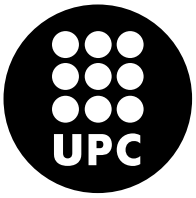


UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONA**TECH**

Facultat d'Informàtica de Barcelona

Master in Innovation and Research in Informatics

High Performance Computing

PRISM: an intelligent adaptation of prefetch and SMT levels

Author: **Cristobal Ortega**

Advisors: **Miquel Moretó**
Barcelona Supercomputing Center (BSC)
Computer Architecture Department, UPC

Marc Casas
Barcelona Supercomputing Center (BSC)

Date: October 20, 2016

Abstract

Current microprocessors include hardware to optimize some specifics workloads. In general, these hardware knobs are set on a default configuration on the booting process of the machine. This default behavior cannot be beneficial for all types of workloads and they are not controlled by anyone but the end user, who needs to know what configuration is the best one for the workload running. Some of these knobs are: (1) the Simultaneous MultiThreading level, which specifies the number of threads that can run simultaneously on a physical CPU, and (2) the data prefetch engine, that manages the prefetches on memory. Parallel programming models are here to stay, and one programming model that succeed in allowing programmers to easily parallelize applications is Open Multi Processing (OMP). Also, the architecture of microprocessors is getting more complex that end users cannot afford to optimize their workloads for all the architectural details. These architectural knobs can help to increase performance but it is needed an automatic and adaptive system managing them. In this work we propose an independent library for OpenMP runtimes to increase performance up to 220% (14.7% on average) while reducing dynamic power consumption up to 13% (2% on average) on a real POWER8 processor.

Acknowledgments

I would like to thanks to my advisors, Miquel Moretó and Marc Casas, for their continued guidance and help in this work. I would not be writing this without them.

And thanks to Ramon Bertran, Alper Buyuktosunoglu and Pradip Bose for everything I learned when working with them.

Also, I want to thank my colleagues from RoMoL at BSC and from UPC.

And finally, thanks to my family for their help through all my life.

This work has been supported by the Spanish Government (Severo Ochoa grants SEV2015-0493), by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P), by the Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), by the RoMoL ERC Advanced Grant (GA 321253) and the European HiPEAC Network of Excellence.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

Nowadays, Moore's Law seems to be ending due to the traditional ways to increase performance are getting more difficult, for example frequency cannot be increased or the transistor size cannot be reduced as it was possible in the last years. But, anyway, actual processors still get more performance each generation by increasing their complexity with diverse techniques within a limited power budget: adding more cores to the processor, using Simultaneous MultiThreading (SMT), smarter data prefetching to improve latency with memory, dynamic voltage scaling, etc.

The techniques just mentioned above were developed to try to overcome different problems found in computer science lately:

- Memory wall. The gap between processing data in the processor and accessing data to memory is huge, and in most programs more than 20% of the instructions are references to memory, to avoid having the processor in idle there are mechanisms such data prefetching to alleviate this problem [41].
- Power wall. As hardware vendors reduced the transistor size and added more transistors to add functionalities to processors power and power density went up; processors have a limited cooling factor (thermal design power), therefore processors cannot dissipate more power than their design allows.

- Instruction Level Parallelism (ILP) wall. Traditionally, one way to speedup workloads were to overlap instructions if they had no dependencies. In the present, finding enough ILP to keep a single-core busy all the time is increasing (because actual processors are faster). Several techniques try to reduce this problem such as adding more cores to the processors, out-of-order execution, instruction pipelining, etc.

But, even more, increasing the complexity of processors lead us to another problem:

- Programmability wall. Getting the peak performance of a processor is a tedious task, because the problem has to be solved having in mind all the architecture details the processor contains: multiple cores, multiple threads per core, data prefetching, dynamic voltage and frequency scaling, etc.

Also, almost all theses techniques that a processor uses are not conscious of the configurations of other techniques. Intuitively we can think that they are not independent, e.g. increasing the number of threads available in the system (number of cores and number of threads per core) will lead to more petitions to memory. And in the case we are prefetching data that can cause cache pollution, stalled threads due to a busy memory, etc.

Having all these into account, one approach that comes to our minds is to profile the workload and then set the different architecture settings to an optimal configuration, but this has 2 problems: (1) we would need a considerable amount of time to profile correctly the workload due to the existence of several settings to tune in present architectures and (2) the workload can have different phases, and probably each phase will need a different hardware configuration.

In order to get the most performance that a processor can offer we need a dynamic coordinated management of the hardware configuration, which is challenging due to the complex interaction between knobs, different intra and inter workload demands, system constraints, large design space options and the difficulty in finding generic solutions as opposed to ad-hoc solutions.

For all the reasons exposed, we need a piece of software/hardware able to tune all the different architecture details of a processor without exposing that complexity to the programmer; something similar to what an operative system does with the

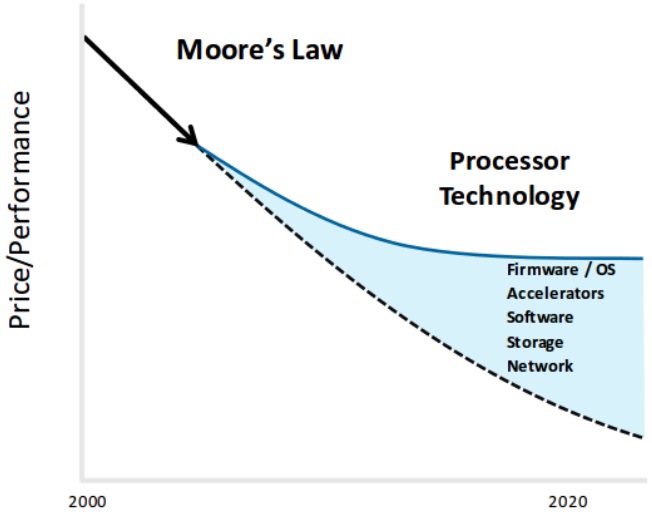


Figure 1.1: Processor technology alone can no longer provide the price/performance gains necessary to sustain Moore's Law [17]

hardware: it exposes the hardware to the user, who does not have to worry on how the hardware behaves or it is controlled.

One solution could be to leave this complexity to a runtime [7, 38]; which takes responsibility of the execution. In figure 1.1 we see an example of this trend in the industry, where software is needed to achieve good performance.. Actually almost all parallel programming models are using a runtime software, if these runtimes that are being already used could get the most performance of a processor would be a smooth transition: increased performance, no need to re-code applications and no architecture dependence.

1.2 Goal

The goal of this thesis is to contribute to solve the problems previously mentioned. We will tune different hardware knobs: SMT level and the data prefetcher, which we think they offer the most performance and the operative system does not anything with these settings:

1. SMT is exposed to the operative system as the extra threads per core are

a real physical CPU, not having into account that is better to use all the physical cores of the machine before starting to use the SMT in a single core.

2. Data prefetcher is not even taken into account. Usually the processor sets the data prefetcher in the boot process to a default behavior and it remains as it.

Even more, in some processors SMT and data prefetcher are not possible to change once the operative system has booted, this is a reason to use the platform we explain in detail later in section 5.1 that allows us to change these setting among others.

SMT level and data prefetch techniques have evolved in a decoupled manner for different reasons: (1) less complexity if they are designed separated and (2) different timescale granularities, SMT level affects only on how many processes can run simultaneously but data prefetch is affecting individually to all the processes.

Also it is important to see how they interact: a system with a lot of processes running and an aggressive data prefetch can impact performance negatively and maybe, running less processes would be more beneficial for performance. Therefore, their independent actuation can lead to conflicting decisions that jeopardize system power-performance efficiency: a robust coordination protocol is necessary.

We will focus on parallel workloads coded in OpenMP, which is the de-facto programming model to parallelize applications. OpenMP works in almost all compiler and all platforms.

In order to achieve our objective we created libPRISM, an auto-tunning library for the SMT and the data prefetch. libPRISM is aware of the machine where is running and it is able to interact with the OpenMP runtime and reconfigure the architecture in order to reduce the execution time and power consumption.

1.3 Thesis organization

This thesis document is structured in the following way: Chapter 1 (this chapter) is the introduction and describes the motivation and goal of this thesis. Chapter

2 gives the planning followed for this work. Chapter 3 summarizes the state of the art for the different technologies used in this work. Chapter 4 explains in detail the libPRISM library, how is designed and how it works. Chapter 5 introduces our experimental platform: processor used, metrics we collected and the evaluated benchmarks. In chapter 6 we evaluate libPRISM with different benchmarks. Finally, chapter 7 presents the conclusions and the final work.

Chapter 2

Planning

This work started in July 2015 with a 3 months internship at IBM Thomas J. Watson Research Center to Pradip Bose's group (Reliability-Aware Microarchitectures) as part of a collaboration between the Barcelona Supercomputing Center and IBM.

The development of our library could be separated in the following tasks as we can see in the Gantt chart:

- Develop a prototype, which needs to be able to do all the desired functions with a small overhead
- Test the infrastructure we developed
- Develop an oracle policy in order to see possible benefits in using the library
- Code a smart policy, here we can differentiate several tasks:
    - Develop and test the SMT level
    - Develop and test the data prefetcher knob
    - Later, we saw that we needed to treat task parallelism different
- Benchmarking, carry out different tests in order to check that everything works fine

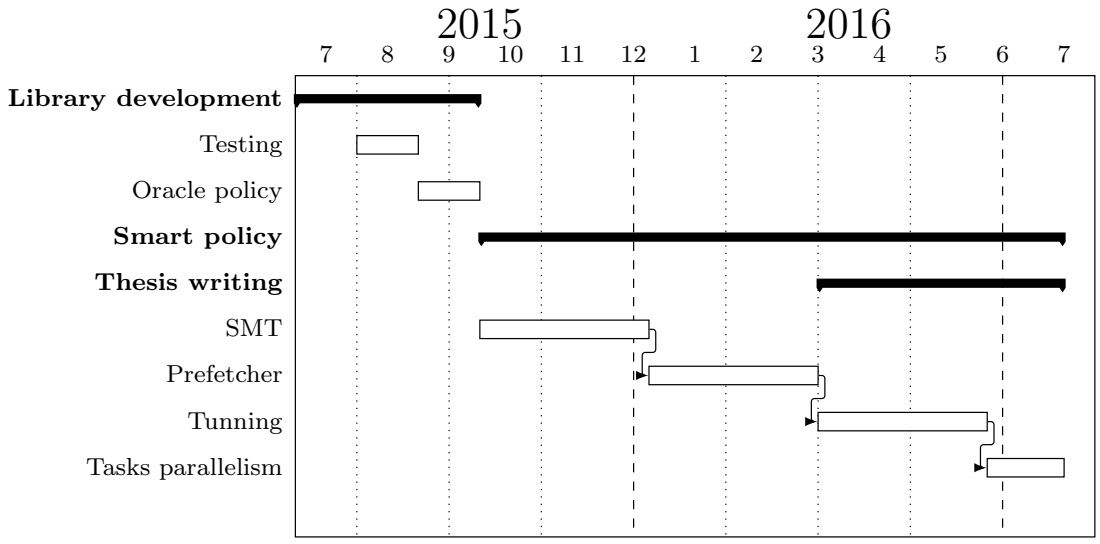
During the internship we developed and tested the infrastructure we used in this thesis.

Once we could guarantee that the infrastructure worked, we started to test a simple algorithm to check the possible gains we were able to obtain.

After the internship we started to select more benchmarks that had bigger inputs and were more used by the community, the suites we used are explained in section 5.3. This task has been the most time-consuming by far, as explained later, some benchmarks used have as a purpose to test High Performance systems, therefore input sets are big and one execution of a benchmark can take several hours.

Once we successfully compiled the selected benchmarks we proceed to start to develop our smart policy for the different knobs we were trying to optimize: the SMT level and the data prefetcher.

While benchmarking our library we saw that we were not able to obtain speedup in some benchmarks due to their nature: they were task-based benchmarks. We came up with one solution to fix it and be able to do obtain the final results.



Chapter 3

State of the art

In this chapter we analyze the different technologies we use in this work are done nowadays and the different contributions in the field.

3.1 Simultaneous MultiThreading

SMT is a hardware technique that permits several independent running threads to issue instructions to the execution stage on the processor, therefore multiple threads can being executed at the same time.

Nowadays, it is implemented in several microprocessors such as IBM POWERPC, Intel Core i Series or UltraSPARC. [12, 24, 27]

The objective of SMT is to not stop the execution of instructions because of possible long accesses to memory or the lack of parallelism at thread level without the need to perform any hardware switch [37]. As seen in 3.1, the processor fetches instructions from different software threads and put them on the instruction queue. Then, in the execution stage all threads executes at the same time.

The trend in processor design has been towards increasing the SMT capabilities. Today, processors implement up to a SMT level of 8 concurrent threads executing on a IBM POWER8 processor [27] (previous version of the IBM processor had only up to a SMT level of 4 threads [34]).

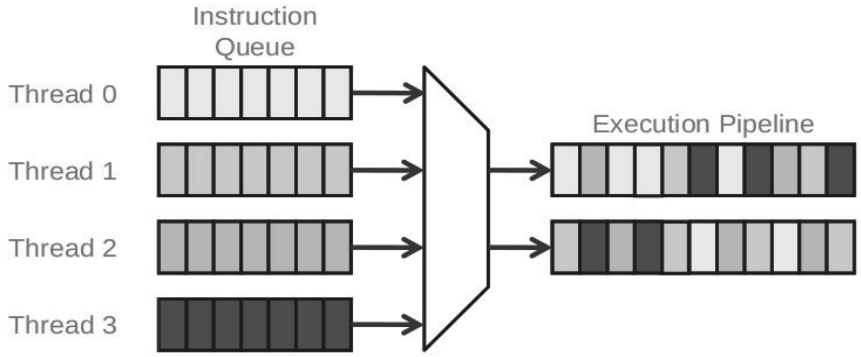


Figure 3.1: Example of a SMT processor fetching and executing from 4 threads

The design of SMT processors are done based on the wide issue technique, which fetches more than 1 instruction of a execution thread from memory to the execution stage. These fetched instructions will have more instruction parallelism since they come from different threads, and ideally, the processor will have enough functional units to execute them all.

In SMT processors, the different threads running simultaneously have to compete for the shared hardware resources, ideally, those threads should not interfere with each other, leading to a performance boost because of a better utilization of the hardware resources (i.e. increasing the total Instructions Per Cycles). One example of this performance boost is in figure 3.3, where 3 different workloads get more performance with the Hyperthreading technology (SMT implementation by Intel)

But in a bad scenario, those competitor threads can interact poorly. For example, if the running threads use a large portion of the cache they may cause a lot of cache misses because they can evict data from another threads, if the threads issue more floating point instructions than the available functional units, etc. This would harm the overall performance.

Also, we need to take into account that SMT is flexible when varying the thread count. If the running threads are low, their performance per-thread will be high and, if the number of threads are high, their performance per-thread will be lower.

While SMT is considered beneficial to increase performance in general, this increment gain is variable due to the current workload the machine is running, because

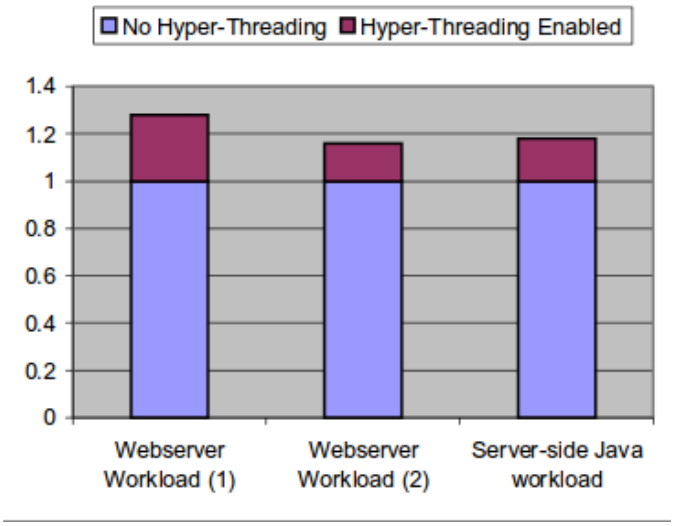


Figure 3.2: Hyperthreading performance (SMT technology by Intel) [12]

as said before, the workload specific characteristics can benefit or harm our performance.

One of the reasons for the loss in performance and fairness is the fact that operative system (OS) perceive the physical core and the hardware contexts (e.g. the different threads a SMT core can run) as virtual cores, which implies that when the OS schedules a process to an available core it has not into account if that core is part of a more busy physical core than other physical core. Therefore, the OS can think is doing a good job scheduling, but actually it could be hurting performance by occupying a single physical core, while there are others idle.

Then, even thread placement is important for performance [26], some workloads can run with higher performance in some specifics hardware threads because of how the SMT was implemented.

Fairness in SMT systems have attracted researchers to find a solution to run different workloads at the same time without degrading too much their performance. Cazorla et al. tried different mechanism to improve performance and fairness [8,9] Research by Boneti et al. seeks a better utilization of the hardware resources at a thread level to reduce load imbalance, therefore increasing performance [2–4]

Lately, researchers have focused on how adjust the SMT level per different work-

loads to avoid performance losses. Tembey et al. propose a mechanism for change the SMT level in order to get same performance but reducing the power energy. [36], something similar Vega et al. do but also using dynamic voltage and frequency reduction [39, 40]. Moseley et al. [29], Snavely et at [35] and Feliu et al. [14, 15] tried to predict IPC when running in a SMT processor and schedule applications accordingly.

Most of the previous work have been developed for serial applications or a combination of serial applications, but little research has been done when referring to parallel workloads. And because the behavior of parallel applications, where usually all the threads are doing the same type of task, it is more probably that exists more conflicts due to threads waiting to be able to use the functional units.

Zhan et al. propose a loop scheduler for SMT processors to obtain performance [42, 43], Heirman et al. try to do automatically choose the best SMT level [19]. Creech et al. implemented a mechanism to address fairness in SMP systems [11], managing the number of threads each application can have; which is similar to managing the number of threads in a SMT system.

3.2 Data prefetch

Nowadays almost all the processors that are made include a data prefetch engine because it is a proven and powerful technique to reduce the problem with memory; as said in section 1.1, memory is becoming the main bottleneck in performance.

A microprocessor can execute an instruction in nanoseconds, but memory needs microseconds to serve data. This difference can affect negatively the performance of our microprocessor and therefore, of our application.

One of the solutions for the memory wall is bring more data than asked and keep it near the microprocessor (e.g. caches), this is done by the data prefetch engine; hardware architects try to do a good design in order to reduce at minimum the waiting for memory.

Of course, there are workloads that have no benefit at all from the prefetcher, even, it can harm the performance.

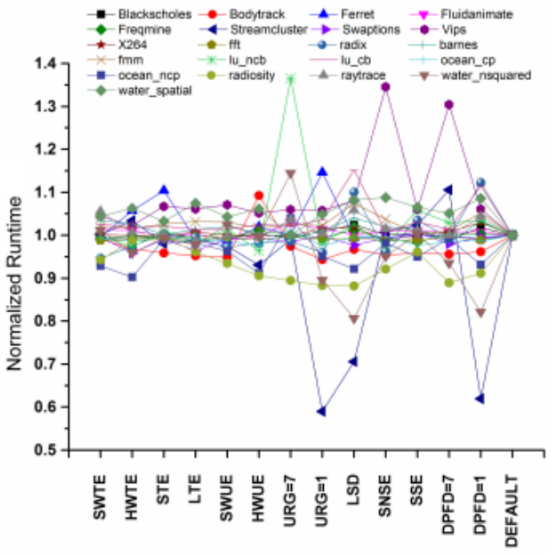


Figure 3.3: Runtime for different configurations for the prefetcher of a POWER8 processor (these options are explained in detail in section 5.1.1.2) [25]

If a workload needs no data prefetch can get worse performance due to the data prefetch engine doing automatically access to memory, even worse, because those data prefetches without any benefit for the workload will cause more bandwidth consumption and more energy waste. In figure **??** we see how choosing a different prefetcher configuration can boost performance or slowdown the execution time in a POWER8 processor (the same we will be using).

Also it can happen that a workload reuse data previously prefetched but, between the 2 accesses the data has been evicted. Again, a bad use for data prefetching.

We can say that the right configuration ultimately depends on the specific characteristics of the workload running. But the algorithm for the data prefetching is *hardcoded* in the processor design; there was no possibility to modify it, vendors often added instructions to let the programmer or the compiler do software prefetching; this adds a step in the optimization process of a code.

Current processors give to the user or programmer some freedom to tune the hardware prefetcher (e.g. IBM POWER8 [27]). Usually, the user is able to do prefetches on loads, but it can be configured to do it also for stores, or bring more lines to the caches, etc. This can save time in the optimization process (i.e. we

just need to profile the workload with different prefetcher configurations once the code is optimized) and in some cases save bandwidth and energy.

As said, data prefetching is workload dependent, this is the reason researchers have investigating how detect different workloads (or phases inside a workload) to adapt the prefetch behavior.

Jimenez et al. have been working in detecting phases of applications in runtime and change the data prefetch accordingly [21, 22]. Minghua Li et al. applied machine learning to smart prefetching depending on different workloads [25] based on performance counters. Hur et al. observe the spatial locality of workloads to apply a better prefetching [20]. Casas et al. evaluated how memory resources behave on HPC applications and how this interacts with the increasing number of cores per chip [5, 6].

Even there is some work in another aspects of the memory hierarchy such as the work done by Moretó et al. where the shared caches are dynamically partitioned to improve performance [28]. Bitirgen et al. go further and try to increase performance managing different aspects of the memory hierarchy with machine learning techniques [1].

All this previous work was done around serial workloads; but less attention was given to parallel workloads.

Previously to this thesis, we focused on parallel task workloads implementing a smart data prefetch mechanism as a part of the a task-based runtime [32].

3.3 Programming models

As discussed before, the number of programmable cores in our systems is growing, therefore if we want to get the full potential of an architecture we must be able to coordinate all the system to work in a given problem.

Even nowadays, solving a problem using parallel system is challenging due to different parameters we have: memory accesses, shared data, race conditions, synchronization points, etc. To avoid this tedious and huge work there are software

specialized to run codes in parallel, which are known as runtimes.

These runtimes are used as an abstract layer in the software stack to make parallel pieces of code. Usually, they need compiler support to translate from keywords to real code that will be executed: the programmer just needs to use a specific keyword to spawn all the desired threads, share the data among them or synchronize the threads. This is a very good way to avoid bugs in our codes and speed up the consuming-time task of coding.

When parallelizing a code in a shared-memory system, we can apply different strategies:

- Data decomposition. Once we identify the data on which computations are performed we partition this data among our available threads. The programmer needs to decide what data is partitioned: the input, the output, or even create an intermediate state. This decision can affect to the performance of the parallel algorithm.

- Exploratory decomposition. If the problem to solve involves the exploration or search of a state space of solutions we may want to use this strategy, where each thread will work in a *possible* solution. One intuitive example would be how to solve a sudoku, where we try to fit a number in a square and a thread will continue as it that number actually fits (leading to a complete puzzle or a unfinished puzzle).

- Recursive decomposition. When using a divide-and-conquer algorithm it can be useful to assign each division to a thread, because divisions should not be affected for the others divisions, which is good because it will have not shared data with others threads.

- Hybrid decomposition. We can combine different strategies mentioned before to increase performance.

If we need more performance (and we are using all the resources of one machine) we can use more than one computer, i.e. a distributed memory system. Usually, when working distributed memory systems is to use message passing between nodes with the same strategies explained.

Once we know how to code our algorithm we need to choose a mechanism to run our parallel work:

- Threads. Here the programmer takes full responsibility for controlling the threads: create and destroy them, synchronization points, etc. It is a way to have maximum control of the code but also, it increases the possibility of introducing a bug due to the programmer is the only one working in that code.

- Runtime. If we choose to use a runtime because its advantages: easier to code, existing code is (usually) reviewed by the community or a company, more documentation, etc. and there are several runtimes available to use:

    - OpenMP. Actually, it is a standard for shared-memory programming. The OpenMP Architecture Review Board (ARB) publish an API for anyone willing to implement it. There are different runtimes based on this API, normally each compiler implement their own runtime, therefore there is one runtime for The GNU Compiler Collection (GCC), icc (Intel Compiler), xl (IBM compiler), etc. [31] The programmer needs to declare where wants to use parallel code with explicitly directives that later will be translate to calls to the OpenMP runtime routines.

    - Threading Building Blocks (TBB) by Intel. Runtime by Intel similar to OpenMP standard but with some extra features as concurrent data structures or scalable memory allocator, but it lacks of others as affinity support or static scheduling. [33] TBB and OpenMP can be used together.

    - Chapel by Cray. It aims to make easier the coding task of large-scale computer, trying to keep the performance or improve it respect other programming models mentioned in this section. [10] It separates parallelism and locality, it enables to describe how run things and how store things. Also it is capable to use parallelism beyond intra-node (e.g. it can do message passing)

    - OmpSs by BSC. OmpSs extends OpenMP tasking functionalities, since this runtime is based on task parallelism. Here the programmer just

needs to explicitly declare the inputs and outputs of the tasks, then the runtime will solve at run time the dependencies, executing first the task with outputs needed for another tasks. [13] This has a lot of potential since, ideally, the run time can accelerate the critical path in the dependency graph. Also since it supports the OpenMP standard it is very easy to translate code written in OpenMP to OmpSs

Chapter 4

libPRISM

Our proposed solution for the mentioned problematic in the introduction is libPRISM: an external library that takes care to adjust the hardware configuration to obtain the best performance without any interaction by the final user and without the need to recompile any other library on the system as it could be the OpenMP library. libPRISM was named after the words: **lib**rary, **Pr**efetcher and **I**ntelligent **SM**T. Mainly, because the prefetcher and the SMT level are our main targets when developing this library.

As it can be seen in the figure 4.1 our idea is to be *part* of the runtime, but one of our goals is to be independent of the runtime, leaving to the user the choice of which runtime use; this is why we will use library interposition: (1) to be (at some level) part of the runtime, (2) be independent on the choice of runtime and (3) be architecture independent.

We will focus on the OpenMP specification [31] mainly for 2 reasons:

- It is the de-facto standard for parallel programming on shared memory system
- GCC has a fully functional OpenMP implementation

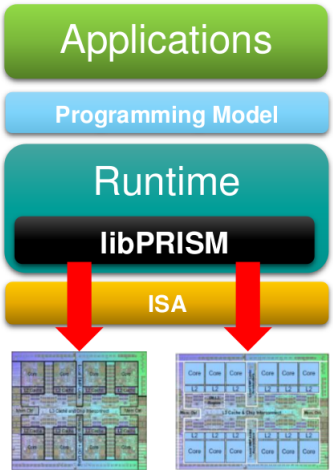


Figure 4.1: libPRISM position on the execution stack for an application

4.1 Design overview

In order to keep the design simple, usable and extendable; libPRISM is divided in 2 main parts as we can see in the figure 4.2:

- Wrapper, which is the code responsible for the library interposition. It will translate calls to different OpenMP runtimes to libPRISM. Explained with more details in section 4.1.1

- Driver, which is the main part of the library. It has to treat data from hardware sensors, apply the algorithms to find the best hardware configuration and change it. Explained with more details in section 4.1.2

Following this design we can achieve (1) modularity: add different OpenMP wrappers keeping the same underlying algorithm, change the wrapper from OpenMP to another type (e.g. based on time), add or change different algorithms without affecting how the library interposition or how data is obtained from the hardware is done. And (2) usability: a new libPRISM user will only have to code a new wrapper and only if he wants to use a different OpenMP implementation than the one provided by GCC; which is GNU OMP (GOMP as we will refer from now on).

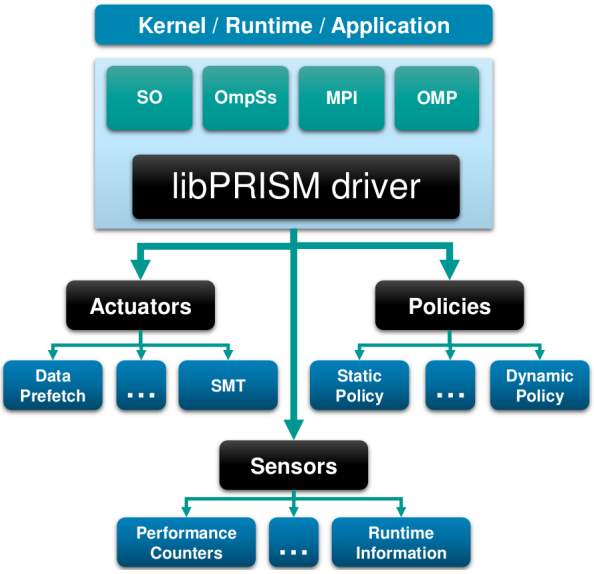


Figure 4.2: libPRISM software design

4.1.1 Wrapping mechanism

As said before, we will focus to do library interposition with the GCC OpenMP implementation. For this task we need to check the OpenMP specification [31] and the code generated from GCC.

We want to be able to profile the loops and, as the specification says, there are some details to consider:

1. There is no method to change the number of threads when we are inside a parallel region. Meaning that we have to change the number of threads (therefore the SMT level) before entering a parallel region

2. Following the previous rule, once inside in a parallel region if we encounter another parallel region (nested parallelism) changing the number of threads will affect the nested parallel region.

   For example: in the code 4.1 we are not changing the number of threads available in the outer loop, therefore the number of threads used in the nested parallel region will be 4 (each existent thread will create 4 more threads). But in the code 4.2 we change the number of threads before entering the

nested *#pragma omp parallel*, therefore each existent thread will generate 8 threads (i.e. *4 (threads of the outer loop) x 8 (threads)*)

```
#OMP_NUM_THREADS=4
#pragma omp parallel for
  //Here will be 4 threads
  #pragma omp parallel for
    /* Each thread of the outer parallel region
     * will create another 4 threads
     * 4 threads x 4 threads = 16 threads */
```

Listing 4.1: Example not changing number of threads in GOMP

```
#OMP_NUM_THREADS=4
#pragma omp parallel for
  //Here will be 4 threads
  omp_set_num_threads(8);
  #pragma omp parallel for
    /* Each thread of the outer parallel region
     * will create another 8 threads
     * 4 threads x 8 threads = 32 threads */
```

Listing 4.2: Example changing number of threads in GOMP

3. When inside a parallel region, if there are a pragma *#pragma omp task* to create tasks we cannot control the number of tasks spawned nor the existing threads that execute them because of the points 1 and 2.

Having in mind those points, we can proceed to code the wrapper for the OpenMP implementation by GCC.

We need to know that the OpenMP specification provides the functions needed to be compliant with the standard, and then the different compilers on the market provides a function to their library with a different name. First of all, we need to know what are the function names we need to intercept, to do this we just need a binary file compiled with GCC (and ideally, the source code of the binary file to know how the different pragmas translates to the different functions). For example, we will use a *Hello world*, which is shown on the code 4.3 to test it:

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
  int th_id, nthreads;

  #pragma omp parallel private(th_id)
  {
    th_id = omp_get_thread_num();
    printf("Hello World from thread %d\n", th_id);

    #pragma omp barrier
    if ( th_id == 0 ) {
      nthreads = omp_get_num_threads();
      printf("There are %d threads\n",nthreads);
    }
  }
  return EXIT_SUCCESS;
}
```

Listing 4.3: Hello world in OMP

After compiling we obtain a binary file from we can read its symbol table in the listing 4.4. The symbol is part of the Executable and Linkable Format (ELF) format, and the section (usually with the name .dynsym) holds the dynamic linking symbol table used for the dynamic linking.

```
 readelf -s hello_world_omp
Symbol table '.dynsym' contains 12 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND GOMP_parallel_start@GOMP_1.0 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND GOMP_barrier@GOMP_1.0 (2)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND GOMP_parallel_end@GOMP_1.0 (2)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND omp_get_thread_num@OMP_1.0 (3)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (4)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (4)
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND omp_get_num_threads@OMP_1.0 (3)
     9: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    10: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    11: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
```

Listing 4.4: Symbol table from Hello World

As we can see the information that is useful for us is:

- GOMP_parallel_start@GOMP_1.0, it indicates when a parallel region is about to start
- GOMP_barrier@GOMP_1.0 (2), it is a synchronization point for all the threads
- GOMP_parallel_end@GOMP_1.0 (2), it indicates when a parallel region is about to end
- omp_get_thread_num@OMP_1.0 (3), function that returns the thread id (from 0 to N, where N is the number of threads created)
- omp_get_num_threads@OMP_1.0 (3), function that returns the total number of threads that are or will be created

In our experiments we have seen something interesting when changing the number of threads, when using the GOMP library the threads are created at the beginning of the parallel region and usually after the parallel region ends the threads remain created and sleeping, waiting for more work to do. But, if between 2 parallel regions the number of threads is changed, in the second parallel region the existing threads will be adapted to the new number of threads: creating or destroying threads. This is important to have in mind since it will difficult how we track and profile the workload. In the listing 4.5 we clarify this concept.

```
1  omp_set_num_threads(4);                          //Existing threads
2  //4 threads will be the limit                    //1
3  #pragma omp parallel                             //4
4    //work
5
6  omp_set_num_threads(8);
7  #pragma omp parallel                             //8 (4 created)
8  //work
9
10 omp_set_nu_threads(5);
11 #pragma omp parallel                             //5 (3 destroyed)
12   //work
```

Listing 4.5: Example of number of existing threads

The last thing we need is to know how to do library interposition. For that purpose we will use the mechanism in Linux *LD_PRELOAD*.

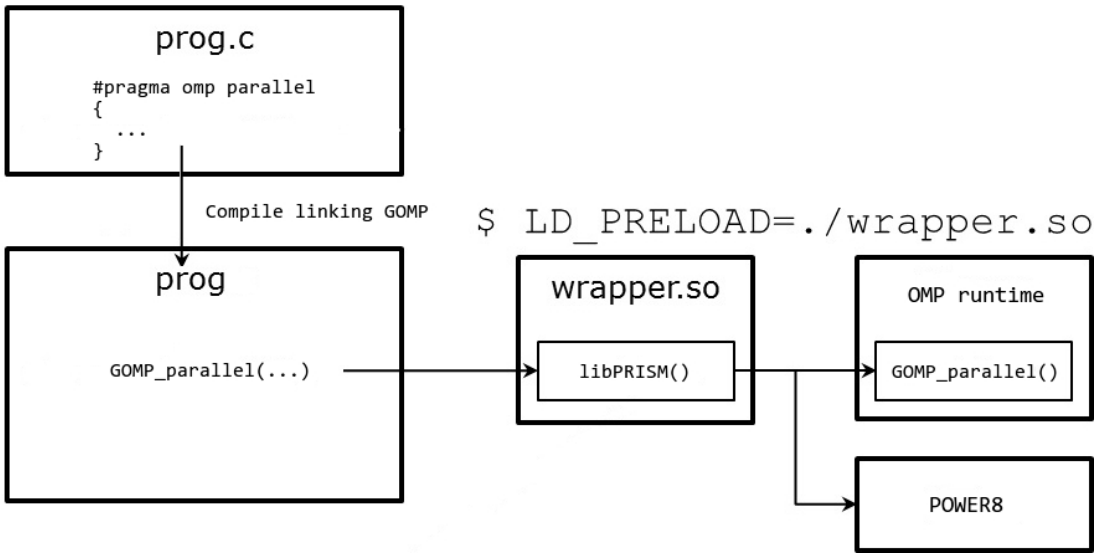


Figure 4.3: Using LD_PRELOAD in a OpenMP application

With this mechanism we can fake a function from a shared object with a function coded by ourselves. This implies that an application linked against that shared object (in our case the shared OpenMP library) will actually call and execute our coded function as it were the real one.

The existing library in Linux that provides library interposition is: *dlfcn.h, Dynamic Linking*, which declares the following functions:

- void *dlopen(const char *, int);
- void *dlsym(void *, const char *);
- int dlclose(void *);
- char *dlerror(void);

In our case we just need to use the *dlsym* function, which allow us to *find* the next address for a given name function.

For example, if we do: *dlsym(RTLD_NEXT, "GOMP_parallelStart")*, it will return us a pointer with the address of the next function called *GOMP_parallelStart*, which the next function is solved in the same order as we specify with the LD_PRELOAD variable. Since we will be calling this function inside our wrapper, the next function

will be the one located in the OpenMP library.

And if we want to create a parallel region for a given code, we just need to jump to that address (with the corresponding parameters). Therefore, we need to find all the addresses of the functions we need to intercept before anything else. Our wrapper will do at the initialization as we can see in the listing 4.6

After applying this mechanism our scheme when running an application that uses the OpenMP library will be as shown in the figure 4.3

We will use this mechanism to be between the application and the OpenMP runtime, in this way we will be able to know when a parallel region starts or ends and we will be able to adjust the hardware configuration on time. Therefore, we need to code a wrapper for the different calls to the OpenMP runtime that our binary files link.

```
1  /* Obtain @ for GOMP_parallel */
2  GOMP_parallel_real =
3  (void(*)(void*,void*,unsigned,unsigned)) \
4  dlsym (RTLD_NEXT, "GOMP_parallel");
5
6  /* Obtain @ for GOMP_parallel_start */
7  GOMP_parallel_start_real =
8  (void(*)(void*,void*,unsigned)) dlsym (RTLD_NEXT, "GOMP_parallel_start");
9
10 /* Obtain @ for GOMP_parallel_end */
11 GOMP_parallel_end_real =
12 (void(*)(void)) dlsym (RTLD_NEXT, "GOMP_parallel_end");
```

Listing 4.6: Example of the dlsym usage for the GOMP wrapper

And then, of course, we need to code our wrap call to the different real functions as we show in the listing 4.7.

```
1  extern "C" void GOMP_parallel_start (void *p1, void *p2, unsigned p3)
2  {
3    int r_PC = -1;
4
5    //Use the PC of POWERPCs as identifier
6    #if defined (__powerpc64__) || defined(__powerpc64__)
7            r_PC = mfspr(8);
8    #endif
9
10   //Follow only master thread
11   if (syscall(SYS_gettid) != master){
12           GOMP_parallel_start_real(p1,p2,p3);
13           return;
14   }
15
16   //Call to libPRISM
17   PRISM_parallelStart(r_PC, 1);
18
19   //Call to the real GOMP_parallel_start
20   GOMP_parallel_start_real (p1, p2, p3);
21 }
```

Listing 4.7: Example of the GOMP wrapper for libPRISM

4.1.2 libPRISM driver

Once we can do the library interposition against GOMP, we need a software that controls the OpenMP runtime based on the behavior of the workload running on the machine, the hardware and the runtime itself.

We want to keep the software simple and modular, for this, we have different modules as we can see in the figure 4.2. These modules are:

- Actuators, they are the responsible to change the underlying hardware (at this moment we just need to set the data prefetcher and the SMT level)
- Sensors, here are the objects that take care of measure the performance in different ways. We use from Performance Monitor Counters to Wall time from the special registers on the POWER8.
- Policies, we want to test different ideas to see which one is the best, for that we create an heritable object from where different policies will easily be

created.

All those object will be controlled by the libPRISM driver, which will be called directly from the wrapper we defined in section 4.1.1 as we can see in the listing 4.7. The libPRISM driver will be an interface to access to the different modules we coded and as shown in the listing 4.8

```
1  void PRISM_parallelStart ( int PC, int nthreads ) {
2    ++level;
3    if( max_level == -1 || level <= max_level ) {
4      timingStart();
5
6      int num_threads = _policy->getNumThreads();
7
8      //Call policy module
9      _policy->parallelStart(PC,num_threads,level);
10
11     timingEnd();
12   }
13 }
```

Listing 4.8: Entry point to libPRISM

For the sensors objects we will use basically 2:

- In order to read the different PMCs we need we use the perfmon 2 library (libpfm version 4.7.0), which let us read from the standard Linux perf_event interface groups of PMCs, in this way we can read all the PMCs at once and then we will be able to correlate their data between them.
- Also for reading the time wall that we spend in parallel regions we will use the timebase special register in the POWER8 architecture, 512000000 ticks of this register are equivalent to 1 second.

For the different actuators, which will be 2 also:

- In order to modify the SMT level we can (1) change the number of threads (POWER8 processor automatically goes to its corresponding SMT level) or (2) set the SMT level by writing in a register (see section 5.1.1.1) libPRISM keeps record of the number of threads that are actually running at the moment to enter the parallel region (in case we want to restore the state once

For each parallel region	For each parallel region executed
Program Counter (PC)	Number of threads used
Number of times executed	Start time
	End time
	SMT configuration used
	Data prefetch configuration used
	Data from PMCs

Table 4.1: Stored data with libPRISM

we exit the parallel region) and the number of threads we run inside the parallel region. Changing the number of threads will require a call to the OpenMP runtime: *omp_set_num_threads(int nthreads)*

- To set the value of the data prefetch is needed to write in a special file as we will detail in the section 5.1.1.2

And the different policies implemented are described in the section 4.3

To achieve a good and modular design we need to keep record of everything we read or do in every parallel region, for that we will store every piece of information we read with the different sensors and the actions we took. Therefore, every time we enter in a parallel region we start the different sensors and when we exit the parallel region we read from the sensors and store the data read in different data structures. We identify each parallel region by its Program Counter and, in case of we encounter the same parallel region multiple times, by the number of times we have executed the parallel region.

4.2 Parallel regions

As described previously, libPRISM works on parallel regions, which in OpenMP are defined with: *#pragma omp parallel*. This pragma spawns the required threads and put them to work.

In general there are 2 methods to parallelize a code with OpenMP, which we need to know in order to be able to capture the behavior and profile them:

- Divide work. *#pragma omp parallel* defines a new parallel region where all the threads will execute the same code and there is the *#pragma omp parallel for* for loops that automatically spawns the desired threads and each of this new threads start to work in a portion of the dataset the for loop is iterating.
- Creating tasks. OpenMP 3.0 introduced the concept of tasks: the programmer can create work tasks and then the threads will work on those tasks.

4.2.1 Dividing work

Usually, they are defined with a *#pragma omp parallel* or with a *#pragma omp parallel for*, the code that is inside those pragmas will be done by all the threads. In the case of just using the keyword parallel the programmer needs to ensure that threads are not actually working on the same data or use synchronization mechanisms.

Here, we proceed as following: once our wrapper for library interposition captures one of those pragmas we read the PC and call to libPRISM. Therefore we have a clear entry and exit point of the parallel region.

4.2.2 Tasks

There are 2 methods to create and work on tasks:

- All the threads create and execute tasks.

  The pattern for this scheme is with a *#pragma omp parallel* and inside there is a *#pragma omp task*. Therefore, all the threads will encounter that pragma, they will create the task and execute it.

  libPRISM treats this method as the mentioned before in section 4.2.1. It just need to support nested pragmas (which it does), then the master thread will be capturing the different tasks it is generating and processing.

  There is only one limitation: created tasks go directly to a task pool and then threads pick tasks from the pool. This means that the number of tasks

we see from the master's perspective (which is the thread libPRISM tracks) are not the total number of tasks generated.

This limitation actually it is not a problem, because the reading realized for the different tasks will be meaningful because all the threads will be working on tasks.

- Only the master threads create tasks.

  Code for this scheme is as follows: first there is a *#pragma omp parallel* to spawn all the threads, then there is a *#pragma omp single* in order to make that region only executable by one thread. Inside that last pragma the only working thread will encounter the different *#pragma omp task*.

  In this case we cannot proceed as the previous method: here the non-master threads go directly to a synchronization point where they will fetch for work from the task pool, while the master thread creates tasks.

  This is actually an important limitation: the master thread will only work on tasks when encounters a synchronization point, and when that happens we cannot get good performance readings: the synchronization point contains the fetch and work function but from outside the runtime we see only the synchronization point.

  In order to sort this out we wait for the synchronization point, and before really enter on it we spend some time to measure the performance (Instructions Per Cycle). This way of measuring performance introduces more overhead than previous methods since we are taking a thread that could be doing useful work to do performance measurements.

4.3 Policies implemented

The modular design of libPRISM explained previously allows to implement a policy only coding a few functions that will be called before and after a parallel region executes (as shown in the workflow of libPRISM in figure 4.4), after the execution of the parallel region we will have available all the information libPRISM stores

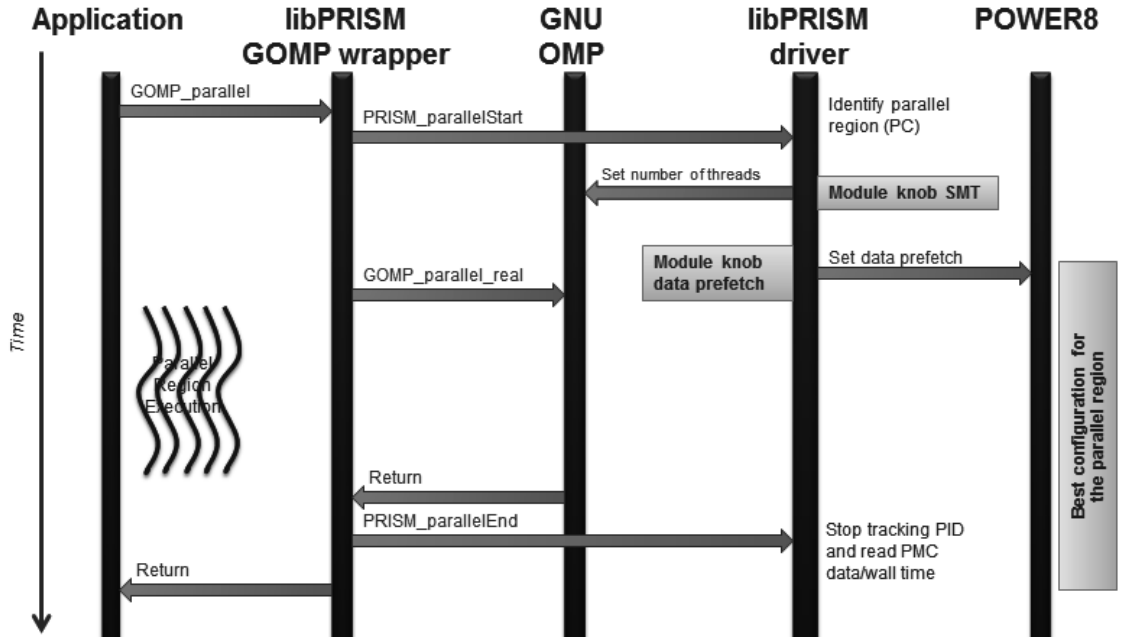


Figure 4.4: libPRISM workflow

per parallel region (see table 4.1).

We coded 2 different policies, the first one purely for showing possible advantages with the approach we are following and the second one, which is the smart policy tries to find the optimal configuration for the hardware by itself.

4.3.1 Oracle

This policy is based on an offline profiling and it was developed to check how much speedup we can obtain when running with libPRISM.

The idea behind this policy is after ran the application with the different hardware configurations in static mode, collect the data generated and then add everything in a file to be read in a next execution. In this file we will store data per parallel region in order to identify what is the best static configuration in terms of performance.

Finally, when running with the Oracle policy we will feed the profile file to the policy, which will set the configuration per each parallel region to the optimal one found in previous executions. Ideally, with this policy we will obtain the maximum

speedup possible in each application ran, but in the process of testing this policy we detected problems with setting the different configurations:

- Overhead due to setting the number of threads. As we described previously in section 4.1.1, GOMP creates and destroy threads to adapt to the number of threads specified by the user, therefore when a parallel region is executed and does not last enough to make up for changing the number of threads we will have an overhead. For example, in figure 4.5 we run multiple times with libPRISM the BT workload (explained in detail in section 6.1.1):
  - In the X axis we have the different parallel regions (identified by their PC) the workload has.
  - In the Y axis we have the execution time normalized to the Best execution time for the parallel region.
  - Per each parallel region we have done several experiments:
    - Run the whole experiment with ST, SMT2, SMT4 and SMT8 and change the SMT level only for the parallel region we are inspecting
    - Run the whole workload with the best SMT level for the parallel region we are inspecting
  - Run the whole experiment dynamically changing the SMT level to the best in all the parallel regions (our oracle policy)

  Here we can see that the last parallel region identified by *0x1000b6a0* takes more time to finish when we are running the whole workload with a number of threads different of its best. This is due to the overhead of changing the number of threads that GOMP has. It is possible to see in table 4.2 that only we can observe a real overhead when the parallel region last around 0,0062 seconds.
- Overhead when setting the prefetcher. As we will explain in section 5.1.1.2, we need to write in a file to set the prefetcher, and because we need to go through the operative system, this can produce an overhead. What we do is discard small region (as in the previous point) and in case of doing more aggressive prefetching increase our performance we require that increment to

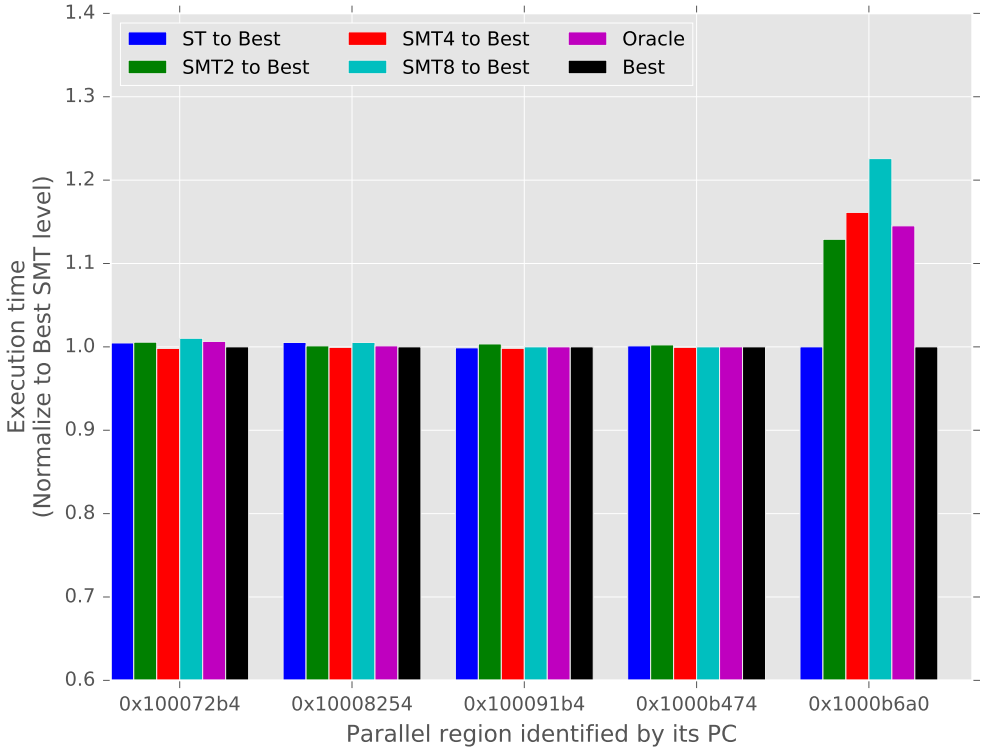


Figure 4.5: Overhead for managing threads

Parallel region	Best SMT level	Best execution time (seconds)
0x100072b4	4	0,107
0x10008254	4	0,1698
0x100091b4	4	0,1715
0x1000b474	4	0,1715
0x1000b6a0	1	0,0062

Table 4.2: Best SMT level and its execution time for different parallel regions in the workload BT

be at least of 0.0015 seconds, due to is our required time to set the prefetcher.

4.3.2 Exploration

This approach is based on the typical behavior of High Performance Computing applications, usually these applications have a main loop that is repeated hundreds of times.

libPRISM takes advantage of this behavior and tries to learn what is the best configuration for the different parallel regions existing in the applications. The idea is to spend a few iterations of an overall of hundreds training libPRISM to know what configuration is the best for the different parallel regions (ideally, we spend a little percentage of iterations to do the training, trying to reduce the overhead to 0). Of course, at the moment that a parallel region is not repeated enough we will not be able to get the best performance for that parallel region.

The first though it came to our mind is that we have 2 knobs to configure, therefore we need to think which of the knobs will be characterize first or if we will characterize both at the same time. Characterizing both at the same time can produce a lot of overhead since we will need to spend more iterations because the number of combinations grow faster:

- SMT levels are 4 (ST, SMT2, SMT4, SMT8)
- Prefetcher has a several bits to configure. In previous experiments realized, we saw that not all the bits affect the same to the performance (all names are as refereed in section 5.1.1.2):
    - Disabling/enabling the prefetcher usually increases the most the performance
    - Depth (how many cache lines the prefetcher brings) offers a performance increment
    - StrideN (the prefetcher will try to recognize simple data access patterns) also offers a performance increment, sometimes bigger than depth
    - Stores, (bring to the nearest cache data when the processor makes a

store), can speed up the performance but usually less than depth or strideN

If we want to try combinations of both of them, we see that:

4 SMT levels x 4 prefetcher fields (just using the default depth and the maximum depth) gives us 16 iterations. That without having in account possible repetitions of the configuration to avoid noise and have a more accurate average of time.

Some codes that we will use have between 75 and 500 iterations of the same parallel region, at the moment we want to get an average, we will increasing the overhead a lot, if we limit the calculation for the average to 3 iterations, then we get 48 iterations wasted in training libPRISM (with the possibility of affecting in a bad way the performance) and 48 iterations translates to a 64% of the iterations in the worst case and 1% in the best case.

In previous experiments we were able to identify that SMT level is the main knob affecting the performance by far. This is the reason we chose a hierarchical exploration for our policy, we will first choose the best SMT level and then the best prefetcher, doing this we can avoid performance loses due to be in a SMT level that is harmful for our performance. The final design can be seen in the figure 4.6 and in the figure 4.7 is a simple view of how we characterize both modules.

The basics of the algorithm are the following: we explore each option until we can observe that we are not gaining performance (or even losing it), then we stop. This also is based on previous experiments, which showed that exists a curve in the performance when increasing the SMT level. When starting with X configuration can happen 2 things:

- Configuration *X+1* gives a speedup, in this case we keep *X+1* as the best configuration or the new X, and proceed to inspect the new *X+1* (or what is the same: the old *X+2*)
- Configuration *X+1* decreases or does not increase the performance, in this case we keep the best configuration (X) and stop the characterization for that hardware knob.

Given these 2 possibilities, the curve of performance is something similar to what

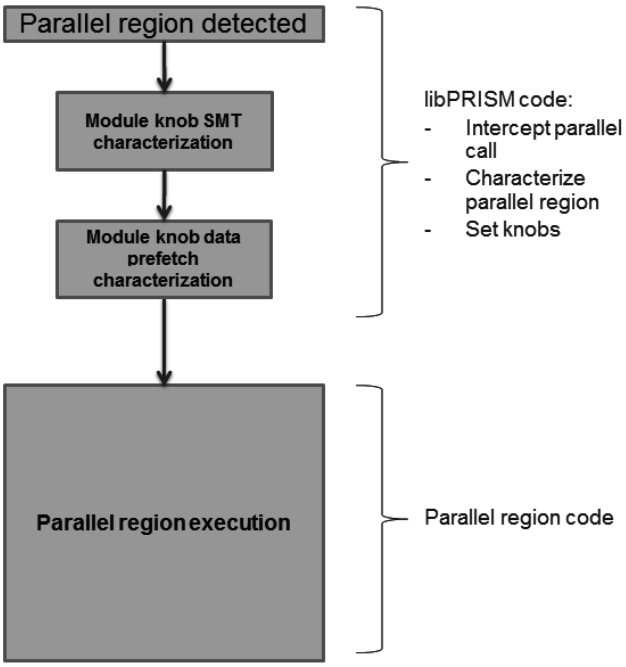


Figure 4.6: Hierarchical design used for the characterization of the different knobs

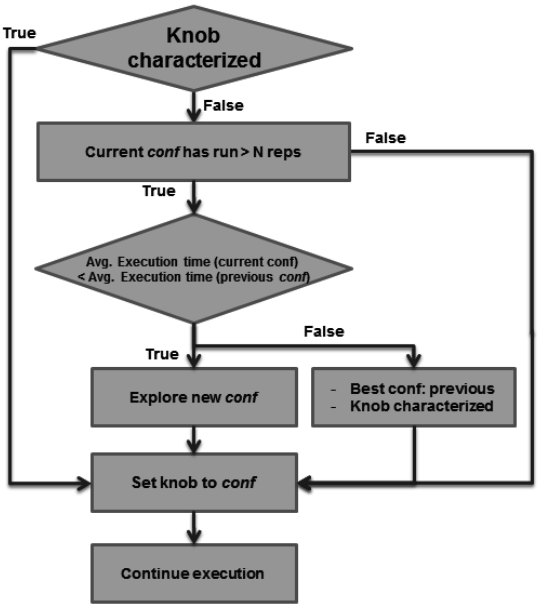


Figure 4.7: Abstract algorithm used to characterize a knob

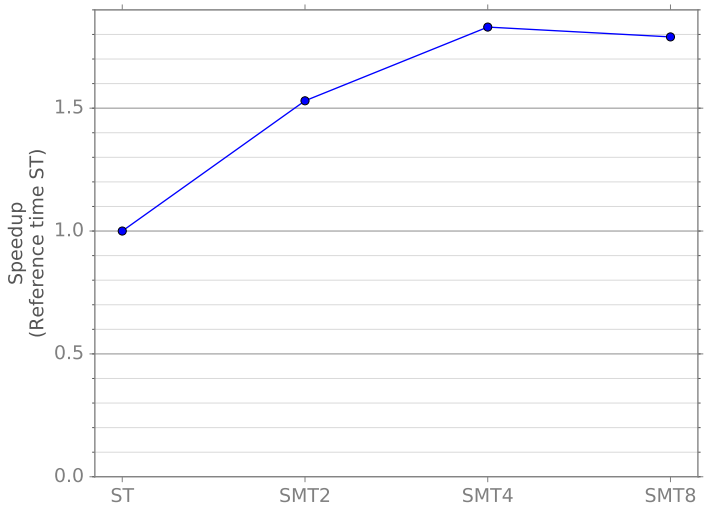


Figure 4.8: Performance curve when increasing the SMT level

it can be seen in figure 4.8, we can observe that usually higher SMT level is better for the performance, that give us a hint in how explore the different configurations: going for the higher SMT level. We have seen few applications where Single Thread (ST) is the best SMT level.

There are several optimizations applied to the algorithm for reducing the overhead of libPRISM:

1. Avoid small parallel regions. Due to the overheads seen in section 4.3.1 we do not set the SMT level nor the prefetcher for parallel regions that last less than 0,01. Instead, we execute the parallel region with the current SMT level and prefetcher configuration.

2. Phase detection. During experimentation we observed a behavior in our workloads: the initialization phase is usually different to the computing phase, even in the same parallel region; parallel regions are shorter at the beginning of the execution. This is the main reason to implement a phase detector in our algorithm. Every N iterations of a parallel region we start to characterize the behavior again, N should be not to large to be able to capture different phases correctly but also not to small to not produce over-

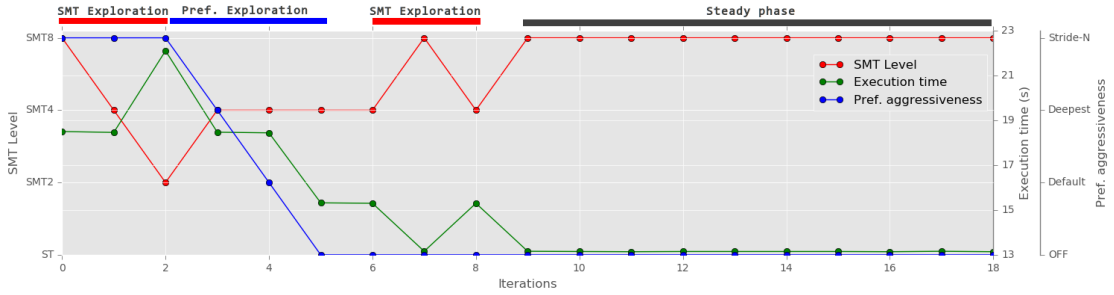


Figure 4.9: libPRISM configuring CG on runtime

heads.

3. Number of repetitions to characterize a parallel region reduced to 1. If we detect that the last execution time of a parallel region differs more than 5% of the best execution time (with the correct SMT and prefetcher configuration) we start to characterize its behavior again. It allows us to reduce the number of repetitions of a parallel region to 1, which decreases the overheads when characterizing. Parallel regions with small number of iterations benefit tremendously from this.

The result of these mechanics are shown in figure 4.9. In this figure we show the main consuming-time parallel region of the benchmark CG from NAS (see section 5.3.1 for further information), in the X axis we show the number of times the parallel region is executed, and in the Y axis we show the SMT level, prefetcher aggressiveness and execution time.

libPRISM starts the most aggressive possible for SMT and data prefetcher, first it selects the best SMT level. Once we know the optimal SMT level for that pre-established prefetcher libPRISM tries to find out the best configuration for the prefetcher. As we can see, it selects to turn off the prefetcher, then libPRISM needs to know if changing the prefetcher had some impact on the best SMT level, and as we can see in the figure the best SMT level for the prefetcher disabled is 8. Then, libPRISM enters in a steady phase until the moment the execution time for a parallel region differs too much from its usual execution time.

Chapter 5

Experimental framework and methodology

5.1 POWER8 processor

For demonstrating purposes we evaluate our solution in a real POWER8 processor (model 8247-42L) with the following hardware specifications:

- Reduced instruction set computing architecture
- 2 sockets, each socket has 12 cores
- 64 GB CDIMM @ 1600 MHz
- L1 64 KB
- L2 512 KB
- Shared L3 48 MB
- Up to 128 MB eDRAM L4 (off-chip)
- Bandwidth with memory of 230 GB/s
- Peak on Input/Output of 96 GB/s
- SMT modes available are SMT1 or ST, SMT2, SMT4 and SMT8

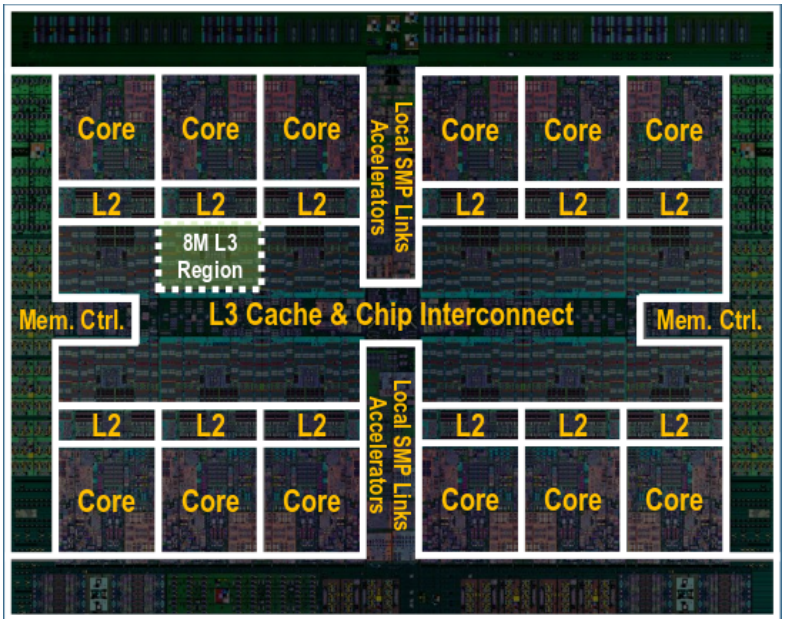


Figure 5.1: POWER8 socket architecture

This model is a scale out system, consisting of a Dual Chip Module (DCM).

This means that a DCM fills 1 socket and a DCM has 2 scale out chips. A POWER8 chip contains 6,8,10 or 12 chiplets (our model contains 6 chiplets). A chiplet consists of one core, 512 KB of SRAM L2 cache and 8 MB of L3 eDRAM shareable among all chiplets as it can be seen in the figure 5.1.

For our experiments we limited the number of cores used to 6 cores (one chip of the 2 available chips) pinning the threads to the processors, the reason for the limitation is to ensure the data is always local, because of how the cores are distributed accesses from one core to another core located in the other chip have different latency and avoid thread migrations. Therefore the experiments will be more repeatable.

Each physical core has the architecture we can see in the figure 5.2 with the following specifications:

- CPU clock rate between 2.5 GHz to 5 GHz
- 16 execution pipes

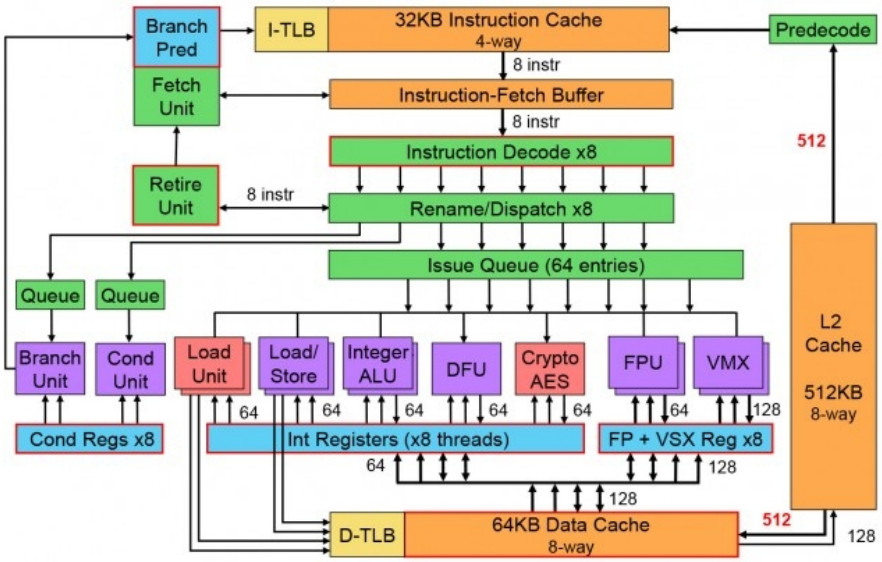


Figure 5.2: POWER8 microarchitecture

- 2 Fixed-point units (FXU)
- 2 Load/Store units (LSU)
- 2 Load units(LU)
- 4 Floating-point units(FPU)
- 2 Vector units for SIMD (VMX)
- 1 Cryptographic unit (Crypto)
- 1 Decimal floating-point unit(DFU)
- 1 Condition unit (CR)
- 1 Branch unit (BR)

The software stack used in all the experiments is:

- Operative system Ubuntu: 14.04
- Compiler: GCC 4.9
- OpenMP library: libGOMP OMP 4.0

- As described in the section 4.1.1, we will develop our inter-positioning calls for these specific library. Changing the OMP library should be as easy as just rename the functions to the new OpenMP library, for example, GOMP has the prefix *"GOMP"* for all the functions, if another OpenMP library has another prefix, we would just need to change the prefix.

5.1.1 POWER8 reconfigurability

The POWER8 processor have different knobs to control different hardware components: the SMT level, thread priorities, how the data prefetch behaves, etc. As said before, we will focus on the SMT level and how the data prefetch behaves; in order to control these knobs the POWER8 processor exposes different writable registers per each virtual core to the operative system [18]

From previous experiments, we saw that the main knob affecting to the performance is the SMT level and secondly, the configuration for the data prefetcher, due to this, we tested different parallel suites to see the different impact of these knobs on real applications.

Our requirements are very simple: the parallel applications have to be programmed with the OpenMP model and we should be able to compile them with gcc 4.9 to do library interposition with the GNU OMP library; this last requirement is to avoid coding different wrappers.

In Linux, the representation of a core of a machine can be seen in the directory */sys/devices/system/cpu/*, in the POWER8 we can see from the folder cpu0 to the folder cpu191 (all the physical cores from both sockets and their virtual cores, these last are the *extra* threads because of the SMT capabilities). Each of those folders have files representing information or registers about the core and some of them are writable. To manage our knobs we need to modify the following files:

online file, only the virtual cores have this file and can be used to turn off the virtual core if we write a 0, or to turn it up if we write a 1. But, thanks to the POWER8 firmware, this is not always needed; POWER8 processor automatically

enables or disables virtual cores depending on how many threads are running on the machine.

DSCR file, it describes with a numerical value how the data prefetch should act, the different values to take into account are described in the section 5.1.1.2.

5.1.1.1 SMT

The operative system sees 192 cores (SMT8 level x 12 cores x 2 sockets), but actually, a group of 8 consecutive cores are representing 1 physical core, i.e. the cores 0,1,2,3,4,5,6,7 correspond to 1 physical core, this representation of a physical core depends of the actual processor (Intel usually uses 0,2,4,etc. as one physical core).

But, the hardware does not behave in the same way when using the first 8 virtual cores (1 physical core) than when using 8 physical cores (with one thread per physical core: cpu 0, cpu 8, cpu 16, etc.) this is because SMT offers more thread capacity but with the disadvantage that those threads will run slower due to the hardware resources are shared.

The trade off for a parallel application is:

- run with more threads, therefore higher SMT level and slower threads
- run with less threads, thus lower SMT level and faster threads.

As we will see in the section 6.1 it really depends on the applications and if the parallel application is CPU-bound or memory-bound.

In order to tweak this knob there are 2 possibilities:

- Change the number of threads running in a physical core. POWER8 firmware automatically goes to the SMT level corresponding to the number of threads running in a core (e.g. when running 2 threads on a physical core, it will go to SMT 2).

  This is used in libPRISM for outer parallel regions.

- Manually enable or disable virtual cores. As said before we can enable or

disable a core by writing a 1 or 0 in the **online** file corresponding to the core. Each physical core are presented to the OS as 8 virtual cores, if we disable 0, 4, 6 or 7 the firmware will force a SMT level of 8, 4, 2 or ST.

This is used in libPRISM for nested parallel regions or tasks.

5.1.1.2 DSCR

Another knob we will be using in our experiments is the DSCR, it controls how the data prefetcher behaves. It contains different fields that are activated writing a 1 or a 0 in the register as seen in table 5.1:

	SWTE	**HWTE**	**STE**	**LTE**	**SWUE**	**HWUE**	**UNT CNT**	**URG**	**LSD**	**SNSE**	**SSE**	**DPFD**
0:38	39	40	41	42	43	44	45:54	55:57	58	59	60	61:63

Table 5.1: DSCR register layout [18]

Where:

- 39 Software Transient Enable (SWTE)
  Applies the transient attribute to software-defined streams
- 40 Hardware Transient Enable (HWTE)
  Applies the transient attribute to hardware-detected streams
- 41 Store Transient Enable (STE)
  Applies the transient attribute to store streams.
- 42 Load Transient Enable (LTE)
  Applies the transient attribute to load streams.
- 43 Software Unit count Enable (SWUE)
  Applies the unit count to software-defined streams.
- 44 Hardware Unit count Enable (HWUE)
  Applies the unit count to hardware-detected streams.

- 45:54 Unit Count (UNITCNT)
  Number of units in data stream. Streams that exceed this count are terminated.
- 55:57 Depth Attainment Urgency (URG)
  This field indicates how quickly the prefetch depth can be reached for hardware-detected streams.
- 58 Load Stream Disable (LDS)
  Disables hardware detection and initiation of load streams.
- 59 Stride-N Stream Enable (SNSE)
  Enables hardware detection and initiation of load and store streams that have a stride greater than a single cache block
- 60 Store Stream Enable (SSE)
  Enables hardware detection and initiation of store streams.
- 61:63 Default Prefetch Depth (DPFD)
  Supplies a prefetch depth for hardware-detected streams and for software-defined streams
- 55:57 Depth Attainment Urgency (URG)
  This field indicates how quickly the prefetch depth can be reached for hardware-detected streams. Values and their meanings are as follows:
  - 0: Default
  - 1: Not urgent
  - 2: Least urgent
  - 3: Less urgent
  - 4: Medium
  - 5: Urgent
  - 6: More urgent
  - 7: Most urgent

There is little information about what these bits are really used for, we did a previous research on data prefetch based on some previous work [21,22,32] and we found out that the bits that impact the most on performance are:

- LDS: If this bit is 0 there will be no data prefetch for loads
- SNSE: When an application is accessing non-consecutive data with a fix stride, enabling this bit makes the data prefetch bring cache blocks that have a distance of the stride
- SSE: When this bit is enabled and an store instruction is executed the data prefetcher will bring to the L1 cache the cache block corresponding
- URG: With minor impact on performance, these bits indicates how many cache blocks the data prefetch is going to bring to the L1 cache of the core, the default one corresponds to bring 4 caches blocks. In some cases, increasing the number of blocks to be brought can reduce the execution time. In the other hand, in some cases, decreasing the number of blocks to be brought can affect positively to the bandwidth wasted in data that later will not be accessed

5.2 Metrics

In this work we will analyze 2 metrics to evaluate performance of libPRISM: execution time and power. We expect to reduce execution time and power by doing a smarter utilization of hardware resources, but our believe is that power consumption will be more affected by setting the configuration of the data prefetcher. At some point the data prefetcher configuration can use more power due to be more aggressive or in the extreme case that the prefetcher is disabled it will be using much less power.

5.2.1 Performance

In order to measure execution time libPRISM gathers different data (as explain in section 4.1.2) and one of them is elapsed time.

libPRISM reads from the timebase register in our platform, in the case of POWER processors this is the special register 268. This register allow us to measure elapsed time spent since starting the execution to the end with a small overhead cost.

This time measurement is done for the whole execution of the workload and per each parallel region in the workload, having more fine-grain knowledge about the characteristics of the workload and how good libPRISM behaves.

5.2.2 Power and energy

Even libPRISM is already reducing energy consumption due to shorter execution times, we want to measure the power consumption to see if there is any benefit in terms of power using libPRISM; setting the SMT level is affecting in how the workloads behave, therefore it is probably that the processor changes its power consumption. Also, tunning the data prefetcher can affect to the power consumed by the memory, this effect should be more notable when disabling the data prefetcher.

For this purpose we will use a tool from IBM called AMESTER (Automated Measurement of Systems for Energy and Temperature Reporting) [16]. This is a research tool to remotely collect power, thermal and performance metrics from IBM servers. AMESTER is a non-intrusive tool that does not use any of the processing cycles of the system connected to, therefore has no impact on the performance and it does not need any support from the operative system. Also, it allows to use scripts to capture and transcribe the read data to local files.

In order to use it, we will connect to the Flexible Service Processor (FSP) located in the IBM machine (in our case a POWER8) using our laptop, to not disturb performance, and execute a script to read the different sensors we need for our experiments: power for the core, uncore and memory.

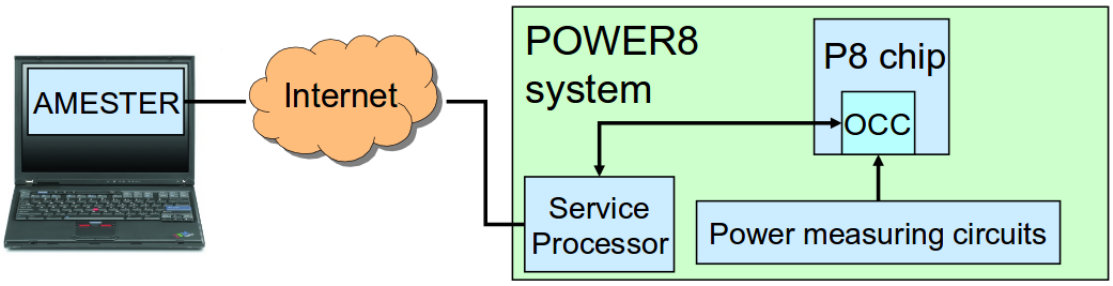


Figure 5.3: AMESTER connection scheme

5.3 Benchmarks

This section describes the different workloads used in this thesis. If possible, we have chosen a full group of workloads predefined and accepted by the community as suite benchmarks.

The suites chosen for the purpose of testing our library are based on several requirements:

- Have to be written in OpenMP. As explained, libPRISM works with OpenMP runtimes.
- Enough execution time to be able to observe different phases and reduce noise between experiments.
- Different input size. To prove that our solution works fine with different inputs and execution times.
- In the best scenario, we do not need message passing workloads since we will just use one machine.
- Ideally, workloads can be compiled in a Power architecture without having to re-write a lot of code.

All the selected suites are tested for the community. The following sections provide information about the selected suites, the workloads part of the suites and the different inputs used.

5.3.1 NAS

The NAS Parallel Benchmarks (NPB) were designed to help evaluate the performance of parallel computers. The suite contains different benchmarks with different predefined problem sizes indicated as classes.

All of them are written in Fortran except for one of them, written in C (IS).

We will be using the NAS version 3.3.1, which includes the benchmarks in MPI, OpenMP and serial versions [23]. From all the benchmarks included, we will just analyze the following benchmarks coded in OpenMP:

Benchmark	**Description**
IS	Integer Sort, random memory access
EP	Embarrassingly Parallel
CG	Conjugate Gradient, irregular memory access and communication
MG	Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive
FT	Discrete 3D fast Fourier Transform, all-to-all communication
BT	Block Tri-diagonal solver
SP	Scalar Penta-diagonal solver
LU	Lower-Upper Gauss-Seidel solver

Table 5.2: NAS Benchmarks description

We only analyze classes C and D, from all the classes: A,B,C,D,E,S (from smaller to bigger input). We picked those 2 because of time (long enough to avoid noise and short enough to run different times) and to prove that with different inputs the applications can have different behavior in terms of performance depending on the SMT level.

5.3.2 SPEC OMP 2012

The SPEC OMP 2012 benchmarks are designed for measuring performance using applications based on the OpenMP 3.1. It contains 14 scientific and engineering

application codes covering a wide range of domains. [30] They are written mostly in C and Fortran, and one of them in C++.

The benchmarks analyzed are:

Benchmark	**Language**	**Application domain**
350.md	Fortran	Physics: Molecular Dynamics
351.bwaves	Fortran	Physics: Computational Fluid Dynamics (CFD)
352.nab	C	Molecular Modeling
357.bt331	Fortran	Physics: Computational Fluid Dynamics (CFD)
358.botsalgn	C	Protein Alignment
359.botsspar	C	Sparse LU
360.ilbdc	Fortran	Lattic Boltzmann
362.fma3d	Fortran	Mechanical Response Simulation
363.swim	Fortran	Weather Prediction
367.imagick	C	Image Processing
370.mgrid331	Fortran	Physics: Computational Fluid Dynamics (CFD)
371.applu331	Fortran	Physics: Computational Fluid Dynamics (CFD)
372.smithwa	C	Optimal Pattern Matching
376.kdtree	C++	Sorting and Searching

Table 5.3: SPEC OMP 2012 Benchmarks description

All the benchmarks have been run with the reference input (the largest one) in order to try to reflect the large HPC applications.

5.3.3 CORAL

As a part of a collaboration between Argonne National Laboratory, Lawrence Livermore National Laboratory and Oak Ridge National Laboratory different representative workloads in the HPC world were selected to study performance in large computers.

The CORAL suite contains different benchmark categories: Scalable science Benchmarks, throughput benchmarks, data centric, skeleton and micro benchmarks;

the total number of existing benchmarks are above 30, not all the codes are in OpenMP, therefore the number of benchmarks we can use is less than 30. Due to this we selected a reduced number of them (see table 5.4) trying to pick one of each category to be fair, but we could not afford to pick one of the "Scalable science" category because of time to run them.

Benchmark	**Category**	**Comments**
LULESH	Throughput	Shock,hydrodynamics for unstructured meshes. Fine-grained loop level threading.
HACCmk	Microkernel	Single,core optimization and SIMD compiler challenge, compute intensity.
graph500	Data-Centric	Scalable,breadth-first search of a large undirected graph.
AMGmk	Microkernel	Three,compute intensive kernels from AMG.

Table 5.4: Selection of CORAL benchmarks

Chapter 6

Evaluation

First of all, we need to check how our exploration policy performs (see section 4.3.2) performs against an ideal execution (see section 4.3.1). With this purpose we used the NAS suite (see section 5.3.1) mainly because the execution time is more affordable than with the other suites. Figure 6.1 displays this comparison. It shows the performance when running in:

- ST
- SMT8
- BSA. Best Static per Application (i.e. run the benchmark with the best SMT level)
- BSPR. Best Static per Parallel Region: After profiling the application we select the best configuration for each parallel region, this information is fed to our oracle policy, which runs the benchmark with the best configuration for each parallel region.
- libPRISM. Using our exploration policy.

Results confirm that our exploration policy is close to the ideal execution. We can highlight some behaviors:

- CG, EP, FT have no improvement in reconfiguring the hardware and libPRISM has no degradation

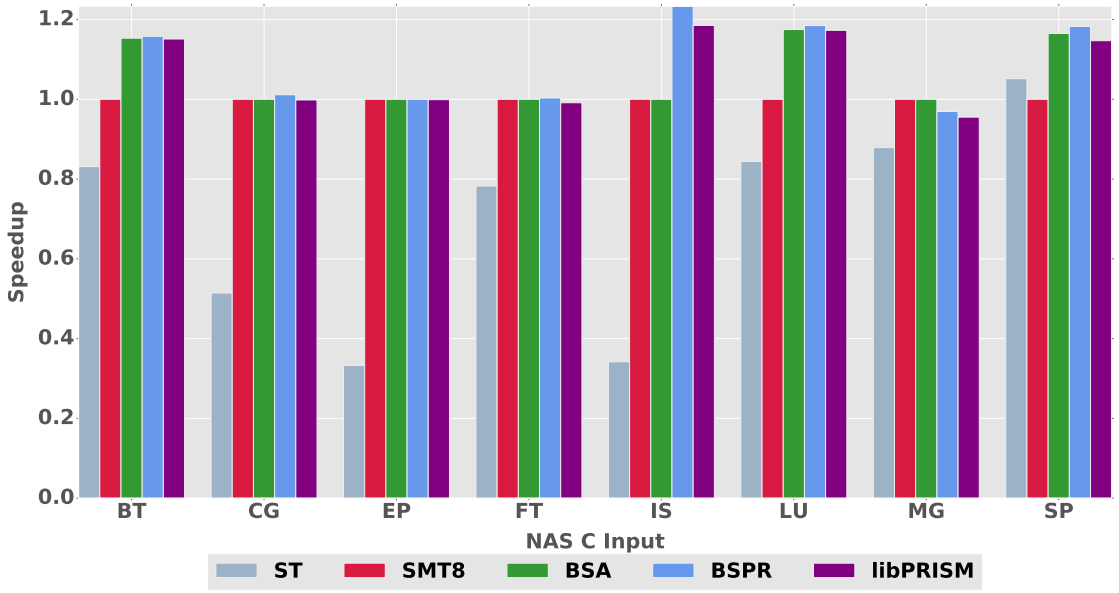


Figure 6.1: Performance comparison when using static SMT levels (ST, SMT8, Best Static Per Application) and dynamic SMT levels (Best Static per Parallel Region and libPRISM)

- BT, IS, LU and SP have an improvement because of the reconfiguration of the SMT level and libPRISM is able to automatically reconfigure the hardware. With libPRISM we have a degradation in IS and SP with respect to the Best Static per Parallel Region: IS has only 11 iterations of a parallel region, this produces a greater overhead when doing the exploration. SP has hundreds of iterations but the behavior of the iterations change over the time and libPRISM needs more time to capture that behavior, leading to a 4% drop in the speedup with respect to the default SMT8 level

- MG actually shows a degradation in performance when using a Best Static per Parallel Region or libPRISM. The reason have been already commented in the previous section 4.3.1; there are small parallel regions where reconfiguring the number of threads has more overhead than the actual work to do inside the parallel region.

Next thing we have to check is the overheads produced by the use of libPRISM. We run libPRISM but this time, libPRISM is not carrying out any hardware reconfiguration; we should appreciate the overheads of the different mechanism we

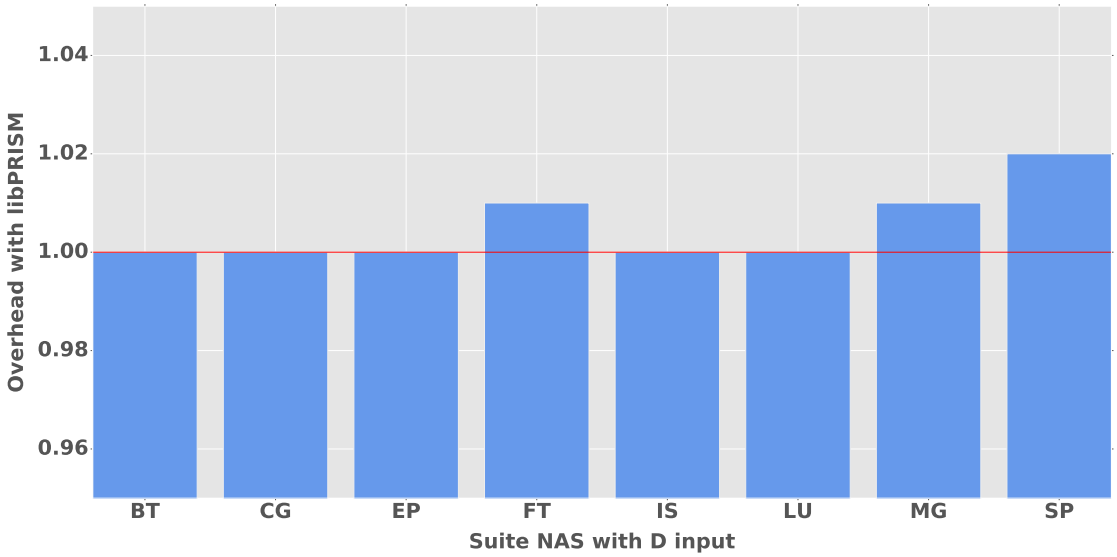


Figure 6.2: Overheads produced by libPRISM

are using. The benchmarks will run as they were with the default configuration SMT8 and prefetcher enabled but with libPRISM's infrastructure. This is shown in figure 6.2, which proves a maximum overhead of 2%.

6.1 Results

6.1.1 NAS

Figure 6.3 shows the results for the NAS suite in terms of speed up with respect to SMT8 level and prefetcher enabled (i.e. bars of SMT8 will be always 1). Plots (a) and (b) show the behavior for C and D input respectively. Again, we show ST mode and Best Static per Application for reference purposes.

At first sight we can see a difference between C and D input: this confirms that the hardware configuration should not be specific for application but for application and input data; which makes more important the runtime that handles this reconfiguration in order to free the programmer to know every detail of the hardware architecture.

Looking at the C input we can see some behaviors:

- BT: with a static SMT level (BSA) is enough to achieve the best performance. libPRISM gets the same performance (15%).
- CG: default configuration is the best configuration. libPRISM does not get any slowdown.
- EP: default configuration is the best configuration. libPRISM does not get any slowdown.
- FT: default configuration is the best configuration. libPRISM does not get any slowdown.
- IS: with a static approach the BSA configuration does not reduce execution time, but libPRISM can get up to 18% speedup. The reason for this we found in how the benchmark works, it has 2 parallel regions that have a different optimal SMT level (SMT4 and SMT8) at the moment we set a static configuration the speedup from a parallel region is not reflected because of the slowdown in the other parallel region.
- LU: as in the BT benchmark, a static configuration is enough to get the best performance and again libPRISM is able to detect it and not lose performance.
- MG: in this case libPRISM is getting a slowdown of 5%. This is due to the libPRISM keeps resettings it exploration phase because the irregularity of the input.
- SP: again a static approach for the hardware configuration is enough. libPRISM gets a slowdown of 2% compared to the BSA because of resetting the exploration phase as in the MG benchmark, but in this case SP has more iterations to be able to reduce the impact of it.

Then, analyzing the data for the D input we can observe differences:

- BT: a static approach is enough to get the best performance, which is a 10% speedup in SMT4 level.
- CG: a static approach is enough to get the best performance, which is a 39%

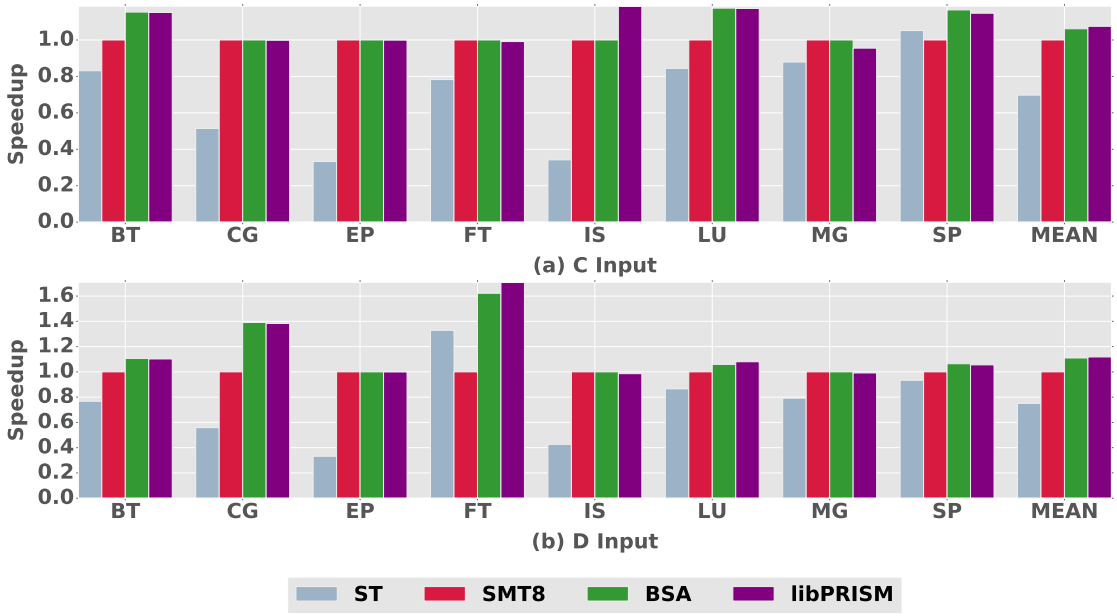


Figure 6.3: Performance using libPRISM in NAS suite with C and D inputs.

speedup in SMT8 level and the data prefetcher disabled.

- EP: default configuration is the best configuration. libPRISM does not get any slowdown.
- FT: this is an example where there are parallel regions with different optimal configuration (SMT4 and SMT8): with a static approach we get a 62% speedup, but with libPRISM we can increase that speedup to 71%
- IS: default configuration is the best configuration. libPRISM does not get any slowdown.
- LU: a static approach with a SMT4 gets a 6% speedup, but libPRISM can get a 8% speedup.
- MG: default configuration is the best configuration. libPRISM does not get any slowdown.
- SP: as explained in the C input, libPRISM loses a 1% of speedup with respect to the BSA corresponding to SMT4 (6% speedup) because of the exploration phase.

In terms of power we show the contribution for the core and memory to the total dynamic power consumption using the default configuration and with libPRISM in figure 6.4 (only D input). Values are normalize to the default configuration.

Generally, there are no big differences in the power breakdown, but there are several important points to highlight:

- Logically, the benchmarks were libPRISM changes nothing on the hardware configuration the power breakdown it is the same.
- BT: libPRISM change the hardware configuration, therefore we can see this reflected in the power consumption. The total dynamic power consumption is reduced by a 10% with respect to the default configuration, which 7% is reduced from memory and 3% from the core
- CG: In this benchmark, libPRISM can turn off the prefetcher, but interestingly, power consumption for memory increases. Actually, with the if we do not turn off the prefetcher, the best SMT level is 4 with a speedup of 2%, but at the moment we turn off the prefetcher the best SMT level is 8. It seems a problem where threads cannot access memory if the prefetcher brings more than the actual and needed line.
- FT: This benchmarks benefits a lot from libPRISM in terms of execution time, but it has a side effect on power. Power consumption of the memory goes down a 5%, but the dynamic power consumption of the core goes from a 66% contribution to the total power to a 75% (9% difference in dynamic power).
- LU: even libPRISM sets the hardware to a different configuration we cannot appreciate a real difference in terms of power consumption.
- SP: again libPRISM reconfigures the hardware to obtain a speedup in terms of execution time and in terms of power this translate to a reduction of 8%, 2% from memory and the other 6% from the core.

In Figure 6.5 we show the energy consumption with libPRISM normalized to the default hardware configuration. We can observe only a change of the energy consumption when libPRISM can increase performance in terms of execution time:

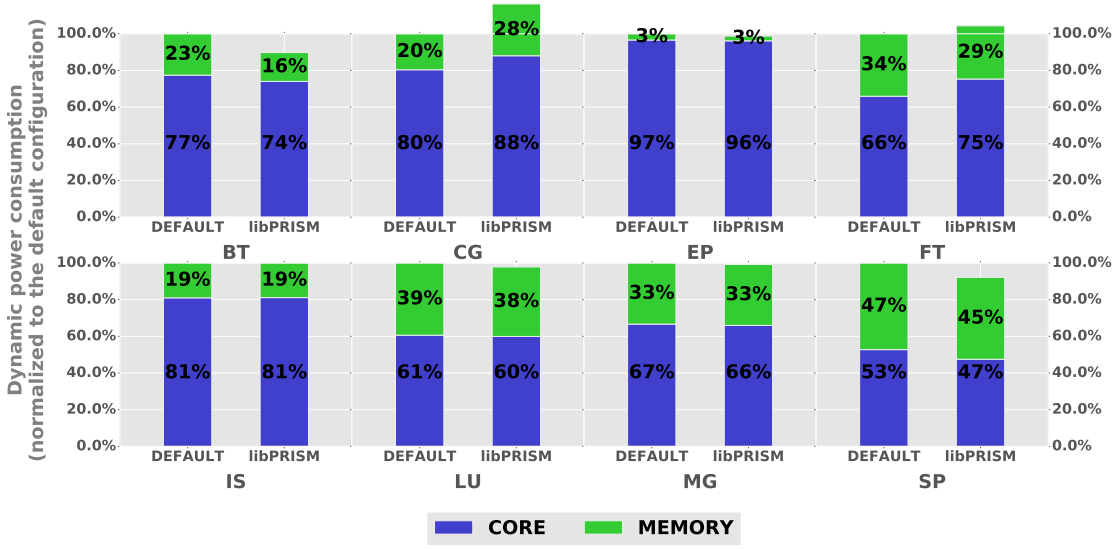


Figure 6.4: Power contribution of NAS suite D Input. Values are normalized to the 100% consumption when running with default configuration

- BT: LibPRISM reduces energy by reducing the execution time and reducing the total power consumption by a 12%.
- CG: The reduction comes mainly by the fact that we can speedup up the execution by turning off the prefetcher, even this implies an increase in the dynamic power consumption of memory.
- FT: In this case power consumption with libPRISM increases, but then libPRISM speedups the execution time. This translates to an energy savings of 78%.
- LU: Energy is reduced a 18% with respect the default configuration.
- SP: LibPRISM reduces energy by reducing the execution time and reducing the total power consumption. This gives us an energy savings of 10%.

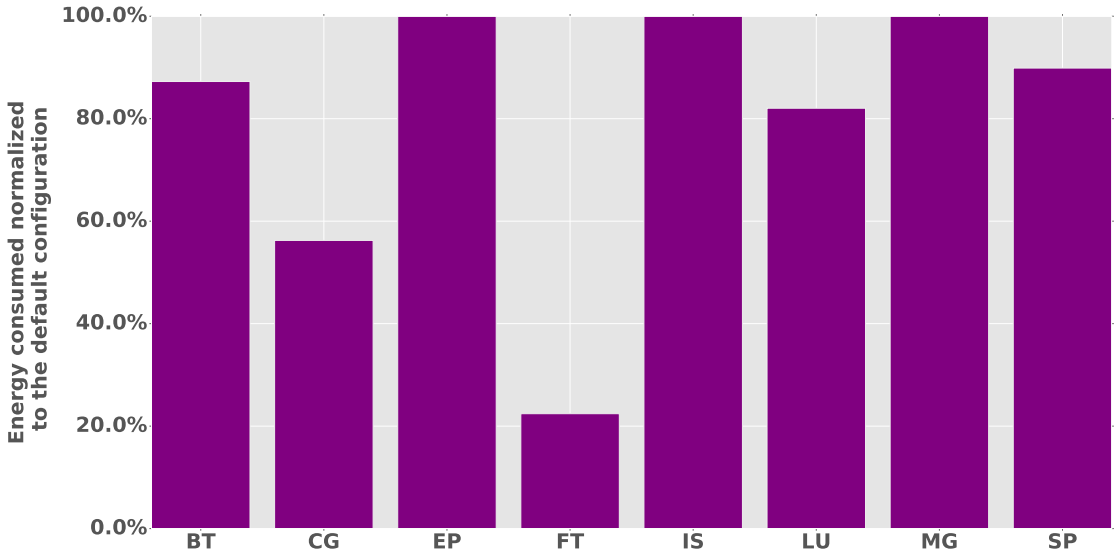


Figure 6.5: Energy consumption with libPRISM for the NAS suite with the D input

6.1.2 SPEC OMP 2012

Respect to SPEC OMP 2012 suite, results are display in figure 6.6. We can see that almost all the benchmarks work great with the default configuration and libPRISM has to change the hardware in few benchmarks:

- Botsalgn: The static approach gets a 9% speedup while libPRISM achieves a 7% speedup. This loss is due to the variability in execution time of tasks (usually smaller and higher variability than normal *parallel for* regions), therefore for tasks we needed to increase the exploration phase to reduce noise. This benchmark benefits of changing the SMT level to 4.
- Botsspar: Both of the static approach (BSA) and libPRISM get a 25% speedup. This benchmark as Botsalgn has only 1 parallel region, which gets the optimal performance with SMT4 and default prefetcher.
- Ilbdc: a static approach for the hardware configuration gets the best performance for this benchmark. BSA and libPRISM get a 12% speedup. The best configuration for Ilbdc is SMT4 with the default prefetcher.
- Mgrid331: BSA gets a 113% speedup with respect to the default configura-

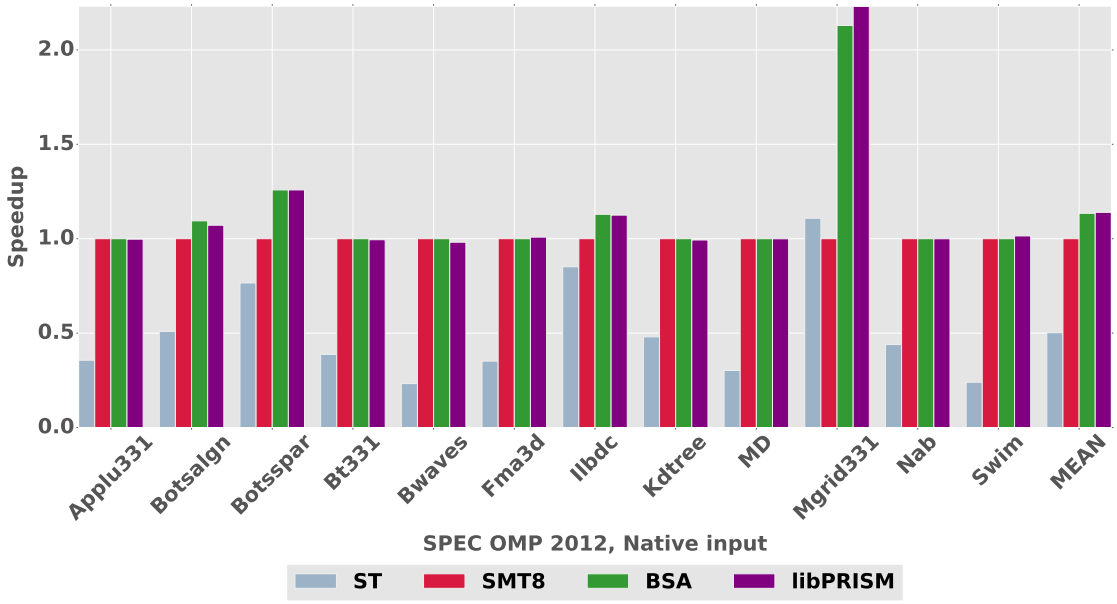


Figure 6.6: Performance using libPRISM in SPEC OMP 2012 suite with native input

tion, but libPRISM (changing the SMT level between 4 and 8) is able to get 10% more (123% with respect to the default configuration) because it has parallel regions with different optimal hardware configuration.

In figure 6.7 we show the dynamic power consumption with the contribution to the total dynamic power consumption for the core and memory hardware components. In the cases where libPRISM can get an speedup we shall expect a variation on the power consumption:

- Botsalgn: Lowering the SMT level to 4 produces a reduction in the dynamic power consumption. Power goes down a 5% of the total dynamic power. The reduction comes only from the core, since it is a benchmark that does not use a lot of power for memory.
- Botsspar: libPRISM is able to get a reduction of a 12% of the total power consumption. The contribution of the memory goes down to a 2% (libPRISM reduced a 3%) and the core goes down a 9%.
- Ilbdc: Setting a different hardware configuration is only reflected on the power consumption by a reduction of 2% of the core.

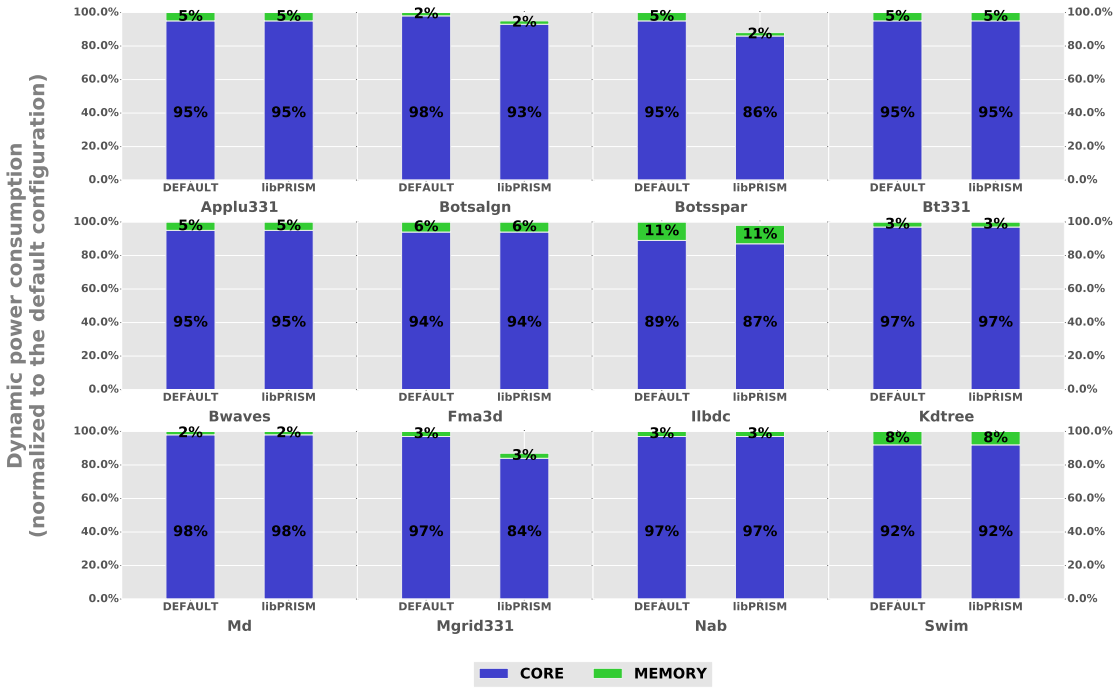


Figure 6.7: Power contribution of SPEC OMP 2012 suite with the native input. Values are normalized to the 100% consumption when running with default configuration

- Mgrid331: Similar to Ilbdc benchmark. libPRISM lowers the power consumption of the core by a 13%.

To finishing the analysis for the SPEC OMP 2012 suite we show the energy consumption in figure 6.8. We see a similar trend to the NAS suite, where the energy reduction mainly comes from the reduction on the execution time. As expected, if the default hardware configuration is the best configuration there are no differences in terms of energy, but libPRISM can actuate in several benchmarks:

- Botsalgn: The speedup obtained in execution time (7%) plus the reduction on the power consumption for the core is from libPRISM gets the energy reduction of a 8%.
- Botsspar: Energy is reduced a 23% with respect to the default execution.
- Ilbdc: Even getting a 1.12x speed up on execution time, energy is only reduced by a 6%.

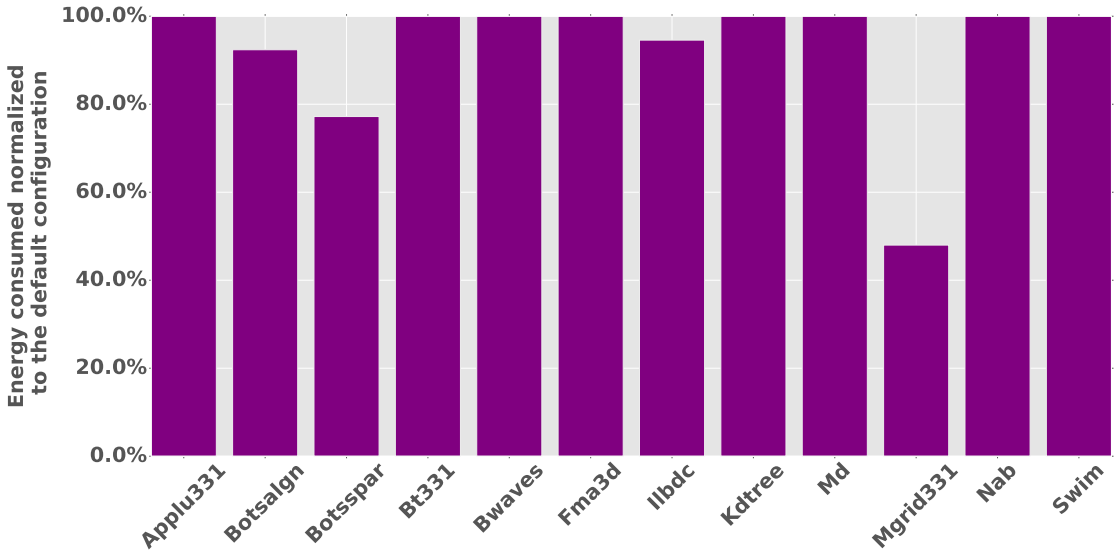


Figure 6.8: Energy consumption with libPRISM for SPEC OMP 2012 suite

- Mgrid331: In this case we can reduce to more than a half the execution time and the energy. Energy is reduced to a 46% normalized to the execution without libPRISM support.

6.1.3 CORAL Benchmarks

Running libPRISM with the CORAL benchmarks produce the results shown in figure 6.9. In this figure we can observe a behavior for each benchmark:

- Lulesh: This is one example of great benefits, setting the correct hardware configuration we can get up to a 1.55x speedup with a static approach. libPRISM is able to capture the behavior and obtain the same speedup automatically.
- HACC: This benchmark works fine with the default configuration with only one detail, it is not using the prefetched data therefore libPRISM detects it and disable the prefetcher.
- graph500: As we can see comparing the default configuration with SMT8 and the Best Static per Application (BSA) the best SMT level is 8. But libPRISM can get a 5% speedup disabling the prefetcher because it produces

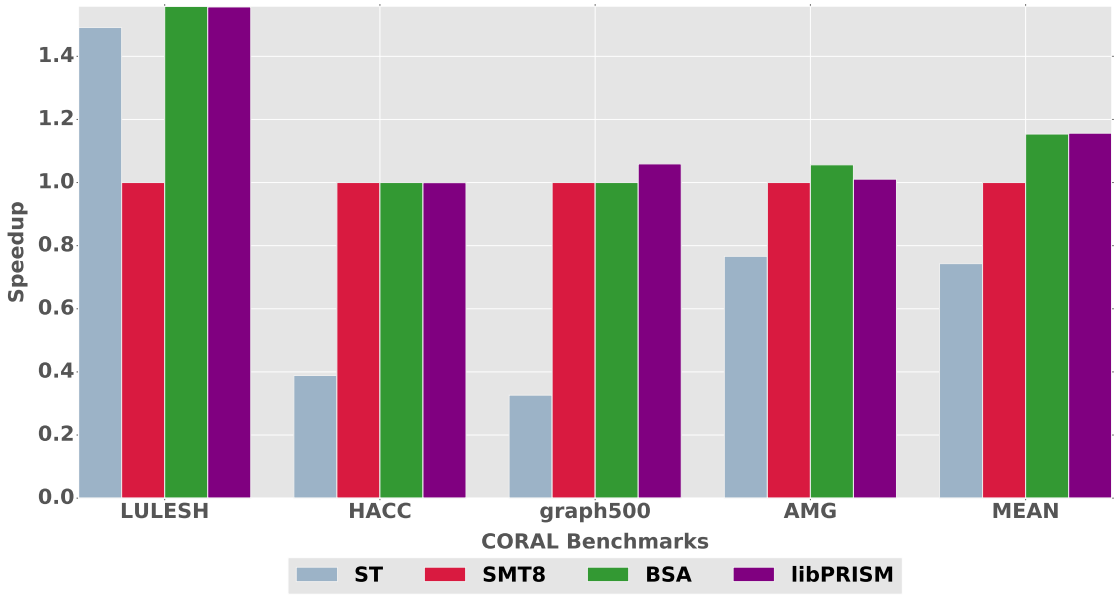


Figure 6.9: Performance using libPRISM in CORAL benchmarks

a slowdown in the benchmark. This is because is an algorithm traversing a graph, therefore will not use the prefetched data (but still will have to prefetch it)

- AMG: This benchmark with a static approach can get a 5% speedup, with libPRISM we cannot achieve that because of the duration of the parallel regions. AMG is composed by one small parallel region repeated thousands of times and, as explained in section 4, libPRISM cannot capture this small parallel regions because of the overhead produced by the GOMP runtime.

Power breakdown (core and memory components) for the CORAL benchmarks are shown in figure 6.10, where we can see some new behaviors:

- Lulesh: Setting the SMT level to a lower level (SMT4) we can obtain a general reduction on power. Power for the core is reduced 6% and memory a 7%.

- HACC: As said previously, this benchmark is able to do all the computation with data on the caches, therefore enabling, increasing aggressiveness or disabling the prefetcher makes no difference. Also, we saw in the performance figure (see 6.9) that HACC runs better with SMT8.

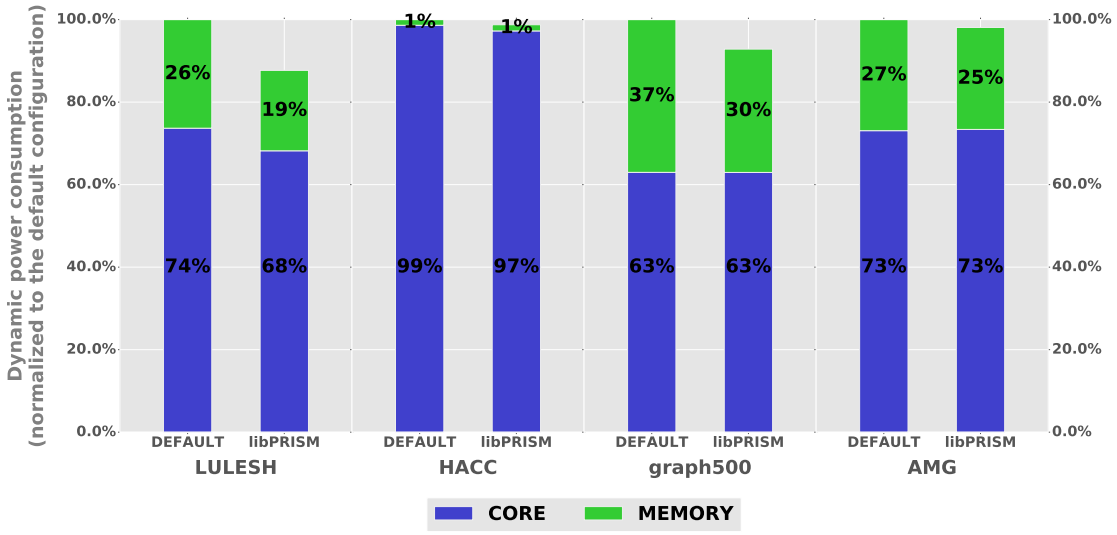


Figure 6.10: Power contribution of CORAL benchmarks. Values are normalized to the 100% consumption when running with default configuration

- graph500: In this benchmark libPRISM is able to disable the prefetcher to speedup the execution time, this is reflected in the drop of the memory component (of 7%) when using libPRISM.
- AMG. Here libPRISM cannot actuate because the reasons stated before, therefore the power breakdown is the same with libPRISM

Energy for the CORAL benchmarks is shown in figure 6.11 from these figures we can observe how the reduction in execution time and power consumption translates to the energy:

- Lulesh: Here we can save up to 37% of energy thanks to libPRISM, which is able to reduce execution time and power.
- HACC: This benchmarks runs better with the default configuration, therefore libPRISM does not change the hardware configuration. No energy differences are appreciated.
- graph500: Thanks to disable the data prefetcher and reduce the execution time energy savings are reduced by a 8% with respect the default configuration.

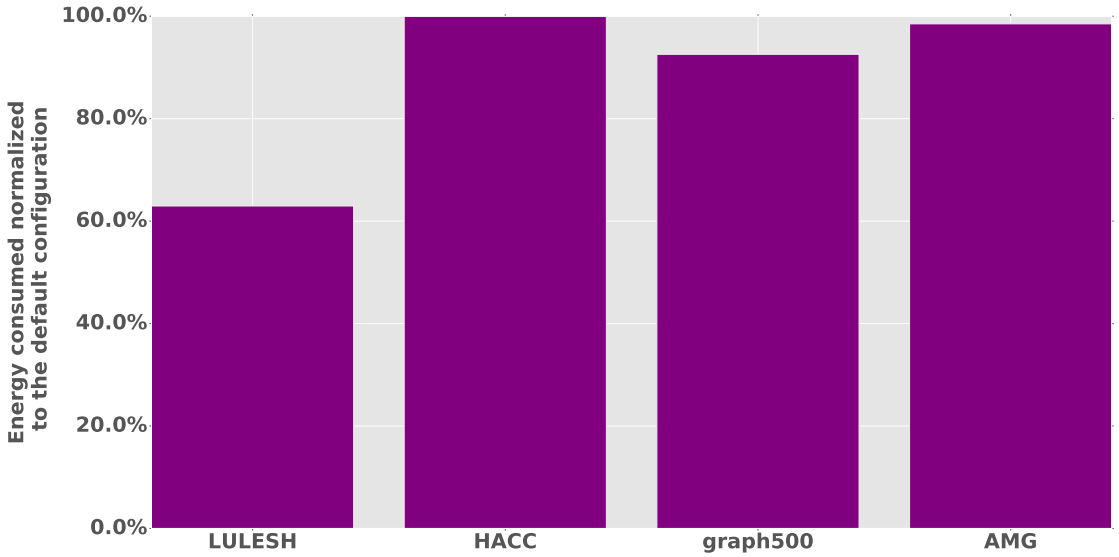


Figure 6.11: Energy consumption with libPRISM

- AMG: This benchmarks runs better with the default configuration, therefore libPRISM does not change the hardware configuration. No energy differences are appreciated.

Chapter 7

Conclusions and future work

As hardware gets more complex it will be needed an abstraction layer to offer to programmers increase their workload performance (e.g. multicore processors can be found in any place and thanks to parallel programming models such as OpenMP we have been able to use them in our benefit).

Hardware vendors keep pushing new features in their processors to increase performance while keeping in mind the current problems (e.g. memory and power wall): data prefetch, SMT, thread priorities, etc.

All these techniques have been proved to increase performance if used correctly; which means that programmers need to be aware of the architecture but also they need to take care of the possible interactions between different techniques.

This programmability wall can have a huge effect on the efforts to develop and maintain software: redesign if the architecture changes, add extra functionalities in order to use new architecture changes, adapt the code if they want to run in a different platform, etc.

In this thesis we contributed to this problematic with libPRISM: an intelligent library that reconfigures the underlying hardware to increase performance and reduce power consumption.

libPRISM does all the analysis of the executed workload and reconfigures the different hardware knobs for the benefit of the end user, and because it runs on

the top of the OpenMP runtime it also does it transparently to the programmer, who just need to code with OpenMP as they were doing before. libPRISM just needs to be able to do library interposition and that is something that almost all operative system already have, in the specific case of Linux is done through the environment variable LD_PRELOAD.

We have tested libPRISM against major and accepted benchmark suites: NAS, SPEC OMP 2012 and CORAL benchmarks. Through this testing we have seen that workloads demands are not only based on the application but also based on the data is processing, this characteristic makes possible intra-phases in the same workload where the optimal configuration can differ from other phases. This fact strongly suggest that we need to reconfigure the hardware per phase instead of per workload: this can be done after a comprehensive profiling or with a runtime mechanism such as libPRISM.

Also, we have seen that disabling the prefetcher helps more to increase performance when the SMT level is higher. Good examples of this behavior is CG from the NAS suite or graph500 from the CORAL suite.

libPRISM can get up to 2.22x speedup (1.15x in average) in execution time while decreasing dynamic power consumption by a 13% (2% in average) by just reconfiguring the SMT level and data prefetcher.

7.1 Future work

One behavior that we would have liked to see more is the relation between SMT and data prefetcher: higher aggressiveness for the data prefetcher implies a lower SMT level, and a lower aggressiveness of the data prefetcher would imply a higher SMT level. We just saw the second behavior, the lowest aggressiveness of the data prefetcher (disabled) with the highest SMT level gives the most performance.

And also related with data prefetcher, we could only apply 2 optimal prefetcher configurations: default and disabled. Probably because the benchmarks tested use a small percentage of bandwidth that our machine can support, therefore, different prefetcher configurations have a very small impact to be noticed. We

could always increase the input size but we did not for 2 reasons: (1) we wanted to use predefined and tested inputs in order to be able to verify at every moment we were getting the desired output and (2) time constraints, as explained in section 5.3 some of the benchmarks have a reasonable time (NAS suite) but the others already took a considerable amount of time to execute with the native input.

We would like to test different benchmarks where we can see those 2 behaviors that we think we are missing, but the number of benchmarks coded in OpenMP are fewer if we compare to benchmarks coded in another parallel programming model such as pthreads, for example.

Another path to follow is to decrease the overhead in training our policy, we could implement a machine learning policy where we train libPRISM before executing applications. This idea can have a major impact in the granularity we have defined (i.e. parallel regions) and we would might want to change it. But this idea could have a negative aspect, as said before we think we are not seeing all the behaviors we would like, then when training our algorithm we could potentially miss some behaviors, therefore our algorithm would not be able to match correctly all the possibilities.

Also, the processor used in this work have more knobs that can be reconfigured. We would like to increase the possibilities of libPRISM to all the knobs in the machine, coordinating all of the hardware at the same time in order to boost performance. One idea that came to our mind was to use thread priorities to make possible to coordinate different workloads to achieve their maximum speedup when they are not running in isolated.

Acronyms

DCM Dual Chip Module. 42

DSCR Data Stream Control Register. vi, ix, 45, 46, *Glossary:* Data Stream Control Register

ELF Executable and Linkable Format. 23

GCC The GNU Compiler Collection. 16, 19–22

GOMP GNU OpenMP implementation. xi, 22, 24, 26, 27, 33, 44

OpenMP Open Multi Processing. i, vii, 4, 16, 17, 19–22, 25–27, 29, 30, 43, 44, 50, 51, 53, 69–71, *Glossary:* OpenMP

PMC Performance Monitor Counter. 27–29, *Glossary:* Performance Monitor Counter

SMT Simultaneous MultiThreading. i, vi, vii, ix, 1, 3, 4, 7–12, 19, 21, 27, 28, 33–36, 38, 39, 41, 44–46, 49, 51, 55–60, 62, 63, 65, 66, 69, 70, *Glossary:* Simultaneous MultiThreading

ST Single Thread. 38, 41

Glossary

Data Stream Control Register Register used to control the degree of aggressiveness of memory prefetching for load and store instructions. vi

OpenMP It is an application programming interface for shared memory processors based on the fork-join model. i

Performance Monitor Counter A set of special-purpose registers built into moder microprocessors to store the counters of hardware-related events that have happened in the system. The number and the possible events to record are hardware dependent but usually a vendor always implement the same events in its different CPUs. 27

Reduced instruction set computing It is a CPU design strategy based on the insight that a simplified instruction set (as opposed to a complex set) provides higher performance when combined with a microprocessor architecture capable of executing those instructions using fewer microprocessor cycles per instruction. 41

Simultaneous MultiThreading Technique for improving the overall efficiency of super scalar CPUs with hardware multithreading. SMT permits multiple independent threads of execution to better utilize the resources trying to fill all the different queues and execution units in a modern processor architecture using more threads per core. i

Bibliography

[1] Bitirgen, R., Ipek, E., and Martinez, J. F. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2008), MICRO 41, IEEE Computer Society, pp. 318–329.

[2] Boneti, C., Cazorla, F. J., Gioiosa, R., Buyuktosunoglu, A., Cher, C. Y., and Valero, M. Software-controlled priority characterization of power5 processor. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on* (June 2008), pp. 415–426.

[3] Boneti, C., Gioiosa, R., Cazorla, F. J., Corbalan, J., Labarta, J., and Valero, M. Balancing hpc applications through smart allocation of resources in mt processors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on* (April 2008), pp. 1–12.

[4] Boneti, C., Gioiosa, R., Cazorla, F. J., and Valero, M. A dynamic scheduler for balancing hpc applications. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC '08, IEEE Press, pp. 41:1–41:12.

[5] Casas, M., and Bronevetsky, G. Active measurement of memory resource consumption. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014* (2014), pp. 995–1004.

[6] Casas, M., and Bronevetsky, G. Evaluation of HPC applications' memory resource consumption via active measurement. *IEEE Trans. Parallel Dis-*

trib. Syst. 27, 9 (2016), 2560–2573.

[7] Casas, M., Moreto, M., Alvarez, L., Castillo, E., Chasapis, D., Hayes, T., Jaulmes, L., Palomar, O., Unsal, O., Cristal, A., Ayguade, E., Labarta, J., and Valero, M. *Runtime-Aware Architectures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 16–27.

[8] Cazorla, F. J., Fernandez, E., Ramírez, A., and Valero, M. *Improving Memory Latency Aware Fetch Policies for SMT Processors*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 70–85.

[9] Cazorla, F. J., Knijnenburg, P. M., Sakellariou, R., Fernández, E., Ramirez, A., and Valero, M. Predictable performance in smt processors. In *Proceedings of the 1st Conference on Computing Frontiers* (New York, NY, USA, 2004), CF '04, ACM, pp. 433–443.

[10] Cray Inc. Chapel language specification, v 0.981, 2016.

[11] Creech, T., Kotha, A., and Barua, R. Efficient multiprogramming for multicores with scaf. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2013), MICRO-46, ACM, pp. 334–345.

[12] D. Marr, e. a. Hyperthreading technology architecture and microarchitecture. *IEEE Micro 6*, 1 (February 2002).

[13] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters 21*, 02 (2011), 173–193.

[14] Feliu, J., Eyerman, S., Sahuquillo, J., and Petit, S. Symbiotic job scheduling on the ibm power8. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (March 2016), pp. 669–680.

[15] Feliu, J., Sahuquillo, J., Petit, S., and Duato, J. Addressing fairness in smt multicores with a progress-aware scheduler. In *Parallel and*

Distributed Processing Symposium (IPDPS), 2015 IEEE International (May 2015), pp. 187–196.

[16] Floyd, M., Ware, M., Rajamani, K., Gloekler, T., Brock, B., Bose, P., Buyuktosunoglu, A., Rubio, J. C., Schubert, B., Spruth, B., Tierno, J. A., and Pesantez, L. Adaptive energy-management features of the IBM POWER7 chip. *IBM Journal of Research and Development 55*, 3 (May 2011), 8:1–8:18.

[17] Gottschalk, K. Industry insights: Openpower roadmap toward coral ibm. HPC Advisory Council Switzerland Conference, 2016.

[18] Hall, B., Bergner, P., Housfater, A., Kandasamy, M., Magno, T., Mericas, A., Munroe, S., Oliveira, M., Schmidt, B., Schmidt, W., et al. *Performance Optimization and Tuning Techniques for IBM Power Systems Processors Including IBM POWER8*. IBM Redbooks, 2015.

[19] Heirman, W., Carlson, T. E., Van Craeynest, K., Hur, I., Jaleel, A., and Eeckhout, L. Automatic smt threading for openmp applications on the intel xeon phi co-processor. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers* (New York, NY, USA, 2014), ROSS '14, ACM, pp. 7:1–7:7.

[20] Hur, I., and Lin, C. Memory prefetching using adaptive stream detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)* (Dec 2006), pp. 397–408.

[21] Jimenez, V., Buyuktosunoglu, A., Bose, P., O'Connell, F. P., Cazorla, F., and Valero, M. Increasing multicore system efficiency through intelligent bandwidth shifting. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on* (Feb 2015), pp. 39–50.

[22] Jiménez, V., Gioiosa, R., Cazorla, F. J., Buyuktosunoglu, A., Bose, P., and O'Connell, F. P. Making data prefetch smarter: Adaptive prefetching on power7. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2012), PACT '12, ACM, pp. 137–146.

[23] Jin, H.-Q., Frumkin, M., and Yan, J. The openmp implementation of nas parallel benchmarks and its performance.

[24] Kongetira, P., Aingaran, K., and Olukotun, K. Niagara: a 32-way multithreaded sparc processor. *IEEE Micro 25*, 2 (March 2005), 21–29.

[25] Li, M., Chen, G., Wang, Q., Lin, Y., Hofstee, P., Stenstrom, P., and Zhou, D. Pater: A hardware prefetching automatic tuner on ibm power8 processor. *IEEE Computer Architecture Letters 15*, 1 (Jan 2016), 37–40.

[26] Manousopoulos, S., Moreto, M., Gioiosa, R., Koziris, N., and Cazorla, F. J. Characterizing thread placement in the ibm power7 processor. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on* (Nov 2012), pp. 120–130.

[27] Mericas, A., Peleg, N., Pesantez, L., Purushotham, S. B., Oehler, P., Anderson, C. A., King-Smith, B. A., Anand, M., Arnold, J. A., Rogers, B., Maurice, L., and Vu, K. Ibm power8 performance features and evaluation. *IBM Journal of Research and Development 59*, 1 (Jan 2015), 6:1–6:10.

[28] Moreto, M., Cazorla, F. J., Ramirez, A., and Valero, M. Mlp-aware dynamic cache partitioning. In *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers* (Berlin, Heidelberg, 2008), HiPEAC'08, Springer-Verlag, pp. 337–352.

[29] Moseley, T., Kihm, J. L., Connors, D. A., and Grunwald, D. Methods for modeling resource contention on simultaneous multithreading processors. In *2005 International Conference on Computer Design* (Oct 2005), pp. 373–380.

[30] Müller, M. S., Baron, J., Brantley, W. C., Feng, H., Hackenberg, D., Henschel, R., Jost, G., Molka, D., Parrott, C., Robichaux, J., Shelepugin, P., Waveren, M., Whitney, B., and Kumaran, K. *OpenMP in a Heterogeneous World: 8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11-13, 2012. Proceedings.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, ch. SPEC OMP2012

— An Application Benchmark Suite for Parallel Systems Using OpenMP, pp. 223–236.

[31] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, Nov. 2015.

[32] Prat, D., Ortega, C., Casas, M., Moretó, M., and Valero, M. Adaptive and application dependent runtime guided hardware prefetcher reconfiguration on the IBM POWER7. *CoRR abs/1501.02282* (2015).

[33] Reinders, J. *Intel Threading Building Blocks*, first ed. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[34] Sinharoy, B., Kalla, R., Starke, W. J., Le, H. Q., Cargnoni, R., Norstrand, J. A. V., Ronchetti, B. J., Stuecheli, J., Leenstra, J., Guthrie, G. L., Nguyen, D. Q., Blaner, B., Marino, C. F., Retter, E., and Williams, P. Ibm power7 multicore server processor. *IBM Journal of Research and Development 55*, 3 (May 2011), 1:1–1:29.

[35] Snavely, A., and Tullsen, D. M. Symbiotic jobscheduling for a simultaneous multithreaded processor. *SIGARCH Comput. Archit. News 28*, 5 (Nov. 2000), 234–244.

[36] Tembey, P., Vega, A., Buyuktosunoglu, A., da Silva, D., and Bose, P. Smt switch: Software mechanisms for power shifting. *IEEE Computer Architecture Letters 12*, 2 (July 2013), 67–70.

[37] Tullsen, D. M., Eggers, S. J., and Levy, H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on* (June 1995), pp. 392–403.

[38] Valero, M., Moreto, M., Casas, M., Ayguade, E., and Labarta, J. Runtime-aware architectures: A first approach. *Supercomputing frontiers and innovations 1*, 1 (2014).

[39] Vega, A., Buyuktosunoglu, A., and Bose, P. Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. In *Parallel*

Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on (Sept 2013), pp. 245–256.

[40] Vega, A., Buyuktosunoglu, A., Hanson, H., Bose, P., and Ramani, S. Crank it up or dial it down: Coordinated multiprocessor frequency and folding control. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2013), MICRO-46, ACM, pp. 210–221.

[41] Wulf, W. A., and McKee, S. A. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News 23*, 1 (Mar. 1995), 20–24.

[42] Zhang, Y., Burcea, M., Cheng, V., Ho, R., and Voss, M. An adaptive openmp loop scheduler for hyperthreaded smps. In *In Proc. of PDCS-2004: International Conference on Parallel and Distributed Computing Systems* (2004).

[43] Zhang, Y., Voss, M., and Rogers, E. S. Runtime empirical selection of loop schedulers on hyperthreaded smps. In *19th IEEE International Parallel and Distributed Processing Symposium* (April 2005), pp. 44b–44b.