Intelligent Adaptation Of Hardware Knobs For Improving Performance and Power Consumption

Cristobal Ortega, Lluc Alvarez, Marc Casas, Ramon Bertran, Alper Buyuktosunoglu, Alexandre E. Eichenberger, Pradip Bose, Miquel Moreto

Abstract—Current microprocessors include several knobs to modify the hardware behavior in order to improve performance, power, and energy 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 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. libPRISM can minimize execution time, energy-delay product or power consumption by dynamically managing the SMT level, the data prefetcher, and the DVFS hardware knobs. Overall, the proposed solutions increase performance up to 130% (16.9% on average), reduce energy-delay product up to 80%, and reduce power consumption up to 33% depending on the target metric compared to the default knob configuration of the system.

Index Terms—HPC, Parallel Programming, Runtime, SMT, Data Prefetcher, DVFS

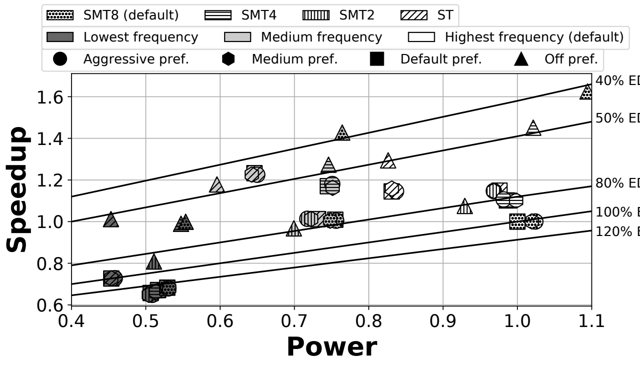


Fig. 1: CG behavior under different hardware knob configurations. The Y axis shows speedup with respect to the default hardware configuration. The X axis shows power consumption normalized to the default hardware configuration. Energy-Delay Product (EDP) normalized with respect to the default hardware configuration is represented with isolines. For power and EDP, lower is better. Default configuration is SMT8, default data prefetcher, and highest frequency.

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, dynamic voltage and frequency scaling (DVFS) levels, the thread 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 [5], [13], [39], [62], [63]. This independent actuation can lead to conflicting decisions that jeopardize system power-performance efficiency [57]. 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.

To illustrate the need for a coordinated adaptive system, Figure 1 shows the performance, the average power consumption, and the Energy-Delay Product (EDP) isolines of the CG benchmark from the NAS Parallel Benchmarks (NPB) [40] suite with different hardware knob configurations[1] with respect to the default hardware knob configuration (SMT8 level, default data prefetcher, and highest frequency). EDP is calculated as $POWER \times EXECUTION\,TIME \times EXECUTION\,TIME$. We report EDP instead of energy since degrading execution time

- C. Ortega, L. Alvarez, M. Moreto and M. Casas are with the Barcelona Supercomputing Center (BSC-CNS) and the Universitat Politecnica de Catalunya (UPC).
E-mail: name.surname@bsc.es
- R. Bertran, A. Buyuktosunoglu, A. Eichenberger and P. Bose are with the IBM T.J. Watson Research Center.
E-mail: {rbertra, alperb, alexe,pbose}@us.ibm.com

1. Section 6 describes the experimental setup in detail.

of the application has a higher penalty with this metric.

In Figure 1, configurations that are above the 100% EDP isoline have a higher efficiency due to reduced power consumption or execution time with respect to the default configuration of the system. There are multiple configurations that have a really low power consumption with respect to the default configuration. Yet, those configurations are inefficient due to a higher execution time (configurations below the 100% EDP line).

Other configurations have different tradeoffs within the same EDP lines. For instance, the best hardware knob configuration in terms of performance achieves an EDP of 41.4% with respect to the default hardware configuration due to a 1.6x speedup and the 10% increase in power consumption. While the best configuration in terms of EDP (37.5% with respect to the default hardware configuration) can achieve a 1.4x speedup while reducing 25% of the power consumption. The best speedup in performance is achieved with a configuration with the highest frequency, a SMT8 level and the prefetcher disabled. While the best EDP is achieved with a medium level of frequency by sacrificing execution time and reducing power consumption.

The previous experiment shows that different knob configurations yield a wide range of speedup and power consumption tradeoffs depending on resource demands from applications. Furthermore, applications can have different intra resources demands, increasing even more the variety of optimal hardware knob configurations. Therefore, hardware knobs must be tightly coordinated to achieve the maximum performance or the minimum EDP.

Performing an exhaustive profiling of each possible configuration (more than 350 possible configurations in our evaluated system) for each application (and each parallel region inside the application) and input data size is a time consuming process. Thus, we believe that using an adaptive online coordinated management of related hardware knobs is a more robust and practical approach to performance tuning than exhaustive offline profiling.

In this paper, we propose libPRISM[2], an infrastructure for shared memory parallel programming models that transparently configures the different hardware knobs available.At 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 the NPB suite [40] on an IBM POWER8 platform. Results show that the best hardware knob configuration depends on the end goal; the best performing configuration found with an offline profiling has a speedup of up to 2.65x with respect to the default configuration, while setting a different hardware knob configuration can reduce power consumption up to 33% with respect to the default configuration and still improve performance.
- We introduce libPRISM, an infrastructure to dynamically manage hardware resources for OpenMP parallel applications in a transparent way to the user, without the need to recompile applications or runtime systems. libPRISM can be extended to support different runtime systems, support hardware architectures, and add more hardware knobs to coordinate.
- We demonstrate speedups of up to 2.3x in execution time (1.69x on average), up to 33% reduction in power consumption (18.0% on average) and up to 80% reduction in EDP (32% on average) when using libPRISM to dynamically find the best hardware knob configuration for these metrics without any prior information of the benchmark. With respect to the static default knob configuration, libPRISM does not introduce any significant slowdown across the benchmarks.

2. libPRISM code available at: https://github.com/criort/libPRISM

This paper is organized as follows: Section 2 provides the required background for this work and Section 3 motivates this work. Section 4 introduces libPRISM and Section 5 introduces our different adaptive policies. Section 6 describes the experimental setup and Section 7 shows the evaluation of our framework. Next, Section 8 discusses the related work and Section 9 presents the conclusions of this paper.

2 Background

This section provides the required background about the SMT, data prefetch and DVFS hardware 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 running 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 different SMT levels (i.e. the number of running threads within the same core), 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, increasing the overall resource utilization and throughput. However, individual thread performance may degrade due the contention on the shared hardware resources.

Multi-programmed workloads can significantly benefit from higher SMT levels, since the different threads stress different functional units or have different memory access patterns. Therefore, the usage of the hardware resources is higher [23], [26], [48], [54]. In contrast, parallel applications that follow a traditional fork-join parallelization scheme execute the very same code on the different threads. In that scenario, all threads are competing for the same hardware resources, leading to a higher contention on shared hardware resources, which might degrade overall system performance [16], [32].

2.2 Hardware Data Prefetching

Hardware data prefetching reduces memory latency by bringing data to the processor caches before it is needed. This reduces stalls due to memory accesses. Almost all current processors include multiple hardware data prefetch engines as it is a powerful technique to improve the overall system 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 it. Also, under certain circumstances, the prefetcher can even degrade performance and energy efficiency. This is because 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; these instructions need to be used consciously by the programmer, who should invest more time in the optimization process of the code. Also, some processors allow the user to configure different parameters of the hardware 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. Overall, a correctly configured data prefetcher can speed up the execution time, save memory bandwidth and reduce power consumption significantly [38], [39].

2.3 Dynamic Voltage and Frequency Scaling

DVFS provides a mechanism to adjust voltage and frequency dynamically on commodity hardware [19], [41].

Nowadays, processors have different DVFS levels (multiple possible combinations of voltage and frequency) in which they can safely operate. Modern operating systems (OS) use the different DVFS levels based on a policy. The most used policy in modern Linux kernels is the *ondemand* policy, which periodically calculates the CPU utilization (non-idle cycles) and sets a corresponding frequency [1], [51]. A small increase on the processor load can increase the frequency to the highest available configuration, diminishing possible power gains and degrading the overall system power-performance efficiency.

DVFS benefits mainly from idle/stall periods of non-critical regions, where frequency can be lowered to save power while achieving the same performance obtained with a higher frequency. For instance, memory bound applications benefit from a lower frequency in terms of EDP, so a user could tune the DVFS hardware knob accordingly to increase energy efficiency [58].

2.4 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. Runtime systems are used as an abstract layer in the software stack to parallelize codes. Usually, they need compiler support to translate from directives to real code that will be executed: the programmer needs to use a specific directive to spawn all the 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 [9], [25], [56].

Specifically, Open Multi-Processing (OpenMP) [50] has become the *de-facto* programming model for shared memory systems. OpenMP is based on directives that are translated to parallel code at compile time. Directives delimit the parts of the source code that are executed in parallel. We refer to this code executed in parallel as parallel region.

Typical HPC applications consist of a set of phases that are iteratively executed [49], [60]. In OpenMP programs, each phase is usually composed of one or more parallel regions that present a regular behavior over time. This program structure allows us to take advantage of the runtime system of parallel programming models to automatically manage hardware knobs during the execution.

In particular, the repetitive behavior of parallel regions across iterations can be exploted to learn the best hardware knob configuration in the first iterations, and apply this one during the rest of the execution.

In addition, the parallel regions naturally delimit the different application phases, so they provide a great opportunity to manage hardware knobs at intra-application granularity by setting the configuration that better suits the characteristic of each application phase.

3 Motivation for multiple hardware knob coordination at runtime

The main goal of this work is to reconfigure the different hardware knobs at a parallel region level in order to achieve a better performance in terms of execution time, power, or energy for each application phase.

The hardware knobs need to be configured according to the resource demands of each parallel region and having into account possible interactions among other hardware knobs to avoid underperforming configurations. For example, a parallel region running with a low frequency could use a higher SMT level to hide memory latency, and the prefetcher should be set accordingly to the access data patterns in order to not waste bandwidth or pollute the cache. On the other hand, a parallel region running with a high frequency, a high SMT level and the most aggressive prefetcher could saturate bandwidth, which could translate to stalled threads and degrade the overall performance.

In addition, our goal is to manage the hardware knobs in a completely transparent manner. To do so, we use library interposition with the runtime system to capture and reconfigure the hardware at the beginning and at the end of each parallel region of the application, without requiring any user intervention, nor modifications to the runtime or application source code, nor recompiling, nor impractical offline profiling.

Next section introduces libPRISM, an infrastructure that leverages these properties to adapt the hardware knobs in runtime systems for shared memory programming models.

4 libPRISM

libPRISM is located on top of the runtime system, as shown in Figure 2, and is composed of several components: (1) the interposition mechanism, (2) monitoring, (3) hardware knob settings, and (4) the policy. libPRISM uses a library interposition mechanism to intercept calls from the application to the runtime. The monitoring components is used to gather data from different sensors of the system such as

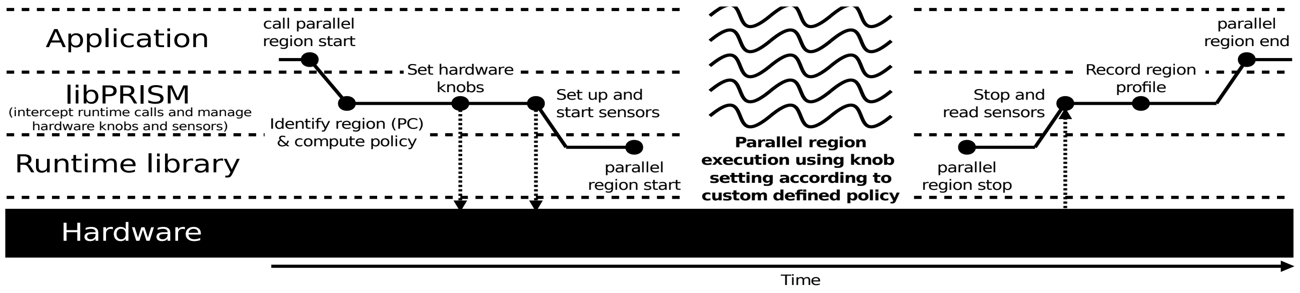

Application
libPRISM
(intercept runtime calls and manage hardware knobs and sensors)
Runtime library
Hardware
call parallel region start
Identify region (PC) & compute policy
Set hardware knobs
Set up and start sensors
parallel region start
Parallel region execution using knob setting according to custom defined policy
parallel region stop
Stop and read sensors
Record region profile
parallel region end
Time

Fig. 2: libPRISM execution stack and work flow.

performance counters, timing, and power consumption. The monitoring is performed by the master thread of the application in order to reduce possible overheads. The hardware knob settings are used to configure the underlying hardware in the system. Finally, the policy leverages the gathered information to configure the hardware knobs in the system aiming to minimize a target metric at a parallel region level.

libPRISM takes care of communicating configuration changes to the runtime system through the master thread and to the underlying hardware. The software stack shown in Figure 2 allows libPRISM to: (1) indicate to the runtime system the available threads for the incoming parallel region and set the hardware knob configuration; (2) gather profiling data from the runtime and hardware; and (3) avoid the need to change code nor recompile the application or the runtime. 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.

When a parallel region starts or ends, the application calls our library instead of the runtime system. 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 its PC, as shown in Figure 2.

Whenever a parallel region starts or ends, libPRISM intercepts the call and informs the policy about the incoming event (*call parallel region start* and *parallel region stop* in Figure 2).

For every parallel region that is executed, the policy records a performance profile under different knob configurations (*Set up and start sensors* and *Stop and read sensors* in Figure 2). The policy builds this performance profile for each parallel region using different performance counters (executed instructions and cycles), the power consumption, and the number of times the region has been executed.

HPC parallel applications consist of a set of phases that are iteratively executed, and each phase is usually composed of one or more parallel regions. Therefore, a given parallel region will be executed multiple times. The policy takes advantage of this repetitive behavior of parallel applications to find the best knob configuration for each parallel region. To do this, the policy uses an iterative learning approach. In the first iterations of each parallel region, the policy explores several possible configurations to build a performance profile and uses it to determine the best hardware knob configuration. Once the policy finds the best hardware knob configuration, libPRISM tracks the behavior of a parallel region and, in case the behavior changes, it starts the knob configuration optimization phase again.

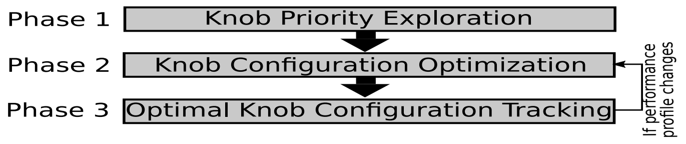

Phase 1
Knob Priority Exploration
Phase 2
Knob Configuration Optimization
Phase 3
Optimal Knob Configuration Tracking
If performance profile changes

Fig. 3: Phases of the generic policy.

We implement different policies using the libPRISM infrastructure to tune the SMT, the data prefetcher, and the DVFS knobs in order to exploit the optimization opportunities to minimize execution time, EDP and power consumption. In the next section, we explain in detail the algorithm of our proposed policies.

5 libPRISM Policies

Policies leverage the gathered data by libPRISM to find the optimal knob configuration for a certain target metric (e.g. execution time, power consumption, or EDP). To reach their goal, policies can manage several hardware knobs such as the SMT level, the data prefetcher and the DVFS knobs.

In this work, we propose a novel policy that minimizes a specified metric. This novel policy is implemented as a generic policy to allow its extension with hardware knobs and metrics (execution time, EDP, or power consumption in this work). Our generic policy uses a vector of hardware knob configurations for each hardware knob to be optimized. This vector contains the different possible configurations a hardware knob can use as an input for our algorithm. The knob vector is useful to reduce the time spent building the performance profile of each parallel region for hardware knobs such as the data prefetcher or the DVFS knobs, which can have hundreds of possible configurations. Users can change the hardware knob vector to better suit their needs. Short vectors converge in the best hardware knob configuration faster than longer vectors, while longer vectors can have more fine-grained configurations than short vectors.

In order to find the best hardware knob configuration per parallel region, our policy goes through the three phases shown in Figure 3.

The first phase is the knob priority exploration shown in Listing 1, which decides the hardware knobs with the most impact on performance. This phase uses the N first iterations of a parallel region, where N is the number of hardware knobs available in the system.

The hardware knobs with a higher boost on performance are explored first in order to improve performance as soon as possible and achieve a closer performance to the best

```
 1 // Call to parallel_region_begin intercepted
 2 function parallel_region_begin_wrapper {
 3   //Select HW KNOB to measure
 4   HW_KNOB = VECTOR_HW_KNOBS[iteration]
 5   set_CONF_NO_AGGRESSIVE(HW_KNOB)
 6   ++iteration
 7
 8   start_measurement()
 9
10   // Return control to runtime
11   parallel_region_begin_real()
12
13   end_measurement()
14   measure_performance(HW_CONF_DEFAULT,
        HW_CONF_NO_AGGRESSIVE)
15
16   if iteration == len(VECTOR_HW_KNOBS):
17     //From highest performance to lowest
          performance
18     sort_vector(VECTOR_HW_KNOBS)
19
20     //This phase is completed
21     next_phase()
22 }
```

Listing 1: Knob priority exploration phase (phase 1) of the proposed generic policy.

performance earlier. Our generic policy learns which hardware knobs have more impact on performance by testing each hardware knob and their performance boost when going from a default configuration to a less aggressive configuration (lines 5 to 14 of Listing 1).

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). Depending on the application running, our policies will explore first the different SMT configurations, the different prefetcher aggressiveness or the different DVFS configurations. Usually, the DVFS knob is explored last due to the SMT level and the prefetcher aggressiveness reporting a higher boost in terms of speedup[3]. In our experiments, the DVFS knob does not lead to any speedup in terms of execution time but it does lead to a reduction on power consumption. This is because the default configuration for the DVFS knob is the highest frequency in the system. Therefore, reducing it can lead to a performance degradation.

Once our policy knows the best order to explore the hardware knobs, it enters in the knob configuration optimization phase (shown Figure 3 and its pseudocode in Listing 2). In this phase, the policy explores different hardware knob configurations specified in the hardware knob vector seeking to minimize a specified metric with respect to the best hardware knob configuration.

In this phase, a hierarchical search algorithm is used as a global algorithm to explore different hardware knobs individually. The algorithm in this phase tunes first the hardware resources that have more impact on the final performance of the application based on the results obtained in the previous phase (line 18 from Listing 1 and line 21 from Listing 2.) Our heuristic-based search allows converging faster to a hardware knob configuration that provides a better performance by taking into account inter-knobs

3. Turbo boost is not enabled in our experiments

```
 1 // Call to parallel_region_begin intercepted
 2 function parallel_region_begin_wrapper {
 3   if targetMetric[PC] > min_time_threshold:
 4       executions[PC] + 1
 5       if executions[PC] == repetitions:
 6         previousMetricPerformance =
               currentMetricPerformance
 7         currentMetricPerformance =
               avgMetricPerformance()
 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 previousMetricPerformance >
         currentMetricPerformance
18   && time > bestTime*(1+Degradation):
19     best_configuration = current_configuration
20     bestMetricPerformance =
           currentMetricPerformance
21     next_HW_knob()
22     if performance_current_knob >>
           performance_previous_knobs
23       reset_previous_knobs()
24   else:
25     move_next_configuration_current_knob()
26   set_current_knob_configuration()
27 }
```

Listing 2: Knob configuration optimization phase (phase 2) of the proposed generic policy.

effects. This reduces the overheads associated to exploring hardware knob configurations.

For each hardware knob, the generic policy implements a greedy search through the different configurations in the vector of configurations of that knob. The use of a greedy algorithm instead of an exhaustive one is needed to reduce the overhead cost of exploring all the possible configurations of the hardware knobs.

The first time a parallel region is executed, libPRISM sets the available hardware knobs to the first hardware knob configuration specified in the vector of hardware knob configurations and records its performance profile. This measurement is repeated a number of *repetitions* 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).

If the duration of the parallel region is too short (i.e. below a threshold), libPRISM stops the knob configuration optimization phase as the cost of reconfiguring the available hardware knobs would neglect the potential performance benefits of an optimized hardware configuration (Line 3 in Listing 2). Therefore, short parallel regions (as well as serial regions) run with the hardware knob configuration already set in the system. Short parallel regions are not aggregated into a larger parallel region due to the possible execution paths and order of execution of the parallel regions, which can change during the execution of the application. Since the time spent changing the specific hardware knobs is paid at least once per parallel region, this threshold needs to be an upper-bound of the worst case scenario when changing all the hardware knobs. We present a detailed overhead

```
1 metric = readCurrentMetric()
2
3 if metric > avgBestMetric*(1+threshold):
4   increase_repetitions()
5   reset_exploration()
6 else:
7   set_best_HW_knob_configuration()
8   execute_parallel_region()
```

Listing 3: Optimal knob configuration tracking phase (phase 3) of the proposed generic policy.

analysis in Section 7.4.

Next time the same parallel region is executed, libPRISM sets the hardware knob configuration to the next possible hardware knob configuration and measures its performance profile again. If setting the next hardware knob configuration leads to a degradation in terms of the target metric, the knob configuration optimization phase for the current knob stops and the previous configuration is selected as the best found performing configuration for this knob (Lines from 17 to 20 in Listing 2). Notice that this mechanism avoids achieving worse performance than the default hardware knob configuration of the system. Then, the policy continues the knob configuration optimization phase with the next knob to configure (Line 21 in Listing 2).

The maximum number of iterations for the knob configuration optimization phase without taking into account reentering in the phase is: $\sum_{i=1}^{N} length\_vector\_HW\_knob_i$, where N is the number of hardware knobs to configure. Vectors of hardware knob configurations can have different lengths. Depending on the application and the selected policy in libPRISM, the number of iterations to find the best hardware knob configuration can vary. For instance, when maximizing performance, libPRISM stops exploring as soon as the performance is degraded, using less iterations for the tuning phase. When minimizing power, libPRISM can explore more configurations as long as the power is reduced, using more iterations for the tuning phase. In our experiments, we observe that the maximum number of iterations is never reached. We measured the number of iterations needed to achieve a steady hardware configuration with libPRISM in our experimental setup[4], our observations show that less than 10 iterations (6.1 on average) are enough to tune non-variable parallel regions when the DVFS knob is not involved. For the DVFS knob there are 22 possible power levels in our infrastructure, and the different policies require different number of iteration to tune it. When minimizing execution time, the maximum observed number of iterations to tune the DVFS knobs is 5. On the other hand, when minimizing power consumption, the number of iterations can reach up to 20 iterations. These typically are a small fraction of the total number of iterations of a parallel region, 338.6 on average in our experimental setup.

Notice that the vector hardware knobs are configured by the user with all the configurations to be tested for each knob. An user could reduce the number of iterations spent tuning different configurations by selecting a reduced number of configurations for each vector hardware knob.

After the knob configuration optimization phase, the policy identifies a competitive performing knob configuration for a particular parallel region and reaches the optimal knob configuration tracking phase where tracks the performance profile of each parallel region as shown in Figure 3. The pseudocode of this phase is shown in Listing 3. 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 performance profile of the parallel region is compared against the average performance profile found during the knob configuration optimization phase.

If the last measured performance of a parallel region differs more than a configurable *threshold* (Line 3 in Listing 3) from the average performance of that parallel region, the knob configuration optimization phase is restarted with an increased number of *repetitions* to obtain a new average performance, which minimizes continuous reconfiguration overheads and takes into account different control flow paths (Line 4 in Listing 3). This threshold needs to take into account the possible variability in the execution time of a parallel region. If the execution time of a parallel region presents a large variability (e.g. because of shared environments or different behavior in different iterations) this threshold needs to be larger. In our experiments, we configure this *threshold* as 5.0%.

We configure our generic policy with different hardware knobs, metrics, and optimization goals. Table Table 1 shows the policies derived from the different configurations that are evaluated in this work. For each policy, we show the possible knob configuration that a hardware knob can use and the inputs metrics and optimization goal of the metric.

The following sections explain in detail how we configured our generic policy to optimize different target metrics.

5.1 MAXPERF Policy

The MAXPERF policy seeks to maximize performance by minimizing the execution time. To that end, we define a metric to minimize execution time of the parallel regions and the hardware knob configuration vectors for hardware knobs: SMT level, data prefetcher and DVFS.

- For the SMT level, MAXPERF explores four SMT levels: SMT8, SMT4, SMT2 and ST.
- For the data prefetcher, MAXPERF explores four configurations: most aggressive, aggressive, default aggressiveness and disabled configurations (shown in Table 1 as 3,2,1,0, respectively).
- For the DVFS knob, MAXPERF only selects the highest frequency, which is the default configuration. In our experiments, lowering frequency only increases the execution time. Therefore, if we seek to minimize the execution time, frequency needs to be set to the highest available configuration.

5.2 MINEDP Policy

The MINEDP policy seeks to minimize the EDP, i.e. maximize speed up while reducing the power consumption. In the MINEDP policy, execution time and EDP are used as input metrics with the constraint that execution time cannot be degraded.

4. This includes all the benchmarks used in our evaluation. Our experimental setup and benchmarks are described in Section 6

TABLE 1: Summary of policies used in this work. SMT can be configured as SMT8, SMT4, SMT2, or ST. Prefetcher can be set to the most aggressive (3), aggressive (2), default (1), or disabled (0). DVFS is explored in steps of 0.06 GHz.

Policy	Knob configurations			Input metrics	Optimization goal
	SMT	Prefetcher	DVFS		
MAXPERF	8,4,2,1	3,2,1,0	3.49 GHz	Execution time	Minimize execution time
MINEDP	8,4,2,1	3,2,1,0	3.49 GHz to 2.06 GHz	Execution time and power consumption	Minimize execution time and power consumption
MINPOWER	8,4,2,1	3,2,1,0	3.49 GHz to 2.06 GHz	Execution time and power consumption	Minimize power consumption with a maximum configurable performance degradation

The MINEDP policy uses 3 hardware knobs and their corresponding hardware knob configuration vectors are the following:

- For the SMT level, the MINEDP policy explores all the SMT levels available in our platform: SMT8, SMT4, SMT2, and ST.
- For the data prefetcher, the MINEDP policy explores four configurations: most aggressive, aggressive, default aggressiveness and disabled configurations (shown in Table 1 as 3,2,1,0, respectively).
- For the DVFS, the MINEDP policy explores 22 configurations, from the highest (3.49 GHz) to the lowest frequency (2.06 GHz) by steps of 0.06 GHz.

Based on the knob priority exploration phase, the DVFS knob is explored the last because the default configuration for the DVFS knob is set to the highest available frequency in the system. Therefore, reducing the frequency can only lead to a performance degradation. The SMT level and the data prefetcher knobs have a higher beneficial impact on performance than the DVFS knob. Therefore, SMT level and the data prefetcher are explored first. This method allows us to achieve a similar performance than the MAXPERF policy and then use the DVFS knob to reduce the power consumption without affecting the performance. As a result, increasing the energy efficiency of the system.

5.3 MINPOWER Policy

The MINPOWER policy seeks to minimize the overall power consumption of the platform with respect to the optimal hardware knob configuration for execution time.

To achieve the optimal configuration, the MINPOWER policy allows changes in the hardware configuration knobs if execution time is improved. In the second phase of our algorithm, libPRISM uses the optimal hardware knob configuration as starting point to reduce power consumption.

This policy allows a performance degradation in terms of execution time with respect to the best hardware configuration to achieve greater savings on power consumption. The performance degradation in terms of execution time can be controlled with a *degradation* threshold defined by the user (Line 18 in Listing 2). The higher the value of this threshold, higher performance degradations are allowed and higher savings in power consumption can be achieved. The user needs to set this threshold according to its needs. The input metrics for this policy are power consumption and execution time.

This policy explores different hardware knob configurations defined in the hardware knob configuration vectors, which are:

- For the SMT level, MINPOWER explores all the SMT levels available in our platform: SMT8, SMT4, SMT2 and ST.
- For the data prefetcher, the MINPOWER policy explores four configurations: most aggressive, aggressive, default aggressiveness and disabled configurations (shown in Table 1 as 3,2,1,0, respectively).
- For the DVFS knob, the MINPOWER policy explores 22 configurations, from the highest frequency (3.49GHz) to the lowest frequency (2.06GHz) by steps of 0.06GHz.

5.4 Case Study: MINPOWER Policy

In this section we illustrate the detailed behavior of the MINPOWER policy to select the best hardware knob configuration for the CG application.

The MINPOWER policy minimizes power consumption while performance is not degraded more than a certain threshold with respect to the maximum performance achievable (10% in this example). For clarification in this example, the MINPOWER policy just explores default aggressiveness and disabled prefetcher configurations for the prefetcher knob and for the DVFS knob it explores from the highest to the lowest frequency by steps of 0.1 GHz. SMT level is explored as explained in Section 5.3.

Figure 4 shows how the knob configuration optimzation phase (shown in Listing 2) is performed on the longest parallel region of CG benchmark. This figure shows the selected SMT level, the prefetcher, and frequency configuration in a particular iteration of the parallel region, as well as the execution time of the parallel region under this configuration. The first iteration of a parallel region runs in the default hardware knob configuration in order to use it as a reference for the knob priority exploration phase. The knob priority exploration phase (shown in Listing 1) is realized from iteration 1 to iteration 3, which detects what are the knobs impacting most the performance. The policy measures speedup in execution time for a lower aggressive configuration of each hardware knob.

Then, libPRISM goes to the knob configuration optimization phase. Since in this application, the prefetcher is the hardware knob with most impact it starts explore the prefetcher aggressiveness from iteration 3, which has no more possible configurations. Therefore, MINPOWER decides to turn it off. Then, in the next iteration 4, the MINPOWER policy lowers the SMT from SMT8 to SMT4 just to realize that it slowdowns the execution time and power consumption is not improved. After exploring the prefetcher aggressiveness and the SMT level, the policy explores the DVFS knob vector. From iteration 5 to 10, the MINPOWER policy lowers frequency until it sees a performance degradation of the specified threshold of 10%. Therefore, it stops the exploration and goes to the optimal knob configuration tracking phase (shown in Listing 3).

In the case that a hardware knob has a performance interaction with other hardware knob that has been previously explored, libPRISM can reset the exploration of all

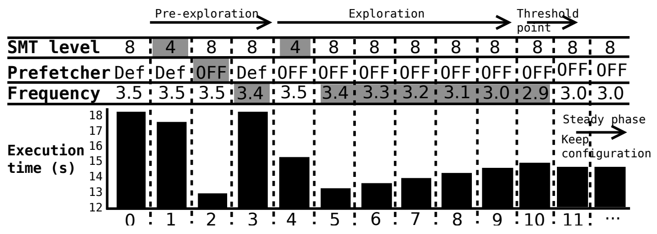


Fig. 4: MINPOWER policy with a 10% threshold in libPRISM to select a competitive performing configuration for SMT level, data prefetcher and DVFS for the CG application. Details on the hardware knob configuration are explained in Section 6. Repetitions is set to 1 (Line 5 in Listing 2).

the hardware knobs in order to consider the interaction, as shown in Lines 22 and 23 in Listing 2.

When an important change in performance during the optimal knob configuration tracking phase happens, the MINPOWER policy starts again the knob configuration optimization. For this case study, the policy does not detect any phase change during the rest of the execution in CG of this parallel region.

In this specific application, we can observe that it is better to use a high SMT level (SMT8), moderately high frequency (3.0GHz), and disable the prefetcher. The largest parallel region of CG does random memory accesses and uses the read data in a simple calculation. Disabling the prefetcher allows to reduce memory bandwidth and, thus, reduce the latency for useful memory accesses, which allows the application to exploit a higher SMT level. As the memory accesses are slow and the computation depends on them, it is possible to run all the threads with a lower DVFS level. This reduces power consumption while not degrading performance more than a given threshold (10%).

6 Experimental Setup

We evaluate our solutions on an IBM POWER8 based system (8335-GTA model) [47]. This system has an IBM POWER8 processor that runs at 3.49GHz with 512GB of DDR3 CDIMM memory running at 1.6GHz. The POWER8 processor in this system is packaged as a single-chip module with 20 cores. Each core has 64KB L1 data and 32KB L1 instruction caches, a 512KB L2 cache and an 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 supports OpenMP 4.0.

6.1 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 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 lowers the SMT level to ST level automatically.

TABLE 2: Voltage used when running a benchmark designed to stress the power consumption of the processor with different frequencies. Voltage is normalized to the highest voltage observed.

Frequency (GHz)	2.06	2.5	2.8	3.1	3.3	3.49
Voltage	0.8	0.86	0.89	0.94	0.97	1

To set the SMT level, we need to specify the number of threads running in a physical core. 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.

6.2 Data Prefetcher

The data prefetcher can be controlled at the core level by a special purpose register called Data Streams Control Register (DSCR) [30], which is exposed by the OS. The DSCR has 12 different fields with 25 bits in total. 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 of 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.

In our experimental setup, we observe that the aggressive prefetcher configuration performs better or equal than the medium configuration in most of the cases. However, in a small amount of cases, we observe that the aggressive prefetcher configuration reduces the hit ratio of the last level cache because it replaces useful blocks to make room for inaccurately prefetched blocks.

6.3 DVFS

The DVFS hardware knob is controlled at the physical core level through an exposed file by the OS. Therefore, libPRISM sets the frequency of the 10 physical cores.

The POWER8 system has 44 possible frequency configurations from 2.0 GHz to 3.49GHz by steps of 0.03GHz and 0.04 GHz as reported by the OS. Each frequency has a determined voltage associated. We run a maximum power stressmark [3] in order to measure the upper voltage limit associated with each frequency. Table 2 shows the processor voltage when executing the benchmark to stress power consumption with a specified frequency. The difference between running the benchmark to stress power consumption at the highest and lowest frequency in terms of voltage is 20%. By default, DVFS selects the highest frequency when running any benchmark we evaluated. Maximum power consumption of the evaluated benchmarks achieves only

42% of the maximum power consumption observed when running the benchmark to stress power consumption.

6.4 Benchmarks

To evaluate the effectiveness of the policies implemented in libPRISM, we use the NAS Parallel Benchmarks (NPB) suite [40] with the class D inputs. The NPB suite is composed of five kernels (CG, EP, FT, IS, and MG) and three pseudo-applications (BT, LU, and SP), which are derived from computational fluid dynamics (CFD).

Benchmarks are executed on 20 cores and pinned to them to avoid thread migration. We pin threads to cores using the environment variable *OMP_PLACES*. Benchmarks run in isolation with 20, 40, 80, or 160 threads for ST, SMT2, SMT4, and SMT8, respectively.

6.5 Metrics

In Section 7, we report speed up in execution time, power consumption and energy-delay product (EDP) for all the benchmarks. We report EDP instead of energy since degrading execution time of the application has a higher penalty with this metric. If energy ($POWER \times EXECUTION\ TIME$) were reported instead of EDP, hardware configurations with a lower power consumption would be reported as efficient hardware configurations, even if those hardware configurations are highly degrading execution time. Therefore, EDP reflects that our dynamic mechanisms have minimal penalties in terms of execution time while reducing power consumption.

We measure wall time for the entire application. When running with libPRISM infrastructure, we also read the timebase register from the POWER8 processor for fine-grained analysis of parallel regions and multiple performance counters (executed instructions and cycles) are collected using `perf` [17].

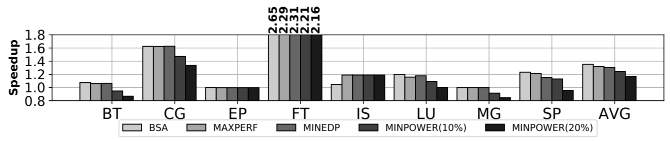
We use AMESTER (Automated Measurement of Systems for Energy and Temperature Reporting) [29] to measure the power consumption of the processor and memory chips. The tool remotely collects power, thermal and performance metrics from the system using the Intelligent Platform Management Interface (IPMI). The IPMI allows reading different hardware sensors without using any of the processing cycles of the system. Therefore, it has no impact on the performance of the running benchmarks.

In Section 7, we report the average power consumption for the total execution and energy-delay product (EDP). Power consumption includes the idle power of the system.

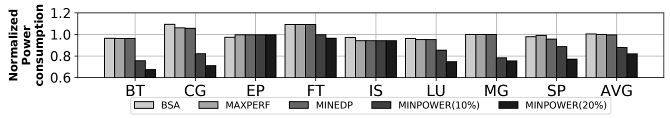
7 Evaluation

In this section we evaluate the behavior of different policies: Best Static per Application (BSA), MAXPERF, MINEDP, MINPOWER (10%), and MINPOWER (20%).

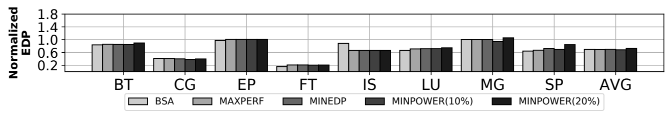
BSA is the best performing hardware configuration found for each application after an offline profiling (352 configurations: 4 SMT levels $\times$ 4 prefetcher aggressiveness $\times$ 22 frequency levels). Notice that the BSA configuration achieves the best possible performance with a static hardware knob configuration and can only be outperformed with a dynamic hardware knob configuration.



(a) Execution time



(b) Power consumption



(c) Energy-Delay Product (EDP)

Fig. 5: Results with respect to default configuration (SMT8, default prefetcher and the highest frequency of 3.5GHz). Best Static per Application (BSA): best SMT level, prefetch aggressiveness and frequency configuration for all the execution found after an offline profiling. libPRISM is running with the MAXPERF, MINEDP, MINPOWER 10% and MINPOWER 20% policies that select the hardware knob configuration for a certain metric per parallel region at execution time.

MAXPERF dynamically sets the hardware knob configuration for every parallel region based on the MAXPERF policy, which seeks the maximum performance in terms of execution time.

MINEDP dynamically sets the hardware knob configuration for every parallel region based on the MINEDP policy, which seeks the minimum EDP within the maximum performance achievable in terms of execution time.

MINPOWER (10%): dynamically sets the hardware knob configuration for every parallel region based on the MINPOWER policy with a threshold of 10%, which seeks the minimum power consumption while sacrificing up to 10% execution time with respect to the execution time of BSA.

MINPOWER (20%): dynamically sets the hardware knob configuration for every parallel region based on the MINPOWER policy with a threshold of 20%, which seeks the minimum power consumption while sacrificing up to 20% execution time with respect to the execution time of BSA.

Figure 5(a) shows the speedup results in execution time with respect to the default hardware configuration (SMT8, default data prefetcher and the highest frequency of 3.5GHz) for the Best Static for Application hardware Configuration (BSA) and all our policies. By comparing the results with respect to BSA, we can observe how the performance degradations introduced by our policies affect performance, power consumption, and EDP.

Figure 5(b) shows the power consumption normalized to the default hardware configuration for the BSA configuration and all our policies on top of libPRISM.

Figure 5(c) shows the EDP normalized to the default hardware configuration for the BSA configuration and all our policies on top of libPRISM.

In the next sections we comment the results for each of our policies: MAXPERF, MINEDP and MINPOWER.

7.1 MAXPERF Policy

7.1.1 Performance

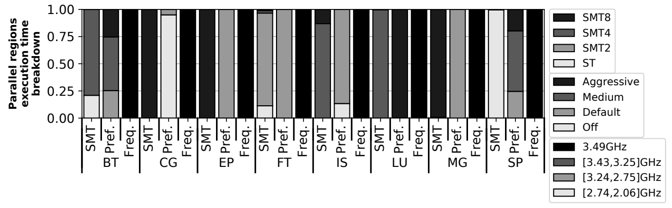


Fig. 6: Final hardware configuration for the different parallel regions when running with libPRISM and MAXPERF policy.

Figure 5(a) shows that the default hardware configuration is already the best performing configuration for 2 out of 8 evaluated benchmarks. For the remaining 6 benchmarks, 5 benchmarks can reach performance improvements above 20% illustrating the need for an adaptive system that manages shared hardware resources. On average, BSA reaches a 35.5% performance improvement over the default configuration. The policy MAXPERF almost achieves the same performance improvement as the BSA (31.6%).

Figure 6 shows the final hardware configuration in terms of SMT level, data prefetcher aggressiveness and frequency for all parallel regions. As we can see, most of the benchmarks run with 1 or 2 different configurations (frequency is set to the highest frequency). For instance, IS runs with SMT4 and the prefetcher disabled for the 13% of the time and with SMT2 and the prefetcher with the default configuration for the remaining 87% of the execution. The difference in the average performance between the MAXPERF policy and the BSA comes mainly from the benchmark FT and IS.

The benchmark FT is composed of several parallel regions, and FT iterates through these parallel regions from 1 to 27 times. In the case of executing a parallel region once, libPRISM cannot improve performance. In the cases of executing a parallel region 27 times, libPRISM spends several iterations to explore and set the hardware knob configuration. This exploration overhead accounts for the difference in execution time.

In the case of IS, the MAXPERF policy improvement over the BSA is due to the dynamic behavior of libPRISM. As we can see in Figure 6, IS runs 13% of the time in SMT8 and 87% of the time in SMT4 and the prefetcher is disabled for 13.3% of the time.

In EP, we observe that all the policies have the same behavior. EP is composed of some time consuming parallel regions that are only executed once and some very short parallel regions that are executed multiple times. In this scenario, libPRISM is not able to tune the knobs for the time consuming parallel regions, so they are executed with the default hardware knob configuration. In addition, libPRISM does not tune the hardware knobs for the short parallel regions because the overhead of reconfiguring the knobs is higher than their execution time. In contrast, BSA has slightly worse performance than the default hardware configuration due to the overheads when setting the hardware knob configuration for the short parallel regions.

The other benchmarks can run with the highest speedup with a static configuration, which is found after an exhaustive offline profiling. The MAXPERF policy is able to dynamically match at runtime the same performance as the BSA configuration without requiring any offline profiling.

7.1.2 Energy Efficiency

Next, we discuss the energy efficiency results obtained with libPRISM using the MAXPERF policy. Figure 5(b) shows the power consumption of the processor when running with the BSA configuration and our policies. Power results are normalized to the default configuration. In the MAXPERF policy, power consumption on average is the same as the BSA configuration (81.6% and 81.4%, respectively). MAXPERF can slightly reduce power consumption on some benchmarks or slightly increase it. The differences come from parallel region that are executed once, therefore, MAXPERF runs those parallel regions with the default hardware knob configuration, which can differ from the BSA configuration.

In terms of EDP, Figure 5(c) shows EDP normalized to the default configuration. Results show that the MAXPERF policy is able to reduce it by 30% with respect to the default hardware knob configuration.

We can appreciate differences between the MAXPERF policy and the BSA configuration in several benchmarks such as FT and IS. In the case of IS, the MAXPERF policy can reduce the EDP up to 5% with respect to the BSA configuration. The difference comes from a better execution time and better power consumption with respect to the BSA configuration. For several parallel regions of these benchmarks, libPRISM adapts the hardware knob configuration to different intra application requirements by lowering the level of different hardware knobs as shown in Figure 6.

7.2 MINEDP Policy

7.2.1 Performance

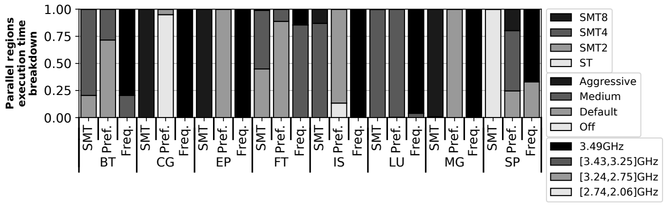
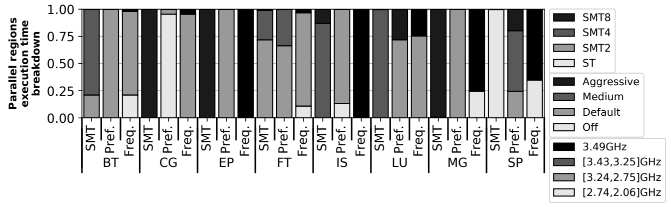


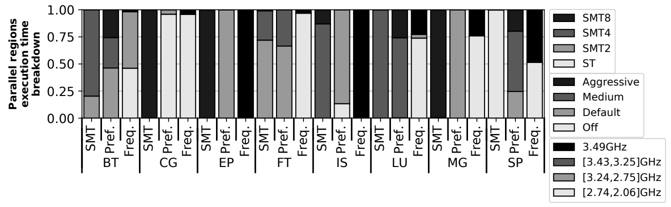
Fig. 7: Final hardware configuration for the different parallel regions when running with libPRISM and MINEDP policy.

In Figure 5(a) we can see that MINEDP achieves a similar performance to BSA and MAXPERF (30.8% on average). Yet, Figure 7 shows that the MINEDP policy is able to reduce frequency in several parallel regions from benchmarks such as FT, LU and SP while achieving the same performance.

In the case of BT and FT, the MINEDP policy is able to reduce frequency to 3.43GHz for 20% of execution time of BT and 85% of execution time of FT. In the case of SP, frequency can be lowered to 3.19GHz for 33% of the execution time. In the case of the other five benchmarks, the MINEDP policy is not able to lower the frequency and keep the same performance due to requirements of the parallel regions. In the case of CG, Figure 5(a) shows that the configuration



(a) 10% Threshold



(b) 20% Threshold

Fig. 8: Final hardware configuration for the different parallel regions when running with libPRISM and MINPOWER policy with thresholds of 10% and 20%.

selected by MINEDP achieves the same performance as MAXPERF. This is caused by the constraint to not reduce performance with respect to the best performing hardware configuration. As shown in Section 7.3, CG can achieve a lower EDP and power consumption if a higher performance degradation is allowed.

7.2.2 *Energy Efficiency*

As we can see in Figure 5(b), power consumption is slightly reduced due the MINEDP policy lowers the frequency for several parallel regions in different benchmarks. In the case of SP, MINEDP policy can reduce power consumption by 4.3% with respect to the BSA configuration while lowering the frequency 2.1% with respect to the BSA configuration.

In terms of EDP (see Figure 5(c)), the MINEDP policy achieves the same EDP as the BSA configuration and the MAXPERF policy since in EDP calculation execution time has more weight than power consumption. The slightly reduced power consumption caused by a lower frequency is not highly reflected in this metric.

From Table 2, we see that the highest drop on voltage is 20%, which happens from the highest frequency to the lowest frequency in an ideal scenario. The MINEDP policy is not able to reduce frequency to the lowest frequency due to the performance constraint and power consumption is not lowered more due to serial regions of the code, overheads of reconfiguring the hardware knobs, and total power consumption from the parallel region. Therefore, in our next evaluated policy we relax the performance constraint.

7.3 MINPOWER Policy

7.3.1 *Performance*

In Figure 5(a), we show two configurations with different thresholds of the MINPOWER policy (10% and 20% maximum execution time degradation).

On average, the MINPOWER policy is still able to significantly improve execution time with respect to the default

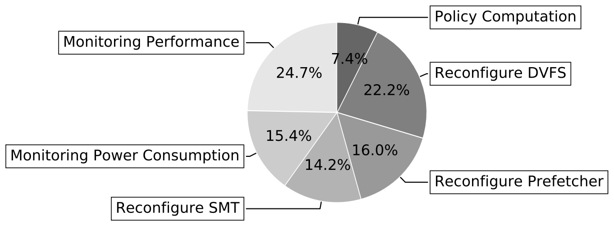


Fig. 9: Contribution to the total overhead of a single parallel region execution of all the libPRISM components.

hardware configuration more than 20%: with a 10% threshold, execution time is improved by 24% and with a 20% threshold, execution time is improved by 17%. With respect to BSA, the MINPOWER policy degrades execution time by 7% and 15% with a 10% and 20% threshold, respectively.

The main difference on execution time comes from a lowered frequency as we can see in the breakdown of the parallel execution time for 10% and 20% thresholds in Figures 8(a) and 8(b), respectively. As we can see in these figures, there are several sections of the code where frequency cannot be lowered in order to not degrade execution time. On the other hand, several parallel regions can run at the lowest frequency with only the 10% execution time degradation threshold.

7.3.2 *Energy Efficiency*

The MINPOWER policy is able to reduce power consumption as we can see in Figure 5(b). This reduction in some cases is greater than the execution time degradation. For instance, in BT we can reduce power consumption up to 30% while execution time is increased by 20% with respect to the default hardware configuration. For CG, MINPOWER can reduce power consumption up to 29% while still achieving a speedup of 47% with respect to the default hardware configuration with a 20% threshold. Notice that MINPOWER can achieve a 2% lower EDP than MINEDP in CG when allowing a higher performance degradation of 10%

In contrast, for FT and MG, increasing the execution time threshold does not achieve the same power consumption reduction, as seen in Figures 8(a) and 8(b).

Figure 5(c) shows the results for the EDP metric. The MINPOWER policy is able to significantly decrease power consumption and improve the average EDP. When the MINPOWER policy uses a 10% threshold can improve EDP up to 1.5% with respect to the BSA configuration. On the other hand, allowing a 20% execution time degradation achieves a worse EDP than the BSA configuration by 3.0%.

In the case of CG, the MINPOWER policy lowers EDP an extra 2.1% with respect to the MINEDP policy and an extra 3.5% with respect to the BSA configuration. In contrast, benchmarks such as BT or SP see a worse EDP only when we allow a higher execution time degradation (going from a 10% threshold to a 20% threshold).

7.4 Overhead Analysis

In this section, we study in detail the overheads introduced by libPRISM and how we mitigate them.

Benchmark	Smallest PR	Largest PR	Most representative PR
BT	0.0025	1.26	1.06
CG	0.0029	6.87	6.87
EP	0.0067	82.29	82.29
FT	0.0047	7.43	7.43
IS	0.0049	17.78	17.78
LU	0.0022	4.02	4.02
MG	0.0021	0.14	0.09
SP	0.0019	1.53	0.83

TABLE 3: Execution time in seconds of the benchmarks from the NPB suite of the shortest, largest, and most representative parallel region in benchmarks from NPB. The most representative parallel region is the parallel region that contributes the most to the total execution time taking into account the number of iterations of all the parallel regions.

libPRISM overheads are mainly introduced by reading different sensors, compute the selected policy, and configuring the different hardware knobs at a parallel region level: (1) Reading performance: as mentioned earlier, we use *perf* to read several performance counters such as instructions and cycles; (2) reading power consumption: AMESTER updates the power consumption every 250 microseconds and it is read at the end of a parallel region; (3) reconfiguring SMT, prefetcher, and DVFS knobs: as explained in Section 6, libPRISM needs to modify several registers exposed to the OS; and (4) Policy Computation: libPRISM needs to process the available information to configure the hardware knobs according to an user-selected policy.

These overheads have a magnitude of microseconds and their weights are shown in Figure 9. From these overheads, 5 of them are unavoidable: measuring performance, power consumption, reconfiguring SMT, prefetcher, and DVFS knobs. The only overhead we can mitigate is the policy computation, which is defined by the algorithm implemented.

We need to keep a lightweight policy computation due to the nature of the benchmarks. Several benchmarks used in this work have thousands of short parallel regions (execution time of a single iteration is shorter than a second) as shown in Figure 10 , that can represent a large percentage of the total execution of the benchmark (e.g. LU, MG, and SP). Therefore, unavoidable overheads from libPRISM can represent a considerable percentage of the total execution of a parallel region.

Also, in Table 3 we measure the execution time of a single iteration of the shortest, longest, and most representative parallel regions in the benchmarks from the NPB suite. The shortest parallel regions are usually executed within microseconds, while the largest parallel regions can take seconds to complete. Several short parallel regions have a shorter time than the unavoidable overheads of libPRISM. Notice that in MG and SP the most representative parallel region is a short parallel region. Therefore, possible overheads can degrade performance in parallel regions that are representative of the overall performance of the benchmark.

In order to mitigate these overheads, libPRISM relies on 2 mechanisms configurable by the user: (1) policies and (2) filtering of parallel regions with short duration.

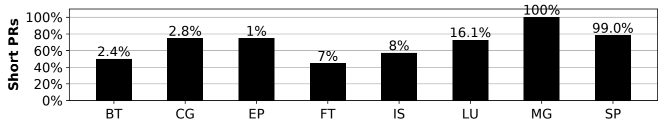


Fig. 10: Percentage of short parallel regions (execution time of a single iteration is shorter than 1 second) of the benchmarks from NPB suite. Percentage on top of each bar represents the total time spent in short parallel regions with respect to the total time spent in parallel regions.

Policies described in this paper implement a greedy search for the decision algorithm. This decision algorithm achieves the best hardware knob configuration while minimizing the policy computation overhead. Figure 9 shows the breakdown of all overheads introduced by libPRISM in a single iteration of a parallel region. As we can see, the policy computation is the smallest overhead. Also, notice that other overheads are unavoidable and cannot be reduced.

Since several overheads are unavoidable, libPRISM implements an user-specified threshold to not explore parallel regions that are shorter than a threshold. This mechanism avoids introducing overheads to short parallel regions where the overheads are larger than the parallel region itself.

Finally, we measure the total overhead introduced 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 short 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.

7.5 Discussion

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

Although we only demonstrated the usage of libPRISM for coordinating the management of SMT, prefetcher and DVFS 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 or serial 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. Using libPRISM on a different architecture or system only requires changing the way that the hardware knob configurations are passed to the architecture, and the way the power measurements are obtained from the system. All the other parts of libPRISM, including the algorithms, are independent of the architecture. However, the potential of libPRISM can change depending on the system. In particular, the most common case for x86 architectures is to offer only up to SMT2 level, the data prefetcher knobs are limited to enabling or disabling the individual prefetchers present in the architecture [35], and the power-performance efficiency of DVFS can change depending on the processor implementation.

8 Related Work

As far as we know, this is the first work to combine multiple hardware knobs such as the SMT level, the data prefetching, and the DVFS hardware knobs of a real system, evaluate the interaction between each other, and achieve a jointly-optimized configuration in runtime.

Previous work has looked at reconfiguring the hardware structures of a processor. Petrica et al. dynamically adapt the number of lanes in the front end, execute, and memory stages of a multicore processor to achieve a higher performance in a power-constrained system [36], [52]. At intervals of 100ms, they evaluate different configurations and run with the best configuration until the next interval. Jha et al. coordinate several hardware structures, cache size, and DVFS level to increase the performance under a given power budget [36]. They sample statistics from every core of a multicore processor to calculate a subset of possible configurations, and then they test different configurations of this subset to find the best configuration for the execution until a new application phase is encountered. These works need to add hardware support, while libPRISM relies on a runtime process that is able to identify phases. Therefore, libPRISM does not need any hardware support nor to continuously sample the application, reducing possible overheads.

8.1 Simultaneous Multi-threading

Previous work on SMT is focused to achieve fairness [5], [6], [7], [11], [12], [55]. 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 [26], [27], [28], [48], [54]. And other authors use SMT to achieve bet 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. [62], [63] and Heirman et al. [32] propose a dynamic algorithm inside the OMP runtime in order to choose the best number of threads. Jia et al. [37] propose a machine learning model to predict the best SMT level with the aim to boost performance. Their solution considers only a very small search space, since the SMT level has only 4 possible configurations. Increasing the amount of hardware knobs to be predicted would also increase the training set and the training time to predict the best performing hardware knob configuration.

8.2 Data Prefetching

There are previous works that propose hardware modifications of the prefetcher implementations in order to improve performance on multicore chips [2], [20], [21], [22], [59], [61], [64]. Heirman et al. [31] track late prefetches in serial and parallel applications with the hardware. Then, using this information, they tune the hardware prefetcher aggressiveness in order to reduce late prefetches and increase useful prefetches. 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. These works do not take into account possible effects with other hardware knobs such as the SMT or the DVFS.

In terms of software, most of the previous work has been developed for serial applications or multi-programmed workloads [34], [43], [44], [46]. 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 [38], [39]. 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 several hardware knobs (i.e. the SMT level, data prefetcher and DVFS knobs). In addition, in [38], [39], 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 [15]. Wang et al. uses information at compile time to correctly set the data prefetcher aggressiveness [59]. 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. [45] apply a machine learning model to predict the best settings for the data prefetcher in different parallel workloads. Their search space is small compared to our work and they do not consider possible interactions between knobs. Prat et al. added intelligence to a task-based runtime to automatically manage the aggressiveness of the data prefetcher for parallel workloads [53]. These works lack the control of the number of threads working in the same task. Therefore, the possible interaction with the SMT level, the data prefetcher and the DVFS knobs is missing.

8.3 Dynamic Voltage and Frequency Scaling

Previous work on DVFS are focused to achieve a better energy-efficient system with serial applications or multi-programmed workloads.

DVFS is usually used to reduce power consumption in program phases where the highest frequency is not needed to achieve the best performance. Hsu et al. determines theses phases at compile time [33], while Keramidas et al. [42] and Eyerman et al. [24] determine these phases at runtime with support of performance counters. In this work, we determine phases dynamically at runtime with support of a standard runtime such as OpenMP while coordinating the DVFS with multiple hardware knobs.

Some other work coordinates DVFS with other techniques to save energy. Vega et al. uses DVFS and core folding in order to reduce power consumption of the system [57]. Deng et al. uses DVFS to coordinate CPU and memory power management to reduce power consumption while remaining within some performance bounds [18]. Bitirgen et al. manage multiple hardware resources (cache partitioning, memory bandwidth, and DVFS level) in a coordinated fashion to enforce a higher-level performance objective for serial applications [4]. This approach is not applicable for parallel applications, since all the threads of the application have very similar hardware demands and, thus, it would lead to an equal partition of the hardware resources. In addition, this approach is not able to tune hardware knobs that impact the behavior of multiple threads at the same time such as the SMT level. In this work, our solutions are aimed to parallel workloads common in high performance computing and using a standard runtime such as OpenMP while using several hardware knobs.

Research on DVFS with parallel workloads has focused on using DVFS to accelerate the critical path in applications [8], [10] or to improve the overall energy efficiency of a system when running big data workloads [14]. In this work, we focus on parallel workloads in a widely-used runtime such as OpenMP and with several goals: reduce execution time, reduce EDP, and reduce power consumption.

9 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, the data prefetcher aggressiveness or the DVFS knob. 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 several policies for managing the SMT level, the data prefetcher and the DVFS hardware knobs: the MAXPERF policy with the goal of increasing performance; the MINEDP policy with the goal of reducing the overall EDP, and the MINPOWER policy with the goal of reducing power consumption at the cost of execution time.

We evaluate our solution for a wide set of OpenMP benchmarks running on an IBM POWER8 system. Results show a boost in performance of up to 2.3x (1.69x on average), a power consumption reduction of up to 33% (18% on average) and an energy-delay product reduction of up to 80% (32% on average) when compared to the default static system configuration with our proposed policies.

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 the Generalitat de Catalunya (2017-SGR-1414 and 2017-SGR-1328), and by IBM/BSC Deep Learning Center initiative.

L. Alvarez has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under the Juan de la Cierva Formacion fellowship number FJCI-2016-30984. M. Moreto and M. Casas have been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship numbers RYC-2016-21104 and RYC-2017-23269.

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. This paper is: Approved for Public Release, Distribution Unlimited.

References

[1] Ayoub, R. Z., Ogras, U., Gorbatov, E., Jin, Y., Kam, T., Diefenbaugh, P., and Rosing, T. OS-level Power Minimization Under Tight Performance Constraints in General Purpose Systems. ISLPED '11, pp. 321–326.

[2] Bakhshalipour, M., Lotfi-Kamran, P., and Sarbazi-Azad, H. Domino temporal data prefetcher. HPCA'18, pp. 131–142.

[3] Bertran, R., Buyuktosunoglu, A., Gupta, M. S., Gonzalez, M., and Bose, P. Systematic energy characterization of CMP/SMT processor systems via automated micro-benchmarks. MICRO '12, pp. 199–211.

[4] Bitirgen, R., Ipek, E., and Martinez, J. F. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. MICRO'16, pp. 318–329.

[5] Boneti, C., Cazorla, F. J., Gioiosa, R., Buyuktosunoglu, A., Cher, C. Y., and Valero, M. Software-Controlled Priority Characterization of POWER5 Processor. ISCA '08, pp. 415–426.

[6] 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. IPDPS'08, pp. 1–12.

[7] Boneti, C., Gioiosa, R., Cazorla, F. J., and Valero, M. A Dynamic Scheduler for Balancing HPC Applications. SC '08, pp. 1–12.

[8] Cai, Q., Gonzlez, J., Rakvic, R., Magklis, G., Chaparro, P., and Gonzlez, A. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. PACT'08, pp. 240–249.

[9] Casas, M., Moretó, 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. Euro-Par'15. pp. 16–27.

[10] Castillo, E., Moreto, M., Casas, M., Alvarez, L., Vallejo, E., Chronaki, K., Badia, R., Bosque, J. L., Beivide, R., Ayguade, E., Labarta, J., and Valero, M. CATA: Criticality Aware Task Acceleration for Multicore Processors. IPDPS'16, pp. 413–422.

[11] Cazorla, F. J., Fernandez, E., Ramírez, A., and Valero, M. Improving Memory Latency Aware Fetch Policies for SMT Processors. ISHPC'03. pp. 70–85.

[12] Cazorla, F. J., Knijnenburg, P. M. W., Sakellariou, R., Fernandez, E., Ramirez, A., and Valero, M. Predictable Performance in SMT Processors: Synergy Between the OS and SMTs. *IEEE Transactions on Computers 55*, 7 (2006), 785–799.

[13] Cazorla, F. J., Ramírez, A., Valero, M., and Fernández, E. Dynamically Controlled Resource Allocation in SMT Processors. MICRO'04, pp. 171–182.

[14] Cheng, D., Zhou, X., Lama, P., Ji, M., and Jiang, C. Energy Efficiency Aware Task Assignment with DVFS in Heterogeneous Hadoop Clusters. *IEEE Transactions on Parallel and Distributed Systems 29*, 1 (2018), 70–82.

[15] Chilimbi, T. M., and Hirzel, M. Dynamic Hot Data Stream Prefetching for General-purpose Programs. PLDI'02, pp. 199–209.

[16] Creech, T., Kotha, A., and Barua, R. Efficient Multiprogramming for Multicores with SCAF. MICRO'13, pp. 334–345.

[17] de Melo, A. C. The New Linux perf tools. In Slides from Linux Kongress 2010.

[18] Deng, Q., Meisner, D., Bhattacharjee, A., Wenisch, T. F., and Bianchini, R. CoScale: Coordinating CPU and Memory System DVFS in Server Systems. MICRO'12, pp. 143–154.

[19] Donald, J., and Martonosi, M. Techniques for Multicore Thermal Management: Classification and New Exploration. ISCA'06, pp. 78–88.

[20] Ebrahimi, E., Lee, C. J., Mutlu, O., and Patt, Y. N. Prefetch-aware Shared Resource Management for Multi-core Systems. ISCA '11, pp. 141–152.

[21] Ebrahimi, E., Mutlu, O., Lee, C. J., and Patt, Y. N. Coordinated Control of Multiple Prefetchers in Multi-core Systems. MICRO'09, pp. 316–326.

[22] Ebrahimi, E., Mutlu, O., and Patt, Y. N. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. HPCA'09, pp. 7–17.

[23] Everman, S., and Eeckhout, L. A Memory-Level Parallelism Aware Fetch Policy for SMT Processors. HPCA'07, pp. 240–249.

[24] Eyerman, S., and Eeckhout, L. Fine-grained DVFS Using On-chip Regulators. *ACM Trans. Archit. Code Optim. 8*, 1 (2011), 1:1–1:24.

[25] Fatahalian, K., Horn, D. R., Knight, T. J., Leem, L., Houston, M., Park, J. Y., Erez, M., Ren, M., Aiken, A., Dally, W. J., and Hanrahan, P. Sequoia: Programming the Memory Hierarchy. SC '06, pp. 4–4.

[26] Feliu, J., Eyerman, S., Sahuquillo, J., and Petit, S. Symbiotic job scheduling on the IBM POWER8. HPCA'15, pp. 669–680.

[27] Feliu, J., Sahuquillo, J., Petit, S., and Duato, J. Addressing Fairness in SMT Multicores with a Progress-Aware Scheduler. IPDPS'16, pp. 187–196.

[28] Feliu, J., Sahuquillo, J., Petit, S., and Duato, J. Perf&Fair: A Progress-Aware Scheduler to Enhance Performance and Fairness in SMT Multicores. *IEEE Trans. Comput. 66*, 5 (2017), 905–911.

[29] 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.

[30] 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 [Online]. Available: http://www.redbooks.ibm.com, 2015.

[31] Heirman, W., Bois, K. D., Vandriessche, Y., Eyerman, S., and Hur, I. Near-side Prefetch Throttling: Adaptive Prefetching for High-performance Many-core Processors. PACT '18, pp. 28:1–28:11.

[32] 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. ROSS '14.

[33] Hsu, C.-H., and Kremer, U. The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction. PLDI '03, pp. 38–48.

[34] Hur, I., and Lin, C. Memory Prefetching Using Adaptive Stream Detection. MICRO'06, pp. 397–408.

[35] Intel 64 and IA-32 Architectures. Optimization Reference Manual, September 2019. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.

[36] Jha, S. S., Heirman, W., Falcón, A., Carlson, T. E., Van Craeynest, K., Tubella, J., González, A., and Eeckhout, L. Chrysso: An integrated power manager for constrained many-core processors. In *Proceedings of the 12th ACM International Conference on Computing Frontiers* (2015), CF '15, pp. 19:1–19:8.

[37] Jia, Z., Xue, C., Chen, G., Zhan, J., Zhang, L., Lin, Y., and Hofstee, P. Auto-tuning Spark Big Data Workloads on POWER8: Prediction-Based Dynamic SMT Threading. PACT '16, pp. 387–400.

[38] Jimenez, V., Buyuktosunoglu, A., Bose, P., O'Connell, F. P., Cazorla, F., and Valero, M. Increasing multicore system efficiency through intelligent bandwidth shifting. HPCA'15, pp. 39–50.

[39] Jiménez, V., Gioiosa, R., Cazorla, F. J., Buyuktosunoglu, A., Bose, P., and O'Connell, F. P. Making Data Prefetch Smarter: Adaptive Prefetching on POWER7. PACT '12, pp. 137–146.

[40] Jin, H., and Ma, F. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Tech. Rep. NAS-99-011, NASA Ames Research Center, October 1999.

[41] Kaxiras, S., and Martonosi, M. *Computer Architecture Techniques for Power-Efficiency*, 1st ed. Morgan and Claypool Publishers, 2008.

[42] Keramidas, G., Spiliopoulos, V., and Kaxiras, S. Interval-based Models for Run-time DVFS Orchestration in Superscalar Processors. CF '10, pp. 287–296.

[43] Khan, M., Laurenzanoy, M. A., Marsy, J., Hagersten, E., and Black-Schaffer, D. AREP: Adaptive Resource Efficient Prefetching for Maximizing Multicore Performance. PACT '15, pp. 367–378.

[44] Khan, M., Sandberg, A., and Hagersten, E. A case for resource efficient prefetching in multicores. ISPASS'14, pp. 101–110.

[45] 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. CAL'16, pp. 37–40.

[46] Luk, C. K., Muth, R., Patil, H., Cohn, R., and Lowney, G. Ispike: a post-link optimizer for the Intel reg; Itanium reg; architecture. CGO'04, pp. 15–26.

[47] 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 (2015), 6:1–6:10.

[48] Moseley, T., Kihm, J. L., Connors, D. A., and Grunwald, D. Methods for modeling resource contention on simultaneous multithreading processors. ICCD'05, pp. 373–380.

[49] Murphy, R., and Kogge, P. On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications. *IEEE Transactions on Computers 56*, 7 (July 2007), 937–945.

[50] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.5, Nov. 2015. http://www.openmp.org/mp-documents/openmp-4.5.pdf.

[51] Pallipadi, V., and Starikovskiy, A. The ondemand governor: past, present and future. In *Proceedings of Linux Symposium, vol. 2, pp. 223-238* (2006).

[52] Petrica, P., Izraelevitz, A. M., Albonesi, D. H., and Shoemaker, C. A. Flicker: A dynamically adaptive architecture for power limited multicore systems. ISCA '13, pp. 13–23.

[53] Prat, D., Ortega, C., Casas, M., Moretó, M., and Valero, M. Adaptive and application dependent runtime guided hardware prefetcher reconfiguration on the IBM POWER7. ADAPT'15.

[54] Snavely, A., and Tullsen, D. M. Symbiotic jobscheduling for a simultaneous multithreaded processor. ASPLOS'00, pp. 234–244.

[55] Tembey, P., Vega, A., Buyuktosunoglu, A., da Silva, D., and Bose, P. SMT Switch: Software Mechanisms for Power Shifting. CAL'13, pp. 67–70.

[56] Valero, M., Moretó, M., Casas, M., Ayguadé, E., and Labarta, J. Runtime-Aware Architectures: A First Approach. *International Journal on Supercomputing Frontiers and Innovations 1*, 1 (2014), 29–44.

[57] Vega, A., Buyuktosunoglu, A., Hanson, H., Bose, P., and Ramani, S. Crank It Up or Dial It Down: Coordinated Multiprocessor Frequency and Folding Control. MICRO'13, pp. 210–221.

[58] Wang, W., and Leon, E. Evaluating DVFS and Concurrency Throttling on IBM's Power8 Architecture. Research Poster at SC'15.

[59] Wang, Z., Burger, D., McKinley, K. S., Reinhardt, S. K., and Weems, C. C. Guided Region Prefetching: A Cooperative Hardware/Software Approach. ISCA '03, pp. 388–398.

[60] Weinberg, J., McCracken, M. O., Strohmaier, E., and Snavely, A. Quantifying Locality In The Memory Access Patterns of HPC Applications. SC '05, pp. 50–50.

[61] Wu, C.-J., Jaleel, A., Martonosi, M., Steely, Jr., S. C., and Emer, J. PACMan: Prefetch-aware Cache Management for High Performance Caching. MICRO'11, pp. 442–453.

[62] Zhang, Y., Burcea, M., Cheng, V., Ho, R., and Voss, M. An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs. PDCS'04, pp. 256–263.

[63] Zhang, Y., Voss, M., and Rogers, E. S. Runtime Empirical Selection of Loop Schedulers on Hyperthreaded SMPs. IPDPS'05, p. 44.2.

[64] Zhuang, X., and Lee, H. S. Reducing cache pollution via dynamic data prefetch filtering. *IEEE Transactions on Computers 56*, 1 (2007), 18–31.

AUTHOR BIOGRAPHIES

Cristobal Ortega is a PhD. Student at the Universitat Politècnica de Catalunya (UPC) and the Barcelona Supercomputing Center (BSC). He received his M.S. degree in 2016 and graduated in Engineering in computer science in 2014 from the Universitat Politecnica de Catalunya (UPC). He joined BSC in 2014. His research interests are High Performance Computing (HPC), runtime systems and parallel applications.

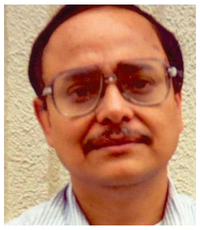
Lluc Alvarez is a postdoctoral researcher at the Universitat Politècnica de Catalunya (UPC) and the Barcelona Supercomputing Center (BSC). He received his B.Sc. degree from the Universitat de les Illes Balears (UIB) in 2006 and his M.Sc. and Ph.D. degrees from the UPC in 2009 and 2015. His main research interests are parallel architectures, memory systems and programming models for high-performance computing.

Marc Casas is a senior researcher at the Barcelona Supercomputing Center. Prior to this, he spent 3 years as a post-doctoral fellow at the Lawrence Livermore National Laboratory (LLNL). He received his B.Sc. and M.Sc. degrees in mathematics in 2004 from the UPC and the PhD in Computer Science in 2010 from the Computer Architecture Department of UPC. His research interests are high performance computing, runtime systems and parallel algorithms.

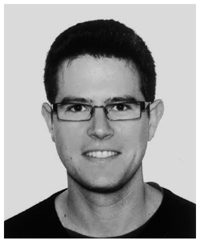
Dr. Ramon Bertran received the Ph.D. degree in computer architecture from the Polytechnic University of Catalonia, Barcelona, Spain, in 2014. He is currently a Research Staff Member with the Efficient and Resilient Systems Research Group, IBM T. J. Watson Research Center, Yorktown Heights, NY, USA. He is involved in research and development work on power-aware computer systems. His main focus is in the area of systematic power/energy modeling and characterization using automatic micro benchmarking for current and future IBM processors (Power Systems, z Systems). He is the lead developer of Microprobe: a micro-benchmark generator framework used to support his research on systematic energy characterization.

Dr. Alper Buyuktosunoglu received PhD degree in electrical and computer engineering from University of Rochester. Currently, he is a Research Staff Member in Efficient and Resilient Systems department at IBM T. J. Watson Research Center. He has been involved in research and development work in support of IBM Power Systems and IBM z Systems in the areas of computer architecture and robust power management. He has over 100 patents, has received several IBM-internal awards, has published over 100 papers, and has served on various conference technical program committees in these areas. He is currently serving on the editorial board of IEEE MICRO. Dr. Buyuktosunoglu is a member of the IBM Academy of Technology, an IBM Master Inventor and an IEEE Fellow.

Dr. Alexandre E. Eichenberger is currently a Research Staff Member in the Advanced Compiler Technologies group of the VLSI Systems department at the IBM T.J. Watson Research Center. My research interests focus on the interaction between compiler technology and microarchitecture design.



Dr. Pradip Bose is a Distinguished Research Staff Member and manager of the Efficient and Resilient Systems Research Group. His primary responsibility is to supervise advanced research and development in support of power-efficient, reliable processor design within IBM Power Systems and z System product offerings.



Miquel Moretó is a senior researcher at the Barcelona Supercomputing Center (BSC). Prior to joining BSC, he spent 15 months as a postdoctoral fellow at the International Computer Science Institute (ICSI), Berkeley, USA. He received the B.Sc., M.Sc., and Ph.D. degrees from UPC. His research interests include studying shared resources in multithreaded architectures and hardware-software co-design for future massively parallel systems.