libPRISM: An Intelligent Adaptation of Prefetch and SMT Levels

Cristobal Ortega[1,2], Miquel Moreto[1,2], Marc Casas[2], Ramon Bertran[3]
Alper Buyuktosunoglu[3], Alexandre E. Eichenberger[3] and Pradip Bose[3]
[1]Universitat Politecnica de Catalunya (UPC) [2]Barcelona Supercomputing Center (BSC-CNS)
[3]IBM T.J. Watson Research Center

ABSTRACT

Current microprocessors include several knobs to modify the hardware behavior in order to improve performance under different workload demands. An impractical and time consuming offline profiling is needed to evaluate the design space to find the optimal knob configuration. Different knobs are typically configured in a decoupled manner to avoid the time-consuming offline profiling process. This can often lead to underperforming configurations and sometimes to conflicting decisions that jeopardize system power-performance efficiency. Thus, a dynamic management of the different hardware knobs is necessary to find the knob configuration that maximizes system power-performance efficiency without the burden of offline profiling.

In this paper, we propose libPRISM, an infrastructure that enables the transparent management of multiple hardware knobs in order to adapt the system to the evolving demands of hardware resources in different workloads. We use libPRISM to implement a policy that maximizes system performance without degrading energy efficiency by dynamically managing the SMT level and prefetcher hardware knobs of an IBM POWER8 system. We evaluate our solution using 24 applications from 3 different parallel benchmarks suites without the need of offline profiling or workload modification. Overall, the solution increases performance up to 220% (15.4% on average) and reduces dynamic power consumption up to 13% (2.0% on average) when compared to the static default knob configuration.

CCS CONCEPTS

•Computing methodologies → Parallel programming languages; •Computer systems organization → Multicore architectures; •Software and its engineering → Parallel programming languages; Software libraries and repositories;

1 INTRODUCTION

Multicore architecture is the main trend in processors development nowadays. Every new generation of processors is increasing the number of cores and the number of threads that can run within the same core (i.e. Simultaneous Multithreading or SMT). As a result, processor shared resources experience contention, which might lead to performance degradation. Processors have several hardware knobs to prevent performance degradation by adapting its behavior to workloads demands, such as the SMT, DVFS levels, the decode priorities or the data prefetcher settings. These knobs allow the user to tune the hardware to adapt it to workload demands.

Multiple policies have been proposed to derive suitable configurations for the hardware knobs, but these policies have always treated them independently of each other [3, 5, 25, 42, 43]. This independent actuation can lead to conflicting decisions that jeopardize system power-performance efficiency [39]. For example, a higher SMT level allows to increase the overall system throughput, but it reduces the effective bandwidth and last level cache size per thread. As a result, coordinating these decisions with other knobs that also contend for the memory bandwidth, such as the data prefetcher or DVFS, is required to optimize the overall system power-performance efficiency.

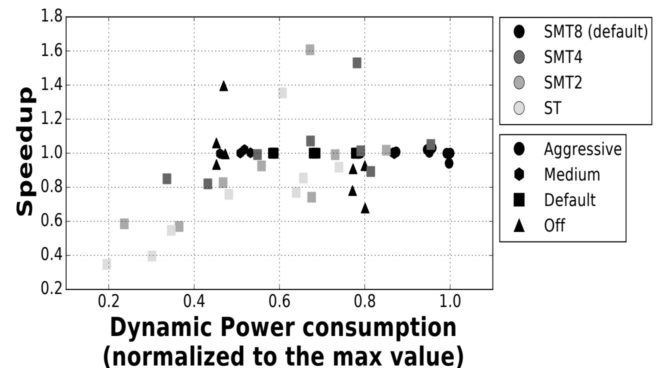

Figure 1: Application behavior of the NPB suite under different hardware knob configurations. The Y axis shows speedup with respect to the default hardware configuration. The X axis shows power consumption normalized to the maximum value observed. SMT level is represented by color and the shape corresponds to the data prefetcher configuration.

To illustrate the need for a coordinated adaptive system, Figure 1 shows the performance and the average power consumption of the NAS Parallel Benchmarks (NPB) [26] suite with different knob configurations: four SMT levels and four levels of aggressiveness for the data prefetcher[1]. Performance is normalized to the default configuration (SMT8 level and default prefetcher setting) and power is normalized to the maximum observed value. The highest SMT level is not always the optimal configuration due to increased last-level cache misses and contention in the execution units. An aggressive configuration for the prefetcher does not imply a better performance either, but it usually ends in more power consumption. In some cases, running in a low SMT level provides a modest 5% performance improvement, while disabling the prefetcher and running in the highest SMT level provides significant performance benefits (up to 40%) and reduces power consumption. Also, Figure 1 shows that different knob configurations yield a wide range of speedup and power consumption tradeoffs across applications. Furthermore, applications can have different intra resources demands, increasing even more the variety of best performing configurations.

An extensive and exhaustive offline profiling is required to discover the best hardware configuration per application. However, given the number of possible hardware configurations, performing an exhaustive profiling of each of them for each application and input data size is a time consuming process. In addition, since application optimal hardware configuration changes during different application execution phases, exploring all the hardware configuration for each application phase becomes unfeasible in a practical amount of time. Thus, we believe that using an adaptive online coordinated management of related hardware knobs is a more robust

[1]Sections 4 and 5 describe the experimental setup in detail.

© ACM, 2017. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in: Ortega, C., Moreto, M., Casas, M., Bertran, R., Buyuktosunoglu, A., Eichenberger, A., Bose, P. libPRISM: an intelligent adaptation of prefetch and SMT levels. A: International Conference on Supercomputing. "ICS'17: proceedings of the International Conference on Supercomputing". Chicago, Illinois: Association for Computing Machinery (ACM), 2017, p. 28:1-28:10. DOI 10.1145/3079079.3079101

and less costly approach to performance tuning than exhaustive offline profiling.

In this paper, we propose libPRISM[2], an interposition library for shared memory parallel programming models that transparently adapts the different hardware knobs available in the architecture. During execution time, libPRISM discovers the best hardware configuration for different fine-grained regions of the application without user intervention and without modifying the original source code of the application.

Overall, the main contributions of this paper are:

- We present a detailed power/performance characterization of a wide set of parallel benchmark suites (NPB [26], SPEC OMP [33], CORAL [9]) on an IBM POWER8 platform. The results show that best performing SMT level and data prefetcher configuration differ between applications and between application phases, leading to differences in performance and power up to 113% and 12% respectively.
- We introduce libPRISM, a library to dynamically manage hardware resources in a transparent way to the user for OpenMP parallel applications without the need to recompile applications or runtimes. libPRISM can be used in different runtimes, with different hardware knobs and it can be easily extended.
- We describe an implementation of an adaptive policy to manage SMT and prefetcher hardware knobs in a coordinated fashion using libPRISM infrastructure. We demonstrate speedups of up to 220% in execution time (15.4% on average) and up to 13% reduction in power consumption (2.0% on average), without any significant slowdown across the suites when compared to the static default knob configuration.

This paper is organized as follows: Section 2 provides the required background for this work, while Section 3 introduces libPRISM and our adaptive policy. Section 4 describes the experimental setup and Section 5 shows the evaluation of our framework. Next, Section 6 discusses the related work and, finally Section 7 presents the conclusions of this paper.

2 BACKGROUND

This section provides the required background about the SMT and data prefetch knobs targeted in this work. The runtime systems for shared memory programming models that we leverage to manage these knobs are also described.

2.1 Simultaneous Multithreading

SMT increases the number of executing threads within the same core, which can be very useful to hide memory latency and exploit more instruction level parallelism (ILP). In a processor with SMT capabilities, the processor fetches instructions from different threads and puts them on a shared instruction queue. Then, in the execution stage, all threads share the hardware resources of the core where they run, effectively increasing the overall resource utilization and the system throughput. However, individual thread performance may degrade due the contention on the shared hardware resources.

Multi-programmed workloads can significantly benefit from SMT capabilities, since the different threads stress different functional units or have different memory patterns. Therefore, the usage of the hardware resources is higher [15, 18, 32, 36]. In contrast, parallel applications that follow a traditional fork-join parallelization scheme, execute the very same code on the different threads. Consequently, all threads are competing for the same hardware resources, leading to a higher contention on shared hardware resources, which sometimes degrades overall system performance. Consequently, a higher SMT level can even degrade overall performance [10, 21].

2.2 Hardware Data Prefetching

Hardware data prefetching reduces memory latency by bringing data to the processor's cache before it is needed. This reduces stalls due to memory accesses. Almost all current processors include a hardware data prefetch engine as it is a powerful technique to reduce memory latency, which is one of the main bottlenecks for performance.

Applications with predictable (e.g. regular) memory access patterns and spatial locality significantly benefit from data prefetching. Other workloads with unpredictable (e.g. random) memory patterns do not benefit at all from the prefetcher, and it can even degrade performance. Useless prefetches waste memory bandwidth (increase in power consumption) and pollute the cache hierarchy (decrease in performance).

The data prefetching algorithm is usually hardcoded in the processor design and it is not possible to modify it. Vendors often add instructions to let the programmer or the compiler do software prefetching; this adds a step in the optimization process of a code. Some processors allow the user to configure the data prefetcher to match the workload characteristics by selecting the number of lines to bring ahead of time, prefetch data on load and/or store instructions, etc. A correctly configured data prefetcher can speed up the execution time, save memory bandwidth and reduce power consumption [24, 25].

In this work, we propose an automatic management of the data prefetcher transparent to the user while coordinating it with the SMT knob. This needs to be done in a coordinated fashion because the number of threads impacts the data prefetcher and data prefetcher configuration can determine the optimal number of threads to be used. This will be seen in detail in Section 5.

2.3 Runtime Systems and Shared Memory Programming Models

With the increasing number of cores, orchestrating the parallel execution of an application is becoming more difficult. The usage of a runtime system to manage this complexity is a common practice to exploit the parallelism of multi-core systems. Runtimes are used as an abstract layer in the software stack to parallelize codes. 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 or directive to spawn all the desired threads, share the data among them, or synchronize them. This method reduces the burden of developing parallel applications and drives the design of future architectures [4, 16, 30, 38].

Specifically, Open Multi-Processing (OpenMP) [34] is a programming model for shared memory systems. It has become the *de-facto* programming model for such systems and it is supported by all the major vendors. OpenMP is based on directives annotated by the developer to a sequential source code. Then, these directives are translated to parallel code at compile time. Directives delimit a part of the source code that is executed in parallel. We refer to this code executed in parallel as parallel region. Depending on the specific runtime implementation, at the beginning of a parallel region, the runtime creates or activates the requested number of threads and executes the parallel code. At the end of the parallel region, the runtime destroys the created threads or deactivates them.

[2] libPRISM source code is available at: https://github.com/criort/libPRISM

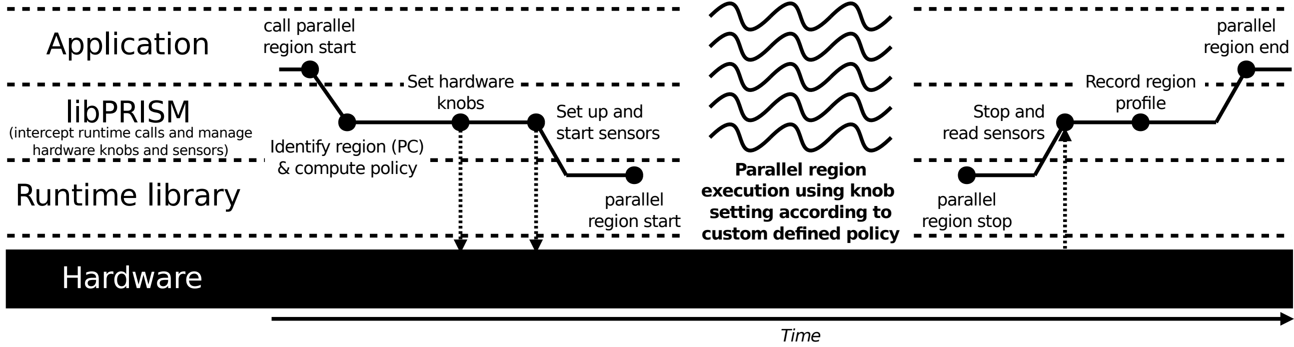

Figure 2: libPRISM execution stack and work flow.

We take advantage of these runtimes in order to automatically manage hardware knobs for several reasons: applications have different phases already annotated, which provides intra-application granularity. This can be exploited to adapt the hardware per phase instead of per application. Those phases usually behave regularly over time and we can learn from their previous executions. Finally, it is possible to use library interposition with the runtime to capture and reconfigure the hardware at the beginning and at the end of each phase of the application.

In the next section, we introduce libPRISM, which leverages these properties to adapt the hardware knobs in runtime systems for programming models based on shared memory systems.

3 LIBPRISM

libPRISM is an interposition library that reconfigures the available hardware knobs at execution time. Its decisions are based on the custom defined policies that are implemented. Policies can leverage the information from the different sensors available, such as performance counters, temperature, power, etc., to drive the different knobs present in the system. Each time the application enters a parallel region, libPRISM asks to the policy the required knob configurations and the sensors to be tracked during the execution of the parallel region. Then, libPRISM sets the different hardware sensors and knobs accordingly. If there are multiple parallel regions, they will be executed with their respective knob configuration according to the implemented policy.

To achieve this goal without recompiling the application or the runtime system, libPRISM is located on top of the runtime system, as shown in Figure 2. When a parallel region starts, the application calls our library instead of the runtime system. Then, libPRISM takes care of communicating configuration changes to the runtime system and to the underlying hardware. The software stack shown in Figure 2 allows libPRISM to: (1) communicate changes to the runtime system; (2) gather data from the runtime and the hardware; and (3) avoid the need to recompile the application or the runtime itself. In this scenario, the application executes as usual without being aware that libPRISM is dynamically adapting the hardware resources based on a custom defined policy.

libPRISM uses a library interposition mechanism to intercept calls from the application to the runtime. Figure 2 gives an overview of the work flow of libPRISM. When the application calls the runtime to start or finish a parallel region, libPRISM intercepts these calls and executes the policy specific code before calling the runtime system. libPRISM records information about the parallel region that is going to be executed and reconfigures the knobs based on the implemented policy. Then, libPRISM calls the runtime system with the selected parameters as if it was the application. As a result, the application executes with the selected best found performing knob configuration without requiring any modification.

Our goal is to implement a policy using libPRISM infrastructure to tune SMT and hardware prefetcher knobs in order to exploit the optimization opportunities to maximize performance and, if possible, reduce the power consumption. libPRISM tracks and profiles at execution time every parallel region of the application. At compile time, parallel regions are transformed into functions that are called by the application. Parallel regions can be identified by their next program counter (PC) in the program stack of the intercepted runtime function calls. libPRISM identifies a parallel region using this PC, as shown in Figure 2. libPRISM passes that information to the policy, which keeps track of the number of times a parallel region is executed. For every parallel region that is executed, the policy records a performance profile under different knob configurations. The policy builds this performance profile for each parallel region using different performance counters (executed instructions and cycles) and the execution time of the region.

Note that, in several programming models, there exists the possibility to use a *master* thread that creates work for the other *worker* threads. This is the case when using the task abstraction available in OpenMP. This behavior is usually not exposed to the user, and it is handled internally in the runtime system. To support this type of parallelism in libPRISM, we use the master thread to measure system performance after it creates all the tasks to be executed by the worker threads without requiring any modification in the runtime.

3.1 Adaptive Algorithm

The MAXPERF policy explores different knob configurations in order to identify the best configuration per parallel region at execution time. The policy manages two hardware knobs that are targeted in this work: the SMT level and the data prefetcher, but it can work for *N* hardware knobs. It is optimized to handle parallel applications that use common runtimes such as OpenMP.

The policy implements a greedy search through the different hardware configurations in order to identify the best found performing configuration. The use of a greedy algorithm instead of an exhaustive one helps to reduce the overhead cost of exploring all the possible configurations of the hardware knobs.

MAXPERF policy adopts a hierarchical search algorithm. It explores different configurations for a particular hardware knob at a time. MAXPERF tunes first the hardware resources that have more impact on the final performance of the application. We base our heuristic on a single factor search over a multi-factor search to reduce the exploration space, therefore, reducing the overhead cost

```
1 // Call to parallel_region_begin intercepted
2 function parallel_region_begin_wrapper {
3   if execution_time[PC] > threshold:
4       executions[PC] + 1
5       if executions[PC] == repetitions:
6         previousPerformance = currentPerformance
7         currentPerformance =  avgPerformance()
8
9         module_HW_knob()
10
11  // Return control to runtime
12  parallel_region_begin_real()
13 }
14
15 /* Hardware knob module */
16 function module_HW_knob() {
17   if previousPerformance > currentPerformance:
18     best_configuration = current_configuration
19     bestPerformance = currentPerformance
20     next_knob()
21     if performance_current_knob >>
           performance_previous_knobs
22       reset_previous_knobs()
23   else:
24     lower_aggressivity_current_knob()
25   set_current_knob_configuration()
26 }
```

Listing 1: MAXPERF policy exploration phase algorithm.

of exploring. Our heuristic allows converging faster to a hardware knob configuration while taking into account inter-knobs effects.

In this work, our heuristic achieves a competitive performing hardware knob configuration with respect to the best static hardware knob configuration found for each application when tuning the SMT level and the prefetcher aggressiveness knobs.

For instance, we have measured that the best performing SMT level can lead to a performance boost larger than 10% (with respect to the default SMT level), while the best performing data prefetcher setting boosts performance around 5% (with respect to the default data prefetcher). As a result, the MAXPERF policy first explores the different SMT configurations from SMT8 to single thread (ST) to find a competitive performing SMT setting. Then it explores the different prefetcher configurations from the most aggressive to the least aggressive one. Starting with the hardware knobs configured to the most aggressive configuration allows the policy to maximize performance, reducing the possibility of degrading performance.

The policy implements an exploration phase followed by a steady state phase. In the steady phase, it is possible to do a correction phase if needed. This is a good approach in order to minimize overhead by leveraging repetitive behavior of the parallel regions of the applications and to correct hardware knobs configuration in case the behavior changes over time.

The pseudo-code of the exploration phase is shown in Listing 1. A parallel region is identified by the PC of the intercepted function call. If the duration of the parallel region is too short (i.e. below a threshold), libPRISM stops the exploration phase as the cost of reconfiguring the available hardware knobs would neglect the potential performance benefits of an optimized hardware configuration (Line 3 in the Listing 1). This threshold has to take into account the time spent in changing the specific hardware knobs.

In the exploration phase, the first time a parallel region is executed, libPRISM sets the available hardware knobs to the most aggressive configuration and measures its performance. This is done to spend the minimum amount of time in a knob configuration that degrades performance, which is usually the least aggressive knob configuration. This measurement is repeated a number of *repetitions*

```
1 performance = readPerformance()
2
3 if performance != bestPerformance:
4   increase_repetitions()
5   reset_exploration()
6 else:
7   set_best_HW_knob_configuration()
8   execute_parallel_region()
```

Listing 2: MAXPERF policy steady-state phase algorithm.

in order to avoid measurement noise due to new knob configuration. For instance, the first parallel region execution after changing the SMT level might suffer from increased number of cache misses (cold cache effects).

Next time the same parallel region is executed, libPRISM lowers the aggressiveness level of the knob and measures performance again. If lowering the aggressiveness of the knob leads to a slowdown in performance, the exploration phase for this knob stops and the previous configuration is selected as the best found performing configuration (Lines from 17 to 19 in Listing 1). Then, the policy continues the exploration phase with the next knob to configure (Lines 20 in Listing 1).

The maximum number of iterations for the exploration phase without taking into account re-explorations is: $2 \times$ *Number of SMT levels + Data prefetch aggressiveness configurations.* In our experiments, we observe that the maximum number of iterations is never reached. Our observations prove that less than 10 iterations (6.1 iterations on average) are enough to tune non-variable parallel regions with our algorithm. This is typically a low number of iterations with respect to the total number of iterations.

After the exploration phase, the policy identifies a competitive performing knob configuration for a particular parallel region and reaches a steady-state phase. The pseudo-code of this phase is shown in Listing 2. Every time the parallel region is executed, the knobs are set to the identified best found performing knob configuration. In order to identify phase changes in the application, the execution time of the parallel region is compared against the average execution time found during the exploration phase. If the last execution time significantly differs from the average execution time, the exploration phase starts again but with increased number of *repetitions* in order to minimize continuous reconfiguration overheads and take into account different control flow paths in the execution of the parallel region (Line 4 in Listing 2). In our experiments, we select a threshold of 5.0% to start again the exploration phase.

3.2 Case Study: SMT level and Data Prefetcher

To illustrate the detailed behavior of the MAXPERF policy, we describe a case study in which we use libPRISM and the MAXPERF policy to select the best SMT level and hardware data prefetcher for the CG application from the NPB suite. In the exploration phase, libPRISM explores SMT8, SMT4, SMT2 and ST for the SMT level; for the data prefetcher it explores aggressive, medium, default aggressive and disabled prefetcher configurations. The policy starts the exploration with the most aggressive configuration, SMT8 level and an aggressive data prefetcher.

Figure 3 shows how the exploration phase is performed on the longest parallel region of CG benchmark. This figure shows the selected SMT level and prefetcher configuration in a particular iteration of the parallel region, as well as the execution time of the parallel region under this configuration. In the first three iterations, the policy lowers the SMT level from SMT8 to SMT2 until there is a

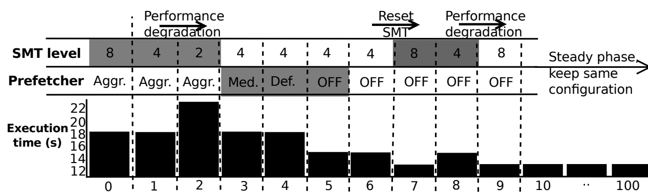

Figure 3: Adaptive algorithm in libPRISM to select a competitive performing configuration for SMT level and data prefetcher for the CG application. Details on the hardware knob configuration are explained in Section 4. Repetitions is set to 1 (algorithm shown in Listing 1).

slowdown in execution time. Therefore SMT4 level is chosen as the best SMT level. Then, in the next four iterations, the policy lowers the prefetcher aggressiveness to the point where it totally disables the prefetcher.

When an important change in performance during the prefetcher tuning happens, the MAXPERF policy starts again to re-explore the SMT level. Since disabling the prefetcher provides a 20% performance improvement, the policy triggers again the exploration phase for the SMT level with the prefetcher disabled. This is shown in Lines 21 and 22 from Listing 1, which correspond to the correction phase. In Figure 3, this is shown in iterations 5 and 6. At the end of iteration 5, the policy knows which prefetcher configuration is competitive in terms of performance for the SMT4 level. In iteration 6, after setting the hardware knobs, the policy realizes it needs to restart the exploration for the SMT knob, which takes place in the iteration 7. In this exploration phase, the policy just lowers the SMT level to 4, which offers worse performance than SMT8 level. As a result, this parallel region will be reconfigured every time to SMT8 level and disabled prefetcher configuration, which leads to a 38.3% performance improvement with respect to the default configuration. The policy does not detect any phase change during the rest of the execution in CG for this particular parallel region.

4 EXPERIMENTAL SETUP

We evaluate the solution on a POWER8 based system (8247.42L model) [31]. The system has an IBM POWER8 processor that runs at 3.15GHz with 512GB of DDR3 CDIMM memory running at 1.6GHz. The POWER8 processor in this system is packaged as a dual-chip module where each chip has 6 cores. Each core has 64KB L1 data and 32KB L1 instruction caches, a 512KB L2 cache and a 8MB L3 cache. The system runs Ubuntu 14.10 operating system with the kernel version 3.16. We compile all the benchmarks with GCC version 4.9.3, which fully supports OpenMP 4.0.

4.1 Simultaneous Multithreading

The POWER8 processor has a maximum SMT level of 8: each core can run simultaneously up to eight threads. It also supports running 1, 2 and 4 threads (ST, SMT2 and SMT4 levels). The operating system (OS) sees a physical core as a group of 8 virtual cores. When the machine boots, it automatically sets the SMT level to 8. If no application is running, the SMT level is adjusted automatically by the hypervisor based on the utilization of the system. For example, when the system is in SMT8 level, the OS exposes 8 virtual cores per each physical core. When just one of those virtual cores is used, the system sets the SMT level to ST level automatically, making all the core hardware resources available to the application.

To set the correct SMT level, we need to specify the number of threads running in a physical core. This can be done manually by setting the desired number of threads of the application and pinning threads to physical cores accordingly. Also, it can be done by disabling the virtual cores through specific *online* registers exposed by the OS. In OpenMP, the required number of threads can be defined through an environment variable or directly from the application code with specific calls to the runtime. By default, the parallel applications evaluated use all the threads available in SMT8 level.

4.2 Data Prefetcher

The data prefetcher can be controlled at the core level by a special purpose register called Data Streams Control Register (DSCR) [20], which is exposed by the OS. The DSCR has 12 different fields, offering a total of 2^{25} possible configurations. The most relevant fields are the following ones:

- LDS: Enables data prefetching for load instructions.
- SNSE: Enables data prefetching for load and store instructions that have a stride bigger than a cache block.
- URG: Number of cache blocks that will be prefetched, from 1 cache block up to 7 cache blocks.

When the machine boots, it automatically sets the prefetcher to the default configuration: LDS activated, URG set to 4, and all the other options disabled. libPRISM considers this default configuration, as well as three more prefetcher configurations. When disabling the data prefetcher, we disable all its available options. The medium configuration has URG set 7, LDS activated and all the other options disabled. The aggressive configuration has URG set to 4, LDS and SNSE activated, and all the other options disabled.

4.3 Benchmarks

To evaluate the effectiveness of the policy implemented in libPRISM, we use a wide set of benchmarks from three different suites: NPB [26] with the class D inputs, SPEC OMP 2012 [33] with the reference input, and a subset of the CORAL [9] benchmarks with the recommended input size.

The NPB suite is composed of 5 kernels and 3 pseudo-applications, which are derived from computational fluid dynamics (CFD). The SPEC OMP 2012 suite contains 14 applications from CFD to image modeling. They are focused on compute intensive performance. All SPEC OMP benchmarks are evaluated except `imagick` and `smithwa`, as these two benchmarks did not pass SPEC's validation tools in our environment. The CORAL suite tests different parts of the systems, from CPU to network performance. It includes applications from scalable science benchmarks, data-centric benchmarks or kernels. We selected four of the most relevant benchmarks in the suite: `Lulesh`, `HACC`, `graph500` and `AMG`. All the benchmarks are parallelized with OpenMP and written in C, C++ or Fortran.

Benchmarks are executed on 6 cores and pinned to them to avoid thread migration. We pin the different threads with the environment variable *OMP_PLACES*. Benchmarks can run with 6, 12, 24 and 48 threads for ST, SMT2, SMT4 and SMT8 levels, respectively, and they are executed in isolation until completion.

4.4 Metrics

In Section 5, we report speed up in execution time, power consumption and energy-delay product (EDP) for all the benchmarks. We measure wall time for the entire application. When running with libPRISM infrastructure, it also reads the timebase register from the POWER8 processor for fine-grained analysis of parallel regions. To

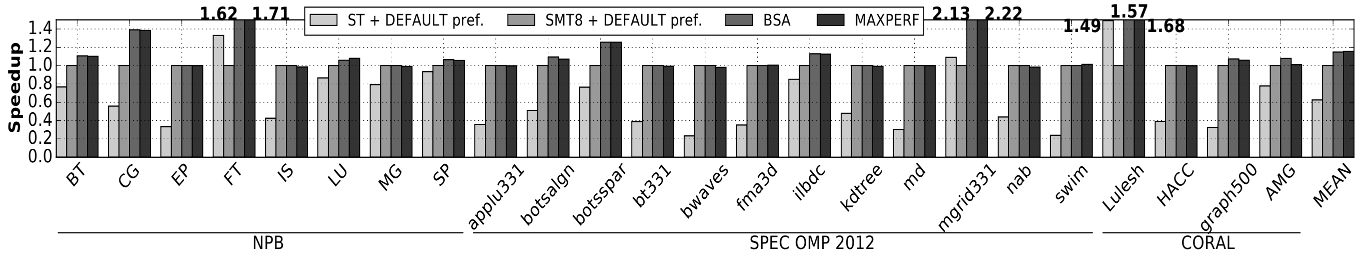

Figure 4: Performance results with respect to default configuration (SMT8 and default prefetcher). Best Static per Application (BSA): best SMT level and prefetch aggressiveness configuration for all the execution; found after an offline profiling. libPRISM is running with the MAXPERF policy that selects the hardware knob configuration per parallel region at execution time.

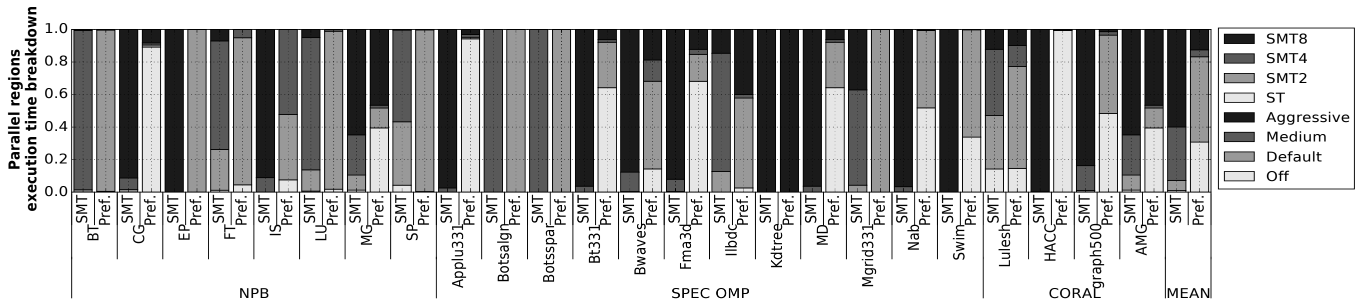

Figure 5: Percentage of time spent on each knob configuration when running with libPRISM and MAXPERF policy.

analyze the execution of the different benchmarks, multiple performance counters are collected using `perf` [11].

We use AMESTER (Automated Measurement of Systems for Energy and Temperature Reporting) [19] to measure the power consumption of the processor and memory chips. The tool remotely collects power, thermal and performance metrics from the system using the Flexible Service Processor (FSP). The FSP allows reading of different sensors from the system without using any of the processing cycles of the system. Therefore, it has no impact on the performance of the running benchmarks. In Section 5, we report the average power consumption for the total execution and energy-delay product (EDP). Power consumption results do not include the idle power of the system to put more emphasis on active power consumption savings. When reporting EDP, we report energy (taking idle power of the system into account) multiplied by execution time.

5 EVALUATION

In this section we evaluate the execution time, power consumption and EDP of libPRISM and the MAXPERF policy.

5.1 Performance

We compare the policy against different static predefined configurations. Figure 4 shows the execution time speedups for the following configurations:

- ST + DEFAULT prefetcher: Single thread (ST) and default data prefetcher.
- SMT8 + DEFAULT prefetcher: Default configuration when the machine boots (SMT8 level and default data prefetcher), used as the baseline to normalize speedups.
- Best static per application (BSA): Best hardware configuration found for each application after an exhaustive offline profiling.
- MAXPERF: Dynamically sets the hardware knobs configuration for every parallel region in the application based on the MAXPERF policy, which seeks the maximum performance in terms of execution time. This policy uses the libPRISM infrastructure.

Figure 4 shows that the default hardware configuration is already the best performing configuration for 10 out of 24 evaluated benchmarks. For the remaining 14 benchmarks, half of them can reach performance improvements above 10%, illustrating the need for an adaptive system that manages shared hardware resources. On average, BSA reaches a 14.9% performance improvement over the default configuration. The policy MAXPERF slightly increases this performance improvement (15.4%) and achieves competitive results for all benchmarks without any significant slowdown across the benchmarks and without requiring any offline profiling.

Figure 5 shows the breakdown of the selected hardware configurations during the execution with libPRISM and the MAXPERF policy. A first observation from this figure is that only 4 benchmarks run 90% of the time with the default configuration. A second observation is that 8 benchmarks have parallel regions with different hardware requirements. libPRISM and the MAXPERF policy effectively detect this situation and select the appropriate hardware configuration per parallel region.

In the case of the NPB suite, there are several benchmarks that achieve the best performance with the default hardware configuration: `EP`, `IS` and `MG`. Even in those cases, libPRISM does not show any significant slowdown. Then, we have different behaviors among the others. `BT` executes better with SMT4 level and needs to have the prefetcher turned on. `CG` gets the maximum speedup with SMT8 level and the prefetcher disabled. If the prefetcher is enabled, `CG` achieves better performance with the SMT4 level, as explained in Section 3.2. In `FT`, the MAXPERF policy gets a speedup of 1.71x by selecting SMT2 and SMT4 levels in different parallel regions, and keeping the prefetcher in the default configuration, as shown in Figure 5. `LU` and `SP` get a 8.0% and 5.5% speedup respectively

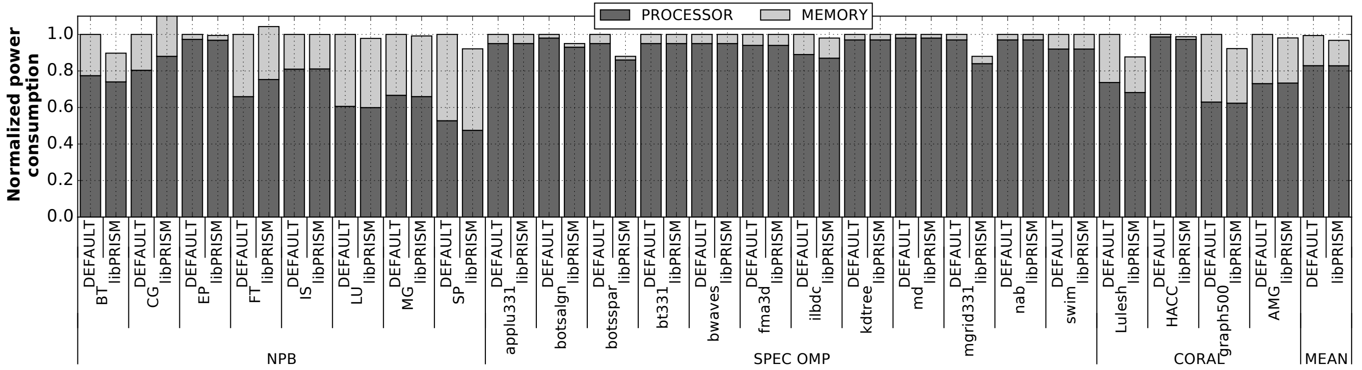

Figure 6: Power consumption when applications are executed with the default hardware knob configuration and when using libPRISM and the MAXPERF policy. Values are normalized to the default hardware knob configuration.

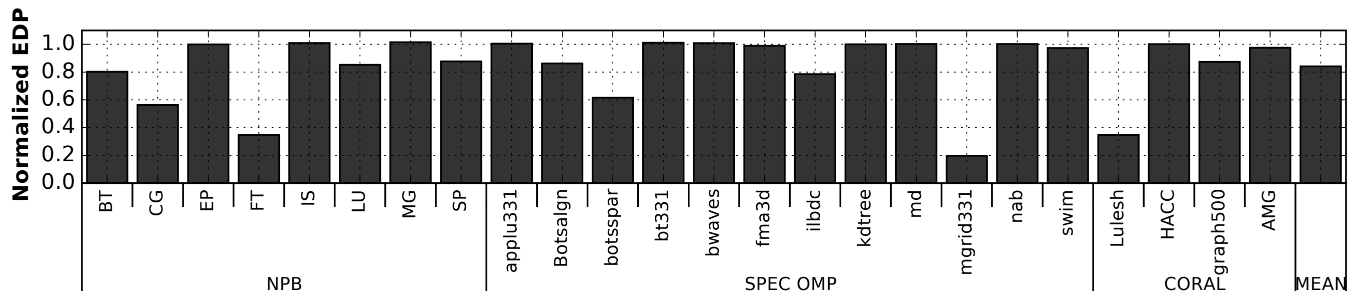

Figure 7: Energy-Delay Product (EDP) when applications are executed with libPRISM with the MAXPERF policy. Values are normalized to the EDP when executed with the default hardware knob configuration. EDP is computed as energy multiplied by execution time.

lowering only the SMT level. `LU` uses SMT4 the most, and `SP` needs to use SMT4 56.3% of the time and SMT2 the remaining of the time. However, there is no performance difference between BSA and the MAXPERF policy as the performance in SMT4 and SMT2 levels is very similar. MAXPERF always chooses the least aggressive configuration possible if multiple configurations have similar performance. This reduces the power consumption as Section 5.2 shows.

From the evaluated benchmarks in the SPEC OMP 2012 suite, 8 of them run faster with the default configuration. In the case of `bostalgn` and `botsspar`, they get a 7.1% and 25.6% performance improvement, respectively, by lowering the SMT level to SMT4. `Ilbdc` also needs a SMT4 level and the prefetcher enabled to get a 12.6% improvement. Finally, `mgrid331` achieves up to a 2.2x speedup. Some parallel regions of this benchmark suffer significant performance slowdowns in the default configuration. MAXPERF combines SMT4 and SMT8 levels for different parallel regions to achieve its maximum speedup, as shown in Figure 5. In some benchmarks such as `applu331`, `bt331`, `fma3d` or `md`, MAXPERF policy decides to turn off the data prefetcher. However, no performance improvement in performance is achieved as the memory component for those benchmarks is very low.

In the case of the CORAL benchmarks, results are also workload dependent. `Lulesh` achieves a 1.7x speedup with the MAXPERF policy, with mostly a combination of SMT2 and SMT4 levels (32.9% and 40.7% of the time is spent in these configurations), while the prefetcher needs to be enabled but its aggressiveness does not matter. In `HACC`, a SMT8 level is enough to get the best performance, but MAXPERF disables the data prefetcher without impacting performance. In the case of `graph500`, the policy disables the data prefetcher in some parallel regions to boost performance by 5.9%. In `AMG`, the best configuration is SMT4 level and the most aggressive prefetcher. However, MAXPERF does not achieve the performance of BSA because this benchmark is composed by many small parallel regions (less than 1ms) where MAXPERF decides to maintain the default configuration to avoid reconfiguration overheads.

5.2 Energy Efficiency

Next, we discuss the energy efficiency results obtained with libPRISM using the MAXPERF policy. Figure 6 shows the power consumption of the processor and the memory components when running in the default configuration and when using libPRISM with the MAXPERF policy. Power results are normalized to the default configuration. The processor power represents 82.9% of the power in both configurations. In the MAXPERF policy, the memory power is reduced from 16.5% to 13.9%. However, different benchmark suites show very different power profiles: NPB and CORAL benchmarks have a high memory power consumption, while SPEC OMP 2012 spends 95.1% of the power on the processor.

To enhance the analysis of the energy efficiency of libPRISM and the MAXPERF policy, Figure 7 shows the energy-delay product (EDP) when using libPRISM and MAXPERF normalized to the default configuration. This figure considers the entire system power consumption, including the idle power consumption. The combination of better performance results (as shown in Figure 4) and reduced power consumption (as shown in Figure 6) explain the significant

reduction in EDP (15.9% on average), being above 19.8% for seven of the evaluated benchmarks.

In the case of the NPB suite, the memory power ranges from 15.7% to 47.3% of the total power consumption. In the case of EP, IS, and MG, MAXPERF does not reduce the power consumption and the EDP. In the case of BT, LU and SP, setting a less aggressive configuration implies a reduction in the power consumption. BT reduces processor and memory power consumption by 3.4% and 6.9%, respectively, and EDP is reduced by 19.8%. LU reduces memory power consumption by 1.5% and EDP by 14.8%. SP reduces processor and memory power consumption by 5.3% and 2.6%, respectively, and EDP by 12.3%. Finally, in the case of CG and FT, MAXPERF slightly increases the power consumption as these benchmarks exhibit a much higher performance with libPRISM and the MAXPERF policy. CG increases processor and memory power consumption by 7.6% and 8.6%, respectively, but EDP is reduced by 43.8% thanks to the reduced execution time. FT increases processor power consumption by 9.4% and reduces memory by 5.1%, while EDP is reduced by 65.4%.

The SPEC OMP 2012 suite is mostly CPU-intensive, as can be seen by the power distribution in Figure 6. Although MAXPERF disables the data prefetcher in multiple cases (see Figure 5), the overall power consumption is not significantly reduced. MAXPERF is able to speedup execution and lower the power consumption and the EDP of different benchmarks. Botsalgn and botsspar reduce the SMT level to SMT4, which reduces processor power consumption by 5.6% and 9.0%, respectively, and EDP by 13.8% and 38.4%, respectively. A similar situation happens in mgrid331, with a 13.1% reduction in processor power and 80.2% in EDP. SPEC OMP 2012 suite shows that when no performance optimization opportunities exist with respect to the default knob configuration, libPRISM with the MAXPERF policy does not introduce noticeable overheads.

Finally, in the case of the CORAL benchmarks, we see different behaviors on the power consumption. Power consumption of the processor ranges from 73.1% to 98.6% and the memory component goes from 1.4% to 26.9%. libPRISM and the MAXPERF policy improve the energy efficiency and the overall performance. In Lulesh, libPRISM with MAXPERF policy reduces processor and memory power consumption by 5.5% and 6.8%, respectively. EDP is reduced by 65.4%. HACC runs faster with the default configuration, and even if MAXPERF disables the data prefetcher, there is no performance and power difference as this benchmark has very low memory utilization. graph500 shows a reduction of memory power consumption of 7.1% after disabling the prefetcher, while EDP is reduced by 12.7%. AMG runs better with a lower SMT level, but as described in the previous section, short parallel regions prevent libPRISM from reconfiguring the hardware and no differences in execution time are seen. Nevertheless, power consumption and EDP are slightly reduced.

5.3 Individual Performance Analysis

In this section, we provide a detailed performance analysis of three interesting benchmarks: CG, FT and Lulesh. For this purpose we read the required performance monitoring counters (PMC) to obtain the CPI breakdown [20]. We focus on these PMCs: *GRP_CMPL*, completed instructions; *GCT_NOSLOT_CYC*, cycles when there are no instructions from threads; *CMPLU_STALL_VSU*, cycles stalled by the vector-and-scalar unit; *CMPLU_STALL_DMISS_L2L3*, completion stall by a data cache miss which is resolved in L2 or L3 caches; *CMPLU_STALL_DMISS_L3MISS*, completion stall due to a cache miss the L3; *CMPLU_STALL_THRD*, a thread could not complete an instruction because the completion port was being used by another thread; *CMPLU_STALL_DCACHE_MISS*, cycles stalled by data cache misses in the L1 cache; These PMCs are the most significant ones for the

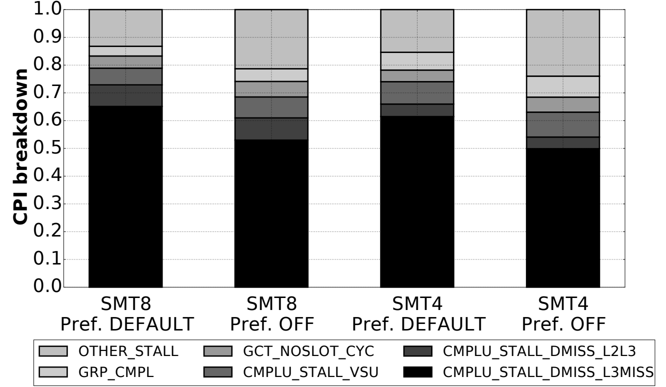

Figure 8: CPI breakdown of CG benchmark. Turning off the prefetcher reduces the L3 misses, which are the main contributors to the CPI breakdown.

```
 1 !$omp parallel default(shared) private(j,k,cgit,suml
       ,alpha,beta)
 2 ...
 3 !$omp do
 4   do j=1,lastrow-firstrow+1
 5     suml = 0.d0
 6     do k=rowstr(j),rowstr(j+1)-1
 7       suml = suml + a(k)*p(colidx(k))
 8     enddo
 9     q(j) = suml
10   enddo
11 !$omp end do
12 ...
```

Listing 3: Relevant parallel region of CG.

applications shown. The rest of PMCs from the CPI breakdown are represented as *OTHER_STALL*.

CG has several parallel regions, but one of them covers more than 96% of the total execution time. Figure 8 shows the CPI breakdown for this parallel region when using SMT8 or SMT4 and the prefetcher set to default or disabled. libPRISM chooses SMT8 level and disables the prefetcher, as shown in Figure 5. In the default configuration, the main reason for stalled cycles is data cache misses as reflected by *STALL_DMISS_L2L3* and *STALL_DMISS_L3MISS* CPI breakdown components in Figure 8. When changing from SMT8 to SMT4 with the prefetcher enabled (third bar in Figure 8), we see these two PMCs decrease by 3.2% and 3.6%. When the prefetcher is turned off, there is a reduction of 12.1% in SMT8 level and of 11.8% in SMT4 level. Overall, the main performance bottleneck in CG is the large number of cache misses, which are reduced when disabling the prefetcher or lowering the SMT level.

Using libPRISM we can relate the executed parallel region with the source code, which is shown in Listing 3. This code iterates through a vector and accumulates its values with a non-regular access pattern (p(colidx(k))). For this type of access patterns, not only the data prefetcher is unable to bring useful data to the cache but it also degrades the performance of the benchmark by polluting the cache and reduced effective memory bandwidth.

In the case of FT, MAXPERF selects the default prefetcher and reduces the SMT level to SMT4 and SMT2. Figure 9 shows the CPI breakdown and the instructions per cycle (IPC) for a parallel region of the FT benchmark when executed in different SMT levels with the default prefetcher configuration. The IPC is maximized when we execute the parallel region in SMT2 level. Stalls in SMT8 and SMT4 levels are mainly due to cache misses, which are reduced by

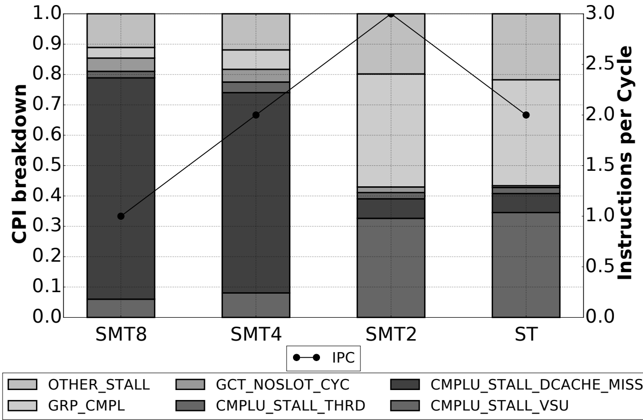

Figure 9: CPI breakdown of the FT benchmark for different SMT configurations with the default prefetcher setting. Reducing the SMT level reduces the number of cache misses and increases performance until other CPI components become the main performance bottleneck (e.g. CMPLU_STALL_VSU and GRP_CMPL in ST configuration).

lowering the SMT level. Once running in SMT2, the prefetcher can hide the memory latency of cache misses and the main bottleneck becomes the number of available execution units: a large percentage of stalled cycles in the vector-scalar unit (CMPLU_STALL_VSU CPI component). Lowering the SMT level to ST exacerbates this problem (less ILP) and reduces the total throughput.

Finally, we analyze the `Lulesh` benchmark. This benchmark has 17 short regions, which are not reconfigured by libPRISM (their duration is below the specified threshold) and 7 long parallel regions, which are executed 2500 times and have an execution time between 2 and 4 milliseconds. As shown in Figure 5, MAXPERF selects all possible SMT levels for these parallel regions, achieving speedups from 1.03x to 2.42x. MAXPERF also selects all possible prefetcher configurations for different parallel regions, with speedups ranging from 1.01x to 1.04x. In `Lulesh`, the parallel regions access nodes from a list in a non-regular pattern. As a result, running with less threads does a better usage of the memory bandwidth and the last level cache. In 2 of the relevant parallel regions, SMT4 level is the best SMT level because threads are not only loading from memory but also doing intensive CPU operations with the loaded data. In contrast, SMT2 level provides better performance for those parallel regions where threads are memory intensive and perform few operation with the loaded data.

5.4 Discussion

In this section we discuss potential applicability of libPRISM together with its limitations.

In order to define a good policy, basic knowledge of the architecture is needed and some experimental process is required to identify the order in which different hardware knobs are explored. After this basic profiling, the exploration does not hurt overall performance. This can be done by running a small training set of benchmarks.

Although we only demonstrated the usage of libPRISM for coordinating the management of SMT and prefetcher knobs for OpenMP applications on a POWER8-based system, the infrastructure can be leveraged for other purposes. For instance, other shared memory programming models that mark parallel regions can be supported by libPRISM using the same library interposition mechanism. Also, other hardware knobs and sensors can be used by the policies implemented within libPRISM. This is enabled by the generic, modular, extensible and architecture-agnostic design of libPRISM.

The library interposition mechanism, reading the sensors and the configuration of hardware knobs can add overheads to the execution of the application. We measured this overhead by running the benchmarks with and without libPRISM infrastructure. In this experiment, libPRISM only tracks and profiles the different parallel regions without reconfiguring the hardware knobs. The measured overhead in terms of execution time is always below 2.3% (1.0% on average), mainly because of monitoring small parallel regions. After selecting an appropriate threshold to control which parallel regions are explored, the exploration overhead is effectively reduced to less than 1.0%, which makes the energy overhead negligible as well.

6 RELATED WORK

As far as we know, this is the first work to combine SMT with data prefetching knobs to see the interaction between each other and achieve a jointly-optimized configuration of these hardware knobs.

6.1 Simultaneous Multi-threading

Previous work on SMT is focused to achieve fairness [1–3, 6, 7, 37]. Other authors predict IPC when running in a SMT processor and schedule serial applications on virtual cores in order to boost the overall performance of the system [17, 18, 32, 36]. These works focus on multi-programmed workloads. This is in contrast to this work, which targets parallel workloads.

There is work on dynamically choosing the best SMT level for parallel workloads. Zhang et al. [42, 43] and Heirman et al. [21] propose a dynamic algorithm inside the OMP runtime in order to choose the best number of threads. Jia et al. [23] uses machine learning to boost performance by setting the correct SMT level. Besides not configuring multiple hardware knobs, our work differs from these previous works in 2 aspects: (1) their solutions are implemented inside the runtime, limiting the possibilities of usage of the work and (2) their search space is small compared to ours.

6.2 Data Prefetching

There are previous works that propose hardware modifications of the prefetcher implementations in order to improve performance on multicore chips [12–14, 40, 41, 44]. Our proposal benefits from already implemented data prefetchers, therefore there is no extra cost and it can be used in current existing hardware to improve performance.

In terms of software, most of the previous work has been developed for serial applications or multi-programmed workloads [22, 27, 29]. Using similar workloads, Jimenez et al. detects phases of applications at runtime and changes the data prefetch configuration according to the overall demands of the applications running on the system [24, 25]. These phases are not explicitly defined in the workloads, therefore, the algorithm constantly iterates through the different data prefetch configurations.

In this work, we use the already annotated parallel regions as phases. Phases are re-explored only when their behavior change, reducing exploration time and minimizing possible slowdowns due to low performing hardware knob configurations. Also, we take into account possible inter-effects between the SMT level and the data prefetcher knobs. In addition, in [24, 25], the operating system needs to be modified. Our solution works without any modification on the software stack.

Also, Chilimbi et al. make use of software prefetching to speedup applications at execution time [8]. Wang et al. uses information at compile time to correctly set the data prefetcher aggressiveness [40]. In contrast, in this work we focus on parallel workloads that are common in high performance computing.

Few research has been done when referring to parallel workloads. Li et al. applied machine learning to automatically reconfigure the data prefetcher for different workloads [28]. Prat et al. added intelligence to a task-based runtime to automatically manage the aggressiveness of the data prefetcher for parallel workloads [35]. These works lack the control of the number of threads working in the same task. Therefore, the possible interaction with the SMT level and the data prefetcher is also missing.

7 CONCLUSIONS

Because of the potential resource contentions among threads in the memory subsystem, current processors offer the user a wide range of configurable knobs such as the SMT level or the data prefetcher aggressiveness. Unfortunately, finding the optimal settings of these knobs is difficult because of the large search space, the strong interactions between different architectural knobs and the different hardware demands of application phases.

In this work we introduce libPRISM, an infrastructure for parallel applications to dynamically adapt the architectural knobs based on a custom policy. On top of libPRISM we develop the MAXPERF policy, which manages the SMT level and the data prefetcher aggressiveness with the goal of increasing performance.

We evaluate our solution for a wide set of OpenMP benchmarks running on a POWER8 system. Results show a boost in performance of up to 220% (15.4% on average), a dynamic power consumption reduction of up to 13% and an energy-delay product reduction of up to 80% when compared to the default static system configuration.

ACKNOWLEDGMENTS

This work has been supported by the RoMoL ERC Advanced Grant (GA 321253), by the European HiPEAC Network of Excellence, by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272) and by IBM/BSC Deep Learning Center initiative. This research was developed in part with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

M. Moreto has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship number JCI-2012-15047. M. Casas is supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (Contract 2013 BP_B 00243).

REFERENCES

[1] Boneti, C., et al. Balancing HPC applications through smart allocation of resources in MT processors. IPDPS'08.
[2] Boneti, C., et al. A Dynamic Scheduler for Balancing HPC Applications. SC'08.
[3] Boneti, C., et al. Software-Controlled Priority Characterization of POWER5 Processor. ISCA'08.
[4] Casas, M., et al. Runtime-Aware Architectures. Euro-Par'15.
[5] Cazorla, F., et al. Dynamically controlled resource allocation in SMT processors. MICRO'04.
[6] Cazorla, F., et al. Improving Memory Latency Aware Fetch Policies for SMT Processors. ISHPC'03.
[7] Cazorla, F., et al. Predictable Performance in SMT Processors: Synergy Between the OS and SMTs. TC'06.
[8] Chilimbi, T., et al. Dynamic Hot Data Stream Prefetching for General-purpose Programs. PLDI'02.
[9] CORAL Benchmarks. https://asc.llnl.gov/coral-benchmarks/.
[10] Creech, T., et al. Efficient Multiprogramming for Multicores with SCAF. MICRO'13.
[11] de Melo, A. C. The new linux perf tools. 2010.
[12] Ebrahimi, E., et al. Coordinated Control of Multiple Prefetchers in Multi-core Systems. MICRO'42.
[13] Ebrahimi, E., et al. Prefetch-aware Shared Resource Management for Multi-core Systems. ISCA'11.
[14] Ebrahimi, E., et al. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. HPCA'09.
[15] Everman, S., et al. A Memory-Level Parallelism Aware Fetch Policy for SMT Processors. HPCA'07.
[16] Fatahalian, K., et al. Sequoia: Programming the Memory Hierarchy. SC'06.
[17] Feliu, J., et al. Addressing Fairness in SMT Multicores with a Progress-Aware Scheduler. IPDPS'16.
[18] Feliu, J., et al. Symbiotic job scheduling on the IBM POWER8. HPCA'15.
[19] Floyd, M., et al. Adaptive energy-management features of the IBM POWER7 chip. 2015.
[20] Hall, B., et al. Performance Optimization and Tuning Techniques for IBM Power Systems Processors Including IBM POWER8. 2015.
[21] Heirman, W., et al. Automatic SMT Threading for OpenMP Applications on the Intel Xeon Phi Co-processor. ROSS'14.
[22] Hur, I., et al. Memory Prefetching Using Adaptive Stream Detection. MICRO'06.
[23] Jia, Z., et al. Auto-tuning Spark Big Data Workloads on POWER8: Prediction-Based Dynamic SMT Threading. PACT'16.
[24] Jimenez, V., et al. Increasing multicore system efficiency through intelligent bandwidth shifting. HPCA'15.
[25] Jiménez, V., et al. Making Data Prefetch Smarter: Adaptive Prefetching on POWER7. PACT'12.
[26] Jin, H., et al. The OpenMP implementation of NAS parallel benchmarks and its performance. 1999.
[27] Khan, M., et al. A case for resource efficient prefetching in multicores. ISPASS'14.
[28] Li, M., et al. PATer: A Hardware Prefetching Automatic Tuner on IBM POWER8 Processor. CAL'16.
[29] Luk, C., et al. Ispike: a post-link optimizer for the Intel reg; Itanium reg; architecture. CGO'04.
[30] Manivannan, M., et al. Runtime-Guided Cache Coherence Optimizations in Multi-core Architectures. IPDPS'14.
[31] Mericas, A., et al. IBM POWER8 performance features and evaluation. *IBM Journal of Research and Development* (2015).
[32] Moseley, T., et al. Methods for modeling resource contention on simultaneous multithreading processors. ICCD'05.
[33] Müller, M., et al. OpenMP in a Heterogeneous World: 8th International Workshop on OpenMP. IWOMP'12.
[34] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.5.
[35] Prat, D., et al. Adaptive and application dependent runtime guided hardware prefetcher reconfiguration on the IBM POWER7. CoRR'15.
[36] Snavely, A., et al. Symbiotic Jobs cheduling for a Simultaneous Multithreaded Processor. ASPLOS IX.
[37] Tembey, P., et al. SMT Switch: Software Mechanisms for Power Shifting. CAL'13.
[38] Valero, M., et al. Runtime-Aware Architectures: A First Approach. *International Journal on Supercomputing Frontiers and Innovations 1*, 1 (June 2014), 29–44.
[39] Vega, A., et al. Crank It Up or Dial It Down: Coordinated Multiprocessor Frequency and Folding Control. MICRO'13.
[40] Wang, Z., et al. Guided Region Prefetching: A Cooperative Hardware/Software Approach. ISCA'03.
[41] Wu, C., et al. PACMan: Prefetch-aware Cache Management for High Performance Caching. MICRO'11.
[42] Zhang, Y., et al. An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs. PDCS'04.
[43] Zhang, Y., et al. Runtime Empirical Selection of Loop Schedulers on Hyperthreaded SMPs. IPDPS'05.
[44] Zhuang, X., et al. Reducing Cache Pollution via Dynamic Data Prefetch Filtering. TC'07.