance for each kind of task the programmer had specified in the code. Suffice it to say that the proposed solution is flexible in terms of what performance metrics the user needs. To begin with, the gathered data are the IPC of all tasks executed in each SMT thread.

In this first stage of Nanos++, data are stored in a file in a straightforward format enabling other software to parse the file so as to plot data showing important insights about the execution. These plots will be explained in section 4.3 with further explanation.

Going back to Nanos++ instrumentation capable design and implementation, let's look thoroughly at how PAPI calls are inserted in runtime aiming to monitor performance counters only during the execution of the OmpSs tasks.
Figure 4.1 is a simplification of how to use hardware counters directly on top of Nanos++, suffice it to say that function names and parameters also have been simplified to help to understand their operation. First of all the runtime library (RTL) master thread has to initialize the PAPI library. This is done by calling PAPI_library_init(PAPI_CURRENT_VER) function, here it is important to notice that this function can only be called once so it is mandatory to find a place in Nanos++ where only master thread executes a region of code once during the user application execution. A good place to do it turns to be inside the associateThisThread method in Nanos++. It is only executed by the master thread and it is located in the processingelement file.
PAPI provides the programmer with what is called event sets, these ones are a way to define groups of performance counters so it is much easier to turn on and off different groups of performance counters as they are needed. Going on with the needed PAPI calls, each NANOS++ defined thread has to call PAPI_register_thread(), this is used to force PAPI to initialize a thread that PAPI has not seen before. Because we want to save each performance counter associated to a used thread during its execution, each thread needs to call PAPI_create_event_set(eventsetX)
and PAPI_add_events(eventsetX,eventsv) methods, where eventsetX is a pointer to reference an event set and eventsv is a vector that contains what events do we want to monitor. A reasonable place to call these calls in Nanos++ is in function smp_bootthread of smpthread file as this function is only executed once by all worker threads.
Now that the runtime has initialized PAPI correctly it is time to find how to start, stop and read the performance counters each time a task starts and stops executing. Figure 4.1 shows only the idea of how the system should behave. In practise, this part is much more complex because Nanos++ first uses several functions to start, change and stop tasks and second because a task can potentially be yielded to another task when the former one spins for some amount of iterations in the already mentioned idle loop.

Basically, Nanos++ controls tasks execution starts by means of three methods in smpthread file. They are inlineWorkDependent, switchTo and exitTo methods. The first one is used when one phase of one workdescriptor finishes and then another workdescrip-

tor phase is placed in the same thread to execute the last of its phases to finally go on with the first workdescriptor to execute more of its phases. Here, phases are regions of code where one thread executes only its code. For example: a thread could be executing its task code when a nested pragma would define multiple threads each one needing to start a new phase. The second method, switchTo, is executed when a phase of a workdescriptor that is not the last one finishes and a phase, (that can be the first one or any other one), of another workdescriptor start. The third method, exitTo, is similar to the second one with the difference that the phase of first workdescriptor is the last one. At this point it is only a matter to place PAPI start, reset, and read calls in the right order in these three functions. However, because all of these three functions are only called when there is a change from one workdescriptor phase to another workdescriptor, also a call to read performance counters is needed in the wordescriptor file. Here, the method finish is always executed when a workdescriptor finishes so in this sense it is the right place to do the last read of that workdescriptor and store counters into fast data structures. Finally, the first place to start reading counters is also the same as the one used to initialize each thread eventset because function smp_bootthread not only is executed by each thread but also is executed before the first phase, of each wordescriptor in a thread, takes place.

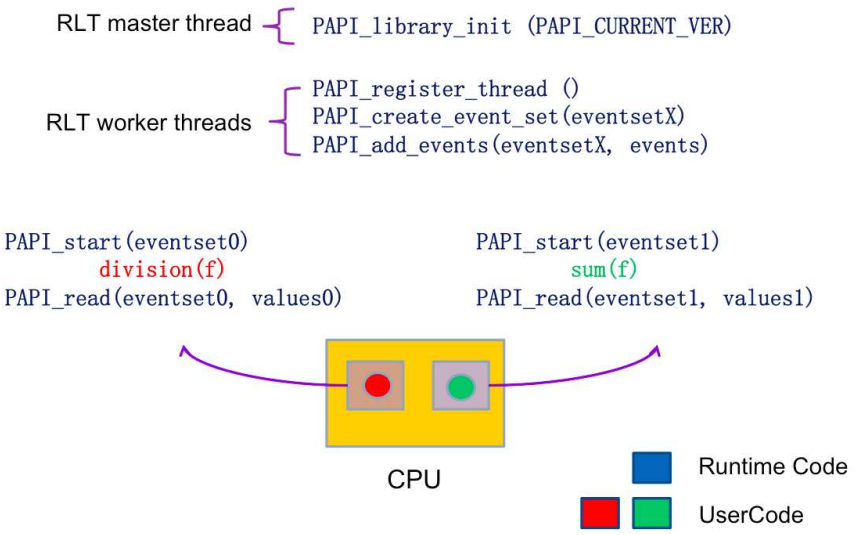

Figure 4.1: Nanos++ PAPI integration

Regarding data structures to gather metric statistics, they are mainly placed in the system class, which is the runtime singleton class. It has been the choice because this

class includes the whole system and is instantiated once during the whole application execution. A part from these general execution structures, each workdescriptor needs to be enriched with its hardware counters metrics because as it has already been mentioned each workdescriptor can have different phases and taking into account that a large system would contain many workdescriptors at the same time, it would be a bad idea to bulk data constantly to a single data structure in the system causing its contention because of the need to be locked and unlocked by each workdescriptor; so as it has been argued each workdescriptor representation has to contain those data as well. Suffice it to say that once a workdescriptor is ended, its representation as an object in the system is erased, therefore, even though a copy of its metrics to the general execution data structure is needed that doesn't represent replication of data.
Having defined a mechanism to pass down data from workdescriptors to the system, now it is only a matter of defining what data do we want to store; the first part of the implementation is getting the IPC, that is done through the use of PAPI calls. Nonetheless, any other metric such as the first three cache levels, load and store instructions or floating point operations are also a feature in this work implementation.
The most representative data structure used in this part of the implementation is very similar to a separate chaining hash table. The purpose of this data structure is to store and retrieve all tasks statistics classified by their task type very quickly so as to avoid runtime performance degradation. Figure 4.2 shows the operation of this structure. When a data tuple with performance counters values from a workdescriptor comes, a hash function is applied using its task type as the key, then contents are stored for their later retrieval also using task type as the key; in the figure each coloured bucket represents a different task type name. C++ standard library implementation may differ with respect to the explained data structure. Nonetheless, its functional operation is the same.

Before executing any set of benchmarks, it is of vital importance to prove that the resultant version of Nanos++, which now is capable of monitoring hardware performance counters, doesn't add considerable overhead as a future gain in performance by adapting the hardware prefetcher, (or other features in future), to the application requirements could well be impaired by that overhead. We choose a demanding benchmark in terms of computation with a great amount of OmpSs tasks in order to put pressure to the runtime. In this case, figure 4.3 shows time each Nanos++ version takes for different number of processors executing Parsec fluidanimate benchmark in MareNostrum III. In the experi-

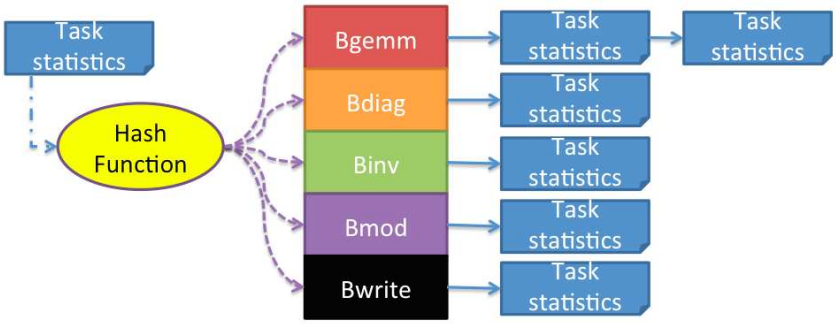

Figure 4.2: Separate chaining hash table for tasks statistics

ments SMT mappings were done so that each thread was alone in one of the cores; in this way the SMT processor feature was not exploited. On the other hand, the right y-axis represents the relative percentage of time increase of the instrumentation capable Nanos++ with respect to normal Nanos++. As it can be appreciated, the newer version of Nanos++ doesn't represent a significant reduction at all in terms of performance because at most is only around 1.75% slower than normal Nanos++.

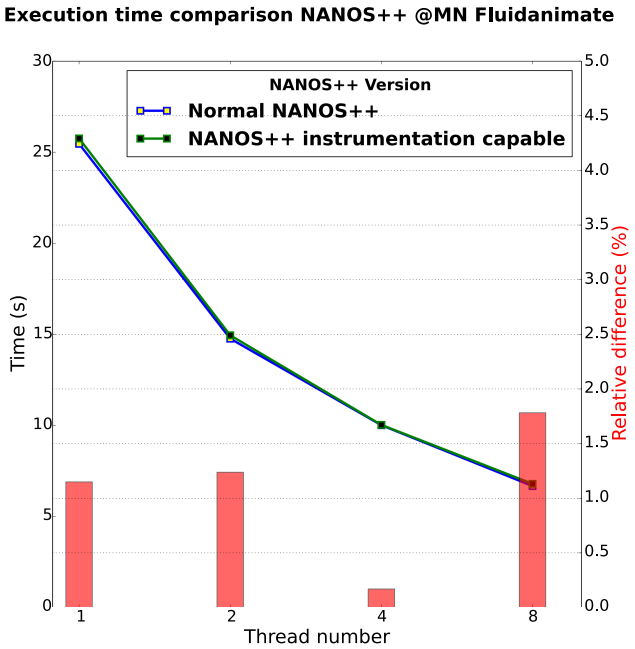

Figure 4.3: Nanos++ VS Nanos++ instrumentation capable execution times

4.2.2 Extracting performance statistics with Nanos++ instrumentation capable

This created Nanos++ version, capable of extracting performance counters, is a first step to achieve a version that can actually decide by itself the best prefetcher configuration dynamically. So in this sense, this first attempt can be tagged as static, because given a configuration of the prefetcher set beforehand, it allows the user to get statistics from all available performance counters of a given processor. In this case, the IBM POWER7 processor. To begin with, Nanos++ instrumentation capable, can be compiled in its original four versions; performance, instrumentation, debug and debug-instrumentation. However, if we'd like to get statistics, the performance version needs to be included. After having our Nanos++ installation, we can run applications getting or not statistics without needing to compile again anything. This is achieved through the use of runtime parameters while launching applications. Table 4.1 shows how to use Nanos++ instrumentation capable. Performance counters should be specified in PAPI format. For example PAPI_TOT_INS or PAPI_TOT_CYC, to request total instructions and total cycles of each OmpSs task. For each performance counter requested, a file with statistics will be generated. File name has the format of #threads.prefetcherconfiguration.performancecounter; (additionally, it can have an instance number to tell it apart from other executions). Regarding file's data format; the structure is one that writes a line for each OmpSs task type, separating fields by colon. Starting with the thread number and task name, each line contains a given performance counter for each task instance of that task type. In this sense, it makes straightforward to treat data with a-posteriori scripts.

Option	Description
–use-papi	requests the runtime statistics from performance counters; if not specified no statistics are requested
–papi-counters	each PAPI counter requested needs from an addition flag; PAPI format should be used to write performance counters

Table 4.1: Nanos++ instrumentation capable options for static prefetcher configuration

4.3 Results

This section is done in basis of a set of benchmarks from Parsec benchmark suite showing different performance metrics for each one of them. At this point, the purpose of these benchmarks is basically to measure IPC for each task type and each SMT thread; by doing this we will be able to see how benchmarks behave in multicore environments.
Machines used for this experiments have been an IBM BladeCenter PS701 and MareNostrum III supercomputer. So before going into any benchmark, is highly recommended to understand the underlying platforms. Chapter 3 gives detailed information about the used platforms.
Results of Nanos++ instrumentation capable are showed and commented for each microprocessor: the IBM POWER7 and Intel's Sandy Bridge-EP E5-2670. Each one assembled in one of the two already mentioned platforms. The former one is more oriented to get performance out of its multithreaded cores whereas the later one offers more resources to fewer threads.

4.3.1 IBM POWER7

At this point, Parsec benchmarks are run on top of an IBM BladeCenter PS701 which contains an IBM POWER7 processor. Here, the IBM POWER7 processor is of special interest because it has much more potential for further steps in adaptability as its architecture is highly focused on resource sharing. For example: IBM POWER7 microprocessor has 4 SMT threads per core enabled and it has a configurable hardware prefetecher. Further specifications are thoroughly explained in sections 3.1 and 3.1.1.

Let's start defining benchmarks and analyse their results. To begin with, what we finally will achieve, in this section and in the next one, is to monitor the IPC for each OmpSs task type and for each core number execution; this way it will possible to see algorithms' behaviours broken down into OmpSs task types depending on the parallelism requested. The benchmark suite chosen for this purpose is Parsec Benchmark Suite not only because its representativeness in terms of applications, but also because of its multicore support implementations as well as there are a great number of them translated

OmpSs programming model.

Figure 4.4 shows IPC variability when running each benchmark with 4 different configurations: 1, 2, 4 and 8 cores. Boxplots contain IPC of all task types merged for each thread number execution. Remember that each thread is mapped to a single processor core. All plots contain data from 4 executions so as to minimize possible noise; for each task IPC, the used value is the average of all four executions. The reading of boxplots going bottom to top is that of minimum value, first quartile, median, third quartile and maximum IPC.

The first aspect that stands out in figure 4.4 is that, in comparison with the same experiment in a MareNostrum III blade, figure 4.8, the POWER7 obtains around half of the performance of the Sandy Bridge in all benchmarks and regardless of the amount of cores used. This is not worth it focusing on to; firstly because they are products launched with a gap of two years and secondly because there are a lot of parameters that affect both execution performances, such as used compiler, power consumption, core complexity, market targeted, etc. Nevertheless, the interesting point here is that IPCs variability follows a similar pattern in all benchmarks but in swaptions with respect to MareNostrum III executions. That is giving the information that somehow applications' behaviour on top of both architectures tends to be slightly similar. Nonetheless, let's go more into detail with POWER7's performance: blackscholes and swaptions benchmarks don't suffer from IPC variability whatever thread number configuration and that is a consequence of having only one OmpSs task in their codes. In this sense, the same kind of task gets the same performance from the POWER7's pipeline regardless the parallelism requested, which is something reasonable and a good start to support heavier workloads.

Regarding bodytrack, fluidanimate and freqmine benchmarks; their variability goes according to their higher number of different OmpSs tasks. Bodytrack's IPC variability in POWER7 is lower than in Sandy Bridge-EP E5-2670 but the median is still around half of it. A possible explanation for that is that while Sandy Bridge cores offer more complexity for a single thread, this advantage cannot be always exploited and therefore we appreciate in figure 4.8 that higher variability. Another interesting benchmark here is freqmine. Is the only benchmark that has a low regular IPC with spikes across all thread number configurations. As it will be shown in next figures, this is caused by one of the

OmpSs tasks, resulting in this spikes in the whole application execution.

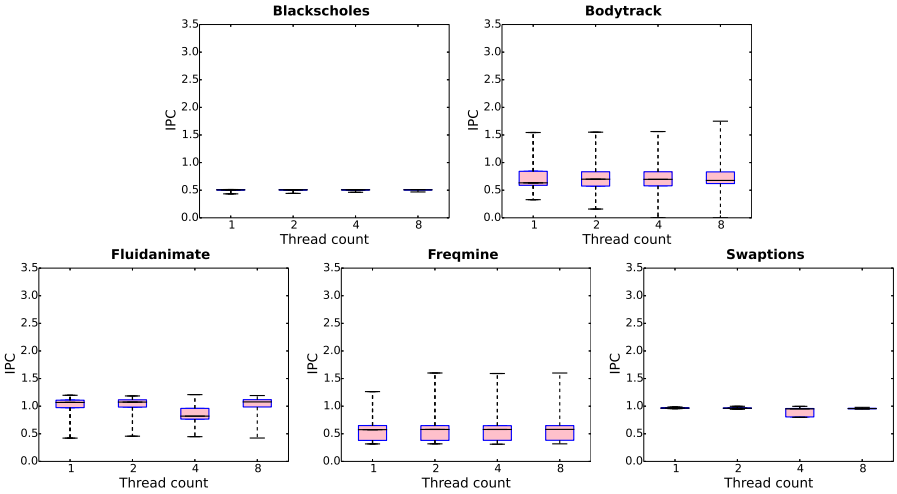

Figure 4.4: IPC variability when running with 1, 2, 4 and 8 cores on IBM BladeCenter PS701

Executions are also shown in figures 4.5 and 4.6. However, these ones split IPCs according to each task type. As it has already been commented, this is an interesting idea as it provides finer granularity to study applications' behaviour. For example: applications usually have different methods associated to OmpSs tasks where each task can put pressure to different processor's shared resources. So in this sense, this finer granularity can help to construct an integrated algorithm in the runtime because, having this insights, the integrated algorithm could better distribute OmpSs task types across different SMT threads, which, depending on used shared-resources, should or should not be mapped within the same chiplet.

In figure 4.5, bodytrack benchmark IPCs go from 0.5 to 1.5. The comparison with the Sandy Bridge execution unveils that some tasks perform much better in Sandy Bridge whereas other ones have a similar IPCs. This reveals that Sandy Bridge has some architectural components that accelerate certain kind of operations or operation series whereas other tasks are more limited by some bottleneck in the architecture. On the other hand, fluidanimate benchmark performs kind of the same in POWER7 than in Sandy Bridge, tough with half of the IPC. Regarding freqmine benchmark, although Sandy Bridge ex-

ecution doesn't show spikes, POWER7 has them in one task type. However, if we have general look to all tasks types, the Sandy Bridge delivers more performance. In this sense the POWER7 presents some spikes of high performance for one task but the Sandy Bridge positions its average IPC in the POWER7 spikes while having presenting some long third quartiles, which somehow denote a drop in the pipeline.

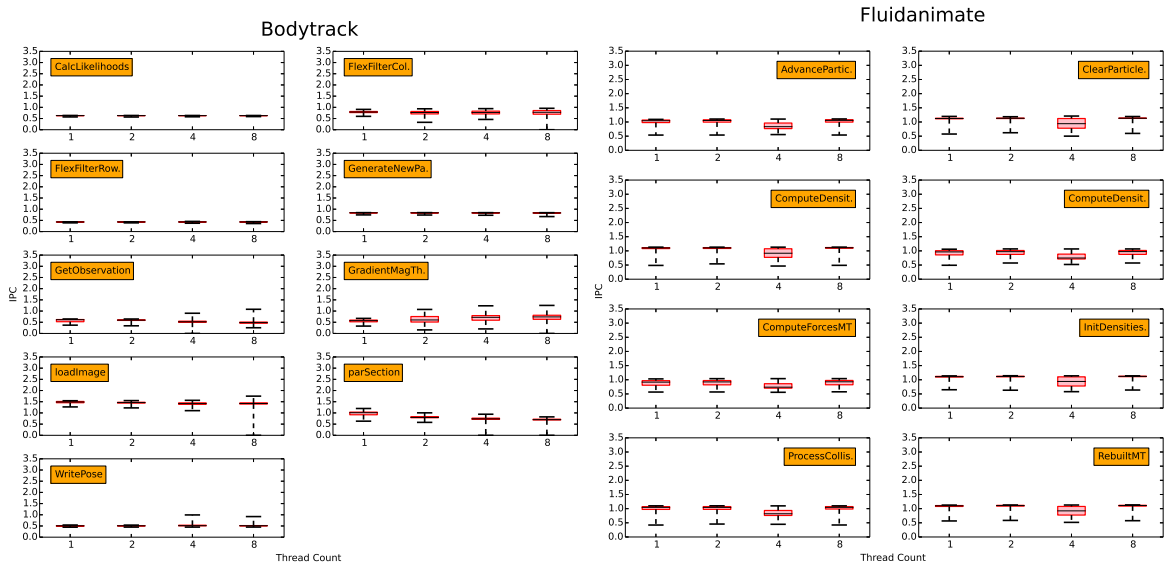

Figure 4.5: IPC variability when running with 1, 2, 4 and 8 cores on IBM BladeCenter PS701

Another interesting experiment with Nanos++ instrumentation capable, which is a step to reach the final goal, consists in relating each task IPC with its number of instructions executed. Figure 4.7 contains all Parsec benchmarks executed previously with this information. The way to show it is through a two dimension histogram, here there is no distinction between different task types. However, the aim of this experiment is that of finding correlations between IPC and number of instructions per task in all benchmarks. Their comparison with Sandy Bridge execution, figure 4.12, is an excellent example of how different ISAs generate different number of instructions executed per task. Notwithstanding this observation and different IPC performances in both microprocessors. Patterns observed for each algorithm are very similar in both cases. This observation can be a used as an statement to rely on when developing policies to change hardware dynamically bearing in mind that they will work well in different architectures. However, this

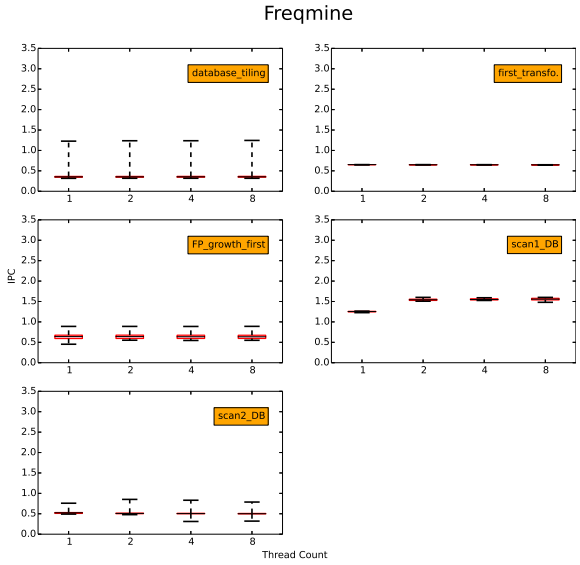

Figure 4.6: IPC variability when running with 1, 2, 4 and 8 cores on IBM BladeCenter PS701

observation is out of the scope at the moment and should fit more in future's work.

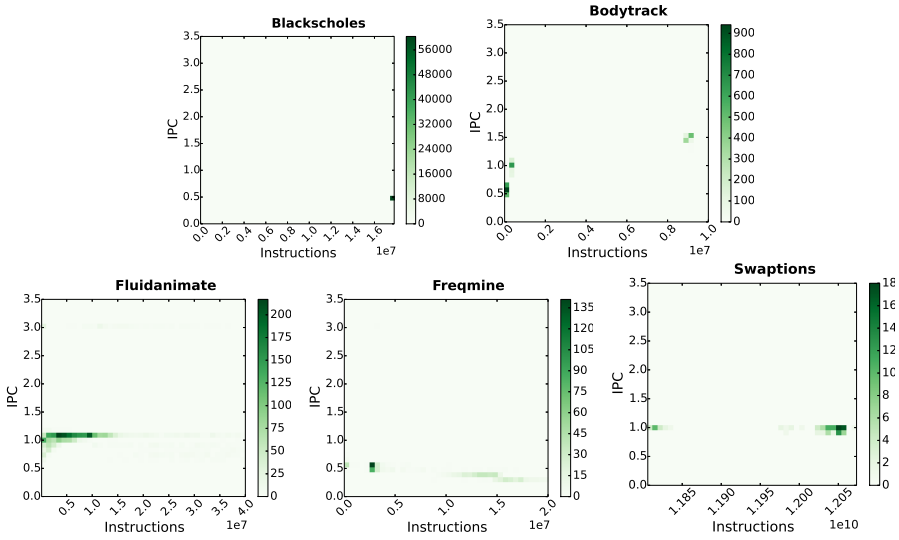

Figure 4.7: IPC VS instruction number heatmap on IBM BladeCenter PS701

4.3.2 Intel Sandy Bridge-EP E5-2670

Looking at figure 4.8, at first sight, it be seen that that whereas some benchmarks experience a rather regular IPC: blackscholes, canneal, dedup and swaptions, others such as ferret, fluidanimate and freqmine suffer from more variability. IPC mean and variability will be keystone in further implementation steps to achieve hardware adaptivity; by taking them into account, it will be possible to analyse the behaviour of multiple algorithms or different algorithm instances in different SMT threads mapped to the same core. That is to say getting the most of processor's resource sharing feature.

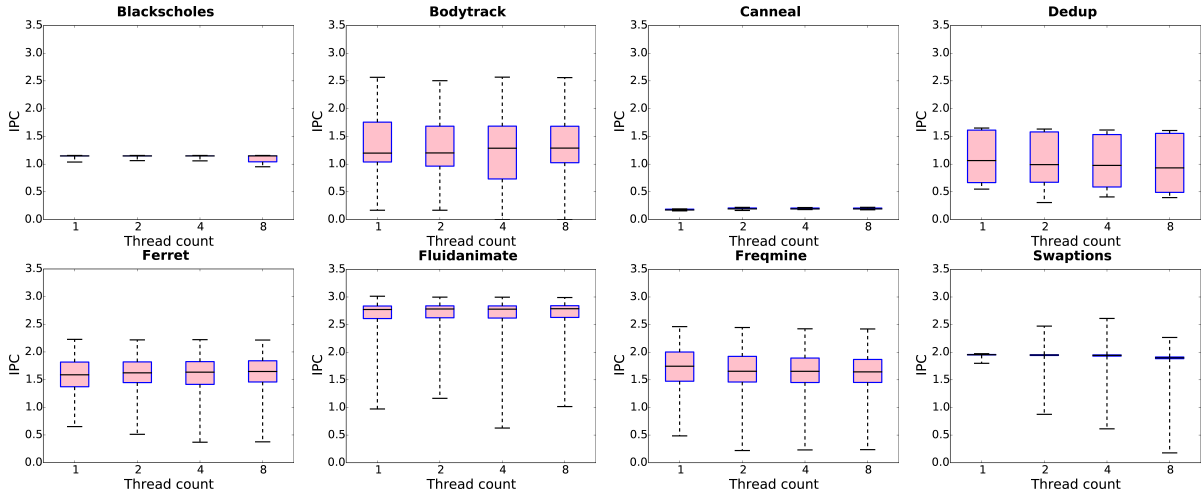

Figure 4.8: IPC variability when running with 1, 2, 4 and 8 cores on MareNostrum III

When it comes to do a break down of applications' behaviours, figures 4.9, 4.10 and 4.11 provide IPC values for each task type of each application. As it can be appreciated, some application benchmarks have similar IPCs regardless of task type whereas other show wide gaps between different tasks types. Examples of similar IPCs across different task types are: fluidanimate and ferret benchmarks. This low variability depending on the task type in some application benchmarks can be attributed to the fact that they have been ported so that different methods with an assigned task were computing very similar kernels or that they ended up using approximately the processor pipeline with the same efficiency. On the other hand, other benchmarks such as dedup and freqmine experience more variability depending on each task type. This can mean that different task types perform very different actions in the machine. A good example can be found in dedup

benchmark where the first task is data compression whereas the second one, that has less IPC, somehow consists of writing that compressed data. An isolated case is bodytrack benchmark; not only has IPC variability across different task type but also very different variability depending on the thread number. This variability when splitting by task type is contained in the large quartiles that appeared in figure 4.8.

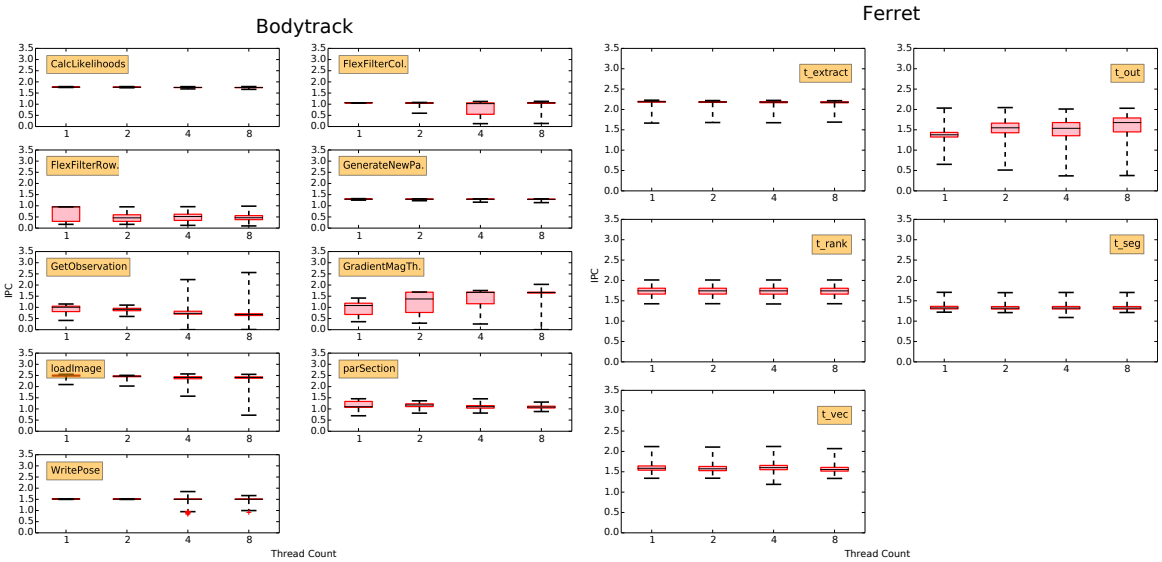

Figure 4.9: IPC variability when running with 1, 2, 4 and 8 cores on MareNostrum III

As for the two dimensioned histograms, at first glance, intuition suggests to think that the greater the number of instructions executed per task the grater the IPC because of processor's pipelined design. In figure 4.12, fluidanimate benchmark, the one that showed IPC variability across different task types and processor number, behaves according to the first impression.

However, it can be seen through different benchmarks that this intuition becomes somehow too simplistic as a general rule of thumb. For instance: canneal, swaptions and dedup don't show this direct correlation. A possible reason for this behaviour can be attributed to the fact that the processor's pipeline, depending on the workload tasks entail, is unable to benefit more form being superscalar even when OmpSs tasks are bigger.

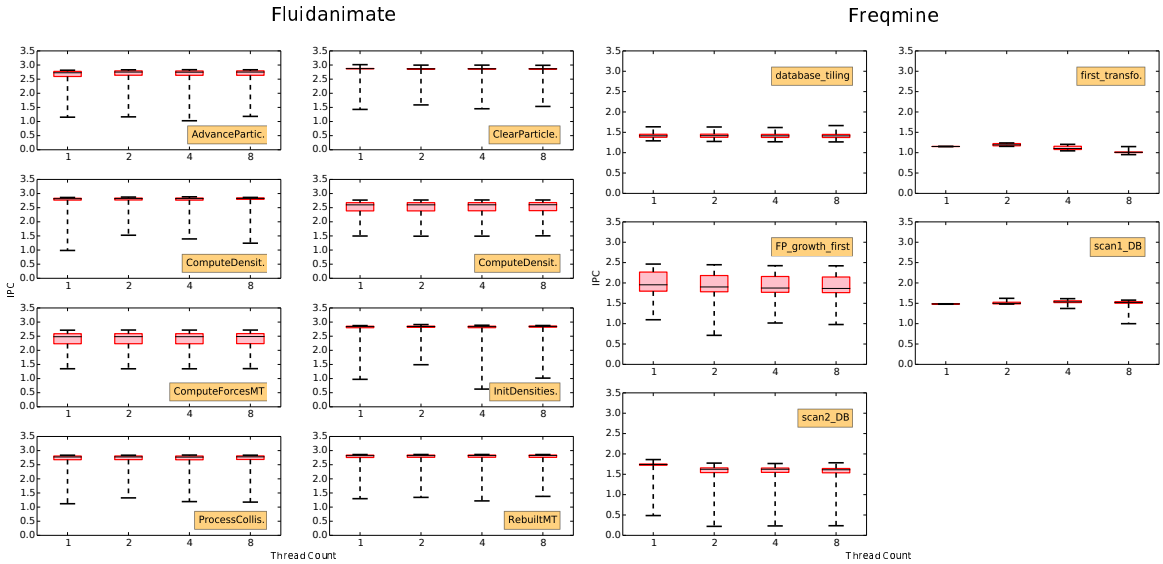

Figure 4.10: IPC variability when running with 1, 2, 4 and 8 cores on MareNostrum III

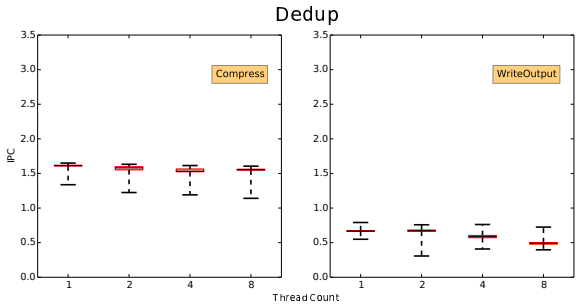

Figure 4.11: IPC variability when running with 1, 2, 4 and 8 cores on MareNostrum III

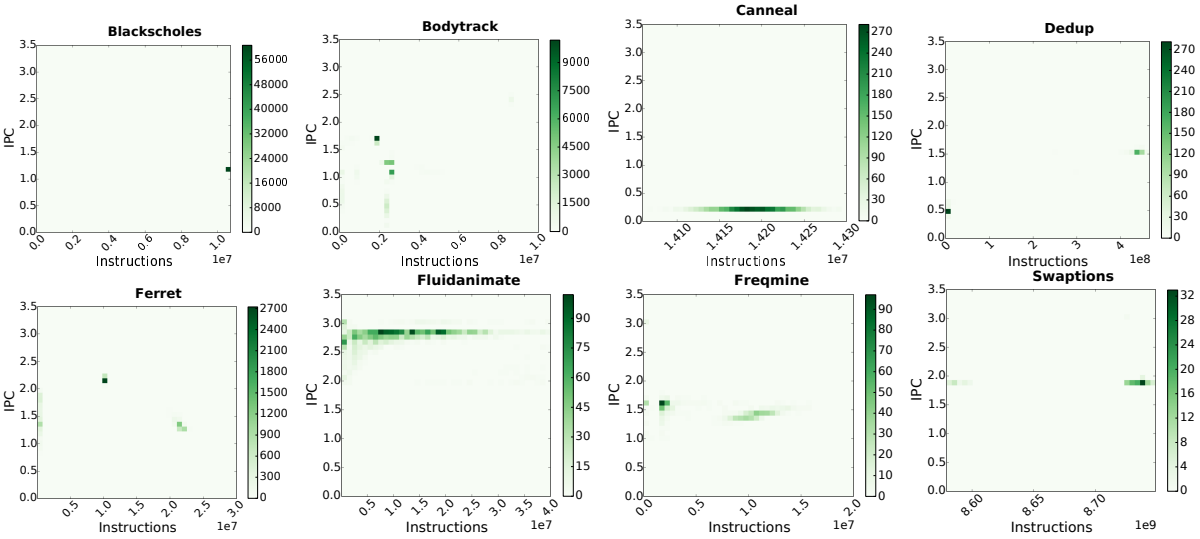

Figure 4.12: IPC VS instruction number heatmap on MareNostrum III

Chapter 5

Runtime adaptability

This chapter explains run-time implemented ability to capture how hardware has to adapt to user applications to maximize performance; so in this sense, run-time adaptability is the fact that it chooses the best hardware configuration during user applications' execution.

5.1 Work granularity

This section focuses on work granularity; that is to say, when to take measurements of hardware configuration or how much work executed take into account for measurements. That leads us to define the concept of work unit, which is a lapse of time run under certain hardware configuration. A previous study from Victor et al. [24], used fixed time for work units and for defining a region of time in which to run with best found configuration. The idea presented in his work was to define two phases: one running 10ms each hardware configuration, with different prefetcher configuration, and a second, leaving 100ms to be running under the best configuration found in the former phase.
From the point of view of this work, this methodology can reap good results. However, the mentioned methodology adds blindness in the search of the best configuration as it always takes the same granularity to define the mentioned work units. Choosing 10ms is based on some empiric results and although it may work, it is rather a small lapse of time which loads system with control code execution and causes overhead; given that this

work is based on a runtime rather than modifying OS kernel and that it aims to be highly scalable, the idea is to benefit from OmpSs programming model to define a work unit as an OmpSs task.

5.2 Developing Hardware adaptive Nanos++ runtime library

As it has been defined in section 1.3, the final part of this work is to get the runtime to configure the processor automatically so as to be running the maximum time with the best configuration of the prefetcher, at that point Nanos++ will be considered as hardware adaptive. For this, a similar operation as in Victor et al. Work [24] is defined; mainly, the idea is to mark execution time with two different phases that keep alternating; one gathers statistics, the other runs the application with the best configuration found. The point in this work, is that workloads are parallel running on top of a multicore, in this case, processor IBM POWER7, so the use of OmpSs task based parallelism as an environment, which perfectly fits in parallel architectures, makes very convenient to make work units as OmpSs tasks. This way, comparisons between different configurations can be made during application execution for each OmpSs task type, thus enabling different prefetcher configurations for potentially different computational requirements.

5.2.1 Design and implementation

Thanks to Nanos++ run-time modularity, it is straightforward to place implementation of the logics that will take care of when to switch form one phase to another for each OmpSs task type. The chosen method is the one that starts each OmpSs task within a workdescriptor. That is to say, start method in the workdescriptor file. The algorithm structure presents the scheme showed in figure 5.1. When a task is to be executed and runtime passes through start method; the first thing to bear in mind is to look for the previous existence of that task type. If this is the first time this task type appeared, the task type should be registered in a structure in the runtime and let it to be executed in exploration

mode. Contrary, if the task type already existed, the algorithm should check out if that task type was ordered to be executed in exploration mode or in stable mode. Although the task will be executed, a counter will be checked to determine if next time this task type needs to execute again, it will be executed in exploration or in stable mode.
An optimization done in the algorithm is one that keeps track of which processing elements have already executed a given task type in stable mode, without any other processing element having executed that task type in exploration mode. This is a way to avoid checking or changing the DSCR register in the cases we know that processing element already had the best configuration set. Let it suffice to say that checking or changing the DSCR is expensive as the register is represented in RAM memory and therefore is a thread of detriment to the overall performance.

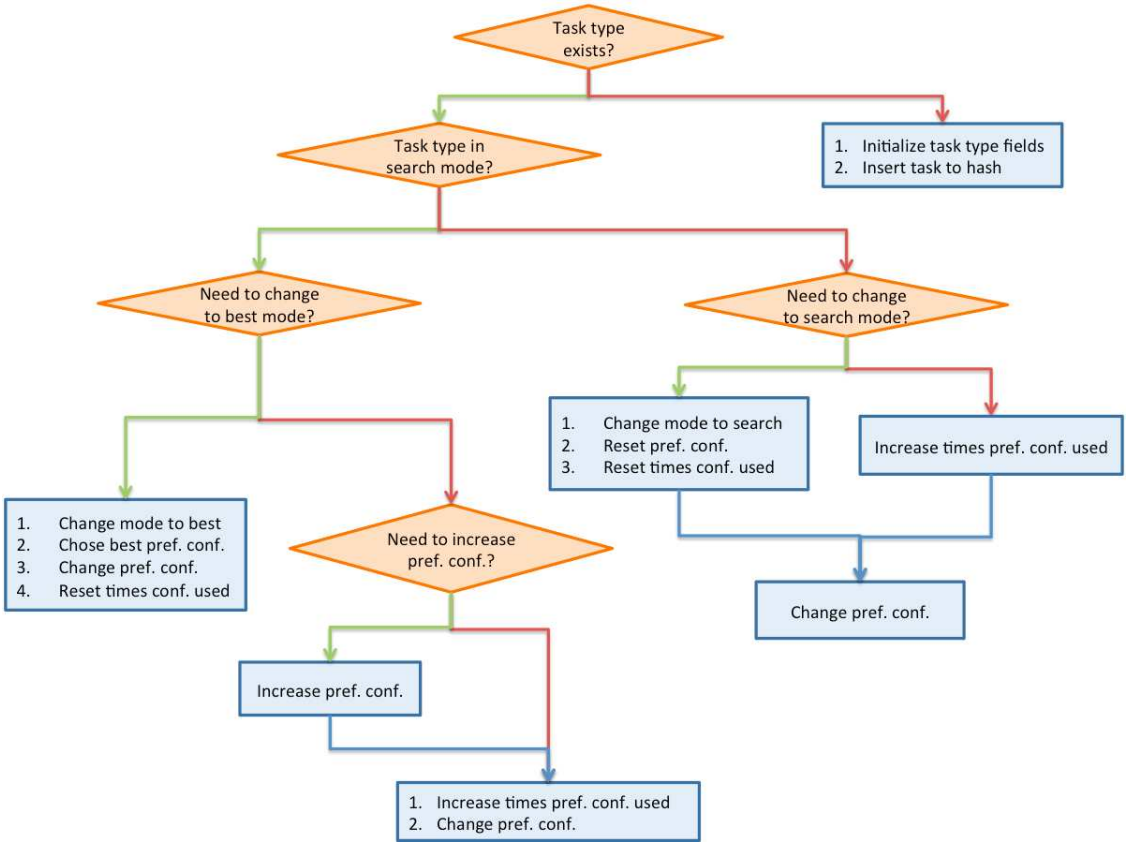

Figure 5.1: Nanos++ flow diagram of each new task executed by a hardware thread

Regarding DSCR management, each hardware thread DSCR readings and writings

are done through the runtime. Because DSCR is a feature of the IBM POWER7 architecture, DSCR reading and writing functions are placed in smpthread.cpp file. This file is not located in the Nanos++ core, which is the common directory for all kind of devices.
Special attention is deserved to the placement of user requested threads to the IBM POWER7 hardware threads. The IBM POWER7 has 32 hardware threads, 8 cores and therefore 4 SMT threads for each core. With this layout, a first approach could be to pin each requested thread consecutively to each hardware thread. However, to squeeze all processor computational capacity, it is more efficient to pin the first 8 threads to each processor core and once all cores have been assigned with one thread, start distributing others to SMT threads across cores in a round robin fashion. With the already mentioned thread layout a formula to fairly distribute threads across cores is equation 5.1

$$Mapping = \lfloor(((x \bmod 32) \times 4) + ((x \bmod 32)/8) \bmod 32)\rfloor \tag{5.1}$$

This way, even if we had more than 32 threads, they would be placed in the same round robin fashion adding concurrency to their corresponding hardware threads.
Let it suffice to say that formula 5.1 is useful inside the runtime. With it, the runtime is able to change each DSCR thread values according to the accorded mapping. Nonetheless, it is the user's responsibility to launch the application with the described mapping; for example, by using taskset command. In other works, the user has to provide the mapping to the operating system whereas the runtime uses the formula 5.1 to read and write each thread DSCR according to the described mapping, which sorts a number of threads requested across the processor's cores and hardware threads in a round robin fashion.

5.2.2 Using Hardware adaptive Nanos++

Hardware adaptive Nanos++ adds intelligence to the previous version, Nanos++ instrumentation capable, as it finds by itself the best prefetcher configuration for each hardware thread associated to an OmpSs task type during execution time. Moreover, it can modify its decisions during execution time as multicore environments are very likely to change their best setup due to variable computational workloads.
Regarding the use of this mechanism, basically, the user doesn't need to do anything else

rather than to link the application with the modified version of Nanos++. This way, it is the runtime which constantly gathers performance metrics and adapts hardware to the most efficient configuration. Table 5.1 shows options of Hardware adaptive Nanos++. It is keystone to understand that this Nanos++ version can make longer or shorter its exploration and stable phases, so depending on machine's workload, finding the best configuration may take more measures in the exploration phase. Moreover, system global phase changes can be more common for example when executing different applications on the multicore or when an application have a dependencies graph with lots of computationally different task types, so a shorter stable phase can also be more appropriate.

Option	Description
–num-ps	tells the runtime how many times a task is executed in search, (exploration), mode before going to best, (stable) mode
–num-pb	tells the runtime how many times a task is executed in best , (stable), mode before going back to search, (exploration), mode.

Table 5.1: Hardware adaptive Nanos++ options for dynamic prefetcher configuration

5.3 Evaluation

Benchmarking section for Hardware adaptive Nanos++ development has several objectives: first it aims to deepen in the performance analysis of Nanos++ instrumentation capable, which needs static configuration of the prefetcher beforehand. After measuring static configurations, optimal parameters for the adaptive mechanism of Hardware adaptive Nanos++ will be found out. Moreover, there also will be a study to compare Hardware adaptive Nanos++ with the instrumentation capable version. For this, a set of plots involving a wide range of OmpSs benchmarks and comparing different prefetcher static configurations with the dynamic/adaptive mechanism will be shown. Finally, Hardware adaptive Nanos++ speed-ups will be put on the spot with a study to analyse what is the mechanism behaviour depending on the algorithms computational requirements as well as their level of parallelism requested.
When looking at the plots, for each application, the basic following outcomes will be

encountered in the analysed metrics: applications get benefit in a given metric when enabling the prefetcher are called prefetcher-friendly. Applications that get the same values in a metric regardless the prefetcher is disabled or enabled regardless of the enabled configuration; these are called prefetcher-insensitive applications. Applications that although have an improvement by using the prefetcher, don't experience difference between different configurations when the prefetcher is enabled; these are called configuration-insensitive. Applications that notice an appreciable difference when changing the configuration with prefetcher enabled; these are called configuration-sensitive. Finally, applications that suffer from a decrement of performance when enabling the prefetcher are called prefetcher-unfriendly.

5.3.1 Static configuration

This section shows the results of different prefetcher configurations for a sort of benchmarks from different suites, so here Nanos++ instrumentation capable is used for this purpose. The goal of this study is to analyse the impact of different prefetcher configurations and to get robust results that can be used later as a reference to the prefetcher reconfigurability mechanism, which is included in Hardware adaptive Nanos++.
Here, the effort is put into compare speed-ups for different configurations, average used bandwidth and amount of last level cache misses per thousands of instructions.
Bear in mind, that the methodology to chose the best prefetcher configuration for the static configuration makes first the user to change the prefetcher configuration on their own before running the whole application. Repeating this step until all wanted configurations are tested thus ending up having what configuration performed best in average for the whole application's execution. In this sense, static version is coarser than dynamic version when comparing prefetcher configurations. Remember that Hardware adaptive Nanos++ is able to compare prefetcher configurations at OmpSs task type level instead of being limited to make an average of the whole application's execution.

Initial experiments using the Parsec benchmark suite, didn't come to fruition as far as speed-ups are concerned when enabling the hardware prefetcher. Figure 5.2 shows this first failed attempt, where execution times are normalized with respect to configuration in which the prefetcher is disabled. In other words, speed-ups got by setting the

prefetcher enabled working with some configuration against not having the prefetcher enabled. Benchmarks had been very recently ported to OmpSs presenting bugs for the POWER architecture and in most of the cases their understanding wasn't ripe enough. We decided to try SPEC benchmarks, which are single-threaded and they are not ported to OmpSs programming model yet. Although they were not matching our needs of testing multithreaded codes nor having support to OmpSs yet, we got quite important speed-ups in many of those benchmarks. Having verified that, at least with single-threaded codes, the hardware prefetcher was useful for some codes, we decided to include other application suites ported to OmpSs already tried in other external projects and also to develop a couple of custom ones.

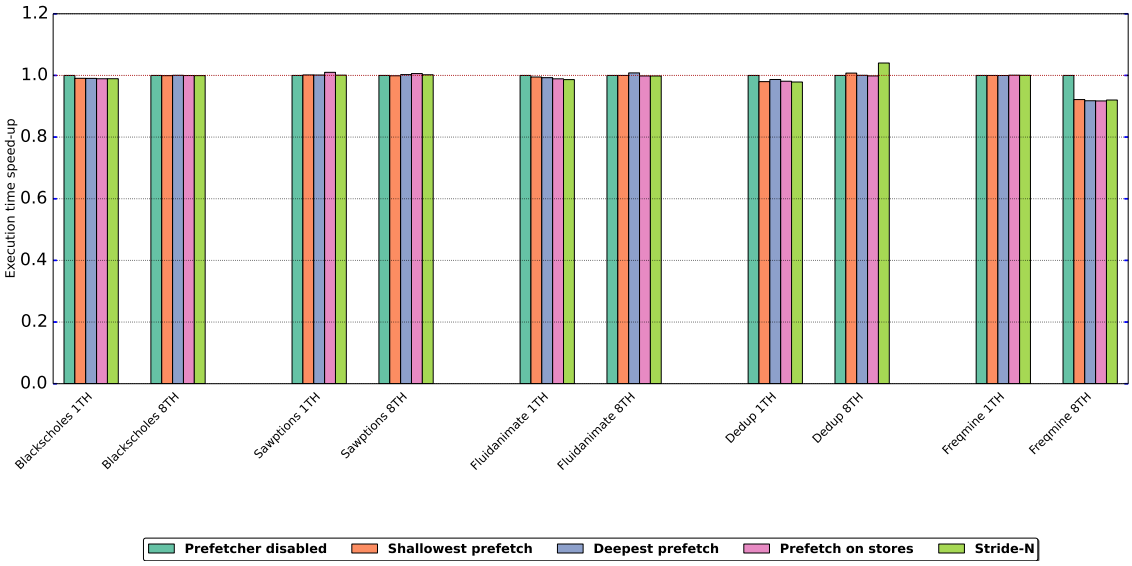

Figure 5.2: Parsec suite benchmarks, execution times for different prefetcher configurations

Next experiments show results for custom developed applications, HCA applications repository and BSC applications repository. Figures 5.3, 5.4 and 5.5 show normalized execution times with respect to configuration in which the prefetcher is disabled. That is to say, speed-ups of having the prefetcher enabled working with some configuration against not having the prefetcher enabled. Algorithms graphics are divided in different graphics because clarity and because they are obtained from different sources. Figure 5.3 contains two custom typical applications: dotprod and matrix multiplication. Figure 5.4 has appli-

cations from BSC HCA group. They are typical computational kernels as well as some interesting High Performance Computing applications. Figure 5.5 includes applications form the BSC repository which has a wide sort of codes ported to OmpSs programming model.

To begin with, let's have a look to figure 5.3 in which 10 different configurations are tested for each algorithm: using a single thread, using one thread per core, using all available hardware threads, launching 1 instance of the same algorithm per core requesting a single thread each one, as well as requesting 4 threads per core in each algorithm instance. All configurations are repeated for a bigger input size that doesn't fit into processor's last level cache. Remember here that thread mappings have been done in a round robin fashion with respect to cores and hardware threads inside each core.
Prefetcher configurations are explained in detail in section 3.1.1. To start with, it is straightforward to appreciate important speed-ups when enabling the prefetcher in both algorithms and across all possible configurations. However, at the same time, a considerable observation is that there is not much difference in the speed-up when changing prefetcher configuration within the same parallel demand. This means, these two algorithms have their load streams well detected by the automatic hardware prefetcher. Nevertheless, there is only difference between different configurations when the prefetcher is set to be shallowest; in other words, when it brings less data and thus, in general, doesn't exploit all possible benefit from spatial locality.
Beyond these main observations, there's more significant information in the plot. For example, dotproduct algorithm collapses its performance improvement when requesting all hardware threads for a single application instance. As long as this phenomena is also happening when instantiating an application for each core and requesting all 4 hardware threads for each core, it means there is contention somewhere between the processor's execution units and the main memory. Going on with dotproduct, it is clear that 8 MB experiments are fitting into the LLC, hence speed-ups due to prefetcher enabled are quite smaller than in the 80 MB experiments.
Regarding matrix multiplication algorithm, there's not much difference depending on different parallelism demands. Here, despite only having a single kind of OmpSs task, like in dotproduct algorithm, the prefetcher finds it harder to find and manage load streams. Special attention is deserved to 8 MB configuration. Although having less potential for

an improvement due to the use of prefetcher and showing it in the first three configurations, their vertical scaling configurations show greater speed-ups than the 50 MB configurations. This observation doesn't imply that the prefetchehr finds it easier to find load streams when dealing with multiple instances instead of the same application at the equivalent level of parallelism but a more reasonable explanation could be due that this is due to the possibility to execute more critical tasks of dependency graph replicas at the same time.

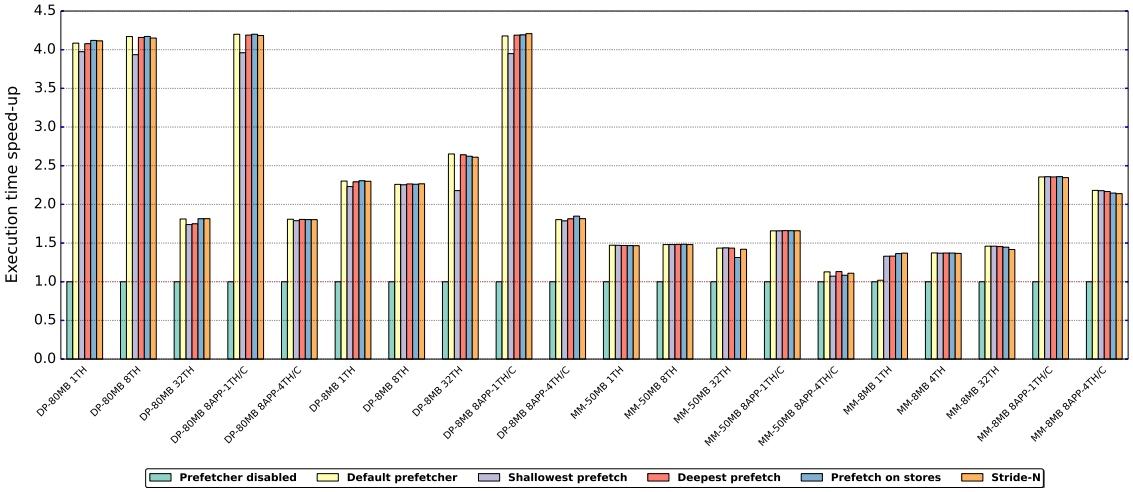

Figure 5.3: Custom applications, execution times for different prefetcher configurations

Figure 5.4 shows more discreet speed-ups across all applications and their configurations. All applications but heat suffer from the same phenomena as in the previous plot. They do not notice a difference in their speed-ups when changing the prefetcher configuration in enabled mode, this is they are configuration-insensitive. Here kmeans and knn algorithms are prefetcher-insensitive and although further tests were done increasing input size far bigger than the LLC size, (32 MB), results were following the same pattern. Special attention is deserved to specfem3D application. This one is prefetcher-friendly but configuration-insensitive. One difference of specfem3D with respect the other applications is that the former one is more complex and more representative of a HPC application, which is an encouraging result for future studies.

Figure 5.5 shows similar speed-ups with respect to figure 5.4. So in this sense, this gives us a notion of what level of speed-ups are we going to get by using current OmpSs

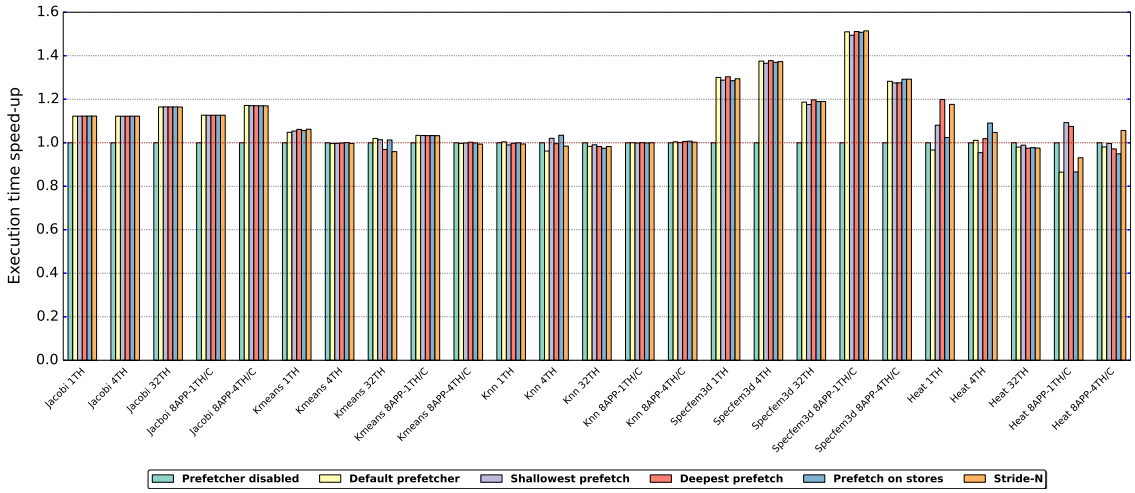

Figure 5.4: HCA applications, execution times for different prefetcher configurations

appplications. Here, a multizone NAS benchmark and a ported hydro to OmpSs are shown. At first glance, it can be seen how 200 MB input size gives slightly better speed-ups than 50 MB input size for NPB-BT-MZ benchmark. Although telling apart different sizes in bars within the same parallel configuration, the difference is so small that all parallel configurations can be considered as configuration-insensitive. In the case of hydro benchmark, it is important to notice a drop in the performance in the third configuration; where 2 MPI processes use all available hardware threads, collapsing the machine. The last two configurations were causing irregularities in the machine due to surpassing machine's available memory, so they can't be considered as 100% reliable.

Figures 5.6, 5.7 and 5.8 show normalized bandwidths with respect to prefetcher disabled configuration in terms of GB/s.

Starting with figure 5.6, the initial idea is that we should see an increment in the used bandwidth as long as we are more demanding in terms of parallelism with the POWER7. According to these expectations, results show used bandwidths increments in terms of horizontal and vertical scaling. For example: 80 MB dotproduct experiments show a 5,5x increment when requesting 8 threads instead of a single thread. This a very reliable indicator that shows the machine is being pushed by OmpSs ability to exploit application's parallelism. However, even more interesting are the vertical scaling results. In the 80 MB

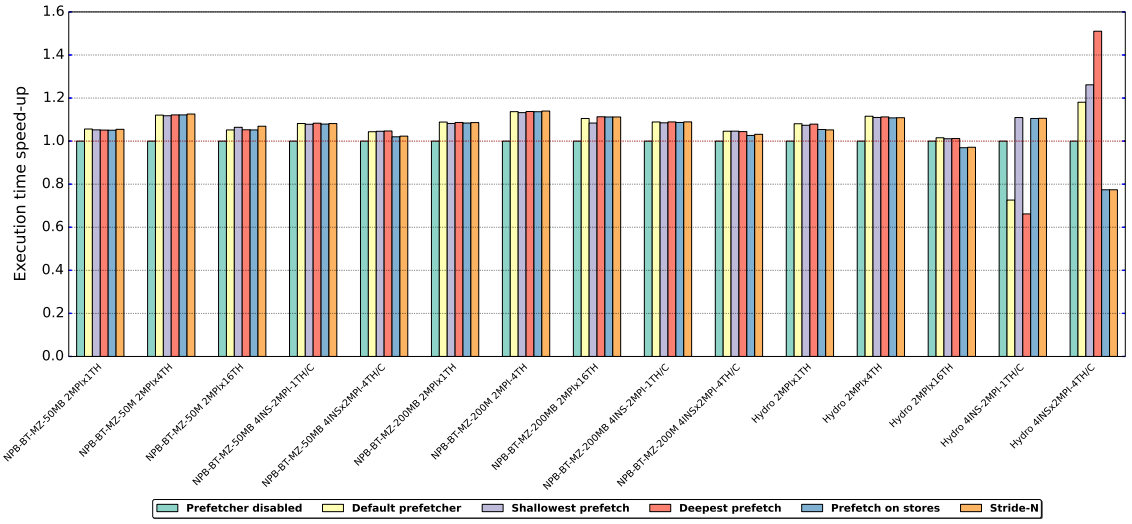

Figure 5.5: BSC applications, execution times for different prefetcher configurations

experiments, vertical scaling results show a noticeable greater amount of used bandwidth with respect to the same parallelism demanded for the horizontal scaling. This is somehow showing that vertical scaling gives the possibility to execute multiple bottlenecks at the same time as it is actually executing multiple application instances and therefore has multiple dependency graphs replicas. A considerable observation is seen in the 80 MB dotprod experiment when requesting all available hardware threads in vertical scaling; 40 GB/s are used. It is a very indicative figure because these 40 GB/s are the maximum amount of bandwidth the machine can deliver. In our case, the 40 GB/s corresponds to an IBM bladecenter PS701 with one memory controller.
On the other hand, matrix multiplication shows quite less significant used bandwidths. Nevertheless, it also can be seen the same tendency for the 50 MB input size. A strange behaviour is observable for the 8 MB input size, where horizontal and vertical scaling use less bandwidth as long as more parallelism is requested. These results are not intuitive but at least, if we go back to figure 5.3 results are coherent for vertical scaling, where there's a performance drop when requesting all hardware threads with respect to requesting one thread per core. A possible explanation is that, when data fits into the LLC, it is very likely that the machine goes under bus contention when requesting so many data.

Figure 5.7 shows this behaviour again for jacobi and specfem3D applications. Nevertheless, this tendency is not observed in other applications, which actually achieve their

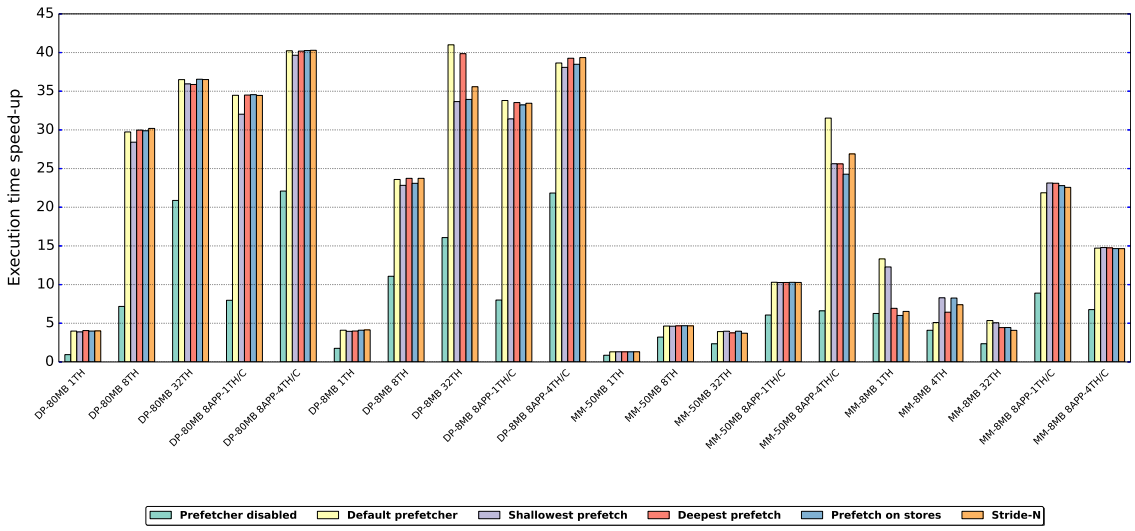

Figure 5.6: Custom applications, bandwidths for different prefetcher configurations

maximum used bandwidth in that parallel configuration. Here a remarkable observation is that shallowest prefetcher configuration gets less used bandwidth in nearly all applications and parallel configurations, which is good news, as it is an indicator that the prefetcher is working as expected. A leading finding is that all these applications achieve much less bandwidth than custom applications, that may be because they have less OmpSs region codes. Anyway, this is quite a reliable clue to understand why these applications have lower speed-ups than custom applications.

Figure 5.8 goes in the same line of figure 5.7 so its results make sure applications are requesting bandwidth as expected. However, an specific observation in the most demanding parallel configuration of vertical speed-up, shows us that prefetch on stores gets slightly less bandwidth. This is a rare behaviour as, in general, horizontal scaling configurations get their peaks in this prefetcher configuration.

Figures 5.9, 5.10 and 5.11 show normalized LLC misses per kiloinstruction with respect to prefetcher disabled configuration.
To begin with, figure 5.9 shows something we would expect. Given that, predominantly, applications are configuration-insensitive, results only show a decrease of misses when comparing prefetcher configuration with disabled mode. An increase of misses is observ-

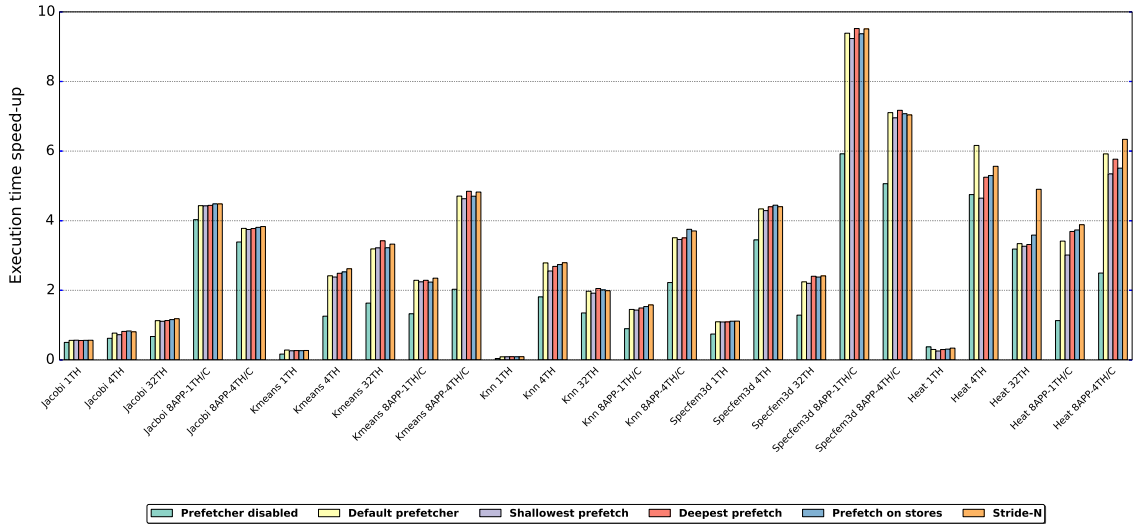

Figure 5.7: HCA applications, bandwidths for different prefetcher configurations

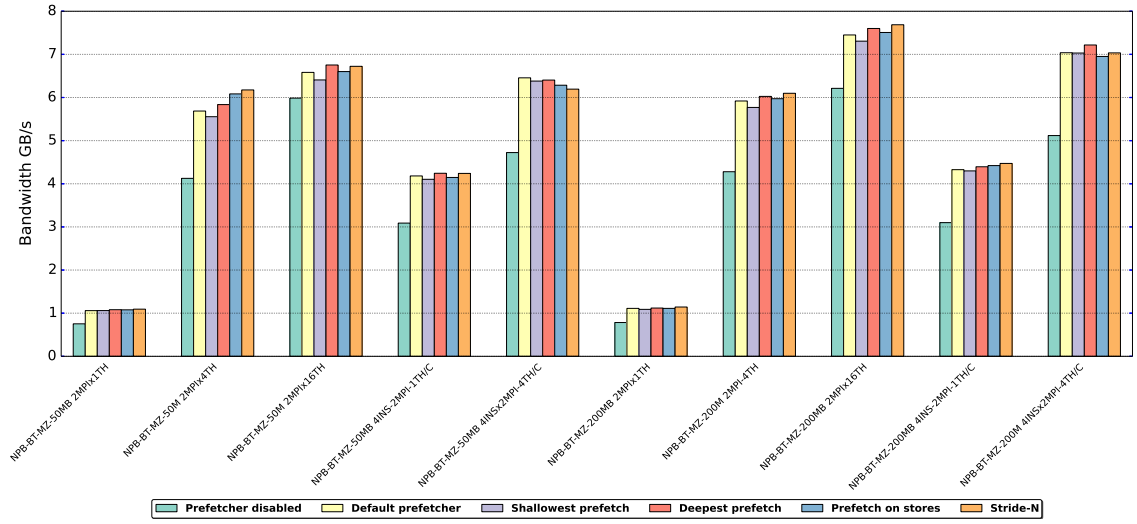

Figure 5.8: BSC applications, bandwidths for different prefetcher configurations

able when requesting maximum parallelism in horizontal as well as in vertical scaling. This can be explained by small false sharing between 4 hardware threads in the L2 cache and/or between all hardware threads in the LLC.

Figure 5.10 also shows expected results. It can be seen, how essentially, shallowest prefetcher configuration obtains the second higher misses after setting disabled prefetcher configuration. That goes according to prediction, as by bringing less data, there are less opportunities to exploit spatial locality.

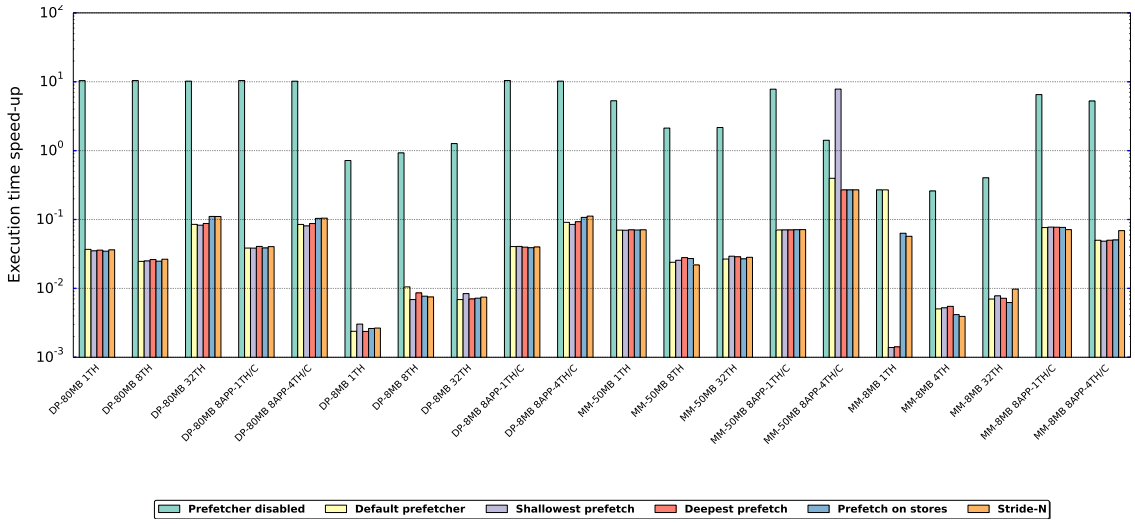

Figure 5.9: Custom applications, L3 misses/kins for different prefetcher configurations

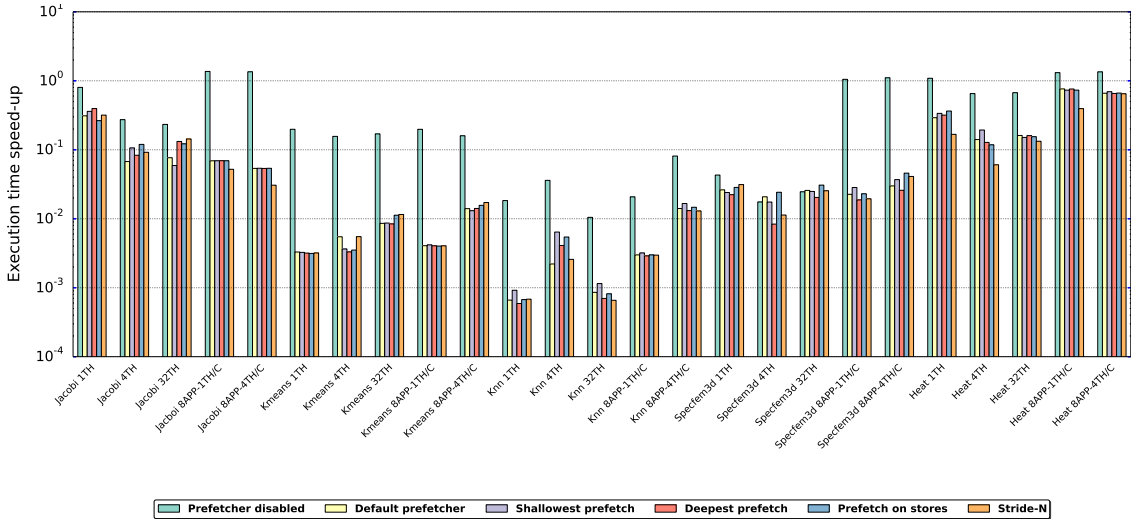

Figure 5.10: HCA applications, L3 misses/Kins for different prefetcher configurations

Figure 5.11 also presents the same information in the results as figure 5.10. Additionally, there are interesting cases in which the deepest prefetcher configuration performs quite worst than other configurations that have the prefetcher enabled. This could be attributed to bringing unused data or data that overwrites some other data needed sooner.

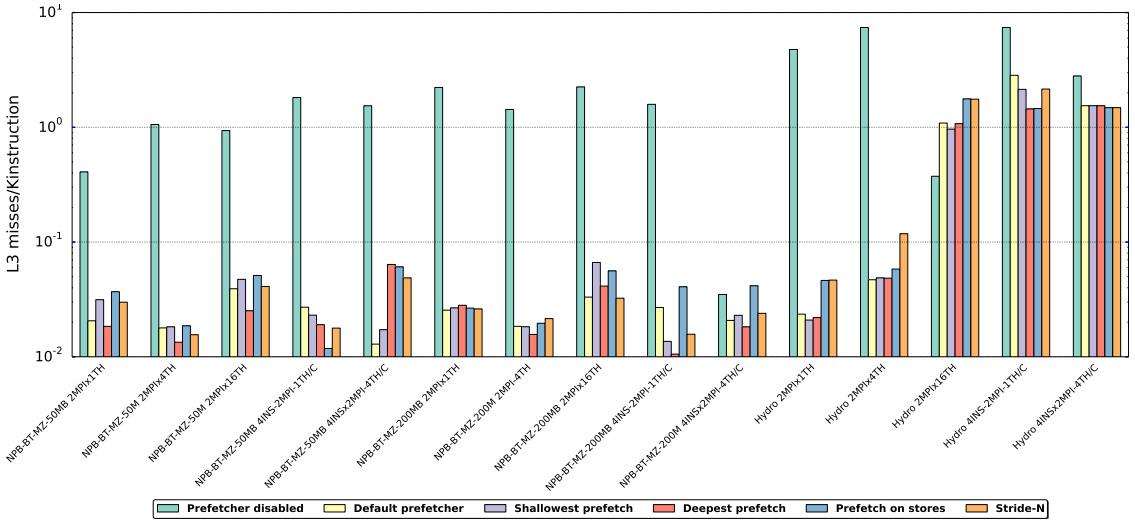

Figure 5.11: BSC applications, L3 misses/Kins for different prefetcher configurations

5.3.2 Dynamic configuration

In this section the dynamic mechanism is put to the test. As has already been commented, the dynamic mechanism has two main parameters that configure its behaviour. The number of tasks executed consecutively under a given prefetcher configuration in exploration phase and the number of tasks executed in the stable phase before switching to exploration mode again.

Big amount of tasks executed in the exploration phase for each configuration means, a priori, more robustness in the IPC measurements, thus more likelihood in choosing the most effective configuration. However, this translates into more time spent executing tasks with a possible performance detrimental configuration, which is something that has direct negative impact in applications' execution time.

On the other hand, large stable phases can achieve good performance in the whole mechanism if there's a high percentage of exploration phases that pick good prefetcher configurations. Nonetheless, if the chip suffers from an environment with multiple applications or/and multiple tasks changing fast their phases of hardware resources demands, that may mean that the mechanism will be less responsive to detect these changes and therefore it may fail to harmonize all hardware threads configurations on time. Let it suffice to say that a too small stable phase could also be harmful for performance as that means switching again to exploration phase trying potentially inefficient prefetcher configurations.

Now that the problematic has been described, let's start to describe a methodology to find some values for these two parameters that actually can work best on average for all kind of applications.
First of all, there's a need to find the shortest number of tasks executed in the exploration phase for each configuration, preserving at the same time, correctness in the election of the prefetcher configuration. In this document, this is is called the window size for each configuration. A way to find this optimization, is to plot the difference in the IPC of each prefetcher configuration in the exploration phases with respect to the corresponding IPC configuration of one execution with a fixed prefetcher configuration, those are the static prefetcher configurations that have already been shown in section 5.3.1. This difference is expressed in terms of the relative error with respect to the static configuration IPC, which is calculated as the IPC average for each configuration of each exploration phase with respect to the total IPC average of the static execution. Figure 5.12 shows the results of this experiment. The IPC difference in terms of the explained error is represented in the ordinate whereas a sort of tried window sizes is represented in the abscissa.

Results of the experiment to find an optimal window size are displayed in figure 5.12. This one shows 4 interesting benchmarks: dotproduct, jacobi, specfem3D and heat, selected from the 3 used benchmarks suites. Horizontal scalability also has been put to the test so as to look for behavioural differences when requesting multiple threads. The plot gives as a very valuable insight; although the relative error presents, in general, low values, most executions tend to lower this relative error reaching their minimum point at 16 tasks in exploration mode. For example: 8 threads dotproduct, 8 threads heat and both specfem3D executions find their optimal in window size 16. From this graphic another very valuable information can be extracted. Given that, as for bigger windows size than 16, the error starts to become higher for nearly all executions, it is important to find out what reasons are behind this behaviour. At first glance, one strong argument can be that inefficient prefetcher configurations are tried too many times. However, when looking at bulked data, an important discovery was that by executing so many times so many prefetcher configurations in exploration phases, some OmpSs tasks types didn't present such an amount of instances in their executions. Hence, as they were missing some values, these ones had to be build by a reconstruction from inside the runtime thus generating another possible source of imprecision in the measurements.

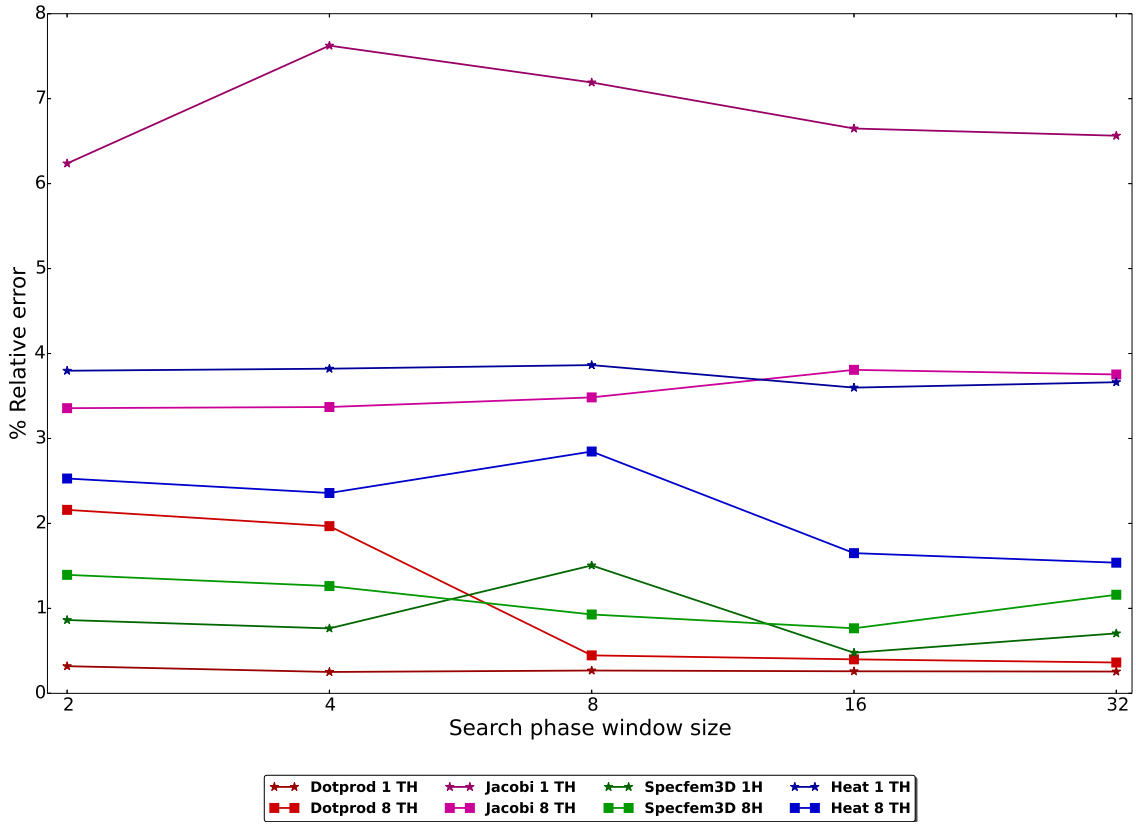

Figure 5.12: Error dynamic IPC VS static IPC measurements

Regarding the size of the stable phase, there is also a need to find a length that no only doesn't switch too often to exploration phase but also doesn't switch too late to detect, to begin with, intra application phase changes. A way to find an appropriate value for this parameter is to compare stable phases IPCs for each possible value of stable phase. This way, final quality of stable phases is measured with respect to changes in the parameter. However, experiments didn't show noticeable differences when testing different orders of magnitude in the stable phase length; for example: we tried 4, 40, 400, 4000 and 40000. Where even high values performed well. This can be attributed to a good working of exploration phase so they choose the best prefetcher configuration in most of the cases. On the other hand, given the results and the scope of these tests, further experiments with new OmpSs applications could bring more workload variety in the whole chip that could create a need to pass through exploration phases more often. For this reason, 400 value

as a conservative option has been the choice for next experiments.

Thanks to the previous experiments, optimal parameters on average have been found for exploration and stable phases. With them, we are ready to test the dynamic mechanism against the best static configuration for each application. Bear in mind that the static mechanism represents itself a worst methodology as it needs the user to first try each prefetcher configuration before they can determine which one is the most performance efficient.
Let's have a look the resultant execution times, which is the most eye-catching metric for this comparison.

Figures 5.13, 5.14 and 5.15 show normalized execution times with respect to configuration in which the prefetcher is disabled including the dynamic mechanism results.
It is straightforward to see that by using the dynamic mechanism, speed-ups suffer from a variable speed-up drop with respect to the best static configuration. However, though few of them have a noticeable drop in the speed-up, the great majority stay vary near to the best static configuration being this very good news for a first design of the dynamic mechanism.

By looking into detail figure 5.13, a general trend is that requesting high parallelism involves a drop in the speed-up. This can be due to a created bottleneck when consulting information about the system state regarding to which hardware thread had what prefetcher configuration for a given task type. This overhead will be discussed later in the document. Apart from this, matrix multiplication presents nearly as good results as dotproduct and this is a good indicator because it means the dynamic mechanism successfully supports two very used computational kernels.

In figure 5.14, a wider range of OmpSs applications is showed. These ones are also very important because they are used in more realistic environments. Results are also very optimistic as dynamic speed-ups are, in general, near to the best static configuration. This can be seen in a prefetcher friendly kernel such as jacobi, in two prefetcher-insensitive applications: kmeans and knn; and in a HPC application such as specfem3D, which is prefetcher-friendly with some differences between configurations with the prefetcher en-

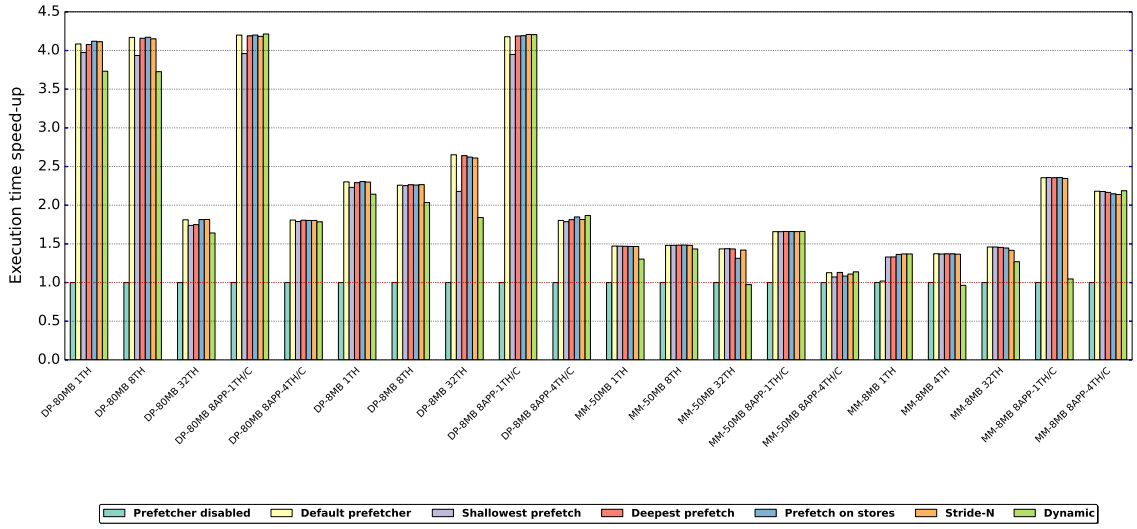

Figure 5.13: Custom applications, execution times for different prefetcher configurations

abled. Finally, in heat application, the dynamic mechanism fails to achieve good speed-ups for some parallel configurations. Nevertheless, their static configuration is prefetcher-unfriendly in those cases and therefore exploration phases are trying a great amount of times all prefetcher configurations with prefetcher enabled, which, in this case, it is something that slows down the whole execution.

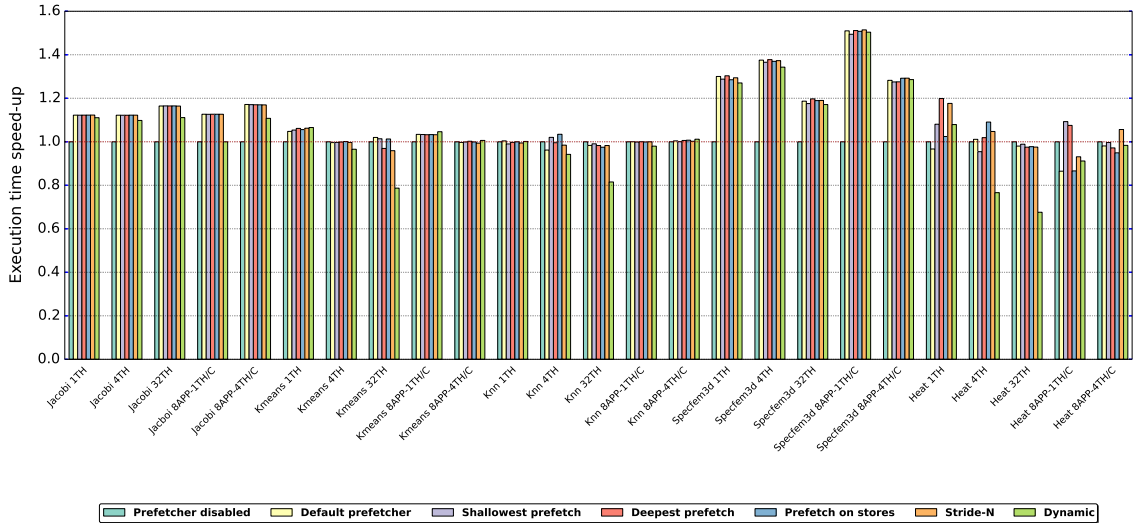

Figure 5.14: HCA applications, execution times for different prefetcher configurations

Figure 5.15 shows NAS NPB-BT-MZ benchmark, which is a MPI benchmark. Nevertheless, this shouldn't suppose any problem as horizontal scaling has already been tested in previous experiments without having an impact in the results. Here, the dynamic mechanism finds it harder to maintain good speed-ups, specially when 2 MPI instances are requested with 16 threads per instance. This is, when the maximum amount of parallelism is requested. Again, a possible collapsed data structure could be the cause of this added drop in the performance.

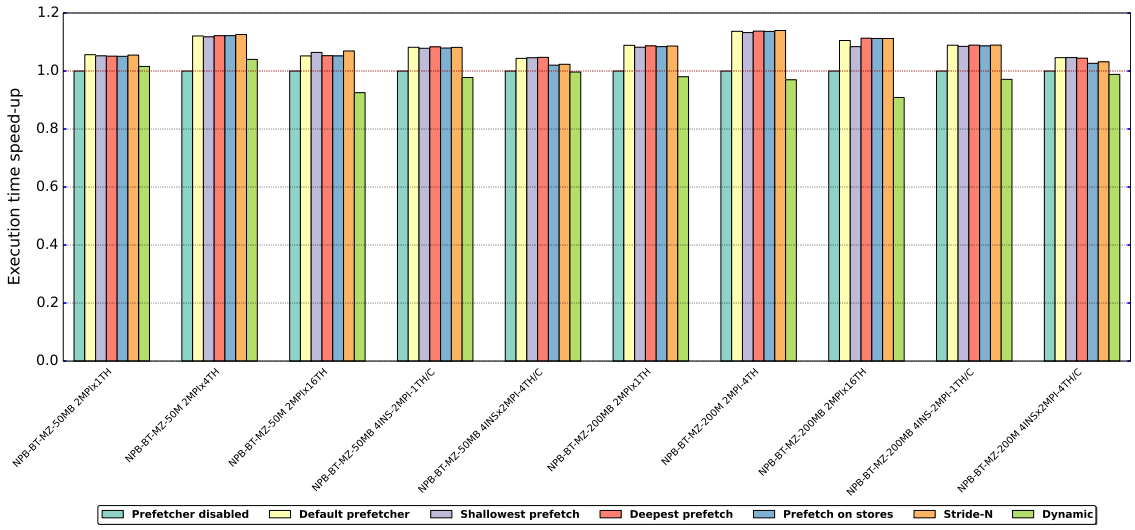

Figure 5.15: BSC applications, execution times for different prefetcher configurations

Going ahead with possible causes in the already showed performance drops: one cause could be a high overhead added by the dynamic mechanism code inserted into the runtime. In this sense, some measurements were done by probing times in the beginning and the end of the dynamic mechanism. Although results are not reported in this document, their figures were, a priori, dismissive as being the cause of a high overhead and their finer refinement certainly involved an accurate study. Notwithstanding two improvements were applied. The first one is explained in section 5.2.1. Whereas the second one consists in saving a copy of the updated DSCR register value in the runtime code. This is a way to avoid making always mandatory to read each hardware thread DSCR register from RAM memory. Although these optimizations were applied, DSCR reads and writes to RAM memory were an issue to take into account and their potential overuse in some algorithms

could be part of an important percentage of these drops.

Chapter 6

Planning

This section explains what planning has been followed in the project. On the whole, the project has benefited from weekly meetings with its supervisors as well as beforehand planed milestone checking with IBM.

6.1 Project tasks

To put it simply, the project has basically two steps. The first one that is checking applications' performance with different prefetcher configurations, changing the prefetcher configuration manually instead of doing it inside the runtime. That is what has been called in this thesis, Nanos++ instrumentation capable version. The second part, is the development of algorithms to make the runtime somehow so intelligent to decide what the best prefetcher configuration is at every time. This is what is called Hardware adaptive Nanos++ version, which is a version that can change the prefetcher configuration dynamically. Here, a keystone idea is that dynamism in the configuration selection is achieved by keep alternating between exploration and stable phases, thanks to them it is possible to find out the best configuration for each OmpSs task type and execute with it. Moreover, it is also important to notice the difference between two concepts mentioned in the document: application phase changes and global phase changes; while the former concept means applications suffer from workload changes during their execution, the later means multicore processors also experience different workload changes across their microachitectural components because of several application executing different phases.

Bearing in mind these two tasks, other tasks appear as a result of different needs. Let's have a look to them.

- MN environment installation: this task contains all related work to install and configure the runtime, the compiler as well as all needed components such as libraries or other tools to start developing our own Nanos++ versions.
- PAPI tests: here all included tests to verify the correctness of PAPI library calls are included. Furthermore first versions of our Nanos++ instrumentation capable version in MareNostrum are included. This included how to get performance counters and analysing that information using Matplotlib as a tool to generate plots.
- POWER7 porting checking: next step is to bring Nanos++ instrumentation capable version to the POWER7 architecture.
- Benchmarking: important and time consuming, benchmarking includes all benchmark suites tuning for Nanos++. This includes, Parsec benchmark suite, some custom developed benchmarks, BSC application repository and benchmarks developed by BSC HCA research group. Apart from this, there's is also an important piece of work to automatise benchmarking tests from the mentioned different suites.
- POWER7 prefetcher register: this task includes doing research about POWER7 capability to change its prefetcher configuration as well as all other modifiable features. There's also a need to get an efficient way to read and write DSCR register from the runtime as well a checking the correctness of each configuration.
- Adaptivity Nanos++ feature: this corresponds with the design and implementation of Hardware adaptive Nanos++. The development of a methodology to choose the best lengths of each phase, (exploration and stable), is also included as well making performance comparisons between static and dynamic versions.
- System integration: this is the last task in which the two Nanos++ versions developed are ensured to work correctly giving meaningful results in terms of performance, used bandwidth and cache misses per executed instructions.
- Thesis report: this task is the writing of this document.

- Thesis defence: this task involves preparing a presentation with a synthesis of the work as well as answering any possible doubts about it.

6.2 Gantt plot

A quick view of the project course is contained in the plot of figure 6.1. It can be appreciated that, at the beginning, project timings are pretty on schedule. However, as time goes by, tasks get delayed. Many issues regarding Nanos++ implementation and some with Mercurium source-to-source compiler appeared. They arose mainly due to lack of knowledge about the environment. Nevertheless, the important problem during the project has been the difficulty to make OmpSs benchmarks to get representative results of different prefetcher configurations, this turned to be a nearly insurmountable problem because no matter how big we increased input size that many benchmarks remained as prefetcher-insensitive. A sanity checking was done developing well-known custom algorithms such as dot product and matrix multiplication, so as to demonstrate the prefetcher has an impact in the performance. To be even more sure about the hardware prefetcher capabilities, SPEC benchamrks, which are out of the scope of this work because they are single-threaded, were also run getting positive results. Although manual tuning of benchmarks as well as additional requirements to show different metrics made the project to delay a great deal of time, the final objective of creating a piece of software that, attached to a runtime, can adapt hardware prefetcher's configuration autonomously to increase microprocessor's effective work has been reached successfully.

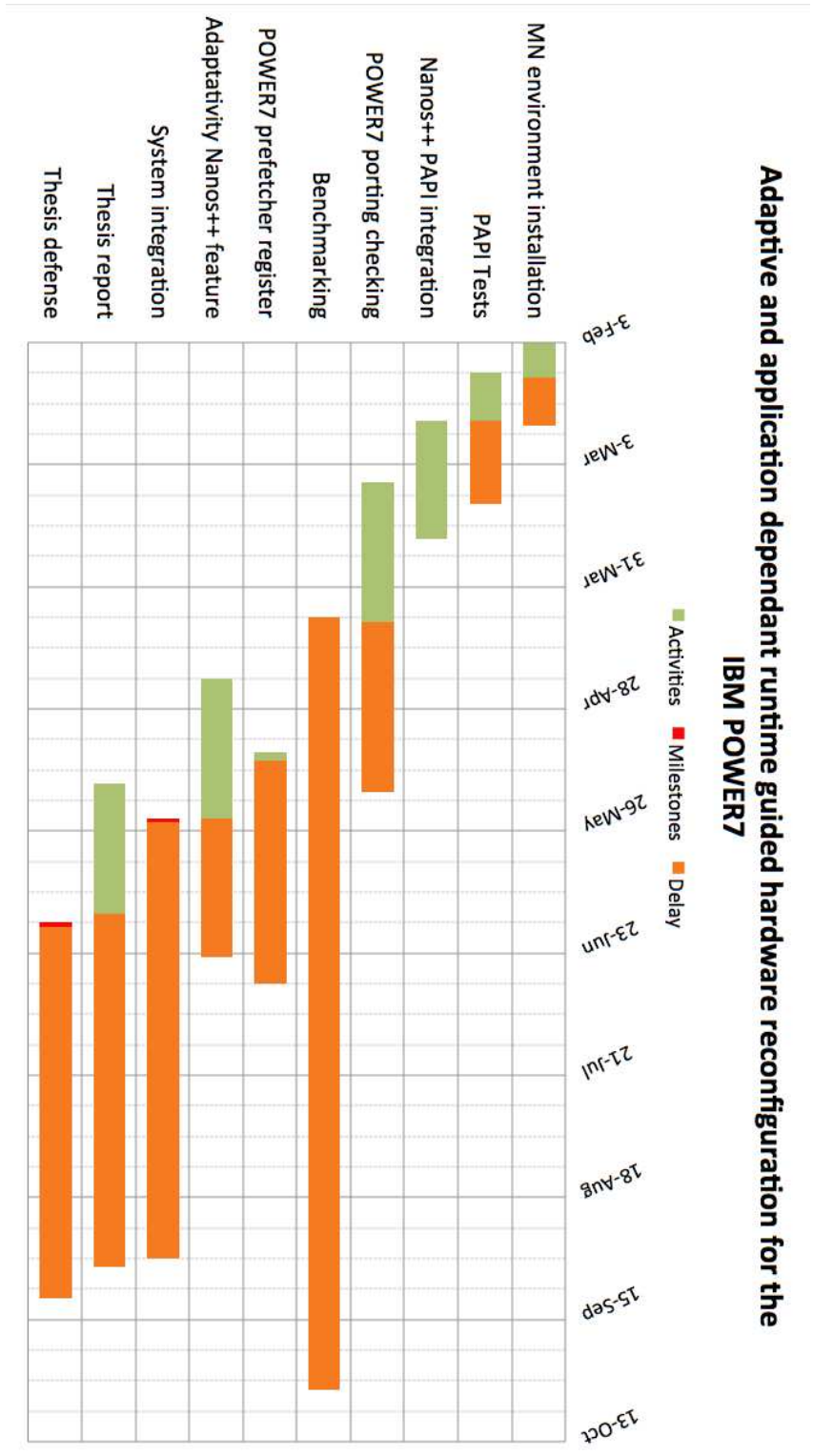

Figure 6.1: Project Gantt plot

Chapter 7

Conclusions and future work

The IBM POWER7 has demonstrated its capability to provide a feature that truly can enhance applications' execution performance without any user action required and very little complication added to processor's architecture. In this work, the focus has been put on the IBM POWER7 hardware prefetcher, which, in the case of Linux operating systems, can be tuned from a register represented in the virtual file system allocated in RAM memory.

In the case of the IBM POWER7, savings in the architecture complexity with respect to the prefetcher management imply that to the get the most out of the prefetcher, there are two options to control its tuning. The first one is to rely on the compiler, letting it try to find load and store streams of data that actually represent spatial locality in which the prefetcher effectiveness is based on; or, also in the same basis, even let the programmer to place their hints in the machine code to provide an aid to the compiler, which is a quite an inefficient solution in terms of productivity. The second option, which is the one presented in this work, is to develop a middleware that can interact with the executed applications and the processor's prefetcher via operating system. Monitoring applications' performance so as to find the best prefetcher configuration constantly, means the need of a runtime system. In this work the use of Nanos++ has been of vital importance. Moreover, its capability to support OmpSs programming model represents releasing full potential of a highly parallel microprocessor as the IBM POWER7.

The first step in this work has involved using PAPI right from Nanos++ runtime system without any other library in between thus avoiding an increase in latency each time

there's a monitoring action. A smart way to store hardware counters measurements and using OmpSs programming model paradigm, which allows to separate pieces of code in tasks that usually put different types of presure in machine's architecture, has allowed me to create a new and very useful piece of software that, to begin with, can make very accurate synthesis of applications' execution metrics; Nanos++ instrumentation capable version. Although this initial version doesn't have any artificial intelligence to find the best prefetcher configuration, it allows to extract valuable conclusions from applications' behaviours and even compare those behaviours in different processors. Section 4.3 is a comprehensive demonstration of such an experiment. Plotting IPC quartiles of each kind of task of a set of benchmarks extracted from Parsec Benchmark suite ported to OmpSs at the BSC, has put on the spot the IBM POWER7 and Intel's Sandy Bridge-EP E5-2670. IBM's microprocessor got less IPC across different benchmarks than Intel's one. Nonetheless, this difference is relative as measurements are taken from each OmpSs task and the IBM POWER7 architecture is more parallel having simpler cores. On the other hand, IPC results grouped by OmpSs task types suppose a unique opportunity for programmers as well as architects to extract conclusions of how different tasks types exploit different components of the microarchitecture. For example: a task type that implies writing massively on disk may get very low IPC with respect to other task types in a processor because, in that processor, pipeline superscalar feature is not exploited. All the same, it can represent not as much performance loss in some other processor if the later one doesn't have very powerful cores. Another utility of this powerful tool is to look at the IPC variability as requested threads increase. Little variability can mean a good multicore design because different threads don't interfere so much with each other inside processor's pipeline and because data is brought smartly to cache hierarchy.

A further experiment has been changing the prefetcher manually and extracting representative metrics such as execution times, used bandwidth between main memory and processor and the amount of LLC misses in the form of misses per kiloinstructions or MPKI. In this tests, a wide range of available OmpSs benchmarks have been used. Notwithstanding needing to change the prefetcher manually, this experiment has been able to put to the test the effectiveness of the prefetcher under OmpSs programming model applications. It turns out that only simple and custom developed OmpSs applications squeeze the machine to its full potential when enabling the prefetcher whereas the other ones have quite more modest gains. Given the proportion of these differences, this means that not

only more complex applications have more regions of serial code or more critical paths in their dependency graphs because of their nature, but also that a more thorough analysis and refinement of their codes could make then to benefit even more from machine's specifications. However, the most remarkable finding has been that nearly all applications presented gain by enabling the prefetcher with very few difference between different depths and other features such as finding store data streams and finding out strides in data streams. This is what in this document has been called applications are prefetcher-friendly but they are configuration-insensitive. This is very important, because in those cases in which execution times are configuration-insensitive and used bandwidths are higher for deeper prefetcher configurations, an intelligent decision is to save power consumed by making the prefetcher to request data streams parts shallower. This action would save power without actually losing performance. Regarding MPKI metric, this one has made even more reliable obtained results. For example: NAS benchmark showed in some parallel configurations an increase of MPKI for the shallower prefetcher configuration while at the same time their correspondent results in used bandwidths showed a decrease in data brought from main memory. Let it suffice to say that, a priori, the less data the prefetcher brings in advance the less bandwidth will be requested.

Finally, the decisive experiment has been putting to the test an autonomous mechanism able to find the best prefetcher configuration for each OmpSs task type through application's execution time; being able to provide this functionality even with multiple applications running at the same time. A latency conscious programming of the mechanism has been born in mind so as to avoid adding overhead each time an OmpSs task is let executed by runtime queues. Minimalism in mechanism's algorithm and aids to avoid executing so often slow operations have been described in section 5. The dynamic mechanism correctness was verified for multithreaded executions tracing each hardware thread DSCR register during each OmpSs task executed by the runtime. As has already been explained, the mechanism tells apart two phases for each task type: the exploration phase, which tests all prefetcher configurations finding the one with higher IPC, and the stable phase, which executes a given amount of times a task type with the previously found prefetcher configuration. By means of experiments, it has been proved that exploration phases should get enough measurements from each prefetcher configuration to increase robustness in the choice. Regarding stable phases, experiments didn't present meaningful results and a conservative decision was made so exploration phases would be short enough to allow

entering the exploration phase so often that a mistake in the chosen configuration not only doesn't make critical harm during the whole application's execution and also leave room to detect intra and inter-applications phases.
Results have been as far as possible satisfactory for all tried applications. A drop in the performance with respect to the best static found configuration can be due two reasons: inefficient prefetcher configurations are always tried in exploration phases, for example: disabled prefetcher configuration can represent up to an 80% of loss in the IPC, for example in dotproduct benchmark. The second reason can be given by the fact that the prefetcher mechanism may find it hard to find the best prefetcher configuration when there IPCs are very similar. Apart from these reasons, but less significant, is the overhead caused by the dynamic mechanism, where read and write operations in DSCR register that can't be avoided have a not mindless weight. Apart from these mild drawbacks, the dynamic mechanism by itself suppose not having to run an application as many times as prefetcher configurations before we actually know what's the best one. Furthermore, heavy workload environments may well change their behaviour from time to time due to their variety of applications running hence this implies that static version approach is not a reliable alternative in multicore environments.
General results have shown applications' performance speed-ups in a great number of cases and although the average speed-up stays, on average, at around 15% for the tested benchmarks, the fact that it is a completely automatic performance enhancement that comes for free and is architecture independent, makes it an absolute useful proposal.
Regarding future's work, a finest refinement of dynamic mechanism is on way to better the mentioned overhead. A first approach will be to assign a drop factor to the worst prefetcher configuration in the exploration mode avoiding its use for a given lapse of time; for example: until a change in workload appears in a multicore environment. Given that this work has probed that it is feasible to benefit from a processor providing architectural tunable features, an interesting contribution of future processors would be to have more features to tune at run-time. This would give more possibilities to develop an intelligence attached to the runtime that could solve, for example by heuristic algorithms, trade-offs when setting parameters/configurations associated to those features. Luckily, IBM announced for their next IBM POWER8 some new features that involve adaptability such as adaptive bandwidths awareness and topology awareness; so in this sense, further research in shared-resources adaptivity is awaiting for us.

Bibliography

[1] J. Baer and T. Chen. An effective on chip preloading scheme to reduce data access penalty. *Proc. ACM/IEEE, Conf. Supercomputing, SC*, pages 176–186, 1991.

[2] B. Barney. https://computing.llnl.gov/tutorials/linuxclusters, 2014.

[3] J. Bartolom. Marenostrum 3, 2013.

[4] C. Boneti, F. Cazorla, R. Gioiosa, A. Buyuktosunoghu, C. Y. Cher, and M. Valero. Software-controlled priority characterization of power5 processor. *In Proc. 35th Int'l Symp. Comp. Arch., ISCA*, pages 415–426, 2008.

[5] B. S. Center. Ompss programming model specification documentation, December 2013.

[6] B. S. Center. *OmpSs user guide*, December 2013.

[7] B. S. Center. http://pm.bsc.es/mcxx, July 2014.

[8] B. S. Center. http://pm.bsc.es/nanox, July 2014.

[9] B. S. Center. http://pm.bsc.es/ompss, July 2014.

[10] B. S. Center. https://pm.bsc.es/projects/nanox, June 2014.

[11] S. Choi and D. Yeung. Learning-based smt processor resource distribution via hill-climbing. *In Proc. 33th Int'l Symp. Comp. Arch., ISCA*, pages 239–251, 2006.

[12] I. Coorporation. http://ark.intel.com/products, 2012.

[13] D.Joseph and D.Grundwald. Prefetching using markov predictors. *Proc. 24th Int'l Symp. Comp. Arch., ISCA*, pages 252–263, 1997.

[14] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[15] E. Ebrahimi, O. Mutlu, and Y. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hibrid prefetching systems. *Proc. 15th Int'l Symp. High Perf. Comp. Arch., HPCA*, pages 7–17, 2009.

[16] B. Elkin and V. Indukuru. *Commonly used metrics for performance analysis POWER7*, September 2011.

[17] P. Emma, A. Hartstein, T. Puzak, and V.Srinivasan. Exporting thee limits of prefetching. *IBM J. R&D*, pages 49(1):127–144, January 2005.

[18] F. C. et al. Predictable performance in smt processors: Synergy between the os and smts. *IEEE Trans. Comput.*, pages 55(7):785–799, July 2006.

[19] S. L. et al. Machine learning-based prefetch optimizaation for data center applications. *In Proc. Int'l Conf. High Perf. Comp. Networking Storage and Analysis, SC*, pages 1–10, 2009.

[20] B. Hall, M. Anand, B. Buros, M. Cilimdzic, H. Hua, J. Liu, J. MacMillan, S. Maddali, K. Madhusudanan, B. Mealey, S. Munroe, F. P. OConnell, S. Reyes, R. Silvera, andy Swanberg, B. Twichell, B. F. Veale, J. Wang, and Y. Yaari. *POWER7 and POWER7+ optimization and tunning guide*. IBM, November 2012.

[21] IBM. *Power ISA, Version 2.06 Revision B*. IBM, July 2010.

[22] IBM. https://www.flickr.com/people/ibmpowersystems/, April 2011.

[23] t. u. o. T. Innovative Computing Laboratory. http://icl.cs.utk.edu/papi/, July 2014.

[24] V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O'Connell. Making data prefetch smarter: adaptive prefetching on power7. In *PACT*, pages 137–146, 2012.

[25] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Proc. 17th Int'l Symp. Comp. Arch., ISCA*, pages 364–373, 1990.

[26] I. C. Laboratory. Papi user's guide.

[27] E. C. J. Lee, O. Mutlu, and Y. Patt. Prefetch-awaare shared resource management for multi-core systems. *In Proc. 38th Int'l Synp. Comp.*, pages 141–152, February 2011.

[28] F. Liu and Y.Solihin. Studying the impact of prefetching and bandwidth partitioning in chip-multiprocessors. *In Proc. Int'l Conf. Measur. and Model. of Comp. Sys. The Filesystem. Proc. Annual Linux Symp*, 2011.

[29] S. Manousopoulos, M. Moretó, R. Gioiosa, N. Koziris, and F. J. Cazorla. Characterizing thread placement in the ibm power7 processor. In *IISWC*, pages 120–130, 2012.

[30] M. Moreto, F. Cazorla, A. Ramirez, R. Skellariou, and M.Valero. Flexdcp: a qos framework for cmp architectures. *SIGOPS Oper. Sys. Rev.*, pages 43(2):86–96, April 2009.

[31] C. L. O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware dram controllers. *In Proc. 41th Int'l Synp. Microarch.*, pages 200–209, 2008.

[32] M. K. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *In Proc. 39th Int'l Synp. Microarch., MICRO*, pages 423–432, 2006.

[33] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linkded data structures. *Proc. 8th Int'l Conf. Arch. Support for Prog. Lang. and Operat. Sys., ASPLOS*, pages 115–126, 1998.

[34] S.Palacharla and R. Kessler. Evaluating stream buffers as secondary cache replacement. *Proc. 21th Int'l Symp. Comp. Arch., ISCA*, pages 24–33, 1994.

[35] V. Srinivasan et al. A prefetch taxonomy. *IEEE Trans. Comp.*, pages 53(2):126–140, February 2004.

[36] J. Strokes. http://arstechnica.com/gadgets/2009/09/ibms-8-core-power7-twice-the-muscle-half-the-transistors/, September 2009.

[37] X. Teruel. Ompss implementation, mercuium and nanos++ overview, 2013.

[38] X. Teruel. Ompss quick overview, a practical approach, 2013.

[39] D. Watts, K. Anders, and B. Patel. *IBM BladeCenter PS700, PS701, and PS702 Technical Overview and Introduction*. IBM, May 2010.

[40] D. F. Wendel, R. N. Kalla, J. D. Warnock, R. Cargnoni, S. G. Chu, J. G. Clabes, D. Dreps, D. Hrusecky, J. Friedrich, M. S. Islam, J. A. Kahle, J. Leenstra, G. Mittal, J. Paredes, J. Pille, P. J. Restle, B. Sinharoy, G. Smith, W. J. Starke, S. Taylor, J. V. Norstrand, S. Weitzel, P. G. Williams, and V. V. Zyuban. Power7;, a highly parallel, scalable multi-core high end server processor. *J. Solid-State Circuits*, 46(1):145–161, 2011.

[41] C. Yang and A. Lebeck. Push vs pull: Data movement for linked structures. *Proc. 14th Int'l Conf. Supercomputing, ICS*, pages 176–186, 2000.

[42] Y.Solihin, J.Lee, and J.Torrellas. Using a user-level memory thread for correlation prefetching. *Proc. 29th Int'l Comp. Arch., ISCA*, pages 171–182, 2002.

Glossary

DSCR The contents of the DSCR, a special purpose register, affects how the data prefetcher responds to hardware-detected and software-defined data streams [20]. 20

IPC instructions per clock (instruction per cycle or IPC) is an indicative aspect of a processor's general performance: it is usually taken as the average number of instructions executed for each clock cycle. It is the multiplicative inverse of cycles per instruction. 44

PAPI provides the tool designer and application engineer with a consistent interface and methodology for use of the performance counter hardware found in most major microprocessors. PAPI enables software engineers to see, in near real time, the relation between software performance and processor events. In addition Component PAPI provides access to a collection of components that expose performance measurement opportunites across the hardware and software stack [23]. 31

RTL is a set of low-level routines used by a compiler and inserted into the compiled executable binary so these routines are called to provide a runtime environment. This environment can provide different features such as guaranteeing a programming model. 32

SMP or symmetric multiprocessing, is a symmetric multiprocessor where identical processors connect to a single, shared main memory, have full access to all I/O devices, and are controlled by a single operating system instance that treats all processors equally, reserving none for special purposes. Regarding multi-core processors, the SMP architecture applies to the cores, treating them as individual processors. 16

SMT is a computer architecture technique in which instructions from different threads can be executing in any given pipeline stage at the same time. 21

SOI in semiconductor manufacturing, silicon on insulator (SOI) technology refers to the use of a layered silicon-insulator-silicon substrate in place of conventional silicon substrates, especially microelectronics, to reduce parasitic device capacitance having a gain in performance. Basically, SOI-based devices differ from conventional silicon-built devices in that the silicon junction is above an electrical insulator, typically silicon dioxide or sapphire (these types of devices are called silicon on sapphire, or SOS). The choice of insulator depends largely on intended application, with sapphire being used for high-performance radio frequency (RF) and radiation-sensitive applications, and silicon dioxide for diminished short channel effects in microelectronics devices. The insulating layer and topmost silicon layer also vary widely with application. The first industrial implementation of SOI was announced by IBM in August 1998. 16

Acronyms

BRU branch execution unit. 17

BSC Barcelona Supercomputing Center. 1

CMOS complementary metal-oxide-semiconductor. 16

CPU central processor unit. 25

CUDA compute unified device architecture. 6

DCBT data cache block touch. 21

DCBTST data cache block touch for store X-form. 21

DFU decimal floating point unit. 17

DIMM dual inline-memory module. 23

DPFD default prefetcher depth. 21

DPLL digital phase-locked loop. 16

DRAM dynamic access random memory. 16

FPU floating point unit. 17

FXU fixed point unit. 17

GPU graphic processing unit. 6

HPC High Performance Computing. 54

IBM International Business Machines. 1

IFU instruction sequencing unit. 16

IPC instructions per cycle. 32

LLC last level cache. 32

LSD load stream disable. 20

OoO out of order. 16

RAM random-access memory. 25

RaW read after write. 7

RTL run time library. 8

SMP symmetric multiprocessing. 22

SMT simultaneous multithreading. 32

SNSE stride-N stream enable. 20

SOI silicon on insulator. 18

SRAM static random access memory. 16

SSE store stream enable. 20

VSU vector scalar unit. 17

VSX vector scalar extension. 19

WaR write after read. 7

WaW write after write. 7

List of Figures

List of Tables