Hardware Knob Coordination in Multi-Threaded Systems

Cristobal Ortega
Computer Architecture Department
Universitat Politècnica de Catalunya

A thesis submitted in fulfillment of
the requirements for the degree of
Doctor of Philosophy in Computer Architecture

Advisors:
Miquel Moretó
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya

Lluc Álvarez
Barcelona Supercomputing Center

PhD thesis
Doctoral programme on Computer Architecture
September, 2022

Abstract

Every new generation of high performance computing (HPC) processors brings higher complexity to boost performance: the increasing number of cores within the processor itself, cores with the ability to run multiple threads (i.e. Simultaneous Multithreading or SMT), increased available memory bandwidth, and multiple architectural improvements. On top of this, HPC systems are typically composed of multiple sockets with Non-Uniform Memory Access (NUMA) to increase the number of available processors in the system. As a result, the application running on the different processors experience contention in the acceess to shared resources, which might lead to individual application's performance degradation.

Modern HPC systems include hardware knobs to adapt architectural parameters to the workload demands with the goal of improving performance, energy, or power consumption. However, a default hardware knob configuration is set by the hardware designers and the responsibility of finding the best hardware knob configuration is left to the user, which is an overwhelming process due to the large search space and the different resource demands of applications. In addition, setting a hardware knob independently of others can result in suboptimal configurations and sometimes to conflicting decisions that jeopardize system power-performance efficiency.

Furthermore, the number and heterogeneity of compute devices, even within a single compute node, has been steadily on the rise. Conflicting decisions can appear when incorporating multiple diverse accelerators within a node to provide efficient performance growth through specialization (e.g. graphics processing unit or GPU), since complex heterogeneous systems with multiple discrete accelerators cannot afford to fully power all the devices simultaneously.

In a HPC system, the number of discrete devices that can run simultaneously at their highest frequency is limited by the globally-imposed power cap. Current systems incorporate a centralized power management unit that statically controls the distribution of power among the devices within the node. However, such static distribution policies are unaware of the dynamic utilization profile across the devices,

which leads to power allocations that end up degrading system throughput performance. The problem is particularly acute in the presence of heterogeneity since type-specific performance-boost capabilities cannot be leveraged via utilization-agnostic static power allocations.

The main goal of this Thesis is to leverage runtime system data to intelligently coordinate hardware knobs while taking into account interactions between each other at application level, hardware level, and system level. This Thesis proposes multiple methods to dynamically coordinate multiple hardware knobs to maximize system power-performance efficiency.

At application level, we propose an infrastructure for shared memory parallel programming models that transparently configures the different hardware knobs available in the architecture. During execution time, the best hardware configuration is discovered for different fine-grained regions of the application without user intervention and without modifying the original source code of the application. This Thesis proposes two mechanisms to discover the best hardware knob configuration: (1) libPRISM: an exploration-based approach useful for reduced search spaces that needs almost no previous knowledge of the system, and (2) MARK: an intelligent approach based on machine learning (ML) to search large design spaces.

At hardware level, this Thesis demonstrates that a hardware knob can benefit from fine-grained coordination between its own possible configurations. We show that runtime metrics can be used to guide the hardware knob to achieve higher power-performance effiency.

At system level, this Thesis proposes a hardware/software power distribution mechanism that maximizes the performance of power-constrained heterogeneous systems by leveraging system information to distribute power among all the devices. Our proposed solution is agnostic of the devices (CPU, GPU, or other accelerators), it uses a simple and scalable heuristic that requires minimal communication between devices, it works for single applications and multiprogrammed workloads, and it can be implemented in current systems without any hardware modification.

Overall, this Thesis demonstrates that fine-grained hardware coordination is needed at different levels to maximize energy-efficiency for parallel applications and multiprogrammed workloads.

Acknowledgements

Looking back after all these years, I realize that the Ph.D. is a long road with many ups and downs, dead-ends, and where you walk without knowing what you will encounter in the next step. On this path, good companions with whom to rest are needed in order to continue advancing. And I have had extraordinary companions.

First and foremost, I would like to thank my advisors, Miquel and Lluc for guiding me along the way. Your patience, motivation, and lessons went beyond this thesis. I have had moments of doubts, but you have always been there to support me day after day without hesitation.

Secondly, I would like to thank Alper, Ramon, and Pradip from the IBM Thomas J. Watson Research Center for guiding me during all these years. All the insights you gave me in our discussions contributed vastly to this thesis and to my life.

Also, I want to thank all the colleagues that I met at ARM Cambridge. Especially to Roxana, who made it possible, Rekai, Miguel, Ilias, and all the others. It was really interesting to learn research from an industry perspective. Not to forget the shared moments, you all made me feel at home with all the board games!

I also want to thank the RoMoL group for their constant support and help, even when I felt unsure about my Ph.D. Back are the Vertex days with all of you Isaac, Ruben, Victor, and Guillem. It's been many years and you always made me feel one more. During the last years of the thesis, your support has been essential.

Thanks to the "Cache Miss" group, Constantino, David, Pedro, and Albert for all the moments shared during so many years. Far behind are those career classes, and the long and well-deserved breaks.

I also want to thank my closest friends for their support, understanding, and motivation. All those Magic nights definitely helped me navigate through the Ph.D. As well as Lucie, my life partner, for her motivation during the tedious writing process.

Last but not least, I would like to thank my family, without them I wouldn't be who I am today.

Thank you all, those who I have mentioned and those who I may have forgotten for accompanying me on this path. I don't know what the future holds, but all of you are part of it.

"For Whom the Bell Tolls; It Tolls for Thee."

John Donne

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The evolution of microprocessors design has changed significantly in the last few decades. For many years, Moore's law drove the progress of every new generation of processors by increasing the number of transistors in the chip and its clock frequency, allowing to build faster and more complex single-core processors that could exploit Instruction Level Parallelism (ILP) of sequential programs. As shown in Figure 1.1, this trend continued until the early 2000s, when Moore's law encountered three fundamental obstacles: the memory wall, the power wall or the end of Dennard's scaling, and the ILP wall.

In the early 2000s, to overcome the performance stagnation of single-core processors, the landscape of microprocessors design entered into the multicore era. Multicore processors provide the desired performance gains by exploiting the Task Level Parallelism (TLP) of parallel programs. However, they face problems concerning the orchestration of parallel workloads such as the communication between cores or the concurrency in memory accesses, giving rise to the programmability wall.

The memory wall is the disparity between the speeds of the CPU and the main memory [165]. For many years, as the cores were increasing their frequency at a very high rate, the frequency of the main memories was increasing at a much lower pace. Due to this trend, the latency of the memory accesses grew significantly in relative terms. To solve this, CPUs include caches and prefetchers. Caches are small and high speed memories that are close to the cores, and prefetchers bring data to the caches before it is demanded by the applications. The addition of caches and prefetchers has been key to reduce the latency of the memory accesses.

The power wall is the difficulty of increasing the power consumption of chips due to the fundamental constraints imposed by affordable power dissipation and delivery [106]. As shown in Figure 1.1, the frequency of CPUs has not increased in the last years. This is due to the temperatures of the CPUs, which have reached a limit where they cannot be tolerated without incorporating extremely costly cooling systems. To solve this, multicore architectures and accelerators are introduced

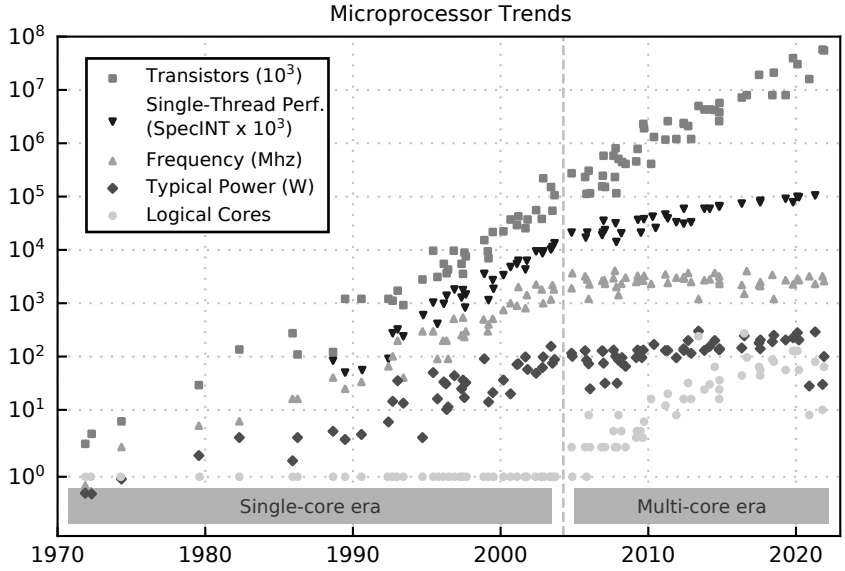


FIGURE 1.1: Trend in the number of transistors, single-thread performance, frequency and power of a core, and number of logical cores. Data up to year 2010 collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. Data spanning 2010-2020 collected by K. Rupp. Data spanning 2017-2020 collected by A. Segura. Data of 2021 collected by C. Ortega.

in modern systems, since CPUs with multiple cores running at lower frequencies are easier to cool. Also, specialized accelerators are used as a way to speedup specific computing problems while not consuming power when they are not working. Adding more accelerators that can have a high power consumption creates the problem of what device (CPU or accelerators) needs to have more power budget since all devices cannot be fully powered at the same time.

The ILP wall refers to the limited ILP in sequential applications. The ILP captures which instructions in a sequence are independent and can be executed in parallel. Dependencies on the instructions have a direct impact on the ILP of an application. For instance, if two instructions A and B are totally independent, they can be executed in parallel. Yet, if there is an instruction C that depends on the result of A and B, C cannot be executed in parallel and needs to wait for A and B to be finished. Simultaneous Multi-Threading (SMT) was introduced to maximize the occupancy of hardware resources. In a SMT processor, multiple independent threads can be executed simultaneously in the same core. Yet, SMT needs to be used correctly to maximize performance gains since scheduling applications with similar needs (i.e. integer calculations, memory bandwidth) can lead to overall slowdowns since applications are competing for hardware resources.

The programmability walls refers to the difficulties of programming complex systems with multiple CPUs and accelerators. Programming in modern systems is more difficult since, in order to maximize performance, programmers need to be

conscious of the architecture they are targeting in order that programs can fully utilize all the available hardware resources of the underlying architecture. To solve this, a new abstraction layer is introduced in the software stack: the runtime system. Runtime systems are libraries that abstract programmers from the low-level intricacies of complex parallel architectures. With this approach, programmers can write high-level parallel code, and the runtime system is responsible for orchestrating the parallel execution and efficiently using the available hardware resources.

1.1 The Need for Hardware and Software Coordination

Nowadays, every new generation of processors achieves better performance than its predecessor at the cost of augmenting the complexity of the architecture. In particular, successive generations of processors have increased the number of cores, the number of threads per core, the available memory bandwidth, and have included new sophisticated microarchitectural techniques. On top of this, most High-Performance Computing (HPC) systems are composed of multiple sockets in Non-Uniform Memory Access (NUMA) configurations. As a result of this trend, the complexity of processors has increased dramatically and the contention in the access to shared resources has been exacerbated.

Complex architectures are generally beneficial for most applications, even if they have very different characteristics. Yet, not all the features of these complex architectures are equally effective across all the applications. For instance, for a particular compute intensive application, increasing the number of cores and threads can be very beneficial, but increasing the available memory bandwidth may not be as relevant. For this reason, modern processors include numerous hardware "knobs" that allow the user to adapt the architectural parameters of the processor to the demands of the workload. This configurability is key to maximize the performance and the energy efficiency of the applications. However, the responsibility of finding the best hardware knob configuration is left to the user, which is an overwhelming process due to the large search space and the different resource demands of applications. In addition, setting a hardware knob independently of others can result in conflicting decisions that lead to underperforming configurations.

Previous research focuses on controlling hardware knobs independently of each other [28, 20, 168, 167, 80, 129, 163, 125]. This independent actuation can lead to conflicting decisions that jeopardize system power-performance efficiency [153]. For instance, parallel applications with a large amount of shared data and synchronization points can run better in a single socket (uniform memory accesses), high SMT level (high computation), and set the prefetcher to bring less cache lines (no cache pollution). Coordinating thread placement, SMT level, and data prefetcher hardware knobs is challenging due to the large search space and possible interactions

between each hardware knob.

Another trend in modern systems is to incorporate multiple diverse accelerators, which provide efficient performance growth through specialization. However, complex heterogeneous systems cannot afford to fully power all the devices simultaneously [47, 66]. To overcome this limitation, modern systems include mechanisms to adjust the power budget of the devices, allowing them to safely operate under a given power cap. The power cap can change over time for a variety of reasons, including safe operation modes when power delivery and thermal limits are reached, battery-saving modes in embedded systems, and economical reasons in servers.

Efficiently distributing power in heterogeneous systems with multiple discrete accelerators and varying power caps is a challenging and non-trivial problem. The growing amount of accelerators requires power distribution algorithms to be simple and scalable to ensure fast response times. Moreover, communication between discrete devices suffers from long latencies, therefore minimizing device communication is a must. On top of this, the proliferation of different types of accelerators (GPUs, FPGAs, ASICs, etc.) manufactured by different vendors imposes power management solutions to use generic knobs that are present in all kind of devices. Another important limitation is the lack of standard interfaces to coordinate the power consumption of every device in the system [127]. Therefore, power management of a system must be handled through software drivers. The nature of heterogeneous workloads further complicates the problem, as their power demands can drastically change due to program phases [108], accelerators being active or idle in different phases [10], and multiple applications running in different devices.

Previous works have studied how to efficiently distribute power under a limited power budget in single chips. However, these approaches are not applicable to heterogeneous system with multiple discrete devices since single chips do not have information about the requirements and the power budgets of the discrete devices.

All the previous optimizations demonstrate that, nowadays, correctly configuring all the hardware knobs present in modern systems is of paramount importance to maximize their performance and power efficiency. Unfortunately, the amount of possible hardware knobs configurations has become extremely large, to the point where finding the best configuration for each application is an insurmountable task for any user. For this reason, an intelligent automatic system is needed to find the best hardware knob configuration and power distribution for each application.

1.2 Contributions

The goal of this Thesis is to pioneer automatic hardware knob coordination mechanisms that maximize the performance and the power efficiency of complex systems.

The Thesis is structured into three different themes, each referring to a different scenario where automatic hardware knob coordination mechanisms can be introduced in an intelligent manner.

1.2.1 Hardware knob coordination for performance

The first contribution of this Thesis tackles the hardware knob coordination at an application level. First, this work demonstrates that the independent actuation of the different hardware knobs in a system can lead to conflicting decisions that jeopardize system power-performance efficiency due to interactions between different hardware knobs. This work also shows that the best hardware knob configuration changes depending on (1) the goal of the end user (increasing performance, reducing power consumption or improving energy-efficiency), (2) the behavior of the application, and (3) the input of the application. To solve the independent actuation, this Thesis proposes a runtime system that coordinates multiple hardware knobs in order to increase performance, power, or energy when a single parallel application is being executed in a shared memory system.

This Thesis first proposes and demonstrates a heuristic solution that is sufficient for a reduced search space. Then, given the limited scalability of heuristic solutions for a large number of hardware knob configurations, this Thesis proposes a solution based on machine learning that is able to scale to a huge search space.

1.2.2 Dynamic hardware knobs in hardware prefetchers

The second contribution of this Thesis demonstrates that allowing the customization of a hardware knob at execution time increases the opportunities to be adapted to the running application. This work analyzes the impact of a single hardware knob when a parallel application is running in a cycle-accurate simulator. Results show that more fine-grained options for customization of a hardware prefetcher can increase performance, reduce power consumption, or improve energy-efficiency. Thus, this work proposes a hardware modification that increases the number of configurations exposed by the hardware prefetcher. These configurations are evaluated to show that using the correct hardware prefetcher achieves performance improvements, reduces power consumption, or improves energy-efficiency.

1.2.3 Hardware knob coordination for power cap

The third contribution of this Thesis consists of a mechanisms that intelligently manages the power budgets of a system with multiple discrete devices. The work analyzes the impact of the independent actuation of a hardware knob of different multiple discrete devices, showing that it leads to performance degradations of the overall system. Then, this proposal presents a hardware/software power distribution

mechanism that carefully balances efficiency and fairness by leveraging dynamic load information, enabling devices that are more utilized to have a higher power budget than devices that are less utilized. The proposed approach is agnostic of the devices (CPU, GPU, or other accelerators), it uses a simple and scalable heuristic that requires minimal communication between devices, it works for single applications and multiprogrammed workloads, and it can be implemented in current systems without any hardware modification. The contribution is demonstrated and evaluated on a system with 2 CPUs and 4 GPUs, showing that it outperforms static power distributions and state-of-the-art solutions.

1.3 Thesis Structure

The contents of this Thesis are organized as follows:

- Chapter 2 presents the required background and terminology of the hardware knobs and systems used in this Thesis. It also presents the state-of-the-art related to the multiple proposals introduced in this Thesis.
- Chapter 3 explains the experimental setup, methodology, metrics, and tools used to analyze and evaluate the proposals introduced in this Thesis.
- Chapter 4 tackles the independent actuation of the different hardware knobs in a system when a single parallel application is running. This chapters proposes a heuristic solution to coordinate the multiple hardware knobs.
- Chapter 5 describes an intelligent solution to coordinate the multiple hardware knobs that is able to scale to large search spaces.
- Chapter 6 demonstrates that configurable fine-grained options in a hardware prefetcher allows improving the overall performance of an application.
- Chapter 7 proposes a centralized solution at a system level to coordinate the power control knob of multiple discrete devices when the system has a limited power budget.
- Chapter 8 presents the final conclusions of this Thesis and exposes directions for future work.

1.4 List of Publications

The following works have been published as a result of this Thesis:

- Cristobal Ortega, Lluc Alvarez, Miquel Moreto, Alper Buyuktosunoglu, Ramon Bertran, and Pradip Bose, "MARK: Machine Learning for Hardware Reconfiguration at Fine Granularity". To be submitted.

- Cristobal Ortega, Lluc Alvarez, Alper Buyuktosunoglu, Ramon Bertran, Todd Rosedahl, Pradip Bose, and Miquel Moreto, "Adaptive Power Shifting for Power-Constrained Heterogeneous Systems", in IEEE Transactions on Computers, May 2022, doi: 10.1109/TC.2022.3174545.

- Cristobal Ortega, Lluc Alvarez, Marc Casas, Ramon Bertran, Alper Buyuktosunoglu, Alexandre E. Eichenberger, Pradip Bose, and Miquel Moreto, "Intelligent Adaptation of Hardware Knobs for Improving Performance and Power Consumption", in IEEE Transactions on Computers, vol. 70, no. 1, pp. 1-16, 1 Jan. 2021, doi: 10.1109/TC.2020.2980230.

- Cristobal Ortega, Victor Garcia, Miquel Moreto, Marc Casas, and Roxana Rusitoru, "Data Prefetching on In-order Processors", in Proceedings of the International Conference on High Performance Computing & Simulation (HPCS), 2018, pp. 322-329, doi: 10.1109/HPCS.2018.00061.

- Cristobal Ortega, Miquel Moreto, Marc Casas, Ramon Bertran, Alper Buyuktosunoglu, Alexandre E. Eichenberger, and Pradip Bose, "libPRISM: an intelligent adaptation of prefetch and SMT levels", in Proceedings of the International Conference on Supercomputing (ICS 2017). Association for Computing Machinery, New York, NY, USA, Article 28, 1–10. doi: 10.1145/3079079.3079101

- David Prat, Cristobal Ortega, Marc Casas, Miquel Moretó, Mateo Valero, "Adaptive and application dependent runtime guided hardware prefetcher reconfiguration on the IBM POWER7" in ADAPT Workshop. 2015.

Chapter 2

State of the Art

This section provides the required background and the state of the art on Dynamic Voltage and Frequency Scaling (DVFS), thread placement, SMT, data prefetching, machine learning, power capping, and runtime systems for shared memory programming models.

2.1 Dynamic Voltage and Frequency Scaling

Nowadays, processors have different DVFS levels (multiple possible combinations of voltage and frequency) under which they can safely operate. DVFS provides a mechanism to adjust voltage and frequency dynamically on commodity hardware [42, 84]. 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. Therefore, a user could tune the DVFS hardware knob accordingly to increase energy efficiency[155]. For instance, the energy efficiency of memory bound applications can be improved by running the cores at a low frequency. Previous works on DVFS focus on achieving a better energy efficiency in serial applications and multi-programmed workloads.

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 [11, 118]. 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. Similarly, NVIDIA GPUs typically use the *Maximum Efficiency mode* [116] policy, which sets the maximum frequency when a workload is running and it lowers the frequency otherwise.

Some works propose using DVFS to reduce power consumption in program phases where the highest frequency is not needed to achieve the best performance. Hsu et al. determine theses phases at compile time[71], while Keramidas et al. [85]

and Eyerman et al. [49] determine these phases at runtime with support of performance counters.

Other works coordinate DVFS with techniques to save energy. Vega et al. use DVFS and core folding in order to reduce power consumption of the system[153]. Deng et al. use DVFS to coordinate CPU and memory power management to reduce power consumption while remaining within some performance bounds [38]. These approaches are not applicable to parallel applications, since all the threads of the application have, in general, very similar hardware demands and, thus, it would lead to an equal partition of the hardware resources. In addition, these approaches are not able to tune hardware knobs that impact the behavior of multiple threads at the same time such as the SMT level.

Research on DVFS with parallel workloads has focused on using DVFS to accelerate task that are in the critical path of the application [26, 23] or to improve the overall energy efficiency of a system when running big data workloads [32].

2.2 Thread Placement and Simultaneous Multithreading

Thread placement is a known hardware knob to impact performance in NUMA systems [125, 140] due to their non-uniform behavior when accessing memory. Parallel applications have a mixture of private and shared resources and, depending on the executed parallel workload, using all cores within all sockets can lead to a degraded performance due to contention on shared resources (e.g., threads lose data locality across sockets). Therefore, using the correct thread placement can boost performance by increasing data locality or reduce data movement across sockets.

SMT increases the number of running threads within the same core, which helps 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 be degraded due the contention on the shared hardware resources.

Multi-programmed workloads can significantly benefit from higher SMT levels when multiple applications stress different functional units or have different memory access patterns. Therefore, the usage of the hardware resources is higher [54, 143, 104, 48]. In contrast, parallel applications that follow a traditional fork-join parallelization scheme execute the very same code on the different threads. In this scenario, all threads compete for the same hardware resources, leading to a higher contention on shared hardware resources, which might degrade overall system performance [67, 37].

Previous works on thread placement focus on achieving better performance of the running workloads in the system. There are multiple works that use exploration-based approaches [129, 163, 125] to set the placement of the threads in a way that maximizes performance. Other works propose using prediction models based on machine learning. Sanchez et al. [140] propose a method driven by machine learning that predicts the best configuration of thread placement and data prefetching in a NUMA system to improve performance. Wang et al. [158] propose a machine learning model to select parallelism mapping for OpenMP workloads. Denoyelle et al. [39] propose a model to select the best thread placement based on the memory access patterns of the threads, among other features. These works focus on minimizing execution time only with thread placement.

In environments with multi-programmed workloads, some works show that the SMT level can be adjusted to improve fairness across applications [30, 29, 148, 20, 19, 21]. In addition, other works predict the IPC of the workloads running on an SMT processor and schedule serial applications on logic cores to boost the overall performance of the system [104, 143, 52, 54, 53].

Parallel workloads have also been targeted by previous works. Zhang et al. [168, 167] and Heirman et al. [67] propose a dynamic algorithm inside the OpenMP runtime system that selects the number of threads that maximize performance. Jia et al. [77] propose a machine learning model to predict the SMT level that maximizes performance in big data workloads, although their solution only considers a very small search space of 4 possible SMT levels.

2.3 Hardware Data Prefetching

Hardware data prefetching reduces memory latency by bringing data to the processor caches before it is needed, reducing stalls due to memory accesses. Most modern processors include multiple hardware data prefetch engines as it is a powerful technique to improve the overall performance of the system.

Applications with predictable (e.g., regular) memory access patterns and spatial locality significantly benefit from data prefetching. However, other workloads with more unpredictable (e.g., random) memory patterns do not benefit from prefetching as much, and under certain circumstances the prefetcher can even degrade performance and energy efficiency [90]. Inaccurate prefetches waste memory bandwidth and pollute the cache hierarchy; therefore, they can cause increased power consumption and decreased performance. Nevertheless, a correctly configured data prefetcher can speed up the execution time, save memory bandwidth and achieve significant reductions in power consumption [80, 79].

Ideally, a hardware data prefetcher brings the exact amount of data needed by the processor in a timely manner and without evicting other useful data that is already

present in the cache. In practice, different application behaviors require hardware data prefetchers to tune their aggressiveness during application execution. To this end, modern architectures expose prefetcher configuration parameters that the users can tune and adapt according to the requirements of the running workload. Some configurable prefetcher parameters are:

- **Degree**: Number of cache lines transferred in every prefetch request. Increasing this parameter can help improve performance by bringing more cache lines. This also leads to an increased prefetcher aggressiveness, which can cause evictions of useful data and memory bandwidth wastage.
- **Distance**: Number of cache lines ahead of the current memory address being accessed that will be prefetched. Increasing this parameter prefetches cache lines further than the offended cache line, which can improve performance by prefetching ahead and, therefore, reducing latency and possible late prefetches. Aggressively increasing the distance can also result in evictions of useful data in the cache and in a high demand of memory bandwidth.
- **Type of accesses:** Enable prefetching for loads, for stores, or for both. Enabling and disabling of prefetches can also consider other aspects of the memory accesses such as their stride, or whether if they hit or miss in the cache.

When measuring a prefetcher performance, we need to take into account whether the prefetcher is bringing in useful data and whether it does so in a timely fashion without evicting useful data. Therefore, the performance of prefetchers is measured using the following metrics:

- **Accuracy** measures the amount of prefetched cache lines that are later demanded by the application.
- **Timeliness** measures the amount of prefetched cache lines that arrive to the cache before the cache line is demanded by the application.
- **Pollution** measures the amount of cache lines that are demanded by the application and have been evicted from the cache to place prefetched cache lines.

Hardware prefetchers are attached to a single cache. Therefore, it is possible to have different types of prefetchers in different cache levels and have different configurations. Typically, complex data prefetchers offer more performance at the cost of more area and increased power consumption. Simpler data prefetchers occupy less area but they usually perform worse than a complex data prefetcher, due to being unable to recognize complex memory access patterns. Some of the prefetchers used in modern systems are:

- **Nextline Prefetcher** A simple prefetcher that detects sequential access patterns and prefetches the next consecutive cache line [142].
- **Stream Prefetcher** In this prefetching scheme, the hardware prefetches consecutive cache blocks after a short training period during which it observes memory access streams. A stream is a group of data references in a short period of time that frequently repeat and are stable [83].
- **Stride Prefetcher** In this scheme, the hardware prefetcher calculates the distance between memory addresses (or stride) from the same instructions. When the prefetcher is trained, upon a cache miss of an instruction that is recognized, the missing cache line and the cache line with the same requested address plus the stride are returned [57].
- **Neighbor Prefetcher** In this scheme, the hardware prefetcher brings to the cache the surrounding cache lines of the demanded cache line [89]. The surrounding cache lines to be prefetched are called the neighborhood, which is composed by different cache lines near to the missing cache block (how near or far are the cache lines depends on the defined size of the neighborhood). The neighborhood needs a training phase to work properly.
- **Correlation** (Global History Buffer prefetcher) In this scheme, the hardware keeps an ordered list of memory addresses generated by the same memory instruction. This information is used in a training phase to observe possible correlations. Those correlations are used to prefetch cache blocks [113].

2.3.1 Data Prefetching Knobs in Modern Systems

The data prefetching algorithm is usually fixed in the processor design and cannot by modified. However, modern architectures include mechanisms to configure the parameters of the prefetcher, in order that the user can tune it according to the workload characteristics by selecting the number of lines to bring ahead of time, prefetch data on load and/or store instructions, etc. In addition, many processors also offer instructions to let the programmer or the compiler do software prefetching, although these instructions need to be used consciously by the programmer because they have a non-negligible cost and can end up deteriorating performance. Overall, a correctly configured data prefetcher can speed up the execution time, save memory bandwidth and reduce power consumption significantly [80, 79].

Previous works propose hardware modifications of the prefetcher implementations to improve performance on multicore chips. Zhuang et al. propose a hardware modification to reduce cache pollution via filtering [169]. Wu et al. incorporates last-level cache information in order to prefetch data [164]. Bakshalipour et al. propose a new hardware data prefetcher to improve performance [13]. Heirman et al. [68]

track late prefetches in serial and parallel applications through hardware. Using this information, they tune the hardware prefetcher aggressiveness to reduce late prefetches and increase useful prefetches. Srinath et al. improve performance by adjusting the prefetching based on several metrics [144]. Ebrahimi et al. provide mechanisms in order to improve performance and fairness of shared resources with data prefetching in a multi-core processor scenario [45, 46, 44]. Nesbit et al. divide the memory address into equal-sized zones and detect patterns within each zone. Then, they adapt the prefetcher aggressiveness and the size of the zones [112].

On the software side, previous works only target serial applications or multi-programmed workloads [95, 72, 86, 87], but not parallel applications. Jimenez et al. detect phases of applications during their execution and change the data prefetcher configuration according to the overall demands of the applications running on the system [80, 79]. Similarly, Navarro et al. [111] and Hiebel et al. [69] propose different methods to control prefetching with different goals. Also, Chilimbi et al. make use of software prefetching to speed up applications [33]. Wang et al. use information at compile time to correctly set the data prefetcher aggressiveness [159].

Some works propose techniques based on machine learning to decide the prefetcher configuration. Rahman et al. [130] apply a model to enable or disable different prefetcher engines present in an Intel processor. Liao et al. [92] create a model to enable or disable those prefetcher engines in data center workloads. Li et al. [91] apply a machine learning model to predict the best settings for the data prefetcher in different parallel workloads.

2.4 Coordination of Multiple Hardware Knobs

Some works in the literature propose techniques to reconfigure multiple hardware structures of a processor in a coordinated manner.

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 [124]. 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 [76]. They sample statistics from every core of a multicore processor to calculate a subset of possible configurations. Then they test different configurations of this subset to find the best configuration for the execution until a new application phase begins.

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[17]. They build a model with multiple inputs that represent the past behavior of the application (L2 cache space, memory

bandwidth, and power budget). Then, at runtime, they gather those metrics at the end of an interval. Then, based on the model they decide the new hardware configuration for the next interval.

Sun et al. manage data prefetching and cache partitioning to reduce prefetch caused intercore interference on Intel multi-core processors without introducing additional hardware support [145]. Similarly, Sanchez et al. investigate the interactions between thread placement and data prefetching in an Intel system [140].

Pothukuchi et al. propose the management of multiple hardware knobs such as frequency, cache size, and number of ROB entries at a core level through a dedicated piece of hardware [126, 128].

2.5 Hardware Reconfiguration using Machine Learning Approaches

The amount of hardware knobs present in systems has steadily increased in the last years. As a consequence, the number of possible knob configurations has grown dramatically, up to a point where the search space is too large to be handled with traditional heuristics. This has created the need for using new scalable methods to coordinate the hardware knobs. A promising solution for a problem of this nature is using ML.

ML algorithms can be used to generate a predictor model. A ML model is built based on samples of data (or training data) to make predictions for new samples. There are different learning approaches to train a model depending on the application of the model: supervised training, unsupervised learning, semi-supervised learning, and reinforcement learning.

Supervised learning is a well known approach to construct the ML model. In supervised learning, the ML model is trained with samples, and each sample includes a label that represents the correct (or ground truth) output the ML model should produce. For instance, a ML model could predict the IPC (output) of an application from the memory bandwidth (feature). The difference between the output and the label (i.e. the error) is passed to a cost function. A typical cost function is the mean-squared error, which is the average squared error between the ML model output and the label for each trained sample.

This process of feeding the ML model with samples is called training. The training is an iterative process that aims at minimizing the results of the cost function. In each training step, the ML model is fed with a different samples. Then, the ML model output is compared to the label to obtain the error, and the cost function is calculated.

A ML model can be used to leverage the large amount of data, process it, and generate a hardware knob coordination that takes into account the multiple interactions between hardware knobs. Tarsa et al. propose a core architecture that selects to be in a high-performance or low-power mode based on the output of a ML model [147]. Lurbe et al. infer a model to set the best prefetcher per application in a multithreaded system [96]. Shi et al. apply ML to the cache replacement problem [141]. Denoyelle et al. use ML for data and thread placement in NUMA architectures [39]. Rahman improves data prefetching with ML on an Intel processor [130], while Li et al. use ML to improve data prefetching on a POWER8 processor [91].

2.6 Power Capping

DVFS provides a mechanism to adjust the voltage and the frequency of different parts of a chip. DVFS exposes different combinations of voltage and frequency in which the hardware can safely operate. DVFS is included in virtually any system, [34, 70, 149, 61, 116, 139, 136], making it the most used hardware knob to control the power consumption of a device. However, previous works show that it is possible to control the power consumption of CPUs using other hardware knobs such as power gating [153], core allocation [35, 134], or SMT levels [166].

To ensure heterogeneous systems with multiple discrete devices do not exceed their power cap, designers estimate the peak power consumption of the individual devices and add them up to get the peak power consumption of the system [55, 137]. However, modern computing systems are often restricted to consume less power than its peak for a variety of reasons, including safe operation modes when power delivery and thermal limits are reached, battery-saving modes in commodity and embedded systems, and economical reasons in servers and data centers. When these situations arise, a power cap is introduced in the system and the available power is distributed among the devices.

Current industrial systems with multiple devices use static power distribution schemes, i.e., for a given power cap, they set a fixed power budget for every device regardless of the characteristics of the workloads running on the devices. For example, the IBM OCC controller of the OpenPOWER architecture provides a mechanism to adjust the power budgets of the group of CPUs and the GPUs when a power cap is introduced. As shown in previous research [136], this mechanism can be used to implement different static power distributions.

In a static power distribution, once the controllers have assigned power budgets to all the devices, these can adjust their running frequency inside a range of available frequencies that honor their power budget. This decoupled scheme loses opportunities for optimization, as the power distribution among the devices does not consider

their utilization, limiting the range of available frequencies for the devices to maximize performance or efficiency.

Static power distributions can be outperformed by dynamic mechanisms that intelligently distribute the available power among the devices. Next sections discuss the approaches proposed in the literature to dynamically distribute power in different types of systems.

2.6.1 Homogeneous Chips

Previous research on power distribution has target homogeneous multicore processors. Isci et al. [74] propose to monitor the performance of the cores and apply DVFS to maximize the overall performance under a power cap. Winter et al. [161] present different scheduling and power management algorithms on a 256-core architecture. Adileh et al. [4, 3] schedule applications into out-of-order or in-order cores based on their performance and power consumption. Some works propose to coordinate the frequencies of the CPU and the memory to maximize performance [55, 109, 93], coordinate DVFS with other techniques such as power gating [153], core allocation [35, 134], and SMT levels [166], save energy [38] under a power cap, and manage power through a resource controller based on a market solution [62, 157] and a machine learning model [18].

2.6.2 Heterogeneous Systems with a Single GPU

Some research works have studied how to distribute power in heterogeneous systems with only one accelerator. These solutions are tailored to a single heterogeneous application, and their algorithms distribute power only when compute kernels start and finish their execution on the accelerator.

Majumdar et al. [98] predicts the best hardware configuration for GPU computation kernels at runtime to improve energy-efficiency by tracking recent execution history. Jiao et al. [78] coordinate the execution of concurrent kernels and DVFS to improve the energy efficiency of a system with a single GPU. Bailey et al. [12] coordinate the DVFS of a power capped system with a single GPU executing a single CPU-GPU application. Similarly, Komoda et al. [88] coordinate DVFS and task mapping in a system with a single GPU.

The main drawback of these solutions is that they do not apply to multiprogrammed workloads, since taking decisions at system level based on the combination of profile information of single applications leads to underperforming configurations. Also, these solutions need to profile the power and performance of the system running the kernels at all the possible frequencies, which requires a huge

amount of offline experiments or long online training periods. In addition, profiling-based solutions can present inaccuracies when the computational kernels have different behaviors in different executions or if the profiled executions are not representative of the real runs.

Sampling-based techniques are used by Indrani et al. to distribute the power in a heterogeneous chip with an integrated GPU to improve energy efficiency [122] and performance [121]. Their proposal only configures the power states of the CPUs among 4 possible preset values, and they rely on the controller to indirectly re-adjust the frequency of the GPU based on the power state of the CPU and the thermal coupling of both devices. Yet, the overall power budget of the chip is not taken into account. This is not applicable to discrete GPUs because they do not have thermal coupling effects. Pathania et al. [120] propose a sampling-based OS-level solution that manages the DVFS of a CPU and an integrated GPU to save power while meeting the performance requirements of 3D games.

2.6.3 Heterogeneous Systems with Multiple GPUs

Tangram [127] is the only previous work that distributes power on a heterogeneous system with multiple discrete devices. Tangram introduces a power management solution that uses a hierarchical organization of robust controllers with a common scalable interface, and it is demonstrated on a system with 2 CPUs and 1 GPU running a single CPU-GPU application. Tangram requires a significant amount of long latency communication between the devices and the Nelder-Mead search method that distributes power suffers from long training periods.

2.7 Runtime Systems and Shared Memory Programming Models

With the increasing number of cores, orchestrating the parallel execution of an application is becoming more difficult. Current parallel programming models leverage the usage of a runtime system to manage this complexity and to exploit the parallelism of multicore 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 [24, 51, 152].

Specifically, OpenMP [117] 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. These directives delimit the parallel regions, which are the parts of the source code that are executed in parallel.

Typical HPC applications consist of a set of phases that are iteratively executed [160, 108, 150, 64]. 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 taking 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 exploited to learn the best knob configuration in the first iterations, and apply this one during the rest of the execution. Also, parallel regions naturally delimit the different application phases; therefore, they provide a great opportunity to manage hardware knobs at intra-application granularity by setting the configuration that better suits the characteristics of each application phase.

2.7.1 Runtime-aware architectures

A Runtime-Aware Architecture [152] is proposed as an approach to improve performance and energy efficiency while minimizing the different constraints or walls in modern systems. In this architecture, hardware and software are tightly coordinated through a runtime layer. This runtime layer is responsible to use and coordinate multiple hardware knobs to further increase performance and energy efficiency. Coordinating multiple hardware knobs can lead to further performance than when multiple hardware knobs are used independently.

Both Garcia et al. [59] and Papaefstathiou et al. [119] propose using the runtime information about dependencies to prefetch data blocks. On the other hand, Manivannan et al. [100, 99] use the dependency and task information to predict dead blocks that can be substituted in the caches. Dimic et al. leverage a runtime to propose cache insertion policies based on the use of reference intervals and to compute reductions in the cache hierarchy [41, 40]. Alvarez et al. leverage compiler and runtime information to manage the coherence of scratchpad memories [7, 6, 9, 8]. Barredo et al. propose a hardware unit to join sparse predicated vector instructions into denser vectors [14]. Gomez et al. propose an efficient method to run Sparse Matrix-Vector multiplication (SpMV) on long vector architectures [60]. Castillo et al. uses a runtime to increase the clock frequency of cores that are running critical processes [26] and improves a runtime system for a better hardware-software coordination [25, 27]. Jaulmes et al. uses runtime information for reliability and fault tolerance [75].

Runtime-aware architectures show that performance can be increase when considering interactions between differents hardware knobs. Since future processors

are likely to become more resource-constrained and with a higher number of hardware knobs, runtime-aware architectures are an interesting area of research. Within this topic, the goal of the work in this thesis is to improve performance, power consumption, and energy-efficiency of real systems through hardware coordination at different levels.

Chapter 3

Methodology

This chapter describes the experimental methodology used in this Thesis. The first section explains the multiple experimental setups used. In this Thesis, several real systems are used to validate and evaluate our proposals. The second section describes the simulation infrastructure. The sections explains the baseline architecture modeled in the simulator and the software stack. The third section details the benchmarks with their characteristics for both the real systems and simulation evaluated in this Thesis. Finally, the fourth section introduces the metrics used to evaluate the proposals of this Thesis.

3.1 Real System Infrastructure

We evaluate our proposals on several IBM-based systems. Specifically, we use an IBM POWER8 and an IBM POWER9 system. Both have several architectural details in common that we introduce in this section.

The POWER processors used in this Thesis have SMT capabilities, meaning that each core can simultaneously run up to N threads, where N is the maximum SMT level. Yet, they also support running at lower SMT levels. For instance, a processor with a SMT level of 4 can run up to 4 threads within the same physical core but it also supports running 1 and 2 threads per core. The OS exposes a physical core as a group of 4 logic cores and, when the machine boots, it automatically sets the SMT level to 4. The SMT level is adjusted automatically by the hypervisor based on the utilization of the system. For example, when the system is in SMT4 level, the OS exposes 4 logic cores per each physical core. When just one of those logic cores is used, the system sets the SMT level to ST level automatically, making all the core hardware resources available to the application.

To force a desired SMT level, we need to specify the number of threads running in a physical core. This can be done manually by setting the desired number of threads of the application and pinning threads to physical cores accordingly. Also,

TABLE 3.1: Layout of the DSCR register. The register is 64-bit wide and 25 bits are mapped to different functions.

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

it can be done by disabling the logic cores through specific *online* registers exposed by the OS. In OpenMP, the required number of threads can be defined directly from the application code with specific calls to the runtime system.

POWER processors include a powerful and configurable data prefetcher that can be controlled at the core level by a special purpose register called Data Streams Control Register (DSCR) [65], which is exposed by the OS. The DSCR register is 64-bit wide where bits control different functions as shown in Table 3.1. The different functions (or fields) in DSCR offer a total of 2^{25} possible configurations. The most relevant fields for our experiments due to differences in execution time, energy, and power are the following ones:

- URG or urgency: How quickly the prefetch depth can be reached when prefetching data. A low URG leads to lower contention in the cache due to lower number of prefetchers realized. A high URG can degrade performance if data needs to be reused.
- 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.
- SSE: Enables data prefetching for store instructions.
- DPDF: Number of cache blocks that will be prefetched, from 1 cache block up to 7 cache blocks.

When a system boots, it sets the prefetcher to the default configuration: URG set to 4, LDS enabled, DPDF set to 4, and all the other options disabled.

DVFS is also available and configurable in both systems. The DVFS hardware knob is controlled at the physical core level through an exposed file by the OS. To set the frequencies of all the devices in the system, we enable the *userspace* policy to change the frequency of the CPUs to the desired value and to disable the OS to set new frequencies.

3.1.1 IBM POWER8

We evaluate our solutions on an IBM POWER8 based system (8335-GTA model) [103]. This system has an IBM POWER8 processor that runs at 3.49 GHz

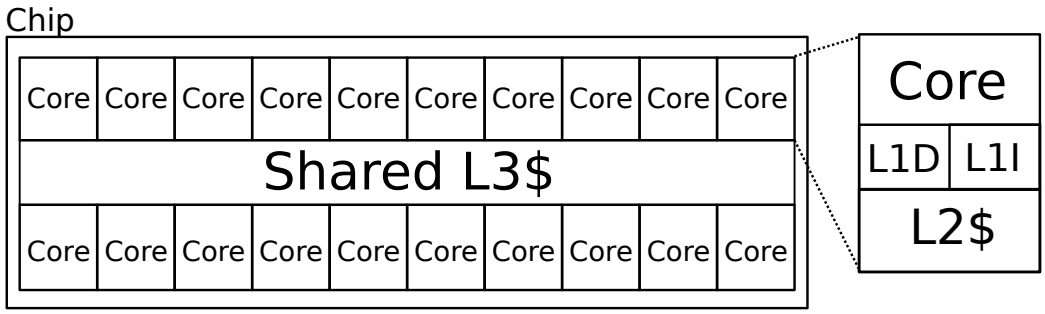


FIGURE 3.1: Architecture schematic of the POWER8.

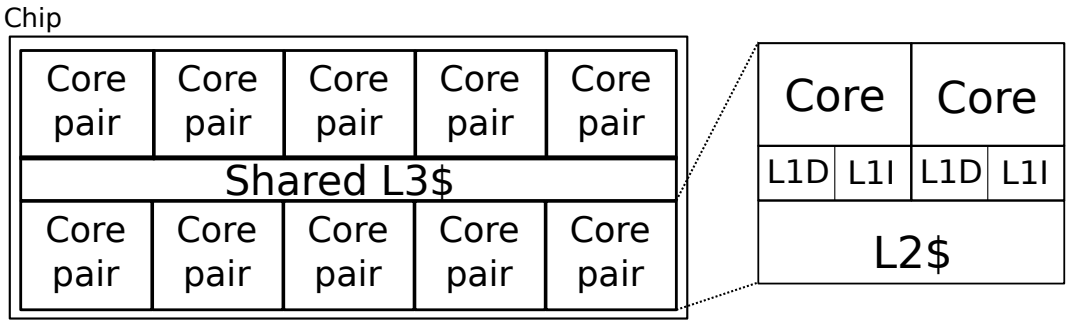


FIGURE 3.2: Architecture schematic of the POWER9.

with 512 GB of DDR3 CDIMM memory running at 1.6 GHz. The POWER8 processor in this system is packaged as a single-chip module with 20 cores. Each core has SMT8 capabilities and 64 KB L1 data and 32 KB L1 instruction caches, a 512 KB L2 cache and an 8 MB L3 cache with a total of shared on-chip L3 cache of 80MB (160MB for both sockets) as shown in Figure 3.1. The system runs Ubuntu 14.10 operating system with the kernel version 3.16.

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.

When running benchmarks in this system, we compile all the benchmarks with GCC version 4.9.3, which supports OpenMP 4.0.

3.1.2 IBM POWER9

We also evaluate an OpenPOWER architecture with a POWER9-based system (PowerNV 8335-GTH) with 2 CPUs and 4 GPUs distributed in 2 sockets. Each socket has an IBM POWER9 processor [139] that runs by default at 3.00GHz with 512GB of DDR4 DIMM memory at 2666MHz, and 2 NVIDIA Volta V100 GPUs [116] that run by default at 1.53GHz with 16GB of HBM2 memory. The POWER9 processor has 20 cores with SMT4 capabilities. Each core has 32KB L1 data and 32KB L1 instruction caches. Each pair of cores has a 512KB L2 cache and a 10MB L3 cache (with a total

TABLE 3.2: Cache parameters on our simulated cores.

Parameter	L1D	L1I	L2	L3
Size	32kB	32kB	256kB	2MB
Hit latency (cycles)	1	1	4	20
Associativy	4	2	16	16

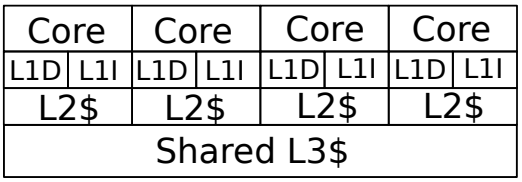


FIGURE 3.3: Architecture schematic of the simulated chip with 4 ARM Cortex-A53-like.

of shared on-chip L3 cache of 100MB) as shown in Figure 3.2. Each NVIDIA Volta V100 has 640 NVIDIA Tensor cores with a memory bandwidth up to 900GB/sec. The operating system is a Red Hat Enterprise Linux Server 7.5 with kernel 4.14.0.

The POWER9 processor has 43 frequency levels (from 3.00GHz to 2.30GHz). While the NVIDIA Volta GPUs can be set to run from 1.53GHz to 135MHz

In addition, our system has 2 sockets, and the system by default adjusts the running threads to be executed across both sockets and all cores to maximize bandwidth. To set the desired thread placement, the function *sched_setaffinity* available in Linux can be used to pin the threads to different sockets.

3.2 Simulation Infrastructure

In order to evaluate the effectiveness of the different prefetchers in a simulated environment, we use a customized version of gem5 [16] to simulate a 64-bit ARMv8 system. We report results while running in full-system mode with a 4-core configurations. These cores are ARM Cortex-A53-like. As shown in Figure 3.3, each core has private L1 and L2 exclusive caches, and a shared, inclusive L3 with the parameters specified in Table 3.2. The replacement policy for cache lines is least recently used (LRU). The system is configured such that prefetches occur on cache misses. The memory used is a DDR4 running at 2.4GHz, with 1 channel and a 16-byte bus. On the simulated system, we run a Linux kernel version 3.16 with a configured base page size of 64kB. We use MPICH3.2 [105] to run MPI benchmarks. Hardware-wise, the L1 data prefetcher is allowed to cross page boundaries if the accessed page is already translated in the translation lookaside buffer (TLB). Therefore, the accessed page is in memory.

3.3 Benchmarks

In this section, we describe the benchmarks use to prove the different proposals introduced in this Thesis. We always use benchmarks well known by the community.

Depending on the proposal, we use different benchmarks to stress and target the issue we want to solve. Therefore, the benchmarks executed can be categorized in terms of what proposal uses them:

- Coordination of multiple hardware knobs: these benchmarks are multi-threaded. They are described along with their inputs in Section 3.3.1.
- Hardware data prefetchers: these benchmarks are multi-threaded and memory intensive. They are described along with their inputs in Section 3.3.2.
- Power shifting for multiple independent devices: these benchmarks can be single or multi threaded and can be CPU-intensive, GPU-intensive, or memory intensive. They are described along with their inputs in Section 3.3.3.

3.3.1 Benchmarks for parallel workloads in real systems

To evaluate the effectiveness of our proposals when coordinating multiple hardware knobs, we use a wide set of benchmarks from the suites NPB [82] with the class D inputs and SPEC OMP 2012 [107] with the reference input.

The NPB suite is composed of 6 kernels and 3 pseudo-applications, which are derived from computational fluid dynamics (CFD). The SPEC OMP 2012 suite contains 14 applications from CFD to image modeling. They are focused on compute intensive performance. All SPEC OMP benchmarks are evaluated except `botsspar` and `smithwa`, as these two benchmarks did not pass SPEC's validation tools in our environment. All the benchmarks are parallelized with OpenMP and written in C, C++ or Fortran.

Benchmarks are executed accordingly to their thread placement and SMT level and pinned to them to avoid thread migration. We pin the different threads with the environment variable *OMP_PLACES* and *sched_setaffinity* function in Linux. Benchmarks are executed in isolation until completion.

3.3.2 Benchmarks for parallel workloads in simulation

In order to evaluate the effectiveness of the different prefetchers implemented, we use benchmarks from different suites: Mantevo [101], HPC Challenge [97], Proxy applications [94], Trinity benchmarks [151] and the High Performance Conjugate Gradients (HPCG) benchmark [43]. These benchmarks are high performance computing-oriented benchmarks and parallelized using OpenMP or MPI.

Brief descriptions, including a list of parameters can be found in Table 3.3.

	Description	Input	Heap Usage (MB)
CoMD	Co-designed Molecular Dynamics a classical molecular dynamics proxy application	20 -N 20 -T 4000	70
DGEMM	Double precision real matrix-matrix multiplication	5000	10
FTT	One-dimensional Discrete Fourier Transform	5000	35
PTRANS	Parallel matrix transpose	5000	65
STREAM	Sustainable memory bandwidth	5000	55
HPCG	High Performance Conjugate Gradient: preconditioned Conjugate Gradient method	24	55
mcb	Monte Carlo Benchmark: a simple heuristic transport equation using a Monte Carlo technique	320000	12
miniFE	Implicit Finite Elements: a proxy application for unstructured implicit finite element codes	860	95
pathfinder	Signature-search mini-application	medlarge1	15

TABLE 3.3: Input and heap usage for the benchmarks used in the evaluation. Heap usage is measured on a physical 64-bit ARMv8 machine, using Valgrind [114]. Total memory footprint used exceeds the total cache size. gem5 reports for all benchmarks a high heap usage (>95%)

3.3.3 Benchmarks for system level in real systems

TABLE 3.4: List of CPU (left) and GPU (right) benchmarks.

Name	Suite	Power	Name	Suite	Power
DAXPY	-	High	Tensorflow	-	High
BT	NAS	High	DGEMM	-	High
EP	NAS	High	Particlefilter	Rodinia	High
FT	NAS	High	Heartwall	Rodinia	Low
MG	NAS	High	Kmeans	Rodinia	Low
UA	NAS	High	Myocyte	Rodinia	Low
CG	NAS	Low	Srad (v1)	Rodinia	Low
LU	NAS	Low	Srad (v2)	Rodinia	Low
SP	NAS	Low	Quicksilver	CORAL	Low
Kripke	CORAL	Low	QMCPack	CORAL	Low
graph500	CORAL	Low	Lulesh	CORAL	Low

When coordinating the power cap of multiple independent devices in a system, we use the benchmarks shown in Table 3.4. Table 3.4 shows the CPU and the GPU benchmarks used in our evaluation in the left and right columns, respectively, and their average power consumption (high/low).

The benchmarks belong to the NPB [82], CORAL [36], and Rodinia [31] suites. For the NPB suite, we use the input class C. When running benchmarks from the CORAL suite, we use the recommended input for a single node. And, for the Rodinia suite, we run with the reference input with an increased number of iterations

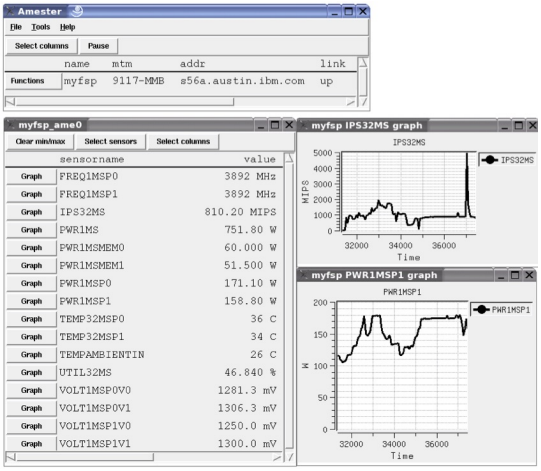


FIGURE 3.4: Visual interface of AMESTER.

to equal the execution time of all the benchmarks.

We also train an Inception v3 neural network [146] with the ImageNet data set [138] in Tensorflow [1], and use a DAXPY kernel for the CPU, and a DGEMM kernel for the GPU. The execution time of all benchmarks is 200s to 375s running in isolation with no power cap.

3.4 Metrics

The following metrics are used across different proposals in this Thesis.

We report speedup in execution time, power consumption and energy-delay product (EDP) for all the benchmarks.

In order to read execution time, we measure the wall time for the entire application or the parallel region in case we only are measuring the parallel region. This wall time includes the time for thread synchronization, which it is fair when measuring time in parallel applications.

We also gather multiple performance counters in order to drive our proposals. These performance counters are collected using perf [102].

To read power consumption in the POWER8, we use AMESTER (Automated Measurement of Systems for Energy and Temperature Reporting) [56] to measure the power consumption of the processor and memory chips (its interface is shown in Figure 3.4). The tool remotely collects power, thermal and performance metrics from the system using the Flexible Service Processor (FSP). The FSP allows reading different sensors from the system without using any of the processing cycles of the system. Therefore, it has no impact on the performance of the running benchmarks. We report the average power consumption for the total execution and energy-delay product (EDP). Power consumption results do not include the idle power of the system to put more emphasis on active power consumption savings. When reporting

EDP, we report energy (taking idle power of the system into account) multiplied by execution time.

When reading power consumption in the POWER9, we do in-band readings from Linux to the OCC [136] for the CPUs and with the NVIDIA Management Library (NVML) [115] for the GPUs. These in-band readings simplify the process of reading the power consumption of the system.

Chapter 4

Exploration for Hardware Knob Reconfiguration

4.1 Introduction

Every new generation of processors is increasing the number of cores and the number of threads that can run within the same core (SMT). As a result, processor shared resources might experience contention, which might lead to performance degradation. Processors have several hardware knobs to prevent performance degradation by adapting its behavior to workloads demands, such as the SMT, DVFS levels, the 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 [28, 20, 168, 167, 80]. This independent actuation can lead to conflicting decisions that jeopardize system power-performance efficiency [153]. 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.

In this chapter, we present a detailed power/performance characterization of an IBM POWER8 and OpenPower system when running parallel applications. We demonstrate that the best hardware knob configuration differs depending on the end goal. We propose libPRISM[1], an infrastructure for shared memory parallel programming models that transparently configures the different hardware knobs available in the architecture. During execution time, libPRISM discovers the best hardware

[1] libPRISM code available at: https://github.com/criort/libPRISM

configuration for different fine-grained regions of the application without user intervention and without modifying the original source code of the application.

4.2 The Need for Hardware Knob Coordination

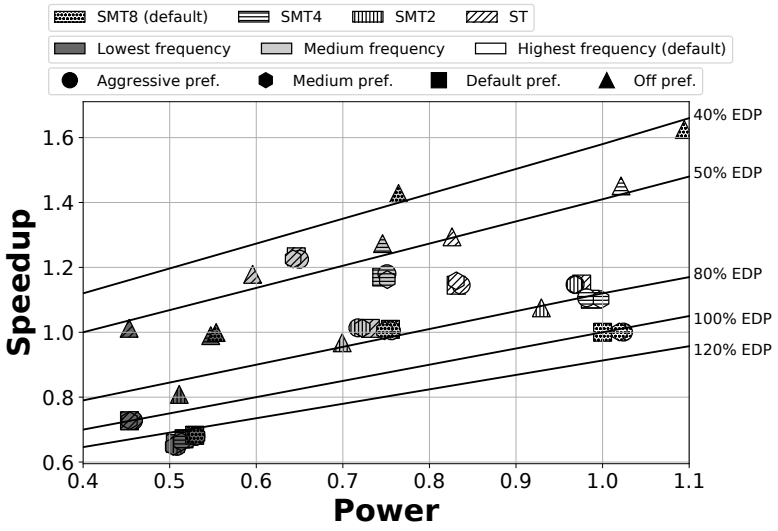


FIGURE 4.1: CG behavior under different hardware knob configurations. The Y and X axis show speedup and power consumption with respect to the default hardware configuration, respectively. Energy-Delay Product (EDP) normalized with respect to the default hardware configuration is represented with lines. For power and EDP, lower is better. Default configuration is SMT8, default data prefetcher, and highest frequency.

To illustrate the need for a coordinated adaptive system, Figure 4.1 shows the performance, average power consumption, and Energy-Delay Product (EDP) lines of the CG benchmark from the NAS Parallel Benchmarks (NPB)[81] suite with different hardware knob configurations 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 of the application has a higher penalty with this metric.

In Figure 4.1, configurations 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 provide 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 claim 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.

4.3 libPRISM

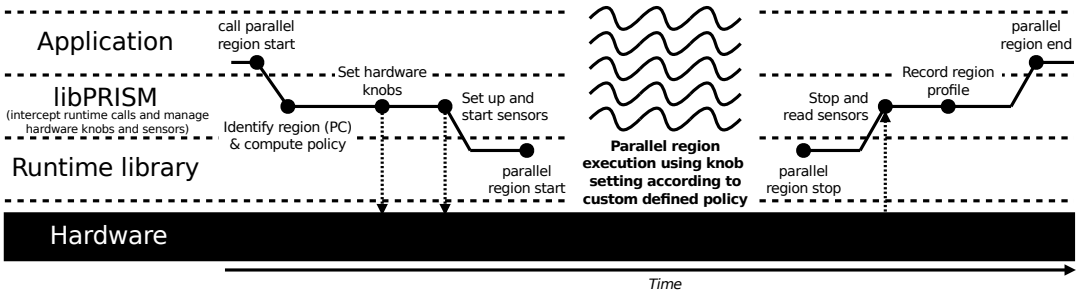


FIGURE 4.2: libPRISM execution stack and work flow.

libPRISM is located on top of the runtime system, as shown in Figure 4.2, 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 component is used to gather data from different sensors of the system such as 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 4.2 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 itself. In this scenario, the application executes as usual without being aware that libPRISM is dynamically adapting the hardware resources based on a custom defined policy.

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 4.2.

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 4.2).

For every parallel region that is executed, the policy records a performance profile under different hardware knob configurations (*Set up and start sensors* and *Stop and read sensors* in Figure 4.2). 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 hardware knob configuration for each parallel region. To that end, 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, in the following iterations it tracks the behavior of a parallel region and, in case the behavior changes, it starts the hardware knob configuration optimization phase again.

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.

4.4 libPRISM policies

Policies leverage the gathered data by libPRISM to find the optimal hardware 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.

We propose a novel policy that minimizes a specified metric. This policy is implemented as a generic policy to allow its extension with hardware knobs and metrics (execution time, EDP, or power consumption). 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 hardware 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 conver the best hardware knob configuration faster than longer vectors, while longer vectors can have more fine-grained configurations than short vectors.

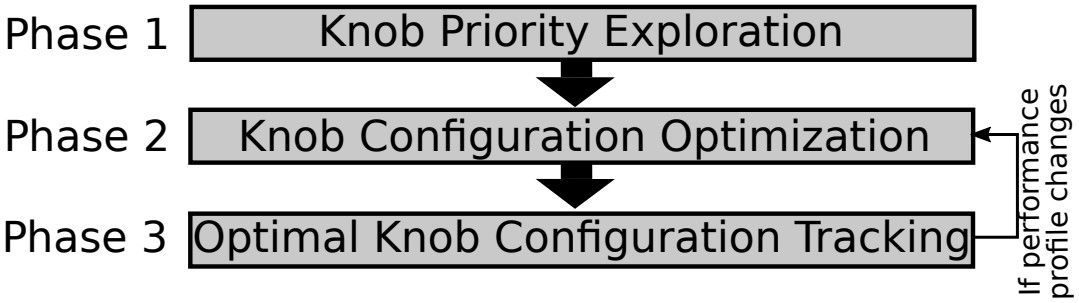


FIGURE 4.3: Phases of the generic policy.

In order to find the best hardware knob configuration per parallel region in the executed application, our policy goes through the three phases shown in Figure 4.3.

The first phase is the knob priority exploration shown in Listing 4.1, 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 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 4.1).

```
1// Call to parallel_region_begin intercepted
2function 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 4.1: Knob priority exploration phase (phase 1) of the proposed generic policy.

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[2]. 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 determines the best order to explore the hardware knobs, it enters in the hardware knob configuration optimization phase (shown Figure 4.3 and its pseudocode in Listing 4.2). 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 generic search algorithm is used to explore different hardware knobs individually. The algorithm in this phase tunes first the hardware resources that have higher impact on the final performance of the application based on the results obtained in the previous phase (line 18 from Listing 4.1 and line 21 from Listing 4.2.) Our heuristic-based search converges faster to a hardware knob configuration that provides a better performance by taking into account inter-knob

[2]Turbo boost is not enabled in our experiments

```
1// Call to parallel_region_begin intercepted
2function 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 */
16function 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 4.2: Hardware knob configuration optimization phase (phase 2) of the proposed generic policy.

effects than exploring all the search space. This reduces the overheads associated with 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 given 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 when the parallel region finishes. This measurement is performed 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 4.2). 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 analysis in Section 4.5.4.

The 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 4.2). Notice that this mechanism avoids achieving worse performance than the default hardware knob configuration of the system. Then, the policy continues the hardware knob configuration optimization phase with the next hardware knob to configure (Line 21 in Listing 4.2).

The maximum number of iterations for the hardware knob configuration optimization phase without taking into account re-entering in the phase with N as the number of hardware knobs to configure is:

$$\sum_{i=1}^{N} length\_vector\_HW\_knob_i$$

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, our observations show that less than 10 iterations (6.1 iterations 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 iterations 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 hardware 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.

```
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 4.3: Optimal hardware knob configuration tracking phase (phase 3) of the proposed generic policy.

After the hardware knob configuration optimization phase, the policy identifies a competitive performing hardware knob configuration for a particular parallel region and reaches the optimal hardware knob configuration tracking phase where tracks the performance profile of each parallel region as shown in Figure 4.3. The pseudocode of this phase is shown in Listing 4.3. Every time the parallel region is executed, the hardware knobs are set to the identified best found performing hardware 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 hardware knob configuration optimization phase.

If the last measured performance of a parallel region differs more than a configurable *threshold* (Line 3 in Listing 4.3) from the average performance of that parallel region, the hardware 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 4.3). This threshold (shared across all applications) 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 4.1 shows the policies derived from the different configurations that are evaluated in this work. For each policy, we show the possible hardware 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.

TABLE 4.1: Summary of policies used in this work. SMT can be configured as SMT8, SMT4, SMT2, or ST. Prefetcher can be set to the mostaggressive (3), aggressive (2), default (1), or disabled (0). DVFS is explored in steps of 0.06 GHz.

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

4.4.1 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 4.1 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.

4.4.2 MINEDP Policy

The MINEDP policy seeks to minimize the EDP, i.e. maximize speedup 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.

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 4.1 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 hardware 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.

4.4.3 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 knob configurations if execution time is improved. Therefore, 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 for the iterations of a parallel region 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 4.2). 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 4.1 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.

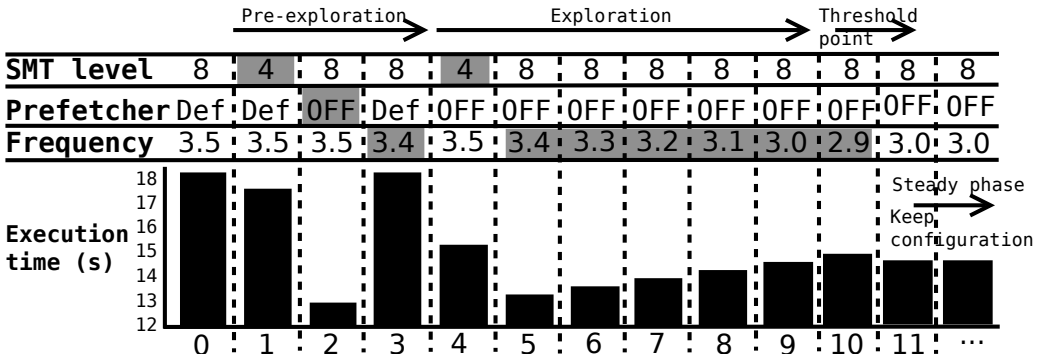


FIGURE 4.4: MINPOWER policy with a 10% threshold in libPRISM to select a competitive performing configuration for SMT level, data prefetcher and DVFS for the CG application. The X-axis shows the iterations of the same parallel region and the Y-axis the execution time for that given iteration. Repetitions is set to 1 (algorithm shown in Listing 4.2).

4.4.4 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 4.4.3.

Figure 4.4 shows how the hardware knob configuration optimization phase (shown in Listing 4.2) 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 hardware knob priority exploration phase. The hardware knob priority exploration phase (shown in Listing 4.1) is realized from iteration 1 to iteration 3, which detects what are the hardware 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 hardware 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 hardware knob configuration tracking phase (shown in Listing 4.3).

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 the hardware knobs in order to consider the interaction, as shown in Lines 22 and 23 in Listing 4.2.

When an important change in performance during the optimal hardware knob configuration tracking phase happens, the MINPOWER policy starts again the hardware 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%).

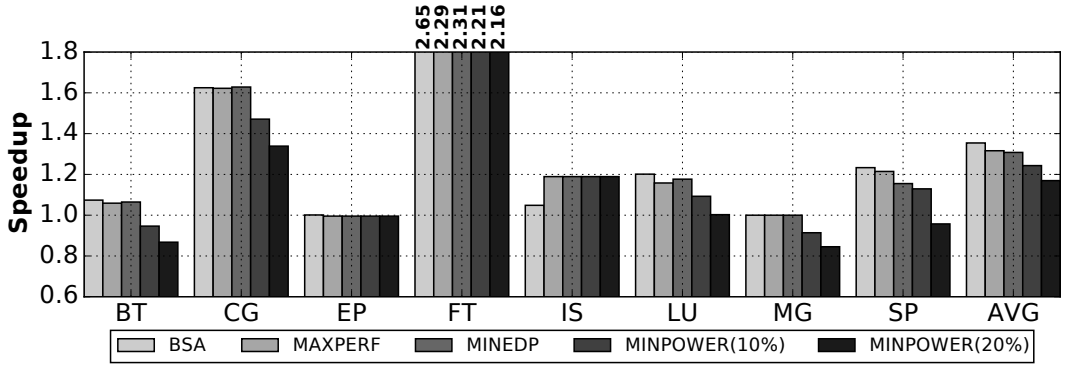
4.5 Evaluation

In this section we evaluate libPRISM in a POWER8 system[3]. We use libPRISM to coordinate the SMT level, data prefetcher, and DVFS knobs.

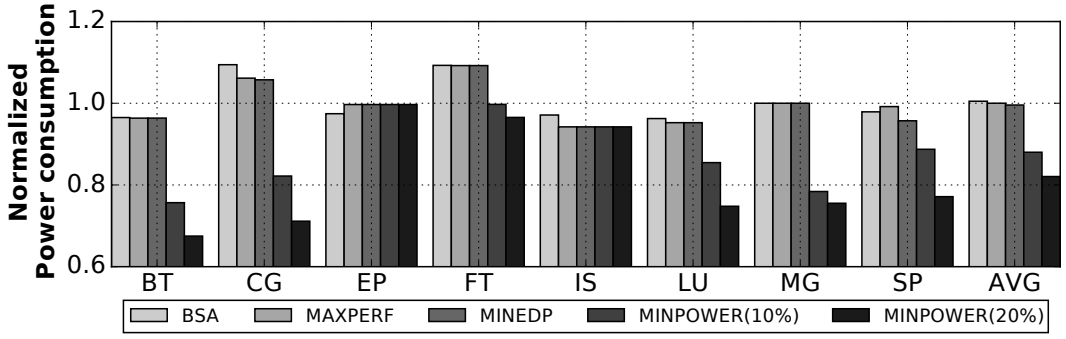
As presented in Chapter 3.1.1, the POWER8 system has 4 SMT levels: 8, 4, 2, and ST. In terms of data prefetching, the most relevant fields of the DSCR register are the following ones:

- 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.
- SSE: Enables data prefetching for store instructions.
- DPDF: Number of cache blocks that will be prefetched, from 1 cache block up to 7 cache blocks.

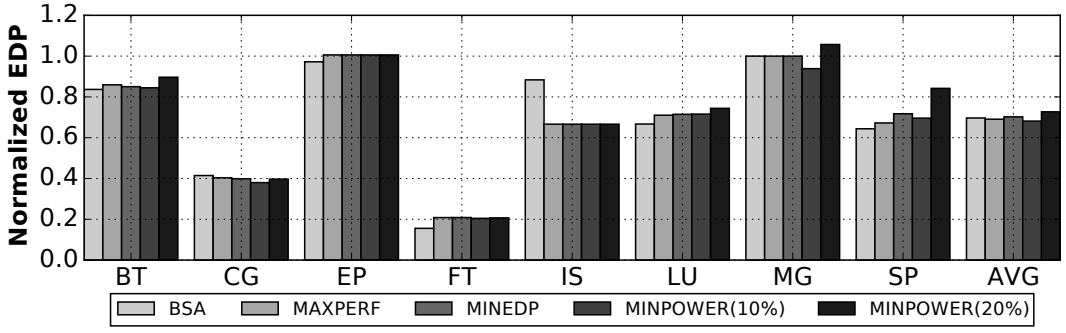
[3]The experimental setup is detailed in Section 3.1.1.



(A) Execution time



(B) Power consumption



(C) Energy-Delay Product (EDP)

FIGURE 4.5: Results with respect to the 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.

Therefore, we select 4 possible different configurations: default, OFF, medium, and aggressive configurations. The details of the DSCR configuration is shown in Table 4.2.

In our experimental setup, we observe that the aggressive prefetcher configuration performs better or equal than the medium configuration in most of the cases.

TABLE 4.2: The aggressiveness levels considered in libPRISM with their corresponding configurations for the DSCR.

Aggressiveness level	LDS	SNSE	DPDF
OFF	Disabled	Disabled	Not considered
Default	Enabled	Disabled	4
Medium	Enabled	Disabled	7
Aggressive	Enabled	Enabled	4

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.

TABLE 4.3: 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	2.50	2.80	3.10	3.30	3.49
Voltage (normalized)	0.80	0.86	0.89	0.94	0.97	1.00

In order to explain the behavior when using the DVFS knob, we run a maximum power stressmark [15] to measure the upper voltage limit associated with each frequency. Table 4.3 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.

In this section we evaluate the behavior of different policies:

- Best static per application (BSA): Best performing hardware configuration found for each application after an exhaustive 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.

- 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 configuration.
- 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 configuration.

Figure 4.5a 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 4.5b shows the power consumption normalized to the default hardware configuration for the BSA configuration and all our policies on top of libPRISM. Figure 4.5c 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 will comment these results individually for each of our policies: MAXPERF, MINEDP, and MINPOWER.

4.5.1 MAXPERF Policy

4.5.1.1 Performance

Figure 4.5a 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 4.6 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.

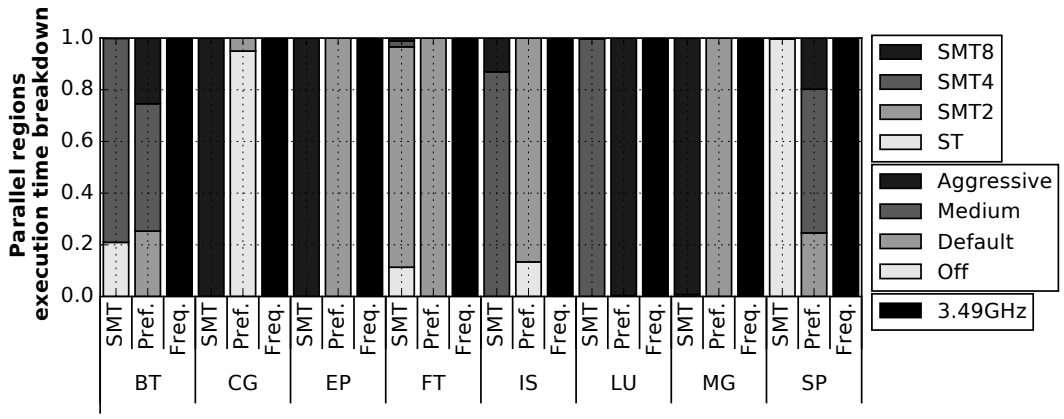


FIGURE 4.6: Final hardware knob configuration for the different parallel regions when running with libPRISM and MAXPERF policy.

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 4.6, 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 hardware knobs for the time consuming parallel regions; therefore, 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.

4.5.1.2 Energy Efficiency

Next, we discuss the energy efficiency results obtained with libPRISM using the MAXPERF policy. Figure 4.5b 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 4.5c 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 4.6.

4.5.2 MINEDP Policy

4.5.2.1 Performance

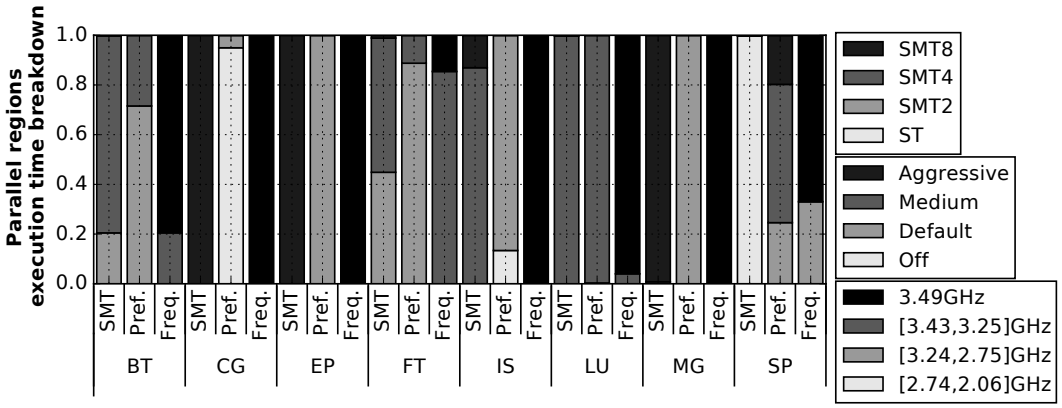


FIGURE 4.7: Final hardware knob configuration for the different parallel regions when running with libPRISM and MINEDP policy.

In Figure 4.5a we can see that the MINEDP policy achieves the similar performance as BSA and MAXPERF (30.8% on average). Yet, Figure 4.7 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 4.5a shows that the configuration 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 4.5.3, CG can achieve a lower EDP and power consumption if a higher performance degradation is allowed.

4.5.2.2 Energy Efficiency

As we can see in Figure 4.5b, 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 4.5c), 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 4.3, 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.

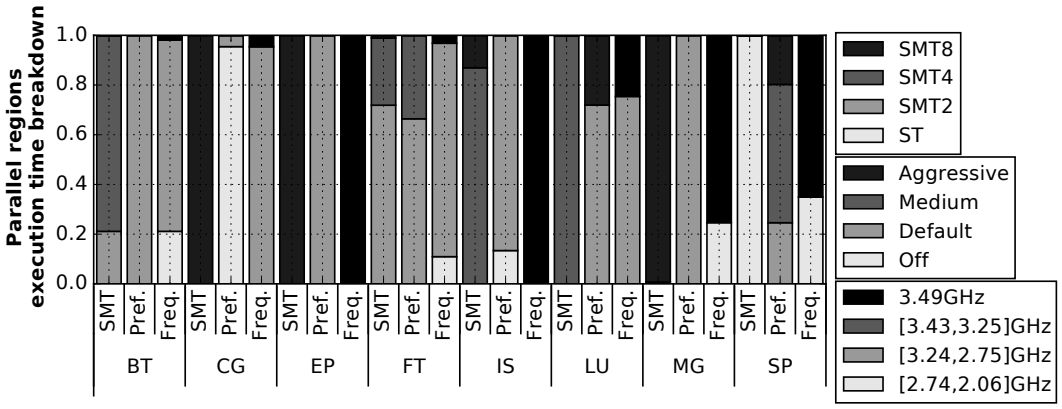
4.5.3 MINPOWER Policy

4.5.3.1 Performance

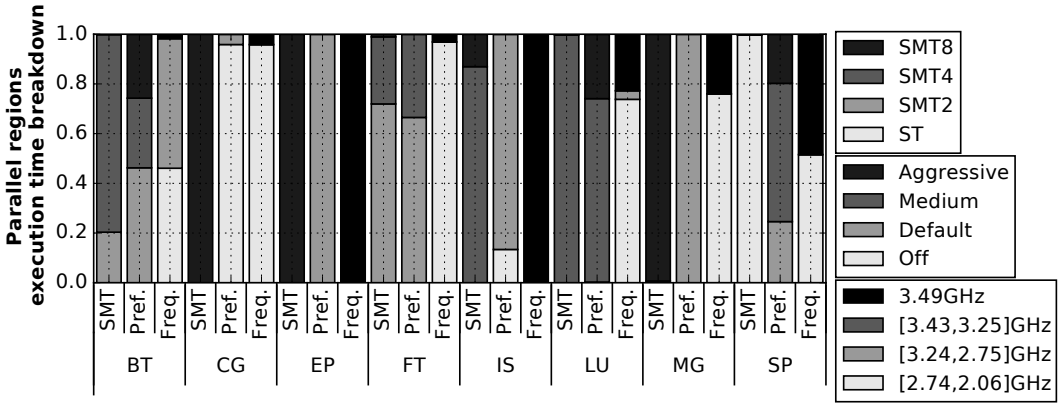
In Figure 4.5a, we show two configurations with different thresholds of the MINPOWER policy (10% and 20% maximum execution time degradation).

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



(A) 10% Threshold



(B) 20% Threshold

FIGURE 4.8: Final hardware knob configuration for the different parallel regions when running with libPRISM and MINPOWER policy with thresholds of 10% and 20%.

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 4.8a and 4.8b, 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.

4.5.3.2 Energy Efficiency

The MINPOWER policy is able to reduce power consumption as we can see in Figure 4.5b. 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. In the

case of CG, the MINPOWER policy 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, in some cases such as FT or MG, increasing the execution time threshold does not achieve the same power consumption reduction, even as seen in Figures 4.8a and 4.8b.

Figure 4.5c 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).

4.5.4 Overhead Analysis

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

libPRISM overheads are mainly introduced by reading the different sensors, compute the selected policy, and configuring the different hardware knobs at a parallel region level:

- Reading performance: as mentioned earlier, we use *perf* to read several performance counters such as instructions and cycles.
- Reading power consumption: AMESTER updates the power consumption every 250 microseconds and it is read at the end of a parallel region.
- Reconfiguring SMT, prefetcher, and DVFS knobs: as explained in Section 3.1, libPRISM needs to modify several registers exposed to the OS.
- 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 4.9. 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 thesis have thousands of short parallel

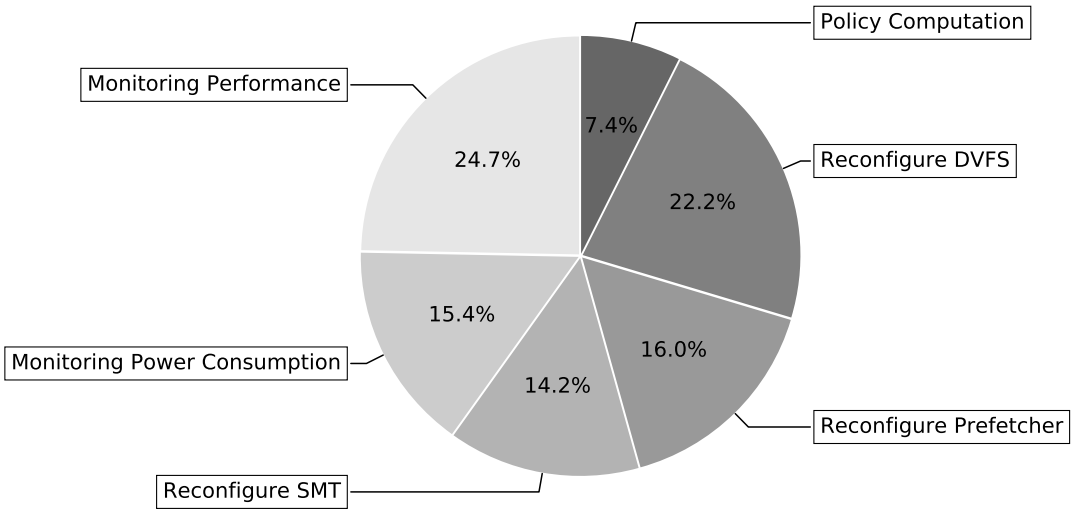


FIGURE 4.9: Contribution to the total overhead of a single parallel region execution of all the libPRISM components.

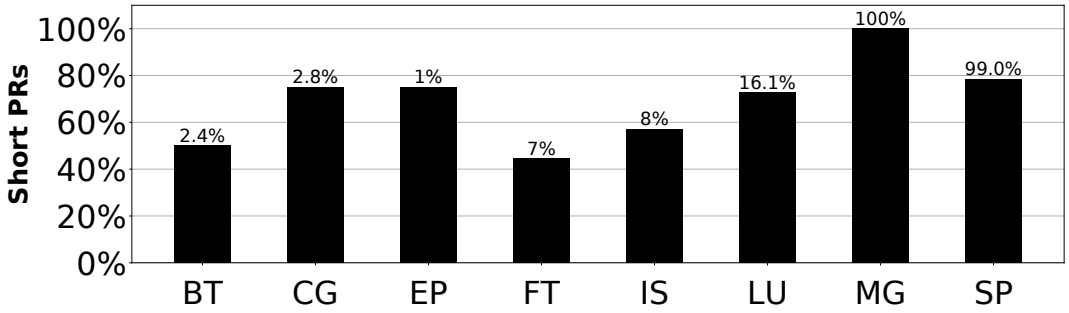


FIGURE 4.10: Percentage of short parallel regions (execution time of a single iteration is shorter than 1 second) of the benchmarks from NPB suite. Percentage on top of each bar represents the total time spent in short parallel regions with respect to the total time spent in parallel regions.

regions (execution time of a single iteration is shorter than a second) as shown in Figure 4.10 , 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 4.4 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.

TABLE 4.4: 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.

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

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.

Policies described in this chapter implement a greedy search for the decision algorithm. This decision algorithm achieves the best hardware knob configuration while minimizing the policy computation overhead. Figure 4.9 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.

4.5.5 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 [4]. 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 [73], and the power-performance efficiency of DVFS can change depending on the processor implementation.

4.6 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 chapter 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, a power consumption reduction and an energy-delay product reduction when compared to the default static system configuration with our proposed policies.

[4] libPRISM source code for x86 architectures is also available at https://github.com/criort/libPRISM/tree/x86

Chapter 5

Machine Learning for Hardware Knob Reconfiguration

5.1 Introduction

The main limitation of exploration-based techniques is that they incur increasingly high overheads and require longer training periods as the search space grows. The search space of possible knob configurations is already very large in current processors, and it is expected to be even larger in the future, since the number of hardware knobs and the amount of configuration options for each knob are increasing in every new generation of HPC processors.

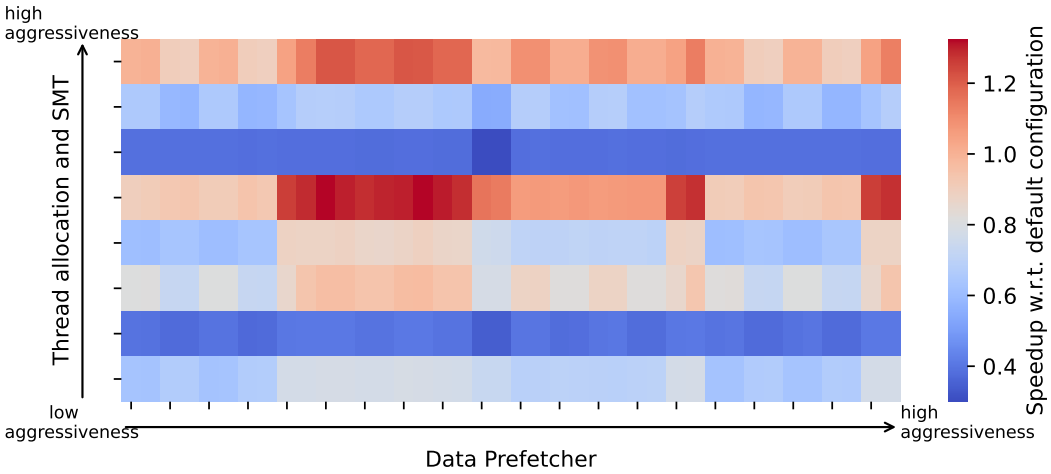


FIGURE 5.1: Speedup of `MD` when running with different knob configurations with respect to the default knob configuration. The aggressiveness of the Y-axis represents the SMT levels of the cores, the number of cores within a socket, and the number of sockets used to run the parallel application.

In a dual socket POWER9-based system, the hardware knobs to control the thread placement, the SMT level, and the data prefetcher combine for a total of

320 possible hardware knob configurations. Figure 5.1 shows the speedup of the MD benchmark from the SPEC OMP 2012 suite when using all the possible knob configurations with respect to the default knob configuration. The axis shows the hardware configuration used to run MD: the Y-axis represents the aggressiveness in terms of number of sockets and SMT level (i.e. higher aggressiveness implies a higher number of sockets or SMT level); and the X-axis represents the aggressiveness of the data prefetcher. The default knob configuration of the system is set to a high aggressiveness for thread allocation and SMT (2 sockets and the highest SMT level) and a medium aggressiveneess for the data prefetcher.

As shown in Figure 5.1, the performance of the MD benchmark is heavily influenced by the knob configuration, and the coordination of the hardware knobs is of paramount importance. In this benchmark, the best performance is achieved using a mid-to-low prefetcher aggressiveness and a medium thread allocation and SMT aggressiveness. However, the same mid-to-low prefetcher aggressiveness degrades performance when combined with other thread allocation and SMT aggressiveness configurations. This also poses a great challenge for exploration-based techniques that use heuristics to avoid the impracticality of exploring all configurations, since they can easily fall into local maximums. All together, exploration-based techniques are not able to find the best knob configuration when the search space is very large and the hardware knobs interact between themselves. Instead, ML techniques have great potential to solve problems of this nature, as they have proven to be an effective solution for decision-making problems with large search spaces, both in terms of accuracy and speed.

This chapter presents Machine Learning for Hardware Knobs (MARK), a technique that dynamically adapts the hardware knobs through a prediction model based on Machine Learning (ML) with the aim of minimizing execution time, power consumption, or Energy-Delay Product (EDP). MARK uses library interposition to identify the parallel regions of parallel applications. During the execution of the program, MARK builds a profile of the parallel regions, it predicts the best knob configuration for them using a neural network model, and re-configures the hardware knobs accordingly. With this approach, MARK avoids the limitations and caveats of exploration-based techniques such as the limited scalability and the overheads they incur when many knobs are considered, algorithm specific search for different metrics, and portability across systems. Instead, MARK uses fast and scalable inferences of the neural network to make the predictions, it considers interactions between hardware knobs, it is completely transparent to the user, and it does not require modifications in the source code of the applications.

TABLE 5.1: Performance Counters measured in order to build the Cycles Per Instruction (CPI) Stack in our POWER9 based system. NTC is the Next-To-Complete instruction

Performance Counter Group	Description	Measured Performance Counters
PM ICT NOSLOT	Cycles that no instruction is available to execute (Branch miss predictions)	IC_MISS, NOSLOT_BR_MPRED, NOSLOT_BR_MPRED_ICMISS, NOSLOT_DISP_HELD
PM ISSUE HOLD	Cycles that the NTC instruction is held in the issue	ISSUE_HELD_DARQ_FULL, ISSUE_HELD_ARB, ISSUE_HELD_OTHER
PM CMPLU STALL	Cycles that instructions are stalled in execution (Stalls due to execution units and cache misses)	BRU, FXU, DP, DFU, PM, CRYPTO, VFX, VDP, LRQ_FULL, PM_CMPLU_SRQ_FULL, LSAQ_ARB, ERAT_MISS, EMQ_FULL, LMQ_FULL, ST_FWD, LHS, LSU_MFSPR, LARX, LRQ_OTHER, DMISS_L2L3, DMISS_L3MISS, LOAD_FINISH, STORE_DATA, LWSYNC, HWSYNC, EIEIO, STCX, SLB, TEND, PASTE, TLBIE, STORE_PIPE_ARB, STORE_FIN_ARB, STORE_FINISH, LSU_FIN, NTC_FLUSH, NTC_DISP_FIN
PM CMPLU THRD CYC	Cycles that instructions are stalled in execution due to the actions of a different thread on the same core (Exceptions and synchronization between threads)	EXCEPTION, ANY_SYNC, SYNC_PMU_INT, SPEC_FINISH, FLUSH_ANY_THREAD, LSU_FLUSH_NEXT, NESTED_TBEGIN, NESTED_TEND, MTFPSCR, OTHER_CMPL
PM 1PLUS PPC CMP	Completed instructions	PM_1PLUS_PPC_CMP

5.2 Machine Learning for Hardware Knobs

This section presents MARK, a technique based on ML that reconfigures multiple hardware knobs during the execution of parallel programs to minimize execution time, power consumption, or EDP.

At execution time, every time a parallel region is executed, MARK is invoked to obtain the best knob configuration for it. For each parallel region of the program, MARK builds a profile with its Cycles Per Instruction (CPI) stack and power consumption. When a parallel region is about to be executed, MARK predicts the best knob configuration for the profile of that parallel region by inferring a machine learning based predictor model, and the knob configuration is applied by setting the hardware knobs accordingly. The following sections explain in detail the different components and steps performed in MARK.

5.2.1 Building a Profile

MARK builds a profile with the CPI stack and the power consumption of all the parallel regions of the program. Modern processors allow to build up a CPI stack [50] to breakdown the cycles a processor is stalled processing or completing an instruction. The CPI stack captures and identifies the behavior of an application and can be leveraged to set the multiple hardware knobs to reduce the number of stalled cycles.

A complete CPI stack is composed of multiple performance counters. Table 5.1 shows all the performance counters that are captured by MARK to build the CPI stacks in our experimental setup. Different performance counters from the CPI stack measure the different causes of the stalls that a thread suffers. For instance, *PM_CMPLU_STALL_CYC* measures stalls due to executions units, cache, or memory. On the other hand, *PM_CMPLU_THRD_CYC* measures stalls due to SMT. MARK reads the complete CPI stack with a single execution of a parallel region. Note that, depending on the system, this approach could lead to low accuracy when reading the performance counters. We further discuss this in Section 5.3.2.

MARK builds the profile of a parallel region the first time it is executed. Then, in the subsequent instances of the parallel region, the profile is used to predict its best

knob configuration. This method minimizes the exploration time of a parallel region to a single execution.

5.2.2 ML Predictor Model

Every time the application enters a parallel region, MARK predicts the best knob configuration for the execution of the parallel region. The predictions made by MARK are based on a knob configuration predictor built offline. ML is needed due to the large search space in current systems, up to 320 possible knob configurations in our setup that considers the thread placement, SMT level, and data prefetcher knobs.

To build a ML model, we go through a process for gathering data, curating data, and training a ML model. To gather the data we use for training, we run a set of benchmarks with all the possible knob configurations, and then we use the results of the experiments to train the model. Figure 5.2 shows the details of this process.

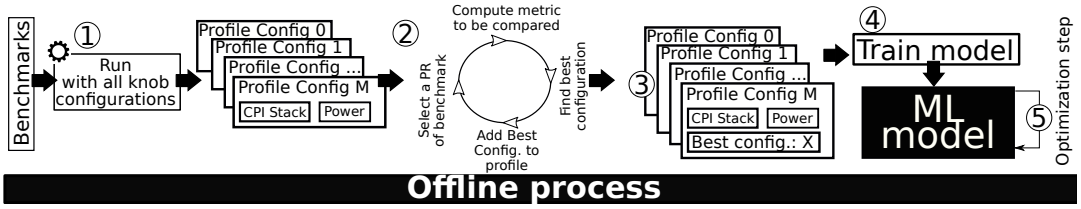


FIGURE 5.2: Steps to build the predictor model incorporated in MARK. All these steps are carried out offline. The predictor model's training process needs to have the information of the execution of the benchmarks with the different knob configurations.

As shown at ① in Figure 5.2, we carry out an extensive offline profiling of all benchmarks with all possible hardware knob configurations. When running the benchmarks with a given knob configuration, we gather multiple information to build the profile of all parallel regions in the benchmark: its CPI stack, power consumption, execution time, and the knob configuration used. In order to read the CPI Stack in our system, MARK reads the 52 performance counters shown in Table 5.1. The result of this profiling is a dataset with all the information about every parallel region in our benchmarks when executed with a given hardware knob configuration.

The dataset is processed in order to find the best knob configuration that minimizes a given metric for every parallel region. For every unique parallel region, we compare the multiple profiles of the parallel region when running with different knob configurations and choose the one that minimizes the given metric as the best hardware knob configuration as shown at ②. Then, we add a new field on the profile with the best hardware knob configuration as shown at ③ in Figure 5.2. At this point, we can manually build an *oracle* that runs every parallel region in a benchmark with the best knob configuration.

This new processed dataset can be leveraged to build an intelligent model with supervised learning as shown at ④ in Figure 5.2. With supervised learning, we can train a model that maps an input (the different profiles from every parallel region) to an output (the best hardware knob configuration). Therefore, we use the characteristics of a parallel region as features of the model: CPI stack, power consumption, and the knob configuration used to gather the data. The desired output value for our model is the best knob configuration that minimizes the given metric.

After the supervised learning process, we have a trained model to minimize a given metric as shown at ④ in Figure 5.2. This model can be optimized by tuning its parameters to increase its prediction accuracy. After an optimization step where different parameters of the predictor are tuned (shown at ⑤), the trained model is ready to predict the best knob configuration from a system state (current knob configuration), a CPI stack, and power consumption.

5.2.3 Runtime Actions

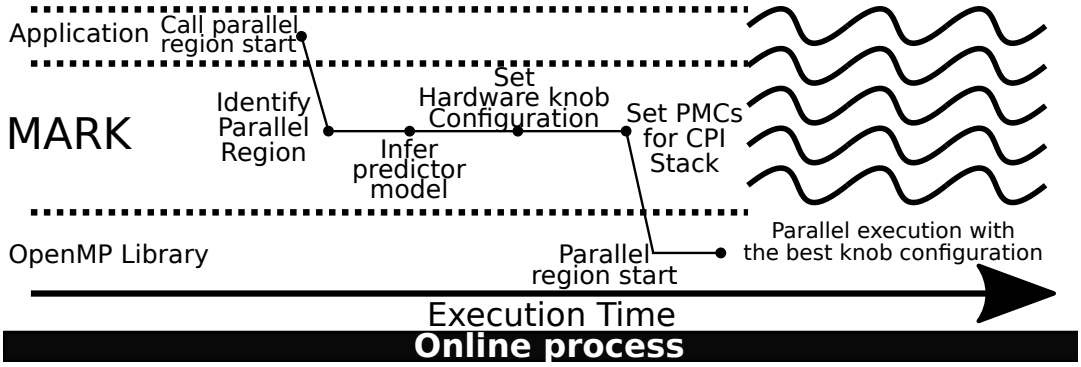


FIGURE 5.3: Runtime for library interposition included in MARK in order to intercept the start or end of a parallel region and reconfigure the hardware knobs specifically for a given parallel region.

MARK implements a runtime library that, using library interposition, captures the calls to the OpenMP runtime library. Library interposition is a well known and commonly used mechanism to gather profiling information and reconfigure the system without modifying or recompiling the source code of the applications. Using the *LD_PRELOAD* environment variable from Linux, we can use MARK on top of the OpenMP library as shown in Figure 5.3. Therefore, when the application calls to the OpenMP library to start or end a parallel region, (*GOMP_start* and *GOMP_end* functions, respectively), the runtime library of MARK intercepts the call, sets the hardware knobs, starts the profiling, and then calls the actual OpenMP library. Note that MARK could be used in many other parallel or task library that exposes the start and end of a parallel region as library functions.

The runtime library of MARK is in charge of:

1. Identifying the parallel region by its Program Counter (PC).
2. Using the predictor model to infer the knob configuration to be used.
3. Setting the hardware knobs of the architecture system through the *sched_setaffinity* function from Linux for the thread placement knob, the *set_num_threads* function from OpenMP for the SMT level knob, and writing to the DSCR register for the data prefetcher knob.
4. Setting the performance counters to be measured at the beginning of the parallel region
5. Stopping and reading the performance counters with *perf* and reading the power consumption of the system with in-band readings at the end of the parallel region.

The first time a parallel region is executed, MARK runs the parallel region with the default knob configuration. The next time the same parallel region is about to be executed, MARK intercepts the parallel region start, calls the ML model, and predicts the best knob configuration. Then, the hardware knobs are set and the parallel region is executed with better performance. Therefore, MARK predicts the best knob configuration from the default knob configuration in the first inference, and in the following instances it predicts the best knob configuration from the previous best knob configuration. This allows MARK to adapt when the behavior of a parallel region changes due to input sensitivity.

The runtime needs to take into account the overheads of setting the hardware knobs. Setting the hardware knobs of a short parallel region can lead to slowdowns due to overheads being longer than the execution of the parallel region itself.

Note that MARK can be easily ported to other systems and hardware knobs by simply re-training the ML model of the predictor on the target system, without any changes in the algorithm of the MARK runtime library.

5.3 Methodology

In this section, we explain in detail the different evaluated hardware knob configurations and how performance counters are gathered. Then, the gathered data are analyzed and the process for training and validating our ML model is detailed.

5.3.1 Hardware Knob Configurations

We coordinate the configuration of the thread placement across sockets, the SMT level, and the data prefetcher aggressiveness.

For the thread placement and the SMT level, Table 5.2 shows the possible combinations of SMT level and thread placement considered for the evaluation of MARK in our experimental setup with a POWER9-based HPC system.

TABLE 5.2: Possible combinations of the SMT level and occupancy of the sockets. A full occupancy of a socket is when all 20 physical cores are executing the application, a half occupancy is when 10 physical cores are used. The logic cores being used by an application depends on the SMT level and the occupancy of the sockets.

SMT level	Occupancy of Socket 1	Occupancy of Socket 2	Num. Threads
4	Full	Full	160
2	Full	Full	80
1	Full	Full	40
4	Full	Empty	80
2	Full	Empty	40
4	Half	Half	80
2	Half	Half	40
4	Half	Empty	40

For the prefetcher, we consider the most relevant fields for our experiments due to differences in execution time, energy, and power consumption as explained in Section 3.1.

When the machine boots, it sets the prefetcher to the default configuration: URG set to 4, LDS enabled, DPDF set to 4, and all the other options disabled. We consider 40 possible configurations for the prefetcher with the fields described in Section 3.1.

5.3.2 Collecting Runtime Data

The data that drives our predictors at runtime are based on the performance counters that build the CPI stack, which consists of up to 52 performance counters in our experimental setup. Unfortunately, reading a large amount of performance counters at the same time is unfeasible. For instance, the POWER9 system we use in our experimental setup has 6 Performance Monitoring Units (PMU); therefore, it can only read up to 6 different performance counters. 2 of these PMUs always measure cycles and completed instructions. To allow reading more performance counters than the available number of PMUs, modern OS automatically handle the switching between groups of a size of the available number of PMUs at a small granularity (10ms in our system). Therefore, there are 2 approaches to gather the required performance counters: (1) set by hand the performance counters in batches of 4 to be read in every execution of a parallel region or (2) read all the performance counters at the same time and rely on the OS to switch between them.

Figure 5.4 shows the difference between (1) reading performance counters in batches of 4 across different executions of a parallel region (Precise) and (2) reading all the performance counters in a single iteration of a parallel region relying on the OS to switch between them at fine granularity (Multiplexing). It can be observed that, for these 5 performance counters, there is no tangible difference between

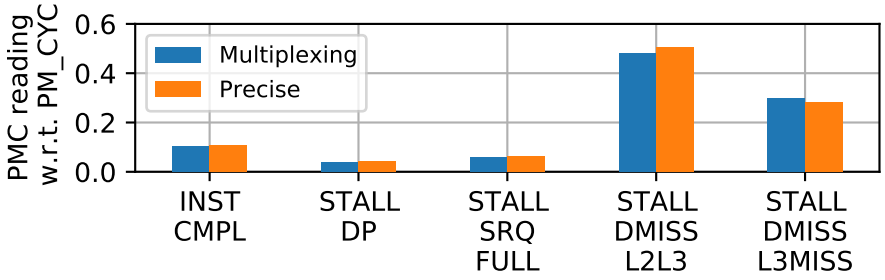


FIGURE 5.4: Difference between reading performance counters in groups of 4 and OS multiplexing at fine granularity. We show 5 different performance counters measuring instructions completed (INST CMPL), stalls due to a double precision execution pipe (STALL DP), stalls due to the store queue was full (STALL SRQ FULL), stalls due to cache misses in the L1 data cache that was resolved in the L2 or L3 cache (STALL DMISS L2L3), and stalls due to cache misses in the L3 cache (STALL DMISS L3MISS).

the two approaches. The rest of performance counters we capture follow the same trends. Since applications can have a low number of iterations, we use the Multiplexing approach that relies on the OS to switch between the performance counters.

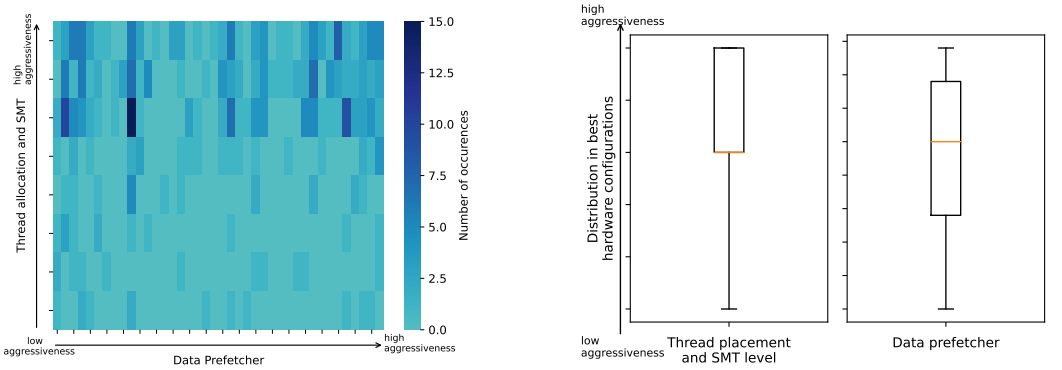
Measuring performance counters with multiplexing within a parallel region can lead to inaccuracies when the parallel region has a low execution time, since the OS switches between groups of 4 performance counters at an interval of 10ms in our setup. Thus, measuring the 52 performance counters that form the CPI stack requires that the parallel regions run for at least 130ms. MARK considers parallel regions with an execution time shorter than one second to be short parallel regions, ensuring that the performance counters are measured correctly while taking into account the overheads described in Section 5.2.3.

5.3.3 Data Analysis

Figures 5.5 (a) and (b) show the distribution of the best hardware configurations when minimizing the execution time of all the parallel regions of all the benchmarks. Figure 5.5 (a) shows the number of parallel regions that, with a given hardware configuration, maximize their speedup with respect to the default configuration. Figure 5.5 (b) shows the distribution of the best hardware configurations when maximizing speedup in terms of aggressiveness for thread placement and SMT level, and data prefetcher.

As shown in Figure 5.5 (a), plenty of hardware configurations do not improve the default hardware configuration when running a given parallel region of a benchmark. Also, we can see that plenty of parallel regions have a unique best hardware knob configuration that maximizes speedup.

Figure 5.5 (b) shows that the hardware configurations maximize the speedup are biased to higher aggressiveness for the thread placement and SMT level. This bias is also present when targeting other metrics such as power consumption or EDP.



(A) Distribution of the best hardware configuration with any speedup percentage with respect to the default configuration.

(B) Distribution of aggressiveness of the hardware configuration for the best hardware configurations when maximizing speedup.

FIGURE 5.5: Number of parallel regions with a given hardware knob as the best hardware knob when maximizing performance. On the left, the speedup with respect to the default hardware configuration of all the best hardware configuration of all parallel regions is shown. On the right, the distribution of aggressiveness of all the best hardware configurations is shown. Boxplots show the 25th, 50th, and 75th percentiles and whiskers show the minimum and maximum value.

These two characteristics (unique best hardware configurations and biased data) can have a negative impact on the training of the machine learning predictor model. As mentioned before, in supervised learning, the training phase is done by feeding inputs to the model and adjusting the model to the desired output. Using an unbalanced dataset can lead to a predictor model with a low accuracy.

In the next section we propose a solution to deal with datasets with such characteristics.

5.3.4 Dealing with Biased Training Data

Biased training data sets can cause accuracy problems in machine learning predictor models, since the training data does not contain all possible scenarios and therefore, they can fail to generalize when predicting for not previously seen data. This is due to the tendency of multiple hardware configurations to perform better for multiple benchmarks as shown in Section 5.3.3. As shown in Figure 5.5 (b), our training data for thread placement and SMT level is biased to higher aggressiveness. On the other hand, for the data prefetcher this bias is not present. This happens because we use benchmarks designed for high performance computing systems, which in most cases benefit from using a higher number of threads and sockets, although it is not the case for all benchmarks. In contrast, the effectiveness of the data prefetcher depends on the data access patterns of the applications, which is more varied.

To avoid accuracy problems caused by biased training data sets, we evaluate a hybrid approach that uses the predictor model only for one dimension (the data

prefetcher) followed by a second step that applies an exploration approach on the other two dimensions (thread placement and SMT level). The search space for the second exploration step uses the combinations shown in Table 5.2.

We evaluate two approaches in Section 5.4. The approach that predicts all the hardware knob configuration (i.e. thread placement, SMT level, and data prefetcher) using the machine learning model is called MARK-OML, and the approach that uses the machine learning model to predict the data prefetcher configuration and then uses a second exploration step to predict the thread placement and the SMT level is called MARK-2step.

In MARK-2step, the prediction of the best hardware configuration from a CPI stack is divided into two steps: (1) the predictor model is inferred to find the best hardware data prefetcher configuration for the current thread placement and SMT level. Then, (2) it performs a limited exploration step for the possible thread placement and SMT level configurations with the predicted hardware data prefetcher, taking 7 iterations of the parallel region. Then the resulting best hardware knob configuration from the exploration is selected as the best hardware knob configuration. Notice that this hybrid approach leads to a reduction of the search space.

In order to react to changes in the same parallel region across multiple instantiations, MARK-2step records the best execution time of the parallel region after the exploration step. The described process is restarted when the current execution time differs more than a 10% with respect to the best execution time.

5.3.5 ML Model Training and Validation

We train the ML model with supervised learning as introduced in Section 2.5. The inputs for the training process are the different CPI stacks of all the parallel regions from all benchmarks and the desired output is set to the best hardware knob configuration found offline.

In order to validate and test our model on unseen data, we validate the model to asset the generality of the model. Therefore, we split the dataset into two datasets: one for training and one for validation. We feed the training dataset to build and train the predictor and the validation dataset is used to validate the built model. We use the leave-one-out cross-validation (LOOCV) method. We select all benchmarks except the one to be validated to build the training dataset and the benchmark to be validated is the evaluation dataset. Notice that a given benchmark can contain multiple parallel regions.

Table 5.3 shows the achieved accuracy when running different predictors for MARK-OML and MARK-2step. The parameters shown for each predictor in Table 5.3 are chosen based on an offline evaluation for different values of each parameter. The chosen parameters are the parameters that offered the highest accuracy. Notice that multiple hardware knob configurations can achieve a similar performance.

TABLE 5.3: The different models explored in this work. For each model, we show the most important parameters and their accuracy when predicting the best hardware knob configuration. MARK-OML column reflects the accuracy when predicting the complete configuration (thread placement, SMT level, and data prefetcher). MARK-2step column reflects the accuracy when predicting the data prefetcher and a limited exploration for thread placement and SMT level. Accuracy represents the percentage of correct predictions with respect to the total predictions. A correct prediction is a prediction of a hardware knob configuration with the same performance as the best hardware knob configuration.

		Prediction accuracy	
Predictor	Parameters	MARK-OML	MARK-2step
Random Forest	Number of trees: 100 Minimum samples to split: 2	50%	81%
KNeighbor	Number of neighbors: 5 Weights: Uniform	44%	78%
ADA	No base estimator Number of estimators: 50	36%	77%
Decision Tree	Minimum samples to split: 2 Criteria for splitting: Gini	36%	77%
MLP	Number of layers: 100 Activation function: sigmoid Solver: lbfgs	22%	74%
GaussianNB	No priorities	34%	67%

Therefore, even though the accuracy in the predictions can be low, the performance of the hardware knob configuration can be similar to the best hardware knob configuration.

From the results of Table 5.3, all the experiments presented in the evaluation use the Random Forest predictor model in both MARK-OML and MARK-2step.

5.4 Evaluation

This section evaluates MARK and compares it to other techniques. The techniques evaluated in this section are:

- Default is the default configuration when the machine boots up. The machine is set to use both sockets with SMT4 enabled and the default prefetcher. The default prefetcher has the LDS enabled, URG set to 4, DPDF set to 4, and all the other options disabled.

- Exploration is an exploration-based approach (see Chapter 4). The exploration technique explores the configurations in the following order: SMT level, prefetcher aggressiveness, and thread placement.

- Best Hardware Configuration (BHC) is the best configuration per parallel region. The best configuration is found after an exhaustive offline profiling (320 possible configurations for a given parallel region).
- MARK-OML is the MARK technique which predicts all the hardware configuration using a Random Forest [22] as underlying predictor.
- MARK-2step is the MARK technique using the 2-step technique to deal with biased data to train a Random Forest [22] as underlying predictor.

In the following sections, we evaluate the performance of MARK to minimize different metrics: (1) execution time in Section 5.4.1, (2) power consumption in Section 5.4.2, and (3) Energy-Delay Product (EDP) in Section 5.4.3. The underlying predictor of MARK can be changed easily, which allows to minimize different metrics without tuning or redesigning the algorithm of MARK. Therefore, minimizing a different metric only needs to train a new model and feed it to MARK.

In many of the evaluated benchmarks the default hardware knob configuration is the best hardware knob configuration in terms of execution time. For clarity, we show the subset of parallel benchmarks for which their best hardware configuration achieves an speedup of at least 10% with respect to the default hardware knob configuration.

5.4.1 Performance

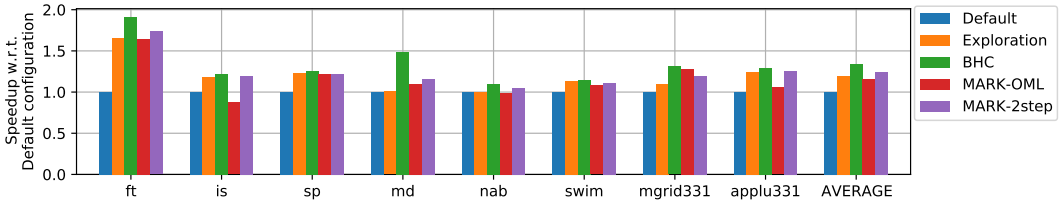


FIGURE 5.6: Results when minimizing execution time. X-axis shows the different evaluated benchmarks and the Y-axis shows the speedup with respect to the default configuration. Higher is better.

Figure 5.6 shows the speedup with respect to the default knob configuration for all the evaluated benchmarks with different techniques. Results show that the best hardware configuration can achieve an average speedup of 1.34x. MARK-OML achieves an average speedup of 1.15x, while Exploration and MARK-2step achieve average speedups of 1.19x and 1.24x, respectively. Next we focus on showing the behaviors of `FT` and `MD`, as they are the benchmarks that benefit the most from a different knob configuration than the default configuration.

`FT` and `MD` benefit the most from a different knob configuration with speedups of 1.91x and 1.48x when running with BHC, respectively. The exploration approach

only achieves a speedup of 1.65x for FT and no speedup for MD. For MARK-OML, FT achieves a similar speedup as Exploration (1.64x) and MD achieves a better performance than Exploration (1.09x). Last, MARK-2step performs better than the exploration technique and MARK-OML, achieving speedups of 1.73x and 1.15x for FT and MD, respectively.

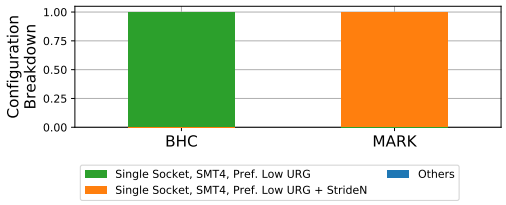


FIGURE 5.7: MD Configuration breakdown for BHC and MARK. The main parallel region is better to be run in a single socket when maximizing speedup.

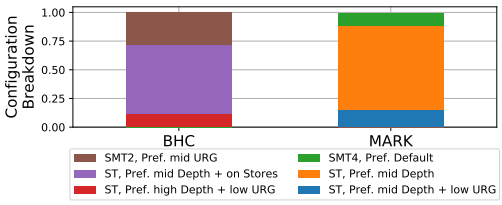


FIGURE 5.8: FT Configuration breakdown for BHC and MARK. For FT, it is better to be executed in both sockets when maximizing speedup.

Figures 5.7 and 5.8 show the configuration breakdown for the whole execution of FT and MD, respectively, when running with BHC and MARK-2step. As shown in the figures, the techniques select different knob configurations.

For FT, different knob configurations are needed to achieve the speedup of 1.91x with respect to the default configuration. BHC always selects configurations using both sockets with lower SMT levels than the default SMT level (SMT2 and ST) with a data prefetcher with low aggressiveness (mid URG, mid Depth, or low URG). On the other hand, our proposed MARK-2step selects similar configurations, yet, the small differences in the configuration lead to a slowdown of 18.0% with respect to the BHC.

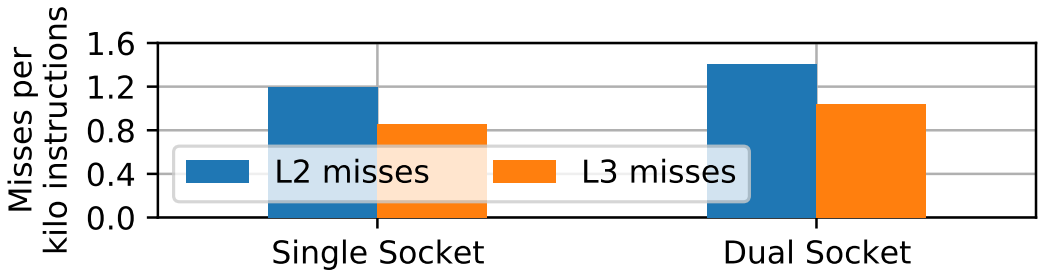


FIGURE 5.9: Cache misses per kilo Instruction for L2 and L3 caches when running the L3 caches for L2 and L3 caches when running the MD benchmark within a single socket and both sockets.

For MD, the best knob configuration is using a single socket with a SMT4 level and a low urgency prefetcher. As we can see in Figure 5.9, the number of misses in the L2 and in the L3 caches increase dramatically when running in both sockets. This is caused by the dynamic task scheduling used in the main parallel region of MD, where threads execute iterations of the parallel loop as soon as their current iteration

is finished. Therefore, threads execute iterations with non-consecutive data, which causes to lose data locality.

Note that MARK-2step is able to predict almost perfectly the knob configuration. MARK-2step selects a low URG prefetcher with the StrideN option, while the optimal configuration is the same low URG prefetcher but without the StrideN option. The small difference in selecting the prefetcher configuration leads to a slowdown of 33% with respect to the BHC.

5.4.2 Power consumption

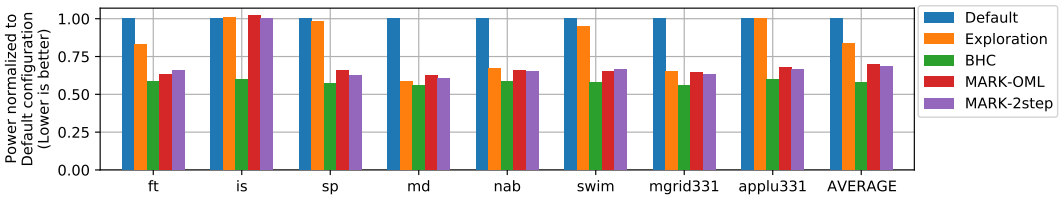
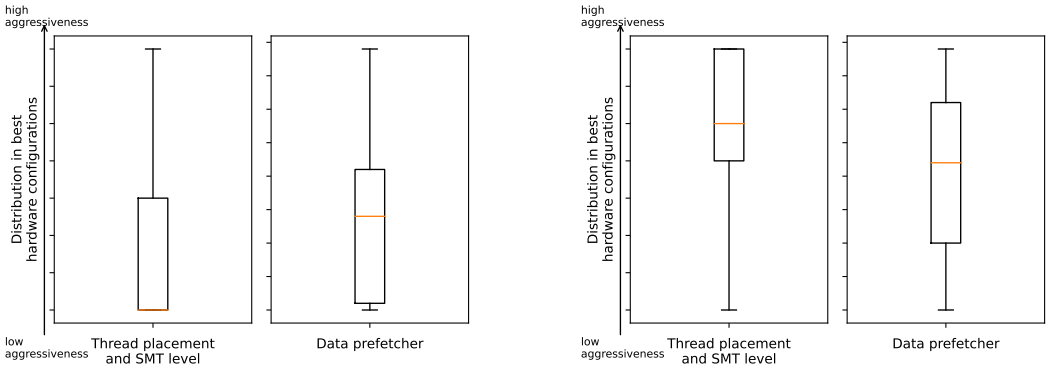


FIGURE 5.10: Results when minimizing power consumption. X-axis shows the different evaluated benchmarks and the Y-axis shows the power consumption normalized with respect to the default configuration. Lower is better.

Figure 5.10 shows the power consumption normalized to the default configuration for all the evaluated benchmarks with the different techniques. It can be observed that MARK-OML and MARK-2step achieve large reductions in power consumption, with averages of 30% and 31%, respectively. In contrast, the exploration technique only achieves an average power reduction of 17%, and in 4 benchmarks it is unable to achieve any significant reductions (`IS`, `SP`, `swim`, `applu331`). The exploration technique is not able to reduce power consumption due to its nature, since the knobs are explored one by one, which can miss opportunities if the exploration order is not the optimal.

Figure 5.11a shows the distribution of the aggressiveness of the best hardware configurations when minimizing power consumption for all parallel regions and benchmarks. We can see that thread placement and SMT level are biased to low aggressiveness hardware configurations (i.e. less sockets and number of threads within the sockets usually will consume less power). Therefore, an exploration technique has to traverse all the search space of the thread placement and SMT level without falling into a local minimum.

It can be observed that no approach is able to reduce the power consumption of `IS`. This benchmark is composed of 3 parallel regions, although 97.2% of the execution time is spent in a single parallel region. The best hardware configuration is using a single socket with a SMT2 level and a high aggressiveness prefetcher (i.e. high URG, high Depth, and prefetch on Stores as well). However, the exploration approach falls into a local minimum and selects the default hardware configuration;



(A) Distribution of aggressiveness of the hardware configuration for the best hardware configurations when minimizing power consumption.

(B) Distribution of aggressiveness of the hardware configuration for the best hardware configurations when minimizing EDP.

FIGURE 5.11: Distribution of aggressiveness of the hardware configuration for the best hardware configurations when minimizing different metrics. Boxplots show the 25th, 50th, and 75th percentiles and whiskers show the minimum and maximum value.

MARK-OML predicts incorrectly the thread placement, the SMT level, and the data prefetcher; and MARK-2step predicts incorrectly the prefetcher configuration, which leads to selecting the default hardware configuration. On the other hand, in `SP`, the exploration technique falls into a local minimum, while MARK-OML and MARK-2step find a hardware configuration that reduces power consumption by 34.2% and 37.4%, respectively.

5.4.3 EDP

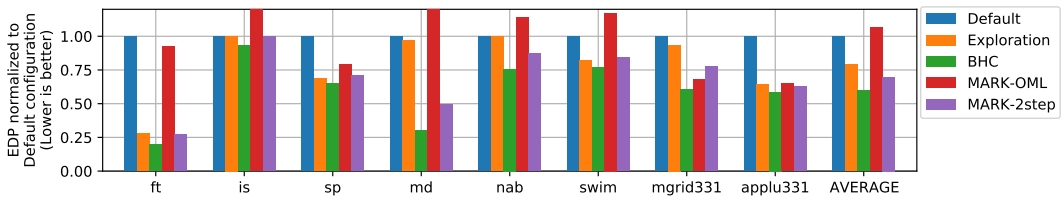


FIGURE 5.12: Results when minimizing EDP (Energy-Delay Product). X-axis shows the different evaluated benchmarks and the Y-axis shows the EDP normalized with respect to the default configuration. Lower is better.

Figure 5.12 shows the EDP normalized to the default configuration for all the evaluated benchmarks with the different techniques. On average, MARK-2step reduces EDP with respect to the Default configuration by 30%, while the exploration approach reduces it by 21%. In contrast, MARK-OML is unable to reduce EDP, and even increases it slightly on average. We also can see that the exploration technique and MARK-OML are unable to find a better hardware configuration than the default hardware configuration for multiple benchmarks (e.g., `IS`, `MD`, `NAB`, `SWIM`, `MGRID331`).

Figure 5.11b shows the distribution of the aggressiveness of the best hardware configurations when minimizing EDP for all parallel regions and benchmarks. Contrary to the distribution when minimizing power consumption (Figure 5.11a), the best thread placement and SMT level configurations are biased to higher aggressiveness. Yet, the 25th percentile shows that low aggressiveness for thread placement and SMT level is still relevant. This leads to the exploration approach falling in local minimums. In addition, this figure shows that MARK-OML is biased to higher aggressiveness in thread placement and SMT level. Therefore, MARK-OML fails to predict correctly in multiple benchmarks. On the other hand, MARK-2step, predicts correctly the hardware configuration with a few exceptions (e.g., `mgrid331`) because it is not affected by the bias of the thread placement and the SMT level.

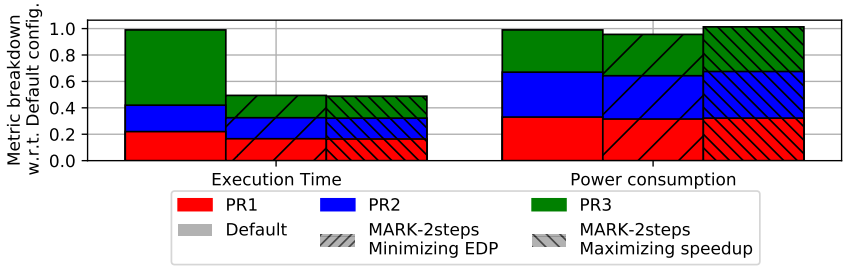


FIGURE 5.13: Execution time and power consumption distribution of the 3 main parallel regions of `FT` for the MARK-2step when minimizing EDP and maximizing speedup with respect to the default configuration.

The EDP of `FT` and `MD` are improved mainly due to the reduction on execution time as explained previously, which translate into selecting the same hardware configuration than MARK-2step when maximizing speedup. Yet, for `FT`, MARK-2step minimizing EDP selects a slightly better hardware configuration in terms of EDP. `FT` has 3 main parallel regions that take up to 85% of the total execution time when running with the default hardware configuration. Figure 5.13 shows the change on execution time and power consumption of these 3 parallel regions (labelled as PR1, PR2 and PR3) when running with the default configuration and MARK-2step when minimizing EDP and maximizing speedup. As seen in this figure, when maximizing speedup, MARK-2step selects a faster hardware configuration while keeping the same power consumption than the default configuration. On the other hand, when minimizing EDP, MARK-2step selects a hardware configuration that reduces power consumption while offering a slightly worse execution time than when maximizing speedup. As a result, when minimizing EDP, MARK-2step reduces EDP by 78.0% and 2.4% with respect to the default hardware configuration and MARK-2step when maximizing speedup, respectively.

For `IS`, the exploration approach falls into a local minimum and ends up selecting the default configuration. On the other hand, both MARK-OML and MARK-2step

fail to predict correctly the best hardware configuration shown with the BHC technique. Yet, MARK-2step is able to select a hardware configuration that achieves a similar EDP than the default hardware configuration.

5.5 Conclusions

Complex HPC systems include many architectural features that are not equally effective across applications. For this reason, modern processors include multiple hardware knobs such as thread placement, SMT level, or the data prefetcher configuration, to adapt the behavior of the processor to the running workload. In this chapter we identify interactions between these knobs, and we demonstrate that they need to be configured at fine granularity and in a coordinated manner to minimize execution time, power consumption, or EDP. However, finding the optimal hardware configuration for these knobs is challenging due to the large search space, the interactions between the hardware knobs, and the different resource demands of the applications.

To overcome these problems we present MARK, a library for parallel applications that adapts the different hardware knobs available in the architecture with a prediction model based on Machine Learning. MARK is able to minimize different target metrics and is completely transparent to the user. We evaluate MARK with multiple predictor models for a wide set of OpenMP benchmarks running on an OpenPOWER system with a POWER9 processor. Compared to the default knob configuration, MARK achieves speedups of up to 1.73x (24.0% on average), power consumption reductions of up to 39.4% (31.3% on average) and EDP reductions of up to 73.1% (30.1% on average) when targeting these metrics.

Chapter 6

Data Prefetching for In-order cores

6.1 Introduction

The High Performance Computing (HPC) field seeks to increase performance of current system to reach exascale, whilst maintaining a 20MW power budget [162]. This level of performance/Watt needs solutions with unprecedented levels of energy efficiency [2, 5].

Current commodity hardware such as embedded and mobile processors is designed to be energy efficient due to constraints such as battery life and over-heating. Usually, these low-power processors contain in-order cores, which are significantly smaller (area, power) and lower performance compared to the typical desktop or server-class out-of-order processors. Low-power processors are promising due to their density for HPC, and they have previously been investigated for such purposes [132, 131, 110, 58]. Yet, adding more cores in a processor puts more pressure on the memory system.

To alleviate the memory wall problem different solutions are available. Out-of-order processors can reorder instructions to avoid stalls due to being waiting for data. Therefore, the order of execution, within consistency constraints, is based on data availability in an order different than the original order of execution. Another option vendors use to reduce memory latency is to include a hardware data prefetcher. The data prefetcher brings data to the processor's cache before it is needed, thus reducing stalls.

Low-power processors are usually based on in-order cores. Therefore, instruction reordering is not possible to reduce memory stalls. However, data prefetching can be used and we explore this solution in this chapter.

In this chapter we simulate and exhaustively analyze the performance of 5 hardware prefetchers in an in-order core in single and multicore systems, with 2 state-of-the-art dynamic mechanisms. Results show that dynamically reconfiguring the data prefetcher on in-order cores can speedup executions up to 1.4x.

6.2 Dynamic Mechanisms

We implement 2 dynamic mechanisms to modify at runtime the behavior of the different possible prefetchers.

Our dynamic mechanisms gather different data prefetcher metrics and, based on those metrics, are reconfigured at the end of an interval. An interval is defined when half of the blocks of the cache are evicted. We use an interval based on cache activity instead of fixed-time, as the cache-related data is more relevant to the prefetcher. We use the global history of the metrics to take into account the global behavior of the application in order to reduce the noise of small application phases.

6.2.1 Dynamic prefetcher aggressiveness

We dynamically tune the prefetcher aggressiveness at execution time following the proposed algorithm in [144]. Due to constraints in our simulation environment, this dynamic mechanism can only be applied to *queue-based* prefetchers. *Queue-based* prefetchers are prefetchers with a buffer that holds data petitions that are handled when the memory controller is idle. On the other hand, non-*queue-based* prefetchers request their own data petitions in an active manner.

TABLE 6.1: Agressiveness (A) and Thresholds (T) values for the dynamic mechanisms used in this chapter.

A_{high}	A_{low}	$T_{lateness}$	$T_{pollution}$	$T_{congestion}$
0.75	0.40	0.05	0.001	0.005

Original thresholds from the work by Srinath et al. [144] are changed to adapt them to our in-order system, which we show in Table 6.1. We needed to increase the lateness threshold ($T_{lateness}$) and reduce the pollution one ($T_{pollution}$) in order to not lose performance since original thresholds were causing a reduction in prefetcher aggressiveness and a slowdown in execution time.

TABLE 6.2: Possible configurations for the dynamic prefetcher aggressiveness

Configuration	Distance	Degree
Very conservative	4	1
Conservative	8	1
Middle-of-the-road	16	2
Aggressive	32	4
Very aggressive	64	4

We consider 5 possible configurations for the prefetcher aggressiveness as shown in Table 6.2. The middle-of-the-road configuration is the default data prefetcher configuration for our simulated core to initialize the dynamic mechanism.

6.2.2 Dynamic destination

We implement a mechanism to decide dynamically where a cache stores the prefetched cache lines for 2 reasons: (1) to be able to prefetch even if the cache is blocked due to having too many demand accesses and (2) to alleviate the memory bandwidth requirements for the caches.

In this mechanism, we select a cache to be a master cache. The master cache decides at each interval where to store the prefetched cache lines. The master cache uses the full memory access stream to train the prefetcher. Then, the master cache can decide to store the prefetched cache line into another cache level if the cache is polluted or to issue a prefetch from another prefetcher if the cache is congested.

In our experiments, we use the L1 cache as master. Therefore, all L1 caches can send prefetches to their respective private L2 or to the shared L3 cache. We set the master cache to the L1 cache for several reasons: (1) L1 cache is the best performing one in terms of latency of all the caches due to its proximity to the core; (2) usually, it is the cache most limited by the memory bandwidth.

At the end of each interval, the master cache checks the pollution and congestion levels of the cache where current prefetches are being stored, starting with itself.

In the case the cache is polluted or congested, new prefetches will be send to the next level cache for the next interval (i.e. pollution level or congestion level are greater than $T_{pollution}$ or $T_{congestion}$, respectively). If that next cache level is polluted or congested, prefetches are sent to the last level cache. This happens irrespective whether the L3 cache is polluted or congested.

At the end of every interval, the master cache checks its own pollution and congestion levels in order to try to prefetch always to itself, which performs better since we set the master cache to the L1 cache, closest cache to the processor.

6.2.3 Metrics

Our dynamic mechanisms are based on the following gathered metrics:

- Prefetch Accuracy: This measures if the prefetcher is bringing in useful data before the cache block is accessed. We define data as useful when it is accessed during its lifetime. We consider its lifetime as all the time that exists in a cache. It is defined as: $\frac{Number\ of\ useful\ prefetches}{Number\ of\ issued\ prefetches}$

- Prefetch Lateness: This measures whether the prefetch requests arrived in a timely fashion, in time to satisfy the demand access. It is defined as:
$$\frac{\textit{Number of late prefetches}}{\textit{Number of useful prefetches}}$$

- Cache Pollution: This measures the useless data brought in by the prefetcher in the cache. It is defined as:
$$\frac{\textit{Number of misses caused by the prefetcher}}{\textit{Number of misses}}$$
In order to track useless data, we track cache lines that are brought by the prefetcher. If there is a miss on a cache line that was brought by the prefetcher, we consider it as useless data brought by the prefetcher.

- Cache Congestion: This measures the time that the cache's prefetcher cannot issue more prefetches due to the unavailability of free Miss Status Handling Registers (MSHRs). It is defined as:
$$\frac{\textit{Number of times cache is blocked due to a MSHR miss}}{\textit{Number of MSHR misses}}$$

6.3 Evaluation

6.3.1 Hardware Data Prefetchers Evaluated

We evaluate 5 different prefetchers, including running experiments without any hardware prefetching. Every cache level can have a hardware prefetcher, and we perform a full design-space exploration, running every benchmark with all the possible combinations of prefetchers. The prefetchers we evaluate are described in Section 2.3: Neighbor, Nextline, Correlation, Stride and Stream.

Notice that, we only apply our dynamic aggressiveness mechanisms on the queue-based prefetchers: Nextline, Stream and Stride. The Queue class in gem5 is a class for existing prefetchers in gem5, which process every memory request ordered by age. Queue-based prefetchers can be generically tuned with parameters such as distance or degree. If they are not queue-based prefetchers, the available parameter set relies on each prefetcher's implementation, which may not have distance or degree exposed.

6.3.2 Region of Interest Simulation

A checkpoint is created at the start of the Region of Interest (ROI) of every benchmark.

We simulate the Region of Interest (ROI) of each benchmark until either the ROI finishes or a maximum number of instructions is reached. We select the ROI manually via source-code instrumentation, as the SimPoint methodology [123] cannot be applied to multi-threaded applications.

Before collecting data to measure the performance of the given configuration, we warm-up the simulation for all benchmarks using 50M instructions. The standard detailed simulation interval is 500M (except in the FFT benchmark, which runs for 1B instructions). The number of instructions simulated is chosen taking into account that every thread must be doing useful work.

We define useful work as progress on the execution, which ensures execution is not stalled in an idle loop waiting for data. We measure this via number of memory, scalar or floating operations executed.

6.3.3 Results Single Core

In this section, we cover results obtained using a single-core system and all benchmarks run using a single thread. All others components are the same as detailed in Section 3.2.

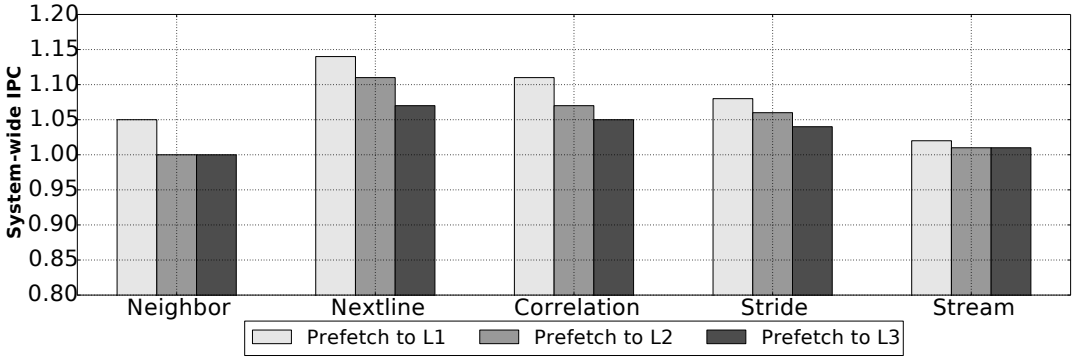


FIGURE 6.1: System-wide IPC in static experiments with different prefetchers for the L1 cache against no prefetching. The L1 prefetcher can prefetch cache lines into the L1 (normal behavior), L2 or L3 caches in order to explore possible benefits using the dynamic destination explained in Section 6.2.2

We start by evaluating the possible performance gains of dynamic destination mechanisms. Figure 6.1 shows the performance in terms of IPC of different L1 data cache prefetchers using dynamic destination, with respect to no prefetching at any cache level. Table 6.3 lists which prefetchers were used for the L2 and L3 caches. Please note that these vary depending on the application, as we chose the best L2 and L3 prefetchers in each case.

We observe that as prefetchers insert the prefetched lines into upper levels in the cache hierarchy, application performance with respect to prefetching to the L1 cache is reduced, as upper cache levels have higher latency.

TABLE 6.3: L2 and L3 prefetchers used in the single core experiments. These configurations are the most performing ones in static experiments.

Benchmark	L2 Prefetcher	L3 Prefetcher
CoMD	Neighbor	Neighbor
DGEMM	Stride	Stride
FFT	Neighbor	Stride
PTRANS	Stride	Stride
STREAM	Neighbor	Stride
HPCG	Neighbor	Stride
mcb	Neighbor	Stride
miniFE	Neighbor	Neighbor
pathfinder	Stride	Stride

The most complex the prefetcher, the most sensitive to prefetching into the upper cache levels. For instance, Correlation and Neighbor prefetchers have a higher penalty when prefetching into the next upper cache level (Prefetch to L2).

In the case of Neighbor, this technique is detrimental: when prefetching to a higher cache level the performance is the same as no prefetching. This is caused by how the prefetchers on all the cache levels interact in this experiment. We cover this in further detail later.

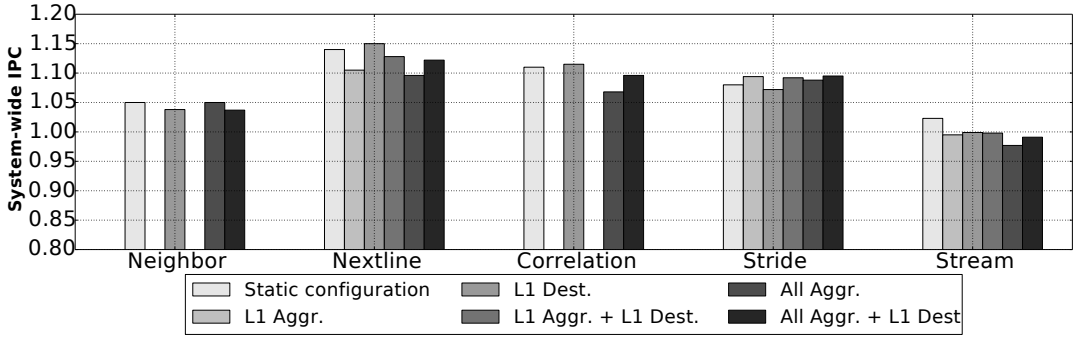


FIGURE 6.2: System-wide IPC with different L1 prefetchers and different configurations with the dynamic mechanisms explained in Sections 6.2.1 and 6.2.2. L2 and L3 prefetchers are fixed across the different configurations to the prefetchers specified in Table 6.3. Due to implementantation limitations, Neighbor and Correlation prefetchers cannot dynamically adapt their aggressiveness.

Figure 6.2 shows the performance of various L1 data cache prefetchers, using the configurations listed in Table 6.3, relative to no prefetching. The L2 and L3 configurations are as outlined in Table 6.4. Results show that, typically, all prefetchers benefit from the dynamic mechanisms, however, at different rates.

TABLE 6.4: Prefetcher configurations used in this work. DA is Dynamic Aggressiveness enabled. DD is Dynamic Destination enabled.

	Prefetchers		
Configuration	**L1**	**L2**	**L3**
1. No prefetcher config.	-	-	-
2. Static config.	-	-	-
3. L1 Aggr.	DA	-	-
4. L1 Dest.	DD	-	-
5. All Aggr.	DA	DA	DA
6. All Aggr. + L1 Dest.	DA & DD	DA	DA

The Neighbor prefetchers shows a small performance reduction when the dynamic destination is enabled. As we explain in detail later, the Neighbor prefetcher has a high accuracy, which leads it to place useful prefetchers further away from the L1 cache, thus affecting the overall latency by keeping in the cache hierarchy useful cache lines.

The Nextline prefetcher gains the most out of the dynamic destination mechanism, as the prefetcher normally issues a large number of prefetches which, if all the prefetched cached lines placed in the same cache, would lead to increased levels of pollution. As such, by using this mechanism we can increase the prefetch data's utilization.

The Correlation prefetcher has performance gains of 10% across the different static and dynamic configurations against not using an L1 data cache prefetcher. The lowest gain is when all prefetchers have the dynamic aggressiveness enabled. This is caused by pollution added by the L2 and L3 cache prefetchers. The dynamic destination mechanism renders the highest performance.

The Stride prefetchers slightly benefits from all dynamic mechanisms. The dynamic destination, for example, reduces cache pollution, while the aggressiveness can help save memory bandwidth by not over-prefetching.

The Stream prefetcher's performance is lower than the other prefetchers. This is caused by the prefetcher not using sufficient memory bandwidth. The Stream perfetcher's performance lowers whenever any dynamic mechanism is used.

6.3.4 Results Multi Core

In this section, we show our experimental results while running a multi-core system.

We report memory bandwidth, cache misses, cache pollution and a classification

in terms of used, late and unused prefetches for every prefetch issued, and Instructions per Cycle (IPC) of the overall system calculated as:

$$\frac{\textit{Total number of instructions}}{\textit{Total number of cycles}}$$

We experiment with several prefetcher configurations, as shown in Table 6.4.

Configuration 1 is a processor with no prefetcher in any cache level. Configuration 2 is a standard configuration for current in-order processors. In configuration 3, we enable the dynamic aggressiveness feature only in the L1 data cache prefetcher. In configuration 4, we instead enable the dynamic destination feature. And in configuration 5 the dynamic aggressiveness is enabled for all prefetchers in the system. Finally, configuration 6 uses the dynamic aggressivenes features for all prefetcher levels and the dynamic destination for the L1 data cache prefetcher.

Results from the rest of the section have a specific prefetcher configuration for the second and last level cache. We set a Nextline prefetcher for the second level cache and a Stride prefetcher for the last level cache. This configuration is one of the most performing ones seen in our experiments, which offers a view of possible performance gains for the dynamic mechanisms.

6.3.4.1 Performance

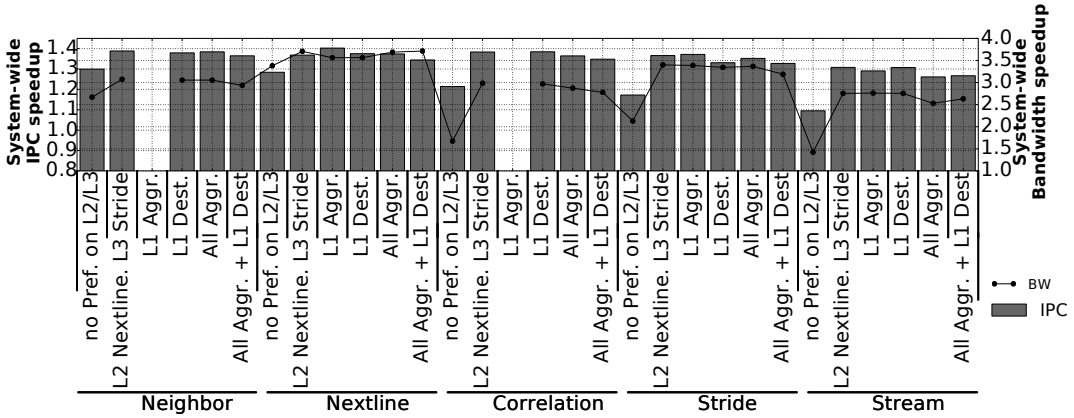


FIGURE 6.3: System-wide IPC and system-wide bandwidth for several configurations with the dynamic mechanisms with respect to no prefetching on L1, L2, nor L3 cache. Due to implementation limitations, Neighbor and Correlation prefetchers cannot dynamically adapt their aggressiveness. Their degree and distance parameters are not reconfigurable.

Figure 6.3 shows the speed-up in terms of execution time and memory bandwidth of different prefetcher configurations with respect to no prefetching.

We observe that the main cause of performance increase is using a prefetcher on the L1 cache, which can speed-up execution time by 28% and up to 65% with the Stream prefetcher offering the least performance increase, while the Neighbor

prefetcher increasing performance the most. As explained previously, the Stream prefetcher memory bandwidth is low compared to the other prefetchers.

The Neighbor prefetcher performance degrades a 2.0% when enabling dynamic destination. This performance degradation is caused by a higher penalty accessing data in L2 or L3 caches, which is higher than the possible benefit of reducing the pollution.

The Nextline prefetcher does not lose performance when enabling any dynamic mechanism. The dynamic aggressiveness mechanism improves performance by 3.2%. This is because the Nextline prefetcher reduces cache misses by bringing as many cache lines as possible before these cache lines are needed. Therefore, the dynamic aggressiveness mechanism chooses a performing configuration for the aggressiveness of the prefetcher while the dynamic destination mechanism reduces pollution in the L1 cache.

The Correlation prefetcher obtains a speed-up by adding a L2 and L3 prefetchers. When the dynamic aggressiveness mechanism is enabled, performance degrades for 2% and 4%. This is caused by a higher miss cache rate in the L2 and L3 caches (3% higher miss cache rate when only the dynamic aggressiveness is enabled and 7% when the dynamic destination is enabled), while a lower aggressive is set due to the added pollution from the L1 prefetcher with the dynamic destination enabled.

The Stride prefetcher maintains performance across the different prefetcher configurations. The only exception is when both dynamic mechanisms are enabled, which leads to performance degradation due to increased pollution. Pollution is increased by 5% in the L3 cache with respect to only enabling the dynamic aggressiveness mechanism. Stride prefetcher shows a lower performance than Nextline prefetcher due to its lower memory bandwidth usage.

The Stream prefetcher suffers from a performance degradation when the dynamic aggressiveness mechanism is enabled, going from 1.30x speed-up to a 1.26x speed-up due to the mechanism choosing a less aggressive configuration. The Stream prefetcher shows a lower performance than Stride prefetcher due to lower memory bandwidth usage.

In the Figure 6.3, we observe that there is a trade-off between memory bandwidth and performance. Enabling our dynamic mechanisms, the same performance can be achieved while decreasing memory bandwidth, cache misses, and cache pollution.

6.3.4.2 Cache misses

Figure 6.4 shows system-wide cache misses across all the cache in order to have a perspective of how data prefetchers affect the entire system.

Cache misses are classified in 3 categories: (1) Half miss, a prefetch was issued, but the prefetch has not arrived to the cache yet; (2) Due to prefetcher, cache misses

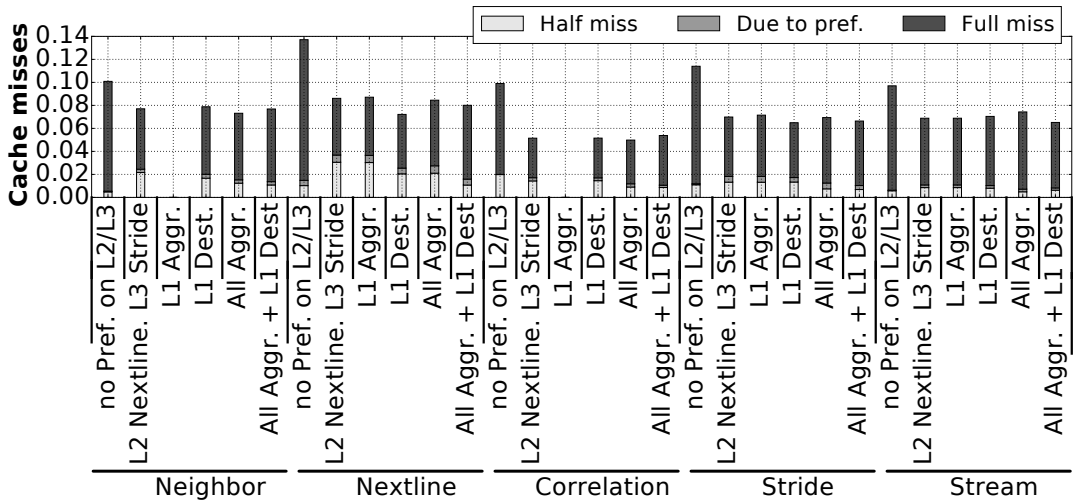


FIGURE 6.4: Overall cache misses in the system. Cache misses in L1, L2 and L3 cache are taken into account. We report half misses: miss in cache, but it hits on the MSHR; due to prefetcher: cache misses that are caused by the prefetcher itself; full misses: miss in cache and the data must be bring from another location.

caused by a prefetcher overwriting cache blocks for prefetched data; and (3) Full miss, the data is in memory and must be brought.

Cache misses are very similar across the different prefetcher configurations. We can see that misses due to the prefetcher are mainly seen in simple prefetchers. Overall, Nextline is the prefetcher that causes most system-wide misses compared to others (even compared to simple prefetchers). This is caused by the high number of issued prefetches to consecutive cache lines.

In terms of the dynamic mechanisms, increasing the aggressiveness of the L1 prefetcher for the Nextline, Stride and Stream prefetchers does not increase performance since they do not reduce cache misses. Therefore, their aggressiveness level is not highly increased due to other constraints such as pollution or congestion. As we can see with the Neighbor and the Correlation prefetchers, they obtain similar performance compared to the simpler prefetchers, while lowering bandwidth usage (see Figure 6.3).

When enabling the dynamic destination, L1 cache misses are increased, yet, we can see that the overall cache misses decrease. This is not highly reflected in terms of performance due to a higher latency access to the L2 and L3 caches but it should impact on overall power consumption.

6.3.4.3 Issued Prefetches

We measure how useful are the issued prefetches in Figure 6.5. We classify every prefetch issued by the L1 prefetcher into: (1) unused, when a prefetched block is not addresses by the application; (2) late, the system performs a demand access to an address, that address misses in the cache and hits on a prefetch register (MSHR);

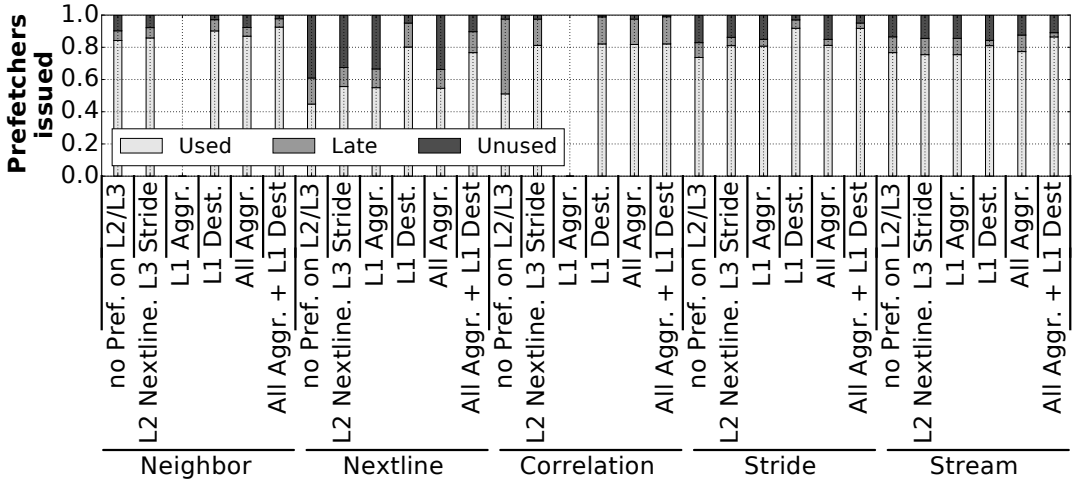


FIGURE 6.5: Classification of the issued prefetchers for different prefetcher configurations. They are classified and unused: the cache block prefetched was not used by the processor; late: the cache line was accessed before the cache line arrived to the cache and used: the prefetcher brought a cache line that was used in time.

and (3) used, when a prefetched block is addressed by the application. When a prefetched block is accessed it is classified as used.

In Figure 6.5, the *Unused* component is mostly present in the simple prefetchers due to their simplistic nature.

The Nextline prefetcher without dynamic destination can waste up to 40% of the issued prefetches. When we enable dynamic destination, the Nextline *Unused* prefetches decrease by up to 5% due to cache lines can be prefetched into an upper cache level to be reused in the future. The Neighbor and Correlation prefetchers lead to a better utilization of the prefetched cache lines since they bring in fewer cache lines and in a more efficient way. They waste up to 10% of the issued prefetches.

In terms of lateness, the prefetcher issuing late requests is Correlation. This behavior is highlighted when there are no prefetchers in the L2 nor the L3 cache. This is casued by the training phase of the Correlation prefetcher, adding a L2 or L3 cache prefetcher can help to reduce the overall latency, and therefore reducing the training phase, which helps to increase timely prefetches.

6.3.4.4 Cache Pollution

In Figure 6.6, we report the overall cache pollution of the system for the different prefetcher configurations. The figure shows the ratio between misses caused by having a prefetcher and the total misses of the system.

Pollution is higher with the Nextline and the Stride prefetchers, both of them are simple prefetchers. The Neighbor and Correlation prefetchers offer the best rate performance to pollution rate.

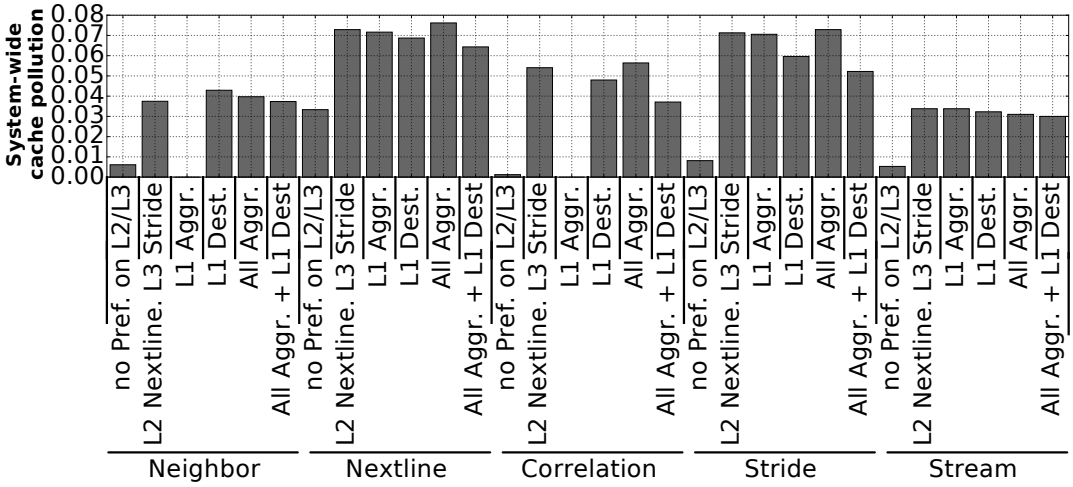


FIGURE 6.6: Cache pollution of the system for different prefetcher configurations. It is measured as the ratio between cache misses caused by having a prefetcher and the total misses of the system.

When evaluating the dynamic mechanisms, we see that dynamic aggressiveness does not affect pollution negatively, unless all the prefetchers in the cache hierarchy have the dynamic aggressiveness enabled (such the case of the Nextline and Stride prefetchers). Dynamic destination can help to reduce cache pollution since cache line can be stored in other cache levels.

6.4 Conclusions

Data prefetching in in-order cores has a major impact on the overall performance since it is a known technique to alleviate the evolving performance gap between processor and memory. There are several data prefetchers available, but research has been focused in out-of-order processors.

In this work we perform an exhaustive analysis of different data prefetchers in terms of performance, bandwidth and cache requirements. We implement 2 state-of-the-art dynamic mechanisms and evaluate them in our in-order core infrastructure.

Results show that there is a trade-off between complexity and memory bandwidth requirements. Simple data prefetchers have a higher memory bandwidth usage, which can be unaffordable for low-power processors. On the other hand, complex data prefetchers can be expensive in terms of area, which can be unaffordable for embedded processors.

Dynamically increasing the aggressiveness of the data prefetcher can increase performance at the cost of a higher memory bandwidth usage. While other mechanisms such as the dynamic destination can increase the efficiency of the prefetchers and the caches. Therefore, simple data prefetchers with dynamic mechanisms can

match the performance of complex data prefetchers while using less area, which can meet the requirements for embedded and low-power processors.

Chapter 7

CPU-GPU Power Distribution under a System Power Cap

7.1 Introduction

In recent years, data centers have started incorporating multiple diverse accelerators per node to provide efficient performance growth through specialization. However, complex heterogeneous systems with multiple discrete accelerators cannot afford to fully power all the devices simultaneously [47, 66].

Efficiently distributing power in heterogeneous systems with multiple discrete accelerators and varying power caps is a challenging problem. The growing amount of accelerators requires power distribution algorithms to be simple and scalable to ensure fast response times. Moreover, the latency of communicating control signals between discrete devices is usually very long; therefore, minimizing device communation is a must. On top of this, the proliferation of different accelerators (GPUs, FPGAs, ASICs, etc.) manufactured by different vendors imposes power management solutions to use generic knobs present in all kind of devices, such as Dynamic Voltage Frequency Scaling (DVFS).

TABLE 7.1: Previos work on distributing power among components in different scenarios.

Scope of work	Target workloads	Work
Core level	Mixed CPU workloads	[74, 55, 153, 38, 161]
Heterogeneous Chip	Single CPU-GPU workloads	[12, 88, 98, 78, 121, 122, 154]
Homogeneous System	Mixed CPU workloads	[157, 18, 166, 35, 134, 135]
Heterogeneous System	Multiple Mixed CPU workloads and CPU-GPU workloads	This work, [127, 62]

Previous works have studied how to efficiently distribute power under a limited power budget in different scenarios, and we summarize them in Table 7.1. All these

techniques are not directly applicable to systems with multiple discrete devices since single chips do not have information about the requeriments and the power budgets of the discrete devices. However, this does not apply to systems with multiple accelerators running multiprogrammed workloads and we summarize their differences with this work in Table 7.2.

TABLE 7.2: Comparison with the state of the art

Related work	Works for single CPU-GPU workloads	Works for multiple CPU-GPU workloads	Power cap in Prediction Failure
[12, 88, 98, 78, 121, 122, 154]	✓	✗	✗
[127, 62]	✓	✓	✗
This work	✓	✓	✓

This chapter presents *Adaptive Power Shifting for heterogeneous systems (APS)*, a technique that maximizes the performance of power-constrained heterogeneous systems by leveraging system information to distribute power among all the devices. APS advocates for a hardware/software power distribution mechanism that carefully balances efficiency and fairness by leveraging dynamic load information. In contrast with previous works, APS is agnostic of the devices (CPU, GPU, or other accelerators), it uses a simple and scalable heuristic that requires minimal communication between devices, it works for single applications and multiprogrammed workloads, and it can be implemented in current systems without any hardware modification, as we demonstrate with the deployment on a real OpenPOWER system with 2 CPUs and 4 GPUs.

APS uses device utilization as the metric for optimization, which is very well suited for heterogeneous systems with devices with different power and performance characteristics, and it is also a very accurate proxy for power consumption. At execution time, APS monitors the power consumption and the utilization of all the devices in the system and dynamically shifts the power between them. To maximize performance, APS enables devices that are more utilized to have a higher power budget than devices that are less utilized. APS also captures changes in the power cap and quickly re-assigns the power budgets of the devices to minimize the time the system is over the specified power cap and to reduce the time a device is assigned an underperforming power budget.

Our results demostrate that current solutions in modern systems can lead to over or under provisioning the power budget of individual devices in a power-constrained heterogeneous system. APS improves performance over static power distributions up to 15.9% for single CPU-GPU applications and up to 20.5% for multiprogrammed workloads. APS improves performance up to 14.9% state-of-the-art solutions [62, 157].

7.2 Motivation

To highlight the challenges of distributing power consumption under a power cap, we run a CPU-bound workload and a GPU-bound workload on an OpenPOWER system with 2 CPUs and 4 GPUs. The CPU-bound workload is composed of a DAXPY kernel in the CPUs and a Speckle Reducing Anisotropic Diffusion (SRAD) kernel [31] in the GPUs. The GPU-bound workload is Tensorflow [1] training an Inception v3 neural network, which uses the GPUs for the main computation and the CPUs for the data transfers, updating the weights, and offloading work to the GPUs.

As explained in Section 2.6, the IBM OCC of the OpenPOWER architecture provides a mechanism to adjust the power budget of the group of CPUs and the GPUs when a power cap is introduced. Following this mechanism, we implement three static power distributions:

- Fairness (Fairness). The system power cap is equally distributed among all devices in the system.
- CPU Priority (CPUprio). The system power cap is distributed between CPUs and GPUs with different weights in favor of the CPUs.
- GPU Priority (GPUprio). The system power cap is distributed between CPUs and GPUs with different weights in favor of the GPUs.

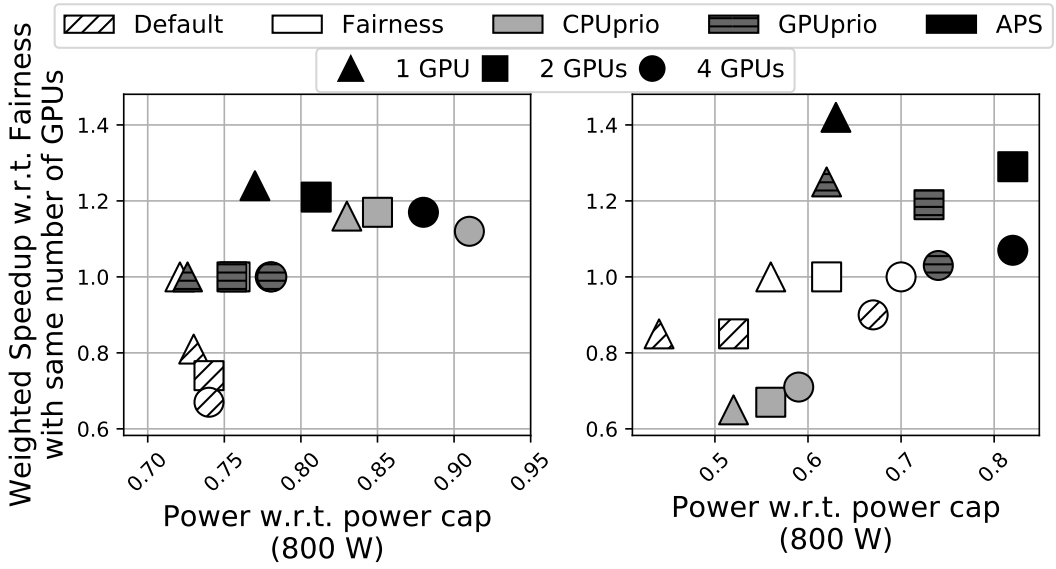


FIGURE 7.1: Execution of DAXPY in the CPUs and SRAD in the GPUs (left), and Tensorflow training an Inception v3 neural network (right).

Figure 7.1 compares the default power distribution of the OpenPOWER architecture (Default[1]), the three static power distribution techniques Fairness, CPUprio, and GPUprio, and APS in a system with a power cap of 800W and different numbers of GPUs used. The y-axis shows the weighted speedup with respect to the Fairness technique and same number of GPUs, and the x-axis shows the average power consumption of the main computation phase normalized to the system power cap of 800W.

The left plot in Figure 7.1 shows the results for the CPU-bound workload running a DAXPY kernel in the CPUs and a SRAD kernel in the GPUs. The default power distribution degrades the performance with respect to Fairness because it drastically reduces the frequency of the CPUs. CPUprio significantly improves the weighted speedup with respect to Fairness by 16.5%, 16.7%, and 12.0% for 1, 2, and 4 GPUs, respectively. Reducing the power assigned to the GPUs does not degrade the performance of SRAD, while the extra power in the CPUs boosts the performance of DAXPY.

Finally, APS improves the weighted speedup with respect to CPUprio by 6.3%, 4.8%, and, 3.8% for 1, 2, and 4 GPUs, respectively. APS shifts power to the GPUs when the CPUs enter an idle state, boosting GPU applications and improving the overall performance. Also, APS reduces the average power consumption with respect to CPUprio (between 2.8% and 5.1%) because it lowers the frequency of the GPUs when they are in a low utilization phase.

The right chart in Figure 7.1 shows the results for the GPU-bound workload running Tensorflow. CPUprio degrades performance up to 34.6% with respect to Fairness because the computation in the GPUs is drastically slowed down, affecting the overall performance even if the data transfers to the GPUs and the weight updates in the CPUs are done at the maximum possible performance. The default power distribution lowers the frequency of the GPUs more than Fairness but not as much as CPUprio, resulting in a slowdown of 15.0%. In contrast, GPUprio achieves speedups of up to 24.7% (with 1 GPU) by accelerating the computation phase in the GPUs.

APS achieves the best performance for this workload by dynamically shifting power between the CPUs and the GPUs. In the phases that perform data transfers and weights update, APS shifts power to the CPUs while, in the compute phases, the power is shifted to the GPUs. As a result, APS improves the weighted speedup with respect to GPUprio by 17.0%, 10.2%, and a 3.9% for 1, 2, and 4 GPUs, respectively.

In conclusion, APS outperforms static techniques due to a better utilization of the available power. Static distributions cannot respond to the changing power requirements of heterogeneous applications that use different devices in program phases. Thus, an intelligent dynamic power distribution specially targeted to heterogeneous

[1]There is no public information on the default behavior of the OpenPOWER architecture. Based on the experimental results, the default behavior sets very conservative power budgets to all the devices.

systems with multiple discrete accelerators is needed to maximize the system power-performance efficiency. To this end, APS provides a hardware/software power distribution mechanism [166, 63, 128] that carefully balances efficiency and fairness by leveraging dynamic load information, increasing the fidelity of the power distributions [156]. Thanks to these properties, APS outperforms other dynamic techniques such as Tangram [127] and market-based allocation [62].

7.3 Adaptive Power Shifting for Multi-Accelerator Heterogeneous Systems

APS maximizes the performance of power-constrained heterogeneous systems by shifting power among devices based on their utilization. Devices with high power budgets and low utilization cannot fully use their power budget, thus APS shifts power to devices with a higher utilization.

7.3.1 Overview

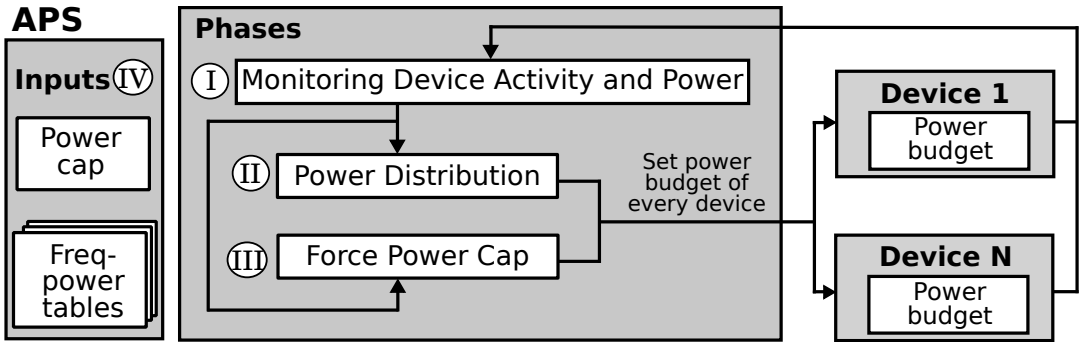


FIGURE 7.2: Overview of a system using APS.

Figure 7.2 shows a system using APS. APS is independent of the underlying devices and of the type of workload, which can be composed of any number of applications using any combination of devices during their execution.

APS constantly monitors the power consumption and utilization of all the devices in the *Monitoring Device Activity and Power* phase (Ⓘ in Figure 7.2). These readings are done through the standard interfaces of each device. As explained in Section 3.4, in our setup we use perf for the CPU utilization, in-band OCC readings for the CPU power consumption, and NVML for the utilization and the power consumption of the GPUs.

After gathering multiple samples of the power consumption and utilization, the algorithm triggers a *Power Distribution* phase to distribute the available power among the devices based on their utilization (Ⓘ in Figure 7.2). If the total power consumption surpasses the power cap, APS goes to a *Force Power Cap* phase to meet

the power cap as soon as possible (Ⓘ in Figure 7.2). The power cap is an input to APS set by the administrator or higher-level power manager (e.g., rack or cluster level). If no power cap is specified, it is set to the node Thermal Design Power (TDP).

The frequency-power (*freq_power*) tables are key elements of the design of APS (Ⓘ in Figure 7.2). APS uses a pre-generated freq_power table for every device type in the system. The freq_power table of each device contains the maximum observed power consumption for all the frequencies, and is generated offline by running a stressmark that intensively uses the hardware and has a high power consumption. In our system we use as stressmarks a DAXPY for the CPUs and a DGEMM for the GPUs. With the freq_power tables APS can quickly set a frequency level that honors the power budget assigned to a device and avoid iterative searches for a budget compliant frequency. These freq_power tables are stored in software, but they could be implemented in hardware since they are small (one entry per DVFS level).

Note that the design of APS is open to using other hardware knobs other than DVFS to adjust the power budgets of the devices. The only requirement to use other hardware knobs is to create a proxy that correlates the configuration values of a knob with the power consumption of the device under that configuration, similarly to the freq_power tables. To combine multiple knobs [133], APS would require a table that contains every combination of the values of the knobs and their power, as well as a mechanism that decides the most convenient configuration when the same power can be achieved with different combinations of the knobs. This scenario with multiple knobs is out of the scope of this work.

APS does not require any modifications in the source code of the applications and it does not instrument the offloading of the GPU kernels nor any application-level interaction between the CPUs and GPUs. Instead, the interaction between the devices is modeled after their utilization and power consumption, which is measured at system level.

7.3.2 APS Phases

APS relies on 3 phases, as shown in Figure 7.2: Ⓘ monitor the system, Ⓘ distribute the available power, and Ⓘ force the power cap when the total system power exceeds it.

7.3.2.1 Monitoring Device Activity and Power

The monitoring phase is responsible of monitoring the system status and triggering the Power Distribution and Force Power Cap phases. The system status includes power consumption and utilization of all the devices, which is computed as the percentage of cycles that a device has been executing a process during the sampling period. The sampling period in our experimental setup is 200ms, as discussed in Section 7.4.1.

```
1 avgSamples = N
2 numSamples = 0
3 while True do:
4   TotalUtil = 0.0
5   TotalPower = 0.0
6   for all devices in the system:
7     freq[device]  ← read device frequency
8     util[device]  ← read device utilization
9     power[device] ← read device power
10    TotalUtil  += util[device]
11    TotalPower += power[device]
12  ++numSamples
13  if numSamples == (avgSamples-1):
14    PowerDistribution(freq,util,power,TotalUtil)
15    numSamples = 0
16  else if TotalPower > PowerCap:
17    ForcePowerCap(freq,power,TotalPower);
```

LISTING 7.1: Algorithm of the monitoring phase.

The monitoring phase iterates indefinitely over the algorithm shown in Listing 7.1. After the initialization, the algorithm gathers the current frequency, the utilization and the power consumption of all the devices (lines 7 to 9 in Listing 7.1). The frequencies, the utilization and the power consumption of the devices are stored in three vectors, `freq`, `util` and `power`, which contain one element per device. In addition, the accumulated utility and power consumption of all the devices is calculated (lines 10 and 11) on `TotalUtil` and `CurrentPower`, respectively. When a number of samples has been taken (5 in our setup), the monitoring algorithm triggers the Power Distribution phase (lines 13 to 15). When the total system power exceeds the power cap, the monitoring algorithm triggers the Force Power Cap phase (lines 16 and 17) to reduce the power budget of every device in order that the power cap is respected.

7.3.2.2 Power Distribution

```
1 Require: freq[], util[], power[], TotalUtil, PowerCap
2 for all devices in the system:
3   RelativeUtil[device] = util[device] / TotalUtil
4 while PowerHeadroom > 0 and PowerHeadroom > minPowerStep:
5   CurrentPower = 0
6   MaxFreqDevices = devices with freq == maxFreq
7   N = Number of devices in MaxFreqDevices
8   if N == Number of devices in the system: break
9   for all devices in the system:
10    TargetPower = RelativeUtil[device] × PowerCap
11    StressmarkPower = freq_power_table[freq[device]]
12    RatioDevice = stressmarkPower / power[device]
13    PowerBudget = TargetPower × RatioDevice
14    freq[device] = freq_power_table[PowerBudget]
15    CurrentPower = CurrentPower + TargetPower
16  PowerHeadroom = PowerCap − CurrentPower
17 apply_frequencies()
```

LISTING 7.2: Algorithm of the Power Distribution phase.

The Power Distribution phase is responsible of setting the power budgets of all the devices in the system according to their utilization. The algorithm that implements this phase is shown in Listing 7.2. The input to the algorithm is the data gathered by the monitoring algorithm and the total system power cap (*PowerCap*).

The Power Distribution algorithm starts by computing the relative utilization *RelativeUtil* of every device with respect to the total utilization of the system (lines 2 and 3). To do so, it divides the utilization of each device by the accumulated utilizations of all the devices *TotalUtil*. For instance, if a system with 6 devices has a *TotalUtil* of 400% and one of the devices has an utilization of 80%, its relative utilization is $0.8/4.0 = 0.2$. Then, the algorithm enters a loop that iterates until there is no power headroom. To control this condition, the *PowerHeadroom* is the difference between the total system power cap and the sum of the power consumption of all the devices in the system, and the *minPowerStep* is the minimum possible increase in power consumption caused by the minimum possible increase of the frequency of any of the devices of the system. The *minPowerStep* avoids re-triggering the outer loop constantly. In our setup *minPowerStep* is 5 Watts, corresponding to the minimum possible increase of 70 Hz in the CPU frequency. Thus, when the *PowerHeadroom* is lower than 5 Watts, it is impossible to assign the *PowerHeadroom* to any device and the Power Distribution phase finishes.

The outermost loop starts by setting the *CurrentPower* to zero (line 5), which is a variable that is going to accumulate the power consumed by all the devices running at a certain frequency. Then, the algorithm checks if all the devices are running at their maximum frequencies (lines 6 and 7). If all the devices are already running at their maximum frequencies, the Power Distribution phase finishes (line 8). Otherwise, the innermost loop iterates over all the devices (line 9). For each device, its *TargetPower* (line 10) is calculated as $RelativeUtil[device] \times PowerCap$. The *TargetPower* of a device represents the maximum power the device can consume, based on the relative utility of the device in the system and the power cap. For instance, for a power cap of 1000W, a device with a relative utilization of 20% will have a *TargetPower* of $0.2 \times 1000W = 200W$. Note that, if two different devices have the same relative utilization, they will get the same portion of the power cap as *TargetPower*. Then the *StressmarkPower* of the device is obtained by consulting the freq_power table (line 11). The *StressmarkPower* represents the power consumption of the device running the stressmark at the current frequency. Then the algorithm calculates the *RatioDevice* of the device (line 12) as $StressmarkPower/power[device]$. The *RatioDevice* represents how much power an application consumes running at a certain frequency compared to the stressmark running at the same frequency. This is done because the freq_power tables store frequency and power values generated with stressmarks that are CPU and GPU intensive, but most workloads have different frequency-power characteristics than the stressmark. For instance, if an

IO-intensive application has a power consumption of 75W running at a certain frequency and the stressmark consumes 150W running at the same frequency, the *RatioDevice* is $150W/75W = 2$. The goal of the *RatioDevice* is to scale up the *TargetPower* of the device before looking for its target frequency in the freq_power table, therefore, the optimal frequency for that device is found in a single step, honoring the power budget and minimizing the power headroom without the need to iteratively refine the power budget. To do so, the algorithm calculates the *PowerBudget* as $TargetPower \times RatioDevice$ (line 13). This *PowerBudget* is searched in the freq_power table for that device type (line 14), and APS selects the highest frequency that satisfies the *PowerBudget*. Finally, the *TargetPower* of the device is accumulated in the *CurrentPower* (line 15). Once the innermost loop has calculated the frequencies of all the devices, the *PowerHeadroom* is calculated as $PowerCap - CurrentPower$ (line 16).

At the end of the algorithm, when all the frequencies of all the devices have been calculated and there is no power headroom available, the new frequencies are applied to all the devices via the DVFS controller (line 17). The new frequency of a device limits its power consumption to its calculated power budget until the monitoring phase triggers again the Power Distribution phase.

Note that this algorithm could be implemented as a reinforcement learning problem where agents or devices take actions by increasing or reducing their running frequencies, and the reward is based on how much power headroom is left or what is their individual throughput. The viability of this approach depends mostly on the trade-offs between the quality of the decisions and the complexity, which can lead to large overheads compared to simpler heuristics. We leave the study of such an approach for future work.

7.3.2.3 Force Power Cap

```
1 Require: freq[], power[], CurrentPower, PowerCap
2 while CurrentPower > PowerCap:
3   NonMinFreqDevices = devices with freq > minFreq
4   N = Number of devices in NonMinFreqDevices
5   if N == 0: break
6   PowerToReduce = (CurrentPower - PowerCap)
7   DevicePowerToReduce = (PowerToReduce / N)
8   CurrentPower = 0
9   for device in NonMinFreqDevices:
10    PowerBudget = power[device] - DevicePowerToReduce
11    freq[device] = freq_power_table[PowerBudget]
12    CurrentPower += PowerBudget
13  MinFreqDevices = devices with freq == minFreq
14  CurrentPower += sumPower(MinFreqDevices)
15 apply_frequencies()
```

LISTING 7.3: Algorithm of the Force Power Cap phase.

This phase forces the power cap when the total power consumption exceeds it. The goal of this phase is to reduce the power consumption as soon as possible to

minimize the time over the specified power cap. For this reason, the Force Power Cap phase quickly reduces the power consumption equally from all the devices, and APS relies on the next Power Distribution phase to optimize the power budgets of the devices under the new power cap. The algorithm that implements the Force Power Cap phase is shown in Listing 7.3. Note that this phase can be triggered at any moment of the execution, as the power cap can be exceeded at any time due to applications entering a phase with higher power consumption, responses to thermal or energetic emergencies, etc.

Therefore, distributing the power in this phase can lead to measurements not representative of the workloads running in the devices (the Power Distribution phase measures 5 samples before distributing the power).

The Force Power Cap algorithm reduces the power budget of the devices that are not running at their minimum frequency until the sum of the power of all devices is less than or equal to the power cap. First, the algorithm selects the devices that are not running at their minimum frequency (lines 3 to 4 in Listing 7.3) and, if all of the devices are running at their minimum frequencies, the algorithm finishes (line 5). Otherwise, APS calculates the amount of power that needs to be reduced, *PowerToReduce*, as $CurrentPower - PowerCap$ (line 6). The *PowerToReduce* is equally divided among these devices by calculating the *DevicePowerToReduce* in (line 7). Then, the *CurrentPower* is set to zero (line 8), and the algorithm enters a loop that reduces the power of all the devices that are not running at their minimum frequency (line 9).

For each device that is not running at its minimum frequency, its new power budget (*PowerBudget* in line 10) is computed by subtracting the *DevicePowerToReduce* from the current power consumption of the device. Then, the device is assigned a new frequency by looking up its new power budget in the freq_power table of that device type and selecting the highest frequency that satisfies it (line 11), and the *PowerBudget* of the device is accumulated on the *CurrentPower* (line 12), which controls the total power of the system. After the innermost loop, the power consumption of the devices that are running at their minimum frequency is also accumulated on *CurrentPower* (lines 13 and 14).

Finally, the frequencies assigned to all the devices are applied (line 15). The new frequencies of all devices ensure that the total power consumption honors the power cap or that all the devices are running at their lowest frequency. If the power behavior changes and exceeds the power cap, the monitoring phase is in charge of triggering this phase again.

APS can use this algorithm thanks to the information of the freq_power tables. Previous work [127] need to lower the frequencies of all devices to their minimum possible frequency level since their mechanism is not aware of the power consumption of every frequency level.

7.3.3 Algorithm Walkthrough for a CPU+GPU

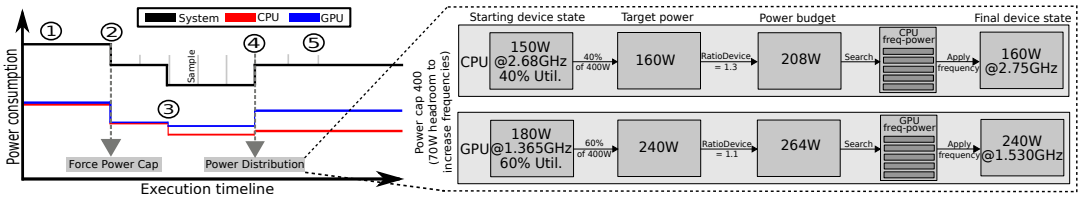


FIGURE 7.3: Synthetic example of APS distributing power in a system with 1 CPU and 1 GPU.

Figure 7.3 shows an example of APS managing a system with 1 CPU and 1 GPU (red and blue lines, respectively). The total power consumption is represented by the sum of both devices (black line). Initially (①), each device is running a workload and no power cap is specified. At ②, a power cap of 400W is introduced. APS invokes the Force Power Cap phase to reduce the power consumption of each device. As a result, the CPU and GPU are set to a lower frequency and their power consumption decreases.

The applications running in the CPU and the GPU enter a phase with a lower power consumption at ③. At ④ APS invokes the Power Distribution phase to set the power budget of the CPU and GPU based on their utilization from the last 5 samples. Since the total system power consumption is lower than the power cap, APS has power headroom to increase the frequencies of the CPU and GPU. The right part of Figure 7.3 shows how this process is done with an example.

At ④, the total system power consumption is 330W, and the CPU and GPU consume 150W and 180W with an utilization of 40% and 60%, respectively. Thus, APS has a power headroom of 70W to boost the performance. The first step is to calculate the *Target power*, that represents the maximum power budget for a device within a power cap. In this example, the target power for the CPU and GPU is 160W and 240W, respectively.

Next, APS calculates the *power budget* of every device. To do so, APS relies on a freq_power table that is generated with a stressmark for that device. Therefore, if an application has a lower power consumption than the stressmark at a given frequency and APS applies this same frequency, the device will not fully use its power budget. In the example, the power consumption of the CPU stressmark running at 2.68GHz is 200W, and the GPU running the stressmark at 1365MHz consumes 200W. Consequently, the RatioDevice for the CPU is $200/150 = 1.3$, and for the GPU the RatioDevice is $200/180 = 1.1$.

In the next step APS looks for the highest frequency that honors the *power budget* in the freq_power table of each device. The power values searched in the freq_power tables of the CPU and the GPU are 208W and 264W, respectively. Note that these power values are higher than the *Target power* of each device, but neither the CPU nor

the GPU will consume that much power because the workloads they are executing have a lower power consumption than the stressmarks, as reflected by a *RatioDevice* higher than 1.

Finally, after applying the new frequencies in the CPU and the GPU, the power consumption of both devices is the same as their *Target power* (note that it could be a bit lower), and the available power is used much more efficiently than in the initial state at ④. After the Power Distribution phase, the power consumption of the devices does not change. Therefore, from ⑤ until the end of the execution, APS keeps monitoring the system but does not change the frequencies.

7.4 Evaluation

This section describes the implementation used to evaluate APS and the baselines used in this chapter. Then, we evaluate APS in a system running a single heterogeneous application (Section 7.4.4) and multiprogrammed workloads (Section 7.4.5). Section 7.4.6 compares APS with multiple state-of-the-art techniques. Finally, Section 7.4.7 discusses different implementation approaches for APS.

7.4.1 Implementation

We implement APS as a process running in the system. The process measures execution time and utilization with perf [102] and does in-band power consumption readings from Linux to the OCC [136] for the CPUs and with the NVIDIA Management Library (NVML) [115] for the GPUs. The APS sampling rate is limited by the overheads of the measurement tools. Reading the power consumption and utilization of the CPUs and the GPUs takes on average 50ms and 105ms, respectively, and can be affected by the load of the system. Therefore, we set a sampling time of 200ms, which allows APS to recognize irregular behaviors.

7.4.2 DVFS Capabilities

The CPUs in our system have 43 frequency levels (from 3.00GHz to 2.30GHz). Lowering the frequency level has a linear impact on power and performance for a DAXPY stressmark as shown in Figure 7.4 (left). The GPUs can be set to run from 1.53GHz to 135MHz through the NVML. We limit APS to use frequencies higher than 500MHz, since lower frequencies have a significant negative impact on performance, as shown in Figure 7.4 (right).

To ensure that the physical power management of the system does not intervene in our experiments, we set an unrealistic high power cap that is never reached.

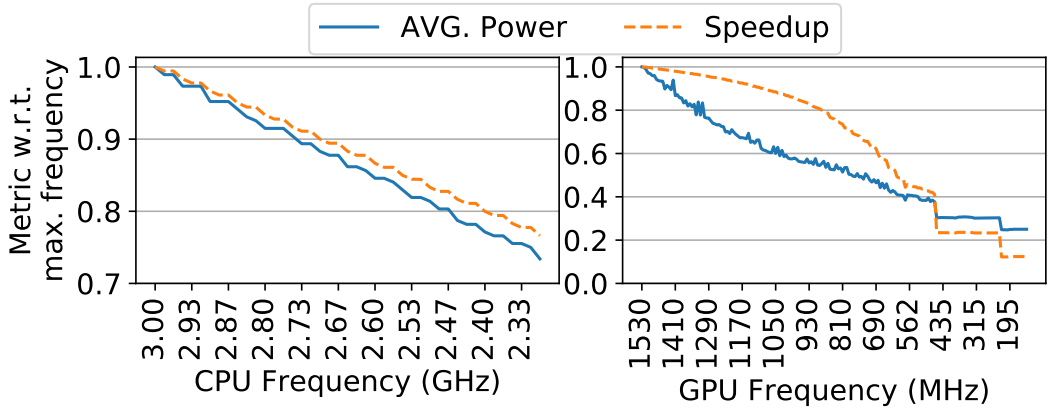


FIGURE 7.4: Average power consumption and speedup when running a stressmark in a CPU (left) and GPU (right). CPU and GPU stressmarks are DAXPY and DGEMM, respectively.

TABLE 7.3: Static power distributions for multiple devices in a system with 2 CPUs and 4 GPUs.

Power Cap (W)	Fairness		CPU Prio.		GPU Prio.	
	CPUs	GPUs	CPUs	GPUs	CPUs	GPUs
1000	166×2	166×4	200×2	150×4	75×2	212×4
900	150×2	150×4	200×2	125×4	75×2	187×4
800	133×2	133×4	200×2	100×4	75×2	162×4
700	116×2	116×4	200×2	75×4	75×2	137×4
600	100×2	100×4	200×2	50×4	75×2	112×4

7.4.3 Baselines

Table 7.3 shows the power budgets of the devices for each policy in a heterogeneous system with 2 CPUs and 4 GPUs and power caps of 1000, 900, 800, 700, and 600W. The static policies set the power budget of the devices to fixed values, without considering their utilization, which can lead to underperforming power distributions. For instance, if a workload is CPU intensive and the power distribution is not CPUprio, CPUs run at a low frequency, leading to an overall slowdown. Moreover, if an application has CPU and GPU phases, static power distributions are unable to boost specific accelerators at the appropriate time.

Once the controllers have assigned power budgets to all the devices, these can adjust their running frequency inside a range of available frequencies that honor their power budget.

We compare APS against these 3 static power distributions (Fairness, CPUprio, GPUprio) and a profile-based approach. We do not include the default OCC of the OpenPOWER system because, as seen in Section 7.2, it is very conservative and it is outperformed by all the other approaches.

For the Fairness, CPUprio, and GPUprio power distributions we set the power budgets of all the devices as specified in Table 7.3. The power budget of the devices is constantly monitored and, if needed, their frequencies are corrected to honor their

power budgets. If a device has available power headroom we increase its running frequency.

The profile-based static approach (Static Prof) finds the best power distribution for mixed workloads using offline profiling of the workloads and computing a Mixed-Integer Linear Programming (MILP) model. We measure offline the maximum power consumption and execution time of every benchmark individually for every available frequency in our system, and then we apply a MILP model (specified in Model 7.1) to select the frequencies of the devices that minimize the overall slowdown of all the benchmarks. At the beginning of the execution, Static Prof sets the frequencies of every device to the frequencies obtained by the model.

$$
\begin{array}{ll}
\text{minimize} & \sum I_{df} \times Slowdown_{df} \\
\text{subject to} & \sum_{i=0}^{6} I_{if} = 1 \\
& \sum I_{df} \times Power_{df} \leq \text{System power cap} \\
& I_{df} = \left\{ \begin{array}{c} 1 \text{ if device } d \text{ runs at frequency } f \\ 0 \text{ Otherwise} \end{array} \right\}
\end{array}
$$

MODEL 7.1: Mixed-Integer Linear Programming to set the frequency of the devices when using Static Prof. $Power_{df}$ is the maximum power measured in an offline profiling when the device d runs at frequency f. $Slowdown_{df}$ is the slowdown when the device d runs at frequency f with respect to the device d running at its highest frequency.

7.4.4 Single CPU-GPU Applications

We evaluate APS with Tensorflow training an Inception v3 neural network using all the CPUs and GPUs in the system. The GPUs perform the main computation phase, but the CPU performance is relevant for (1) updating the weights of the neural network, (2) transferring data between CPUs and GPUs, and (3) doing I/O operations while GPUs are idle.

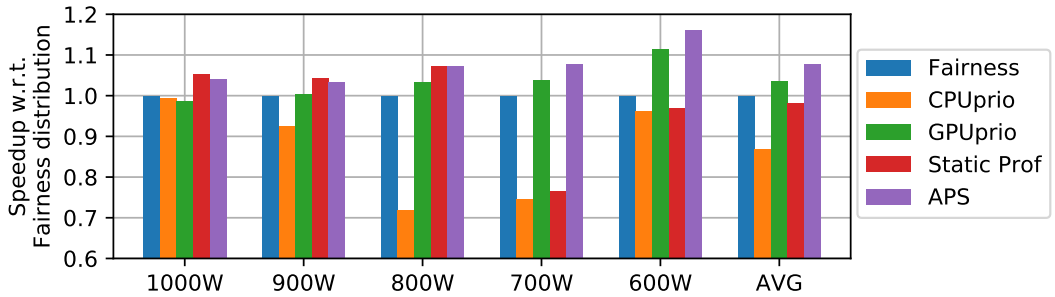


FIGURE 7.5: Speedup of Tensorflow training an Inception v3 network with different power distributions and power caps.

We experiment with Fairness, CPUprio, GPUprio, Static Prof, and APS with power caps of 1000, 900, 800, 700, and 600W. Results are shown in Figure 7.5. The y-axis shows the speedup with respect to Fairness and different power caps are represented in the x-axis. When no power cap is applied, this experiment consumes up

to 1660W (1090W on average). Fairness degrades performance with respect to no power cap by 14.4%, 18.3%, 25.5%, 33.4%, and 48.1% for power caps of 1000, 900, 800, 700, and 600W, respectively.

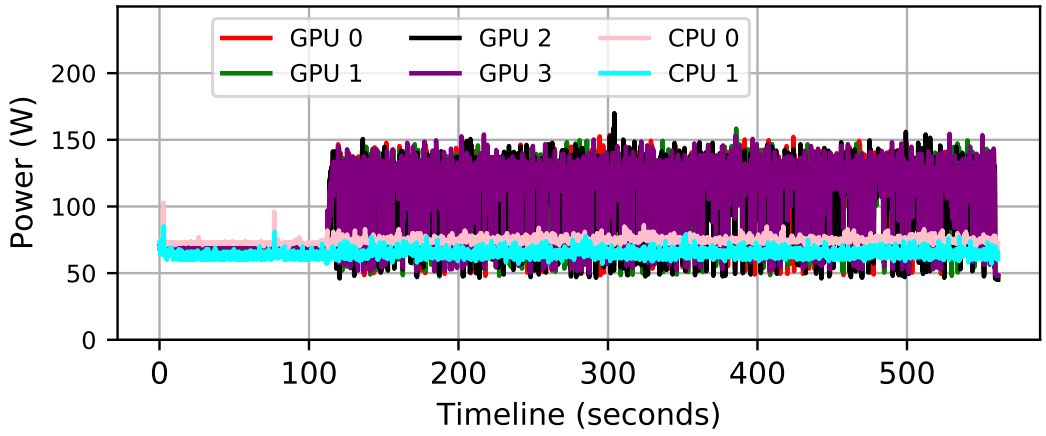


FIGURE 7.6: Power consumption over time of the all the devices running Tensorflow under a Fairness power distribution.

Figure 7.6 shows a timeline of the power consumed by the CPUs and the GPUs running Tensorflow under Fairness with a power cap of 900W. The timeline shows a stable power consumption by the CPUs of between 59W and 75W in the whole execution. In contrast, the power consumption of the GPUs varies widely over time, with high and low power spikes of 50W to 150W. These power spikes are caused by the characteristics of the workload in different program phases, since the power budgets and the frequencies of the GPUs are constant under Fairness. The average duration of the GPU phases with low and high power consumption are 404ms and 470ms, respectively.

Figure 7.5 shows that APS and GPUprio outperform Fairness. This workload relies on GPUs for the main computation; therefore, the performance increases when the power distribution favors the GPUs. APS can respond to this behavior dynamically based on the power consumption and the utilization of the devices. Note that, the lower the power cap, the higher the speedup for APS and GPUprio. This happens because, as shown in Figure 7.4 in the GPU power-performance curve, the power consumption drops faster than the performance. Thus, the performance improvements of APS and GPUprio with respect to Fairness and CPUprio are higher as we lower the power cap. With a power cap of 600W, APS and GPUprio outperform Fairness by 15.9% and 11.5%, respectively. These speedups come from using higher frequencies for the GPUs, and the extra 4.4% of APS is due to running the CPU phases at a higher frequency. APS and GPUprio run the GPUs at a higher frequency than Fairness and APS further improves performance by running the CPUs at a higher frequency for the data transfers, the weight updates, and the I/O phases.

Notice that the data transfers and the execution of the GPU kernels are overlapped, as a result, there are no obvious program phases during the execution. However, the power consumption of the GPUs varies during the execution (as shown in Figure 7.6), which is exploited by APS to efficiently distribute power.

CPUprio assigns power budgets favoring the CPUs. With a power cap of 700W, Fairness runs GPUs at 900MHz while CPUprio sets their frequency to 643MHz. This causes large performance differences because the GPUs power-performance trade-off is much better at 900MHz than at 643MHz, as shown in Figure 7.4. For high and low power caps the performance differences are smaller. For 600W, Fairness sets a very low power budget for the GPUs, which results in similar frequencies to the ones used by CPUprio. For 1000W and 900W there is enough power for CPUprio to set a high GPU frequency similar than Fairness.

Static Prof achieves a performance improvement with respect to Fairness of 5.2%, 4.3%, and 7.3% for power caps of 1000W, 900W, and 800W, respectively. For lower power caps of 700W and 600W, Static Prof has a performance degradation with respect to Fairness of 23.4% and 2.9%, respectively. When setting a power cap of 700W, Static Prof lowers the frequency of the GPUs to 690MHz, much less than the 900MHz set by Fairness, as a result, the performance is greatly degraded. For 600W, the frequency of the GPUs is similar in Static Prof and Fairness, 645MHz and 723MHz, respectively.

In terms of power consumption, GPUprio, Static Prof, and APS have a similar power consumption for a power cap of 1000W. Yet, as we lower the power cap to 600W, GPUprio consumes similar power than APS without providing performance benefits due to the CPUs running at a lower frequency, while Static Prof fails to use the available power headroom due to the high power phases of Tensorflow.

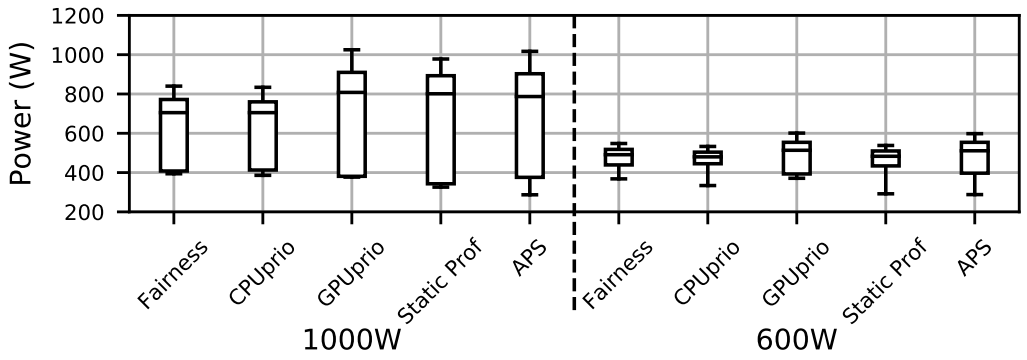


FIGURE 7.7: Power consumption of Tensorflow training an Inception v3 network with different power distributions and power caps. Boxes represent from the 25th to the 75th percentiles and whiskers from the 5th to the 95th percentiles.

Figure 7.7 shows the power consumption of the experiments with power caps of 1000W and 600W. The figure shows power consumption in the y-axis for all the

static and our dynamic power distributions in the x-axis. The boxes represent from the 75th to the 25th percentile, the band inside the boxes represent the median power, and the whiskers represent from the 95th to the 5th percentile. It can be observed that, with a power cap of 1000W, GPUprio, Static Prof, and APS have a similar power consumption. Yet, as we lower the power cap to 600W, GPUprio consumes similar power than APS without providing performance benefits due to the CPUs running at a lower frequency, while Static Prof fails to use the available power headroom due to the high power phases of Tensorflow.

7.4.5 Mixed Workloads (CPU-GPU Applications)

This section evaluates APS in a power capped system running mixed workloads composed of CPU and GPU programs. The mixed workloads are composed of 2 CPU benchmarks and 4 GPU benchmarks randomly selected from Table 3.4. To refer to these mixed workloads, we use the following nomencalture:

$$WorkloadNumber_{CPUHigh-CPULow,GPUHigh-GPULow}$$

where *CPUHigh* and *GPUHigh* are the number of high-power consumption benchmarks (for CPU and GPU, respectively) and *CPULow* and *GPULow* are the number of low-power consumption benchmarks (for CPU and GPU, respectively). The devices start running their assigned benchmarks at the same time and benchmarks run until completion.

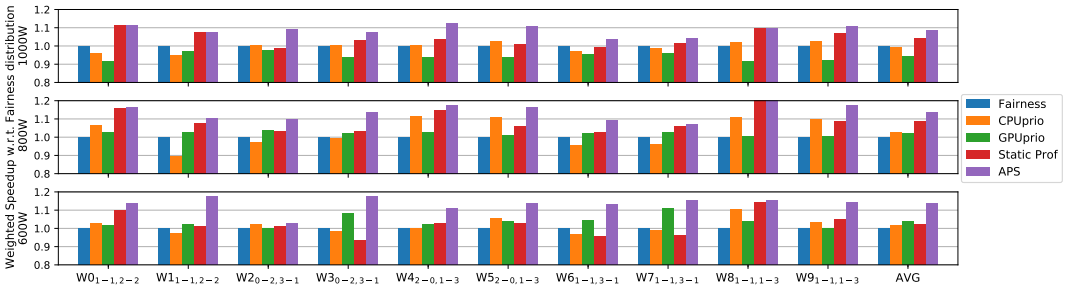


FIGURE 7.8: Weighted speedup of mixed workloads (2 CPU and 4 GPU applications) running under different power caps.

Figure 7.8 reports the weighted speedup for different workloads under 3 power caps (1000, 800, and 600W) using the Fairness, CPUprio, GPUprio, Static Prof, and APS power distributions. The x-axis shows 2 random workloads for 5 scenarios ($CPU_{High} - CPU_{Low}, GPU_{High} - GPU_{Low}$): (1) 1-1, 2-2; (2) 0-2, 3-1; (3) 2-0,1-3; (4) 1-1,3-1; (5) 1-1, 1-3. Figure 7.9 shows the average power consumption and power headroom for the mixed workloads with respect to the power cap. When no power cap is applied, these mixed workloads have a power consumption of up to 1614W (867.5W on average). On average, Fairness degrades performance with respect to

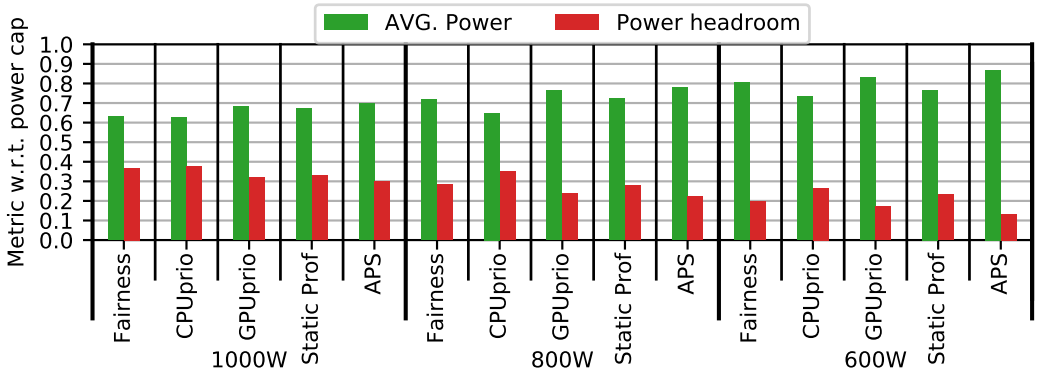


FIGURE 7.9: Average power consumption and power headroom for different power distributions under different power caps when running different mixed workloads.

no power cap by 10.1%, 19.6%, and 30.1% for power caps of 1000, 800, and 600W, respectively.

Figure 7.8 shows that APS achieves the best performance across all the workloads and power caps, achieving average weighted speedups over Fairness of 8.8%, 13.9%, and 13.6% for power caps of 1000, 800, and 600W, respectively. The best power distribution heavily depends on the characteristics of each workload; therefore, the static approaches fail at improving performance for all workloads, while APS consistently achieves speedups in all cases by intelligently shifting the power to the devices that better utilize it. For this reason, Figure 7.9 shows that APS increases the average power consumption over Fairness by 6.8%, 5.9%, and 6.3% for power caps of 1000, 800, and 600W, respectively, while respecting the power cap. Notice that the available power headroom also depends on the workload power consumption behavior. On average, the time over the power cap for APS is 1.2%, 3.1%, 4.9% for power caps of 1000, 800, and 600W, respectively.

CPUprio outperforms Fairness in several workloads. In particular, workloads 4, 5, 8 and 9 are running 3 low-power GPU benchmarks and at least 1 high-power CPU benchmark; consequently, shifting power to CPUs improves performance. The opposite situation happens in workloads 1, 6, and 7, as a result, the performance of CPUprio is worse than Fairness. As seen in Figure 7.9, CPUprio is the approach that uses the least power, with power headrooms of 36.6%, 36.5%, and 29.6% for power caps of 1000, 800, and 600W, respectively.

GPUprio achieves similar or worse performance than Fairness with a power cap of 1000W. This is because CPUs run at the lowest frequency and the GPUs at the highest frequency, and the slowdowns in the CPU benchmarks outweigh the speedups in the GPU benchmarks. With lower power caps, GPUprio achieves better performance than Fairness due to the power-performance curve of GPUs (shown in Figure 7.4), specially in workloads where 3 GPUs are running high-power benchmarks (workloads 1, 3, 6, and 7). Consequently, the average power consumption of

GPUprio over Fairness increases as the power decreases, as shown in Figure 7.9.

Static Prof achieves average performance improvements of 4.3%, 8.8%, and 2.1% with respect to Fairness for power caps of 1000W, 800W, and 600W. For a power cap of 600W, Static Prof degrades performance for workloads 3, 6, and 7 due to GPUs running high-power benchmarks. In higher power caps, this behavior is less noticeable due to the power-performance curve of the GPUs (as shown in Figure 7.4). Static Prof is a conservative approach in workloads with changing power behaviors, which can present a large power headroom in some application phases. As we can observe in Figure 7.9, Static Prof uses 3.9% less power on average than Fairness for a power cap of 600W.

7.4.6 Comparison with State-of-the-Art

This section compares APS with two state-of-the-art techniques: a market-based solution [62] and Tangram [127].

We implement a market solution based on [157], which is aimed to distribute multiple resources (e.g., cache, off-chip bandwidth, and power) among multiple CPUs. We adapt the algorithm to distribute power between CPUs and GPUs. In our market solution (*Market*), agents (e.g., CPUs and GPUs) bid for power based on their expected power utility. Power utility is modeled based on the fact that the length of the compute phase tends to scale linearly with the processor frequency. Agents measure their compute and memory phases from the last interval to compute their power utility. Then, a centralized resource arbiter collects bids, adjusts the price of the resource based on the bids and the power availability, and publishes the new price. Agents bid again based on the new price and their utility. We set the budget for bidding for the agents as their maximum TDP. In our setup, GPUs have a higher TDP and higher budget for bidding than CPUs. If the power surpasses the power cap, all the devices reduce their power budget equally. Note that this implementation, as proposed in the original paper [157], does not take advantage of real-time power measurements, limiting the ability of the bidders to bid accurately. Using real-time measurements in Market is left for future work.

We also compare APS with Tangram [127], which manages the power budgets of the devices in heterogeneous systems to minimize the overall Energy-Delay Product (EDP). Tangram has two operating modes: (1) preconfiguration mode and (2) enhancement mode. The preconfiguration mode is used when the computation only uses one device type, and it applies a static power distribution for all the devices. We extend the static power distributions to fit our larger-scale experimental framework. The enhancement mode is used when all the devices in the system are active. This mode applies a Nelder-Mead search to find the power distribution that minimizes EDP. The Nelder-Mead search is leveraged by throughput, which is read by collecting performance counters in the CPUs and the GPUs. As described in [127], if

the power goes above the power limit, the devices are set to their lowest frequencies and the Nelder-Mead search is restarted. Also, there is 5% probability to restart the search. Since the power is already limited to a given power cap, we set Tangram to maximize overall throughput. Note that, although Tangram is proposed as a hardware technique, in the original manuscript [127] it is evaluated using a software implementation. For our comparison with APS, we use a software implementation of Tangram that has a single Nelder-Mead search.

Note that systems under APS, Tangram and Market (as well as under other state-of-the-art solutions) can surpass the power cap. This happens if, at the moment the power consumption is stable and maximized under a power cap, a device enters a program phase with a higher power consumption, or if the power cap is decreased due to external circumstances such as power or thermal emergencies. For this reason, the three solutions include a control mechanism that ensures the power cap is always respected.

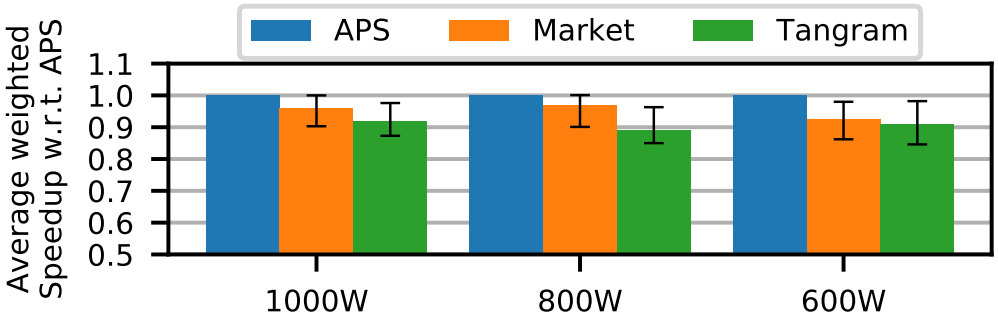


FIGURE 7.10: Average weighted speedup of Market and Tangram with respect to APS for mixed workloads (from Figure 7.8) with different power caps. Whiskers represent the range of speedups achieved in all the workload mixes.

Figure 7.10 shows the average weighted speedup of Market and Tangram with respect to APS for the mixed workloads evaluated in Section 7.4.5 with power caps of 1000W, 800W and 600W. The whiskers represent the range of speedups achieved by Market and Tangram in all the workload mixes.

Results show that, compared to APS, Market presents average slowdowns of 3.5%, 2.9% and 7.5% for power caps of 1000W, 800W and 600W, respectively. In addition, the whiskers show that Market does not improve performance in any workload mix with any power cap compared to APS. Market performs worse than APS in the workload mixes that benefit from assigning more power to the CPUs (workloads 4 and 5) because, in our implementation where the agents have a bidding budget of their maximum TDP, GPUs have a higher bidding budget; as a consequnce, the algorithm is biased to power distributions that favor GPUs while CPUs starve for power. As a consequence, we observe large imbalances in the power budgets of the GPUs and the CPUs, which end up causing performance inefficiencies in the

CPU-intensive mixed workloads because the speedups in the GPU workloads are overweighted by the slowdowns in the CPU workloads. In addition, we also observe that Market introduces performance penalties when GPUs bid intensively for power (workloads 6 and 7). Since agents only consider power utility but not the current power consumption of the devices, they bid for power independently of low or high power phases, which are common in GPU workloads. This can lead to power distributions that do not honor the system power cap, forcing an emergency power reduction in all the devices to lower the consumption below the power cap. All together, we observe that Market is not able to correctly balance the power budgets of the CPUs and the GPUs in some cases. This problem is not caused by the fundamental idea behind the market approaches in general nor by the algorithm that makes the decisions. Instead, the main problem of Market is the implementation of the bidding capacities of the devices. In our implementation the bidding capacity of a device is its maximum TDP, as a result, there is a big disparity in the bidding power of the CPUs and the GPUs. In general, bids in a market mechanism should reflect the actual value that the bidding process obtains from using the resource. Thus, if bids are set heuristically, the Market algorithm is subject to the vagaries of that heuristic. The problem of finding an optimal bidding mechanism for systems with CPUs and GPUs has never been addressed because Market algorithms have never been studied in this context. This research direction is left for future work.

Figure 7.10 also shows that Tangram offers less performance than APS, with average slowdowns of 7.2%, 9.5% and 8.5% under power caps of 1000W, 800W and 600W, respectively. As before, all the workloads mixes perform worse with Tangram than with APS. We observe that the main reason for this performance disparity is that, in some workload mixes, Tangram suffers from slow reactions to changing power behaviors. The Nelder-Mead search is a complex algorithm with a long search phase, which compromises response time and scalability for a large number of devices. Therefore, in workload mixes where power consumption varies considerably along the execution (workloads 4, 6, 7, and 9), the Nelder-Mead search misses opportunities to boost the overall performance. Another important reason for the performance differences are the unbalanced power distributions that Tangram uses for some workload mixes (workloads 0, 1, 2, 3, and 8), which are caused by the nature of the Nelder-Mead search. The algorithm assigns more power to the devices with a higher throughput so, when it assigns more power to the GPU with the highest throughput, its throughput is repeatedly augmented at every iteration of the Nelder-Mead search, while the rest of devices are never assigned more power. In some phases of the execution we see that this behavior creates big imbalances in the power budgets of the devices. As a result, an important speedup is achieved for the single application running on the GPU with the highest throughput, but the performance does not change in the rest of applications running in the other devices.

In contrast, APS distributes the power in a much more balanced way; therefore, the speedup achieved by the application running on the GPU with the highest throughput is not as large as in Tangram, but it is compensated by the speedups obtained in the rest of applications, resulting in an average weighted speedup larger than Tangram.

On average, Tangram performs slightly worse than Market for all the power caps. However, in many workload mixes they achieve very similar performance, as shown by the whiskers. In some workload mixes Tangram is not as effective as Market because of the slower Nelder-Mead search. In other cases (workloads 4 and 9) the reason is the imbalance between the power budgets of the CPUs and the GPUs, which happens in both Market and Tangram, but it is more extreme in Tangram. Tangram assigns as much power budget as possible to the GPUs with maximum throughput and does not increase the power budget of the CPUs. In Market the GPUs have a higher bidding budget than the CPUs; as a consequence, a big portion of the power is shifted to the GPUs, but the CPUs still get a small portion of the power headroom, hence the distribution is not as extremely unbalanced as in Tangram.

7.4.7 APS Design Discussion

This section shows the importance of the RatioDevice, the freq-power tables, and the Force Power Cap phase of APS.

To demonstrate the benefits of the proposed design, we compare APS against 4 alternative implementations: *APS-NoScalePower (APS-NSP)* does not scale the current power consumption; therefore, the RatioDevice is always set to 1. *APS-SpecificBenchmarkTables (APS-SBT)* uses a specific freq-power table for every benchmark instead of a freq-power table generated with a stressmark. The freq-power tables are generated offline by running all the benchmarks on their devices at every available frequency. *APS-NoForcePowerCap-Min (APS-NFPC-Min)* uses a constant RatioDevice of 1 and, when a power cap is introduced, reduces the frequencies of all the devices to their minimum. *APS-NoForcePowerCap-Max (APS-NFPC-Max)* uses a constant RatioDevice of 1 and, when a power cap is introduced, it does not change the frequencies of the devices. The two last variants use the Power Distribution phase to iteratively adjust the frequencies of the devices until their power consumption is equal to their power budget and the system power cap is met.

Figure 7.11 shows the average weighted speedup with respect to Fairness for the mixed workloads evaluated in Section 7.4.5 with a power cap of 800W. Results show that APS outperforms all the alternative designs except APS-SBT. Compared to APS, APS-NSP, APS-NFPC-Min and APS-NFPC-Max present performance losses of 1.5%, 4.6% and 2.2%, respectively. These slowdowns happen because these alternative implementations do not scale the power of the running applications to the

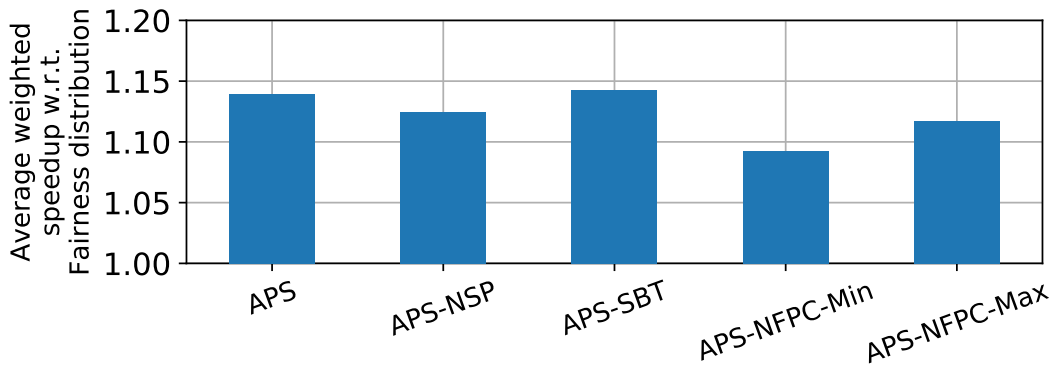


FIGURE 7.11: Average weighted speedup of different APS variants with respect to Fairness for mixed workloads (from Figure 7.8) with a power cap of 800W.

one observed with the stressmark; as a consequence, more iterations of the Power Distribution phase are required to optimally distribute the power. The performance differences between APS-SBT and APS are below 1% because the specific freq-power tables per benchmark achieve the same goal as the one generated by the stressmark. However, generating the freq-power tables per benchmark requires a huge amount of executions to profile all the applications running on their devices at all the frequencies, which can be avoided by combining the RatioDevice and a single freq-power table per device generated with a stressmark.

7.5 Conclusions

Current systems include multiple discrete devices per node. Unfortunately, power constraints limit the number of devices that can operate simultaneously at their highest frequency. To efficiently distribute the power in such systems, dynamic power management techniques that consider device utilization are needed.

In this chapter, we demonstrate that APS, a mechanism that leverages system utilization to distribute the available power among multiple discrete devices in power-constrained multi-accelerator heterogeneous systems. APS dynamically adjusts the power budget of the devices based on their utilization, allowing highly-utilized devices to have a higher power budget than low-utilized devices. Results show that an intelligent power management outperforms static power distributions and state-of-the-art proposals in single CPU-GPU applications and in multiprogrammed workloads.

Chapter 8

Conclusions and Future Work

For years, increasing processor's clock frequency was the main reason for performance improvement in every new generation thanks to Moore's Law. Recent years have proven that frequency cannot keep driving performance improvement due to power constraints. New techniques are constantly being pushed to improve performance, power consumption, and energy efficiency in new systems.

Hardware coordination is a fundamental step in modern systems where independent techniques are used to improve performance, power consumption, and energy efficiency. As more techniques are implemented in future systems, this hardware coordination will become essential to improve future workloads.

As shown in this Thesis, hardware coordination is needed at several levels. At a processor level, it can be used to coordinate multiple aspects of a single technique such as data prefetching. At a system level, it can be used to coordinate multiple independent and single techniques such data prefetching, SMT, socket allocation, and clock frequency. And, at a node level, it can be used to coordinate multiple independent devices within the node to collaborate towards a given metric.

This Thesis has presented several techniques for hardware coordination that leverage information already present in current systems. These techniques are able to improve performance, power consumption, and energy efficiency without overheads or introducing substantial changes in the systems.

We have implemented and assessed hardware coordination as a solution to improve performance, reduce power consumption, and improve energy efficiency in modern systems at the three different levels:

- At a processor level, this Thesis proposes a low level coordination for data prefetching in order to improve performance and reduce power consumption when data prefetching requirements of a workload changes over time.

- At a system level, we introduce a library that intelligently coordinates multiple hardware knobs such as SMT, data prefetching, socket allocation, and clock

frequency to improve performance, reduce power consumption, and improve energy efficiency.

- At a node level, we show that utilization and power requirements change over time and depending on the nature of the device itself. We coordinate the power consumption of multiple independent devices in order to honor an arbitrary power cap.

In this Thesis, we demonstrated that current and future systems need a fine-grained hardware coordination at all levels to maximize their energy-efficiency for parallel applications and multiprogrammed workloads.

8.1 Future Work

This Thesis opens the door to possible lines of future work. In this section we detail the ones with particular potential.

- *Generalization for hardware coordination*. This Thesis proposes the coordination of multiple hardware knobs in a POWER-based system. Yet, other architectures expose hardware knobs as well. A line of future work is to explore what changes are needed to generalize the technique introduced in this Thesis. For this, it is needed to identify the information used to leverage the decisions, evaluate the interactions between different hardware knobs, and find the possible trade offs.

- *Hardware coordination for multiple workloads.* This Thesis proposes a technique to coordinate multiple hardware knobs when a single parallel application is running in the system. Yet, it is possible that a system is running multiple workloads at once. In a multi-core system this can mean that different cores have different hardware requirements. A configuration local to a core affects to the global (or system) performance. Therefore, it exists a trade-off between the global performance and the local performance.

  This scenario is left unexplored in this Thesis. An interesting line of work is to explore how hardware knobs affect to the local and global performance and how manage the different trade-offs.

- *Power shifting with hardware knobs.* We introduce a technique to honor a given power cap in a system with multiple independent devices. This technique is evaluated as a software process, which has limitations such as overheads and gathered information. This technique can be added to the centralized units of modern systems to reduce those limitations.

We believe that without those limitations we can increase furthermore the performance of the workloads running the system due to react faster and with more accurate information.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on pp. 27, 87).

[2] N.R. Adiga, G. Almasi, G.S. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A.A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T.M. Cipolla, P. Crumley, K.M. Desai, A. Deutsch, T. Domany, M.B. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M.E. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R.A. Haring, D. Heidel, P. Heidelberger, L.M. Herger, D. Hoenicke, R.D. Jackson, T. Jamal-Eddine, G.V. Kopcsay, E. Krevat, M.P. Kurhekar, A.P. Lanzetta, D. Lieber, L.K. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J.E. Moreira, B.J. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R.B. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R.K. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B.D. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R.B. Tremaine, M. Tsao, A.R. Umamaheshwaran, P. Verma, P. Vranas, T.J.C. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C.T. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M.K. Seager, J.S. Vetter, and

K. Yates. "An Overview of the BlueGene/L Supercomputer". In: *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. 2002, pp. 60–60. DOI: 10.1109/SC.2002.10017 (cit. on p. 71).

[3] A. Adileh, S. Eyerman, A. Jaleel, and L. Eeckhout. "Mind The Power Holes: Sifting Operating Points in Power-Limited Heterogeneous Multicores". In: *IEEE Computer Architecture Letters* 16.1 (2017), pp. 56–59 (cit. on p. 17).

[4] Almutaz Adileh, Stijn Eyerman, Aamer Jaleel, and Lieven Eeckhout. "Maximizing Heterogeneous Processor Performance Under Power Constraints". In: *ACM Trans. Archit. Code Optim.* 13.3 (Sept. 2016). ISSN: 1544-3566. DOI: 10.1145/2976739 (cit. on p. 17).

[5] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, P. Worley, and W. Yu. "Early evaluation of IBM BlueGene/P". In: *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. 2008, pp. 1–12. DOI: 10.1109/SC.2008.5214725 (cit. on p. 71).

[6] Lluc Alvarez, Lluís Vilanova, Miquel Moretó, Marc Casas, Marc González, Xavier Martorell, Nacho Navarro, Eduard Ayguadé, and Mateo Valero. "Coherence protocol for transparent management of scratchpad memories in shared memory manycore architectures". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*. Ed. by Deborah T. Marr and David H. Albonesi. ACM, 2015, pp. 720–732. DOI: 10.1145/2749469.2750411. URL: https://doi.org/10.1145/2749469.2750411 (cit. on p. 19).

[7] Lluc Alvarez, Lluís Vilanova, Marc González, Xavier Martorell, Nacho Navarro, and Eduard Ayguadé. "Hardware-software coherence protocol for the coexistence of caches and local memories". In: *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*. Ed. by Jeffrey K. Hollingsworth. IEEE/ACM, 2012, p. 89. DOI: 10.1109/SC.2012.61. URL: https://doi.org/10.1109/SC.2012.61 (cit. on p. 19).

[8] Lluc Alvarez, Lluís Vilanova, Marc González, Xavier Martorell, Nacho Navarro, and Eduard Ayguadé. "Hardware-Software Coherence Protocol for the Coexistence of Caches and Local Memories". In: *IEEE Trans. Computers* 64.1 (2015), pp. 152–165. DOI: 10.1109/TC.2013.194. URL: https://doi.org/10.1109/TC.2013.194 (cit. on p. 19).

[9] Lluc Alvarez, Miquel Moretó, Marc Casas, Emilio Castillo, Xavier Martorell, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. "Runtime-Guided Management of Scratchpad Memories in Multicore Architectures". In: *2015 International Conference on Parallel Architectures and Compilation, PACT 2015,*

San Francisco, CA, USA, October 18-21, 2015. IEEE Computer Society, 2015, pp. 379–391. DOI: 10.1109/PACT.2015.26. URL: https://doi.org/10.1109/PACT.2015.26 (cit. on p. 19).

[10] Manish Arora, Siddhartha Nath, Subhra Mazumdar, Scott B. Baden, and Dean M. Tullsen. "Redefining the Role of the CPU in the Era of CPU-GPU Integration". In: *IEEE Micro* 32.6 (2012), pp. 4–16. DOI: 10.1109/MM.2012.57 (cit. on p. 4).

[11] Raid Ayoub, Umit Ogras, Eugene Gorbatov, Yanqin Jin, Timothy Kam, Paul Diefenbaugh, and Tajana Rosing. "OS-level power minimization under tight performance constraints in general purpose systems". In: *IEEE/ACM International Symposium on Low Power Electronics and Design*. 2011, pp. 321–326. DOI: 10.1109/ISLPED.2011.5993657 (cit. on p. 9).

[12] P. E. Bailey, D. K. Lowenthal, V. Ravi, B. Rountree, M. Schulz, and B. R. d. Supinski. "Adaptive Configuration Selection for Power-Constrained Heterogeneous Systems". In: *2014 43rd International Conference on Parallel Processing*. 2014, pp. 371–380. DOI: 10.1109/ICPP.2014.46 (cit. on pp. 17, 85, 86).

[13] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. "Domino Temporal Data Prefetcher". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018, pp. 131–142. DOI: 10.1109/HPCA.2018.00021 (cit. on p. 13).

[14] Adrián Barredo, Juan M. Cebrian, Miquel Moretó, Marc Casas, and Mateo Valero. "Improving Predication Efficiency through Compaction/Restoration of SIMD Instructions". In: *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 2020, pp. 717–728. DOI: 10.1109/HPCA47549.2020.00064. URL: https://doi.org/10.1109/HPCA47549.2020.00064 (cit. on p. 19).

[15] Ramon Bertran, Alper Buyuktosunoglu, Meeta S. Gupta, Marc Gonzalez, and Pradip Bose. "Systematic Energy Characterization of CMP/SMT Processor Systems via Automated Micro-Benchmarks". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 2012, pp. 199–211. DOI: 10.1109/MICRO.2012.27 (cit. on p. 43).

[16] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: http://doi.acm.org/10.1145/2024716.2024718 (cit. on p. 24).

[17] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach". In: *2008 41st IEEE/ACM International Symposium on Microarchitecture*. 2008, pp. 318–329. DOI: 10.1109/MICRO.2008.4771801 (cit. on p. 14).

[18] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach". In: *2008 41st IEEE/ACM International Symposium on Microarchitecture*. 2008, pp. 318–329. DOI: 10.1109/MICRO.2008.4771801 (cit. on pp. 17, 85).

[19] Carlos Boneti, Roberto Gioiosa, Francisco J. Cazorla, and Mateo Valero. "A dynamic scheduler for balancing HPC applications". In: *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 2008, pp. 1–12. DOI: 10.1109/SC.2008.5217785 (cit. on p. 11).

[20] Carlos Boneti, Roberto Gioiosa, Francisco J. Cazorla, Julita Corbalán, Jesús Labarta, and Mateo Valero. "Balancing HPC applications through smart allocation of resources in MT processors". In: *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*. IEEE, 2008, pp. 1–12. DOI: 10.1109/IPDPS.2008.4536293. URL: https://doi.org/10.1109/IPDPS.2008.4536293 (cit. on pp. 3, 11, 29).

[21] Carlos Boneti, Francisco J. Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu, Chen-Yong Cher, and Mateo Valero. "Software-Controlled Priority Characterization of POWER5 Processor". In: *2008 International Symposium on Computer Architecture*. 2008, pp. 415–426. DOI: 10.1109/ISCA.2008.8 (cit. on p. 11).

[22] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32 (cit. on p. 64).

[23] Qiong Cai, Jose González, Ryan Rakvic, Grigorios Magklis, Pedro Chaparro, and Antonio González. "Meeting points: Using thread criticality to adapt multicore hardware to parallel regions". In: *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008, pp. 240–249 (cit. on p. 10).

[24] Marc Casas, Miquel Moretó, Lluc Alvarez, Emilio Castillo, Dimitrios Chasapis, Timothy Hayes, Luc Jaulmes, Oscar Palomar, Osman Unsal, Adrian Cristal, Eduard Ayguade, Jesus Labarta, and Mateo Valero. "Runtime-Aware Architectures". In: Euro-Par'15, pp. 16–27 (cit. on p. 18).

[25] Emilio Castillo, Lluc Alvarez, Miquel Moretó, Marc Casas, Enrique Vallejo, José Luis Bosque, Ramón Beivide, and Mateo Valero. "Architectural Support for Task Dependence Management with Flexible Software Scheduling". In: *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, 2018, pp. 283–295. DOI: 10.1109/HPCA.2018.00033. URL: https://doi.org/10.1109/HPCA.2018.00033 (cit. on p. 19).

[26] Emilio Castillo, Miquel Moreto, Marc Casas, Lluc Alvarez, Enrique Vallejo, Kallia Chronaki, Rosa Badia, Jose Luis Bosque, Ramon Beivide, Eduard Ayguade, Jesus Labarta, and Mateo Valero. "CATA: Criticality Aware Task Acceleration for Multicore Processors". In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016, pp. 413–422. DOI: 10.1109/IPDPS.2016.49 (cit. on pp. 10, 19).

[27] Emilio Castillo, Nikhil Jain, Marc Casas, Miquel Moretó, Martin Schulz, Ramón Beivide, Mateo Valero, and Abhinav Bhatele. "Optimizing computation-communication overlap in asynchronous task-based programs". In: *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019*. Ed. by Rudolf Eigenmann, Chen Ding, and Sally A. McKee. ACM, 2019, pp. 380–391. DOI: 10.1145/3330345.3330379. URL: https://doi.org/10.1145/3330345.3330379 (cit. on p. 19).

[28] F.J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. "Dynamically Controlled Resource Allocation in SMT Processors". In: *37th International Symposium on Microarchitecture (MICRO-37'04)*. 2004, pp. 171–182. DOI: 10.1109/MICRO.2004.17 (cit. on pp. 3, 29).

[29] F.J. Cazorla, P.M.W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. "Predictable performance in SMT processors: synergy between the OS and SMTs". In: *IEEE Transactions on Computers* 55.7 (2006), pp. 785–799. DOI: 10.1109/TC.2006.108 (cit. on p. 11).

[30] Francisco J. Cazorla, Enrique Fernandez, Alex Ramírez, and Mateo Valero. "Improving Memory Latency Aware Fetch Policies for SMT Processors". In: *High Performance Computing*. Ed. by Alex Veidenbaum, Kazuki Joe, Hideharu Amano, and Hideo Aiso. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 70–85. ISBN: 978-3-540-39707-6 (cit. on p. 11).

[31] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. "Rodinia: A Benchmark Suite for Heterogeneous Computing". In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54. ISBN: 978-1-4244-5156-2. DOI: 10.1109/IISWC.2009.

5306797. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1109/IISWC.2009.5306797 (cit. on pp. 26, 87).

[32] Dazhao Cheng, Xiaobo Zhou, Palden Lama, Mike Ji, and Changjun Jiang. "Energy Efficiency Aware Task Assignment with DVFS in Heterogeneous Hadoop Clusters". In: *IEEE Transactions on Parallel and Distributed Systems* 29.1 (2018), pp. 70–82. DOI: 10.1109/TPDS.2017.2745571 (cit. on p. 10).

[33] Trishul M. Chilimbi and Martin Hirzel. "Dynamic Hot Data Stream Prefetching for General-Purpose Programs". In: vol. 37. 5. New York, NY, USA: Association for Computing Machinery, 2002, pp. 199–209. DOI: 10.1145/543552.512554. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/543552.512554 (cit. on p. 14).

[34] L.T. Clark, E.J. Hoffman, J. Miller, M. Biyani, Luyun Liao, S. Strazdus, M. Morrow, K.E. Velarde, and M.A. Yarch. "An embedded 32-b microprocessor core for low-power and high-performance applications". In: *IEEE Journal of Solid-State Circuits* 36.11 (2001), pp. 1599–1608. DOI: 10.1109/4.962279 (cit. on p. 16).

[35] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. "Pack & Cap: Adaptive DVFS and thread packing under power caps". In: *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2011, pp. 175–185. DOI: 10.1145/2155620.2155641 (cit. on pp. 16, 17, 85).

[36] *CORAL Benchmark Codes*. URL: https://asc.llnl.gov/CORAL-benchmarks/ (visited on 10/14/2016) (cit. on p. 26).

[37] Timothy Creech, Aparna Kotha, and Rajeev Barua. "Efficient multiprogramming for multicores with SCAF". In: *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2013, pp. 334–345 (cit. on p. 10).

[38] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. "CoScale: Coordinating CPU and Memory System DVFS in Server Systems". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 2012, pp. 143–154. DOI: 10.1109/MICRO.2012.22 (cit. on pp. 10, 17, 85).

[39] Nicolas Denoyelle, Brice Goglin, Emmanuel Jeannot, and Thomas Ropars. "Data and Thread Placement in NUMA Architectures: A Statistical Learning Approach". In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019. Kyoto, Japan: Association for Computing Machinery, 2019. ISBN: 9781450362955. DOI: 10.1145/3337821.3337893. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3337821.3337893 (cit. on pp. 11, 16).

[40] Vladimir Dimic, Miquel Moretó, Marc Casas, Jan Ciesko, and Mateo Valero. "RICH: implementing reductions in the cache hierarchy". In: *ICS '20: 2020 International Conference on Supercomputing, Barcelona Spain, June, 2020*. Ed. by Eduard Ayguadé, Wen-mei W. Hwu, Rosa M. Badia, and H. Peter Hofstee. ACM, 2020, 16:1–16:13. URL: https://dl.acm.org/doi/10.1145/3392717.3392736 (cit. on p. 19).

[41] Vladimir Dimic, Miquel Moretó, Marc Casas, and Mateo Valero. "Runtime-Assisted Shared Cache Insertion Policies Based on Re-reference Intervals". In: *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*. Ed. by Francisco F. Rivera, Tomás F. Pena, and José Carlos Cabaleiro. Vol. 10417. Lecture Notes in Computer Science. Springer, 2017, pp. 247–259. DOI: 10.1007/978-3-319-64203-1_18. URL: https://doi.org/10.1007/978-3-319-64203-1_18 (cit. on p. 19).

[42] J. Donald and M. Martonosi. "Techniques for Multicore Thermal Management: Classification and New Exploration". In: *33rd International Symposium on Computer Architecture (ISCA'06)*. 2006, pp. 78–88. DOI: 10.1109/ISCA.2006.39 (cit. on p. 9).

[43] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems". In: vol. 30. 1. SAGE Publications Sage UK: London, England, 2016, pp. 3–10 (cit. on p. 25).

[44] E. Ebrahimi, O. Mutlu, and Y. N. Patt. "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems". In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 2009, pp. 7–17. DOI: 10.1109/HPCA.2009.4798232 (cit. on p. 14).

[45] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. "Prefetch-aware shared-resource management for multi-core systems". In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 141–152. DOI: 10.1145/2000064.2000081 (cit. on p. 14).

[46] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. "Coordinated Control of Multiple Prefetchers in Multi-core Systems". In: *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 42. New York, New York: ACM, 2009, pp. 316–326. ISBN: 978-1-60558-798-1. DOI: 10.1145/1669112.1669154. URL: http://doi.acm.org/10.1145/1669112.1669154 (cit. on p. 14).

[47] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark silicon and the end of multicore scaling".

In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 365–376 (cit. on pp. 4, 85).

[48] Stijn Everman and Lieven Eeckhout. "A Memory-Level Parallelism Aware Fetch Policy for SMT Processors". In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 2007, pp. 240–249. DOI: 10.1109/HPCA.2007.346201 (cit. on p. 10).

[49] Stijn Eyerman and Lieven Eeckhout. "Fine-Grained DVFS Using on-Chip Regulators". In: *ACM Trans. Archit. Code Optim.* 8.1 (2011). ISSN: 1544-3566. DOI: 10.1145/1952998.1952999. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/1952998.1952999 (cit. on p. 10).

[50] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. "A Performance Counter Architecture for Computing Accurate CPI Components". In: *SIGOPS Oper. Syst. Rev.* 40.5 (Oct. 2006), pp. 175–184. ISSN: 0163-5980. DOI: 10.1145/1168917.1168880. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/1168917.1168880 (cit. on p. 55).

[51] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. "Sequoia: Programming the Memory Hierarchy". In: *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. 2006, pp. 4–4. DOI: 10.1109/SC.2006.55 (cit. on p. 18).

[52] Josue Feliu, Julio Sahuquillo, Salvador Petit, and Jose Duato. "Addressing Fairness in SMT Multicores with a Progress-Aware Scheduler". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. 2015, pp. 187–196. DOI: 10.1109/IPDPS.2015.48 (cit. on p. 11).

[53] Josue Feliu, Julio Sahuquillo, Salvador Petit, and Jose Duato. "Perf&Fair: A Progress-Aware Scheduler to Enhance Performance and Fairness in SMT Multicores". In: *IEEE Trans. Comput.* 66.5 (2017), pp. 905–911. ISSN: 0018-9340. DOI: 10.1109/TC.2016.2620977. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1109/TC.2016.2620977 (cit. on p. 11).

[54] Josue Feliu, Stijn Eyerman, Julio Sahuquillo, and Salvador Petit. "Symbiotic job scheduling on the IBM POWER8". In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016, pp. 669–680. DOI: 10.1109/HPCA.2016.7446103 (cit. on pp. 10, 11).

[55] Wes Felter, Karthick Rajamani, Tom Keller, and Cosmin Rusu. "A Performance-conserving Approach for Reducing Peak Power Consumption in Server Systems". In: *Proceedings of the 19th Annual International Conference on Supercomputing*. Cambridge, Massachusetts: ACM, 2005, pp. 293–302. ISBN: 1-59593-167-8. DOI: 10.1145/1088149.1088188 (cit. on pp. 16, 17, 85).

[56] M. Floyd, M. Ware, K. Rajamani, T. Gloekler, B. Brock, P. Bose, A. Buyuktosunoglu, J. C. Rubio, B. Schubert, B. Spruth, J. A. Tierno, and L. Pesantez. "Adaptive energy-management features of the IBM POWER7 chip". In: *IBM Journal of Research and Development* 55.3 (2011), 8:1–8:18. DOI: 10.1147/JRD.2011.2114250 (cit. on p. 27).

[57] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. "Stride Directed Prefetching in Scalar Processors". In: *Proceedings of the 25th Annual International Symposium on Microarchitecture*. MICRO 25. Portland, Oregon, USA: IEEE Computer Society Press, 1992, pp. 102–110. ISBN: 0-8186-3175-9. URL: http://dl.acm.org/citation.cfm?id=144953.145006 (cit. on p. 13).

[58] Karl Fürlinger, Christof Klausecker, and Dieter Kranzlmüller. "Towards Energy Efficient Parallel Computing on Consumer Electronic Devices". In: *Proceedings of the First International Conference on Information and Communication on Technology for the Fight Against Global Warming*. ICT-GLOW'11. Toulouse, France: Springer-Verlag, 2011, pp. 1–9. ISBN: 978-3-642-23446-0. URL: http://dl.acm.org/citation.cfm?id=2035539.2035541 (cit. on p. 71).

[59] Victor Garcia, Alejandro Rico, Carlos Villavieja, Paul Carpenter, Nacho Navarro, and Alex Ramirez. "Adaptive Runtime-Assisted Block Prefetching on Chip-Multiprocessors". In: *Int. J. Parallel Program.* 45.3 (2017), pp. 530–550. ISSN: 0885-7458. DOI: 10.1007/s10766-016-0431-8. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1007/s10766-016-0431-8 (cit. on p. 19).

[60] Constantino Gómez, Filippo Mantovani, Erich Focht, and Marc Casas. "Efficiently Running SpMV on Long Vector Architectures". In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 292–303. ISBN: 9781450382946. DOI: 10.1145/3437801.3441592. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3437801.3441592 (cit. on p. 19).

[61] Peter Greenhalgh. *big.LITTLE Processing with ARM Cortex-A15 and Cortex-A7*. Tech. rep. ARM, 2012 (cit. on p. 16).

[62] Marisabel Guevara, Benjamin Lubin, and Benjamin C. Lee. "Navigating heterogeneous processors with market mechanisms". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 2013, pp. 95–106. DOI: 10.1109/HPCA.2013.6522310 (cit. on pp. 17, 85, 86, 89, 103).

[63] Marisabel Guevara, Benjamin Lubin, and Benjamin C. Lee. "Strategies for anticipating risk in heterogeneous system design". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 2014, pp. 154–164. DOI: 10.1109/HPCA.2014.6835926 (cit. on p. 89).

[64] Constantino Gómez, Francesc Martınez, Adrià Armejach, Miquel Moretó, Filippo Mantovani, and Marc Casas. "Design Space Exploration of Next-Generation HPC Machines". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 54–65. DOI: 10.1109/IPDPS.2019.00017 (cit. on p. 19).

[65] Brian Hall, Peter Bergner, Alon Shalev Housfater, Madhusudanan Kandasamy, Tulio Magno, Alex Mericas, Steve Munroe, Mauricio Oliveira, Bill Schmidt, Will Schmidt, et al. *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8*. IBM Redbooks, 2017 (cit. on p. 22).

[66] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. "Toward Dark Silicon in Servers". In: *IEEE Micro* 31.4 (2011), pp. 6–15. DOI: 10.1109/MM.2011.77 (cit. on pp. 4, 85).

[67] Wim Heirman, Trevor E. Carlson, Kenzo Van Craeynest, Ibrahim Hur, Aamer Jaleel, and Lieven Eeckhout. "Automatic SMT Threading for OpenMP Applications on the Intel Xeon Phi Co-Processor". In: *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*. ROSS '14. Munich, Germany: Association for Computing Machinery, 2014. ISBN: 9781450329507. DOI: 10.1145/2612262.2612268. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/2612262.2612268 (cit. on pp. 10, 11).

[68] Wim Heirman, Kristof Du Bois, Yves Vandriessche, Stijn Eyerman, and Ibrahim Hur. "Near-Side Prefetch Throttling: Adaptive Prefetching for High-Performance Many-Core Processors". In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. PACT '18. Limassol, Cyprus: Association for Computing Machinery, 2018. ISBN: 9781450359863. DOI: 10.1145/3243176.3243181. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3243176.3243181 (cit. on p. 13).

[69] Jason Hiebel, Laura E. Brown, and Zhenlin Wang. "Machine Learning for Fine-Grained Hardware Prefetcher Control". In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019. Kyoto, Japan: Association for Computing Machinery, 2019. ISBN: 9781450362955. DOI: 10.1145/3337821.3337854. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3337821.3337854 (cit. on p. 14).

[70] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas

Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS". In: *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*. 2010, pp. 108–109. DOI: 10.1109/ISSCC.2010.5434077 (cit. on p. 16).

[71] Chung-Hsing Hsu and Ulrich Kremer. "The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI '03. San Diego, California, USA: Association for Computing Machinery, 2003, pp. 38–48. ISBN: 1581136625. DOI: 10.1145/781131.781137. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/781131.781137 (cit. on p. 9).

[72] Ibrahim Hur and Calvin Lin. "Memory Prefetching Using Adaptive Stream Detection". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 2006, pp. 397–408. DOI: 10.1109/MICRO.2006.32 (cit. on p. 14).

[73] *Intel 64 and IA-32 Architectures. Optimization Reference Manual*. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf. September 2019. URL: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf (visited on 10/14/2016) (cit. on p. 52).

[74] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi. "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 2006, pp. 347–358. DOI: 10.1109/MICRO.2006.8 (cit. on pp. 17, 85).

[75] Luc Jaulmes, Miquel Moretó, Mateo Valero, Mattan Erez, and Marc Casas. "Runtime-guided ECC protection using online estimation of memory vulnerability". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*. Ed. by Christine Cuicchi, Irene Qualters, and William T. Kramer. IEEE/ACM, 2020, p. 76. DOI: 10.1109/SC41405.2020.00080. URL: https://doi.org/10.1109/SC41405.2020.00080 (cit. on p. 19).

[76] Sudhanshu Shekhar Jha, Wim Heirman, Ayose Falcón, Trevor E. Carlson, Kenzo Van Craeynest, Jordi Tubella, Antonio González, and Lieven Eeckhout. "Chrysso: An Integrated Power Manager for Constrained Many-Core

Processors". In: *Proceedings of the 12th ACM International Conference on Computing Frontiers*. CF '15. Ischia, Italy: Association for Computing Machinery, 2015. ISBN: 9781450333580. DOI: 10.1145/2742854.2742885. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/2742854.2742885 (cit. on p. 14).

[77] Zhen Jia, Chao Xue, Guancheng Chen, Jianfeng Zhan, Lixin Zhang, Yonghua Lin, and Peter Hofstee. "Auto-tuning Spark big data workloads on POWER8: Prediction-based dynamic SMT threading". In: *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 2016, pp. 387–400. DOI: 10.1145/2967938.2967957 (cit. on p. 11).

[78] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra. "Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS". In: *2015 International Symposium on Code Generation and Optimization (CGO)*. 2015, pp. 1–11. DOI: 10.1109/CGO.2015.7054182 (cit. on pp. 17, 85, 86).

[79] Victor Jiménez, Alper Buyuktosunoglu, Pradip Bose, Francis P. O'Connell, Francisco Cazorla, and Mateo Valero. "Increasing multicore system efficiency through intelligent bandwidth shifting". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 39–50. DOI: 10.1109/HPCA.2015.7056020 (cit. on pp. 11, 13, 14).

[80] Victor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. "Making Data Prefetch Smarter: Adaptive Prefetching on POWER7". In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT '12. Minneapolis, Minnesota, USA: Association for Computing Machinery, 2012, pp. 137–146. ISBN: 9781450311823. DOI: 10.1145/2370816.2370837. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/2370816.2370837 (cit. on pp. 3, 11, 13, 14, 29).

[81] H. Jin and Frumkin MA. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance*. Tech. rep. NAS-99-011. NASA Ames Research Center, 1999 (cit. on p. 30).

[82] Hao-Qiang Jin, Michael Frumkin, and Jerry Yan. "The OpenMP implementation of NAS parallel benchmarks and its performance". In: (1999) (cit. on pp. 25, 26).

[83] Norman P. Jouppi. "Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers". In: *Proceedings of the 17th Annual International Symposium on Computer Architecture*. ISCA '90. Seattle, Washington, USA: ACM, 1990, pp. 364–373. ISBN: 0-89791-366-3. DOI: 10.1145/325164.325162. URL: http://doi.acm.org/10.1145/325164.325162 (cit. on p. 13).

[84] Stefanos Kaxiras and Margaret Martonosi. *Computer Architecture Techniques for Power-Efficiency*. 1st. Morgan and Claypool Publishers, 2008. ISBN: 1598292080, 9781598292084 (cit. on p. 9).

[85] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. "Interval-based models for run-time DVFS orchestration in superscalar processors". In: Jan. 2010, pp. 287–296. DOI: 10.1145/1787275.1787338 (cit. on p. 9).

[86] Muneeb Khan, Andreas Sandberg, and Erik Hagersten. "A Case for Resource Efficient Prefetching in Multicores". In: *2014 43rd International Conference on Parallel Processing*. 2014, pp. 101–110. DOI: 10.1109/ICPP.2014.19 (cit. on p. 14).

[87] Muneeb Khan, Michael A. Laurenzanoy, Jason Marsy, Erik Hagersten, and David Black-Schaffer. "AREP: Adaptive Resource Efficient Prefetching for Maximizing Multicore Performance". In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 2015, pp. 367–378. DOI: 10.1109/PACT.2015.35 (cit. on p. 14).

[88] T. Komoda, S. Hayashi, T. Nakada, S. Miwa, and H. Nakamura. "Power capping of CPU-GPU heterogeneous systems through coordinating DVFS and task mapping". In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 2013, pp. 349–356. DOI: 10.1109/ICCD.2013.6657064 (cit. on pp. 17, 85, 86).

[89] D. M. Koppelman. "Neighborhood prefetching on multiprocessors using instruction history". In: *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00622)*. 2000, pp. 123–132. DOI: 10.1109/PACT.2000.888337 (cit. on p. 13).

[90] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. "When Prefetching Works, When It Doesn&Rsquo;T, and Why". In: *ACM Trans. Archit. Code Optim.* 9.1 (Mar. 2012), 2:1–2:29. ISSN: 1544-3566. DOI: 10.1145/2133382.2133384. URL: http://doi.acm.org/10.1145/2133382.2133384 (cit. on p. 11).

[91] Minghua Li, Guancheng Chen, Qijun Wang, Yonghua Lin, Peter Hofstee, Per Stenstrom, and Dian Zhou. "PATer: A Hardware Prefetching Automatic Tuner on IBM POWER8 Processor". In: vol. 15. 1. 2016, pp. 37–40. DOI: 10.1109/LCA.2015.2442972 (cit. on pp. 14, 16).

[92] S. Liao, T. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou. "Machine learning-based prefetch optimization for data center applications". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 2009, pp. 1–10. DOI: 10.1145/1654059.1654116 (cit. on p. 14).

[93] Y. Liu, G. Cox, Q. Deng, S. C. Draper, and R. Bianchini. "FastCap: An efficient and fair algorithm for power capping in many-core systems". In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2016, pp. 57–68. DOI: 10.1109/ISPASS.2016.7482074 (cit. on p. 17).

[94] "LLNL Proxies apps. https://codesign.llnl.gov/proxy-apps.php". In: 2016 (cit. on p. 25).

[95] C.-K. Luk, R. Muth, Harish Patil, R. Cohn, and G. Lowney. "Ispike: a post-link optimizer for the Intel/spl reg/ Itanium/spl reg/ architecture". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 15–26. DOI: 10.1109/CGO.2004.1281660 (cit. on p. 14).

[96] M. Lurbe, J. Feliu, S. Petit, M. Gomez, and J. Sahuquillo. "DeepP: Deep Learning Multi-Program Prefetch Configuration for the IBM POWER 8". In: *IEEE Transactions on Computers* 01 (5555), pp. 1–1. ISSN: 1557-9956 (cit. on p. 16).

[97] Piotr Luszczek, Jack J. Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John Mccalpin, David Bailey, and Daisuke Takahashi. "Introduction to the HPC Challenge Benchmark Suite". In: 2005 (cit. on p. 25).

[98] Abhinandan Majumdar, Leonardo Piga, Indrani Paul, Joseph L. Greathouse, Wei Huang, and David H. Albonesi. "Dynamic GPGPU Power Management Using Adaptive Model Predictive Control". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 613–624. DOI: 10.1109/HPCA.2017.34 (cit. on pp. 17, 85, 86).

[99] Madhavan Manivannan, Miquel Pericás, Vassilis Papaefstathiou, and Per Stenström. "Global Dead-Block Management for Task-Parallel Programs". In: *ACM Trans. Archit. Code Optim.* 15.3 (2018). ISSN: 1544-3566. DOI: 10.1145/3234337. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3234337 (cit. on p. 19).

[100] Madhavan Manivannan, Vassilis Papaefstathiou, Miquel Pericas, and Per Stenstrom. "RADAR: Runtime-assisted dead region management for last-level caches". In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016, pp. 644–656. DOI: 10.1109/HPCA.2016.7446101 (cit. on p. 19).

[101] "Mantevo suite. https://mantevo.org/". In: 2016 (cit. on p. 25).

[102] Arnaldo Carvalho de Melo. "The New Linux perf tools". In: In Slides from Linux Kongress 2010 (cit. on pp. 27, 96).

[103] A. Mericas, N. Peleg, L. Pesantez, S. B. Purushotham, P. Oehler, C. A. Anderson, B. A. King-Smith, M. Anand, J. A. Arnold, B. Rogers, L. Maurice, and K. Vu. "IBM POWER8 performance features and evaluation". In: *IBM Journal*

of Research and Development 59.1 (2015), 6:1–6:10. DOI: 10.1147/JRD.2014.2380197 (cit. on p. 22).

[104] T. Moseley, J.L. Kihm, D.A. Connors, and D. Grunwald. "Methods for modeling resource contention on simultaneous multithreading processors". In: *2005 International Conference on Computer Design*. 2005, pp. 373–380. DOI: 10.1109/ICCD.2005.74 (cit. on pp. 10, 11).

[105] "MPICH 3.2. https://www.mpich.org/". In: (). URL: https://www.mpich.org (cit. on p. 24).

[106] T. Mudge. "Power: a first-class architectural design constraint". In: *Computer* 34.4 (2001), pp. 52–58. DOI: 10.1109/2.917539 (cit. on p. 1).

[107] Matthias S. Müller, John Baron, William C. Brantley, Huiyu Feng, Daniel Hackenberg, Robert Henschel, Gabriele Jost, Daniel Molka, Chris Parrott, Joe Robichaux, Pavel Shelepugin, Matthijs van Waveren, Brian Whitney, and Kalyan Kumaran. "SPEC OMP2012 — An Application Benchmark Suite for Parallel Systems Using OpenMP". In: *OpenMP in a Heterogeneous World*. Ed. by Barbara M. Chapman, Federico Massaioli, Matthias S. Müller, and Marco Rorro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 223–236. ISBN: 978-3-642-30961-8 (cit. on p. 25).

[108] Richard C. Murphy and Peter M. Kogge. "On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications". In: *IEEE Transactions on Computers* 56.7 (2007), pp. 937–945. DOI: 10.1109/TC.2007.1039 (cit. on pp. 4, 19).

[109] N. C. Nachiappan, P. Yedlapalli, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das. "Domain knowledge based energy management in handhelds". In: *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 150–160. DOI: 10.1109/HPCA.2015.7056029 (cit. on p. 17).

[110] H. Nakashima, H. Nakamura, M. Sato, T. Boku, S. Matsuoka, D. Takahashi, and Y. Hotta. "MegaProto: 1 TFlops/10kW Rack Is Feasible Even with Only Commodity Technology". In: *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. 2005, pp. 28–28. DOI: 10.1109/SC.2005.45 (cit. on p. 71).

[111] Carlos Navarro, Josue Feliu, Salvador Petit, Maria E. Gómez, and Julio Sahuquillo. "Bandwidth-Aware Dynamic Prefetch Configuration for IBM POWER8". In: *IEEE Transactions on Parallel and Distributed Systems* 31.8 (2020), pp. 1970–1982. DOI: 10.1109/TPDS.2020.2982392 (cit. on p. 14).

[112] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. "AC/DC: an adaptive data cache prefetcher". In: *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* 2004, pp. 135–145. DOI: 10.1109/PACT.2004.1342548 (cit. on p. 14).

[113] K. J. Nesbit and J. E. Smith. "Data cache prefetching using a global history buffer". In: *IEEE Micro* 25.1 (2005), pp. 90–97. ISSN: 0272-1732. DOI: 10.1109/MM.2005.6 (cit. on p. 13).

[114] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation". In: *SIGPLAN Not.* 42.6 (June 2007), pp. 89–100. ISSN: 0362-1340. DOI: 10.1145/1273442.1250746. URL: http://doi.acm.org/10.1145/1273442.1250746 (cit. on p. 26).

[115] *NVIDIA Management Library (NVML)*. https://docs.nvidia.com/deploy/nvml-api/index.html. 2019 (cit. on pp. 28, 96).

[116] *NVIDIA TESLA V100 GPU ARCHITECTURE*. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. 2017 (cit. on pp. 9, 16, 23).

[117] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.5*. http://www.openmp.org/mp-documents/openmp-4.5.pdf. Nov. 2015 (cit. on p. 19).

[118] V Pallipadi and A Starikovskiy. "The ondemand governor: Past, present and future". In: 2 (Jan. 2006), pp. 223–238 (cit. on p. 9).

[119] Vassilis Papaefstathiou, Manolis G.H. Katevenis, Dimitrios S. Nikolopoulos, and Dionisios Pnevmatikatos. "Prefetching and Cache Management Using Task Lifetimes". In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS '13. Eugene, Oregon, USA: Association for Computing Machinery, 2013, pp. 325–334. ISBN: 9781450321303. DOI: 10.1145/2464996.2465443. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/2464996.2465443 (cit. on p. 19).

[120] Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. "Integrated CPU-GPU Power Management for 3D Mobile Games". In: *Proceedings of the 51st Annual Design Automation Conference*. DAC'14. San Francisco, CA, USA: Association for Computing Machinery, 2014, pp. 1–6. ISBN: 9781450327305. DOI: 10.1145/2593069.2593151. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/2593069.2593151 (cit. on p. 18).

[121] Indrani Paul, Srilatha Manne, Manish Arora, W. Lloyd Bircher, and Sudhakar Yalamanchili. "Cooperative Boosting: Needy Versus Greedy Power Management". In: *Proceedings of the 40th Annual International Symposium on Computer*

Architecture. ISCA '13. Tel-Aviv, Israel: ACM, 2013, pp. 285–296. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485947. URL: http://doi.acm.org/10.1145/2485922.2485947 (cit. on pp. 18, 85, 86).

[122] Indrani Paul, Vignesh Ravi, Srilatha Manne, Manish Arora, and Sudhakar Yalamanchili. "Coordinated energy management in heterogeneous processors". In: *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–12. DOI: 10.1145/2503210.2503227 (cit. on pp. 18, 85, 86).

[123] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. "Using SimPoint for Accurate and Efficient Simulation". In: *SIGMETRICS Perform. Eval. Rev.* 31.1 (June 2003), pp. 318–319. ISSN: 0163-5999. DOI: 10.1145/885651.781076. URL: http://doi.acm.org/10.1145/885651.781076 (cit. on p. 75).

[124] Paula Petrica, Adam M. Izraelevitz, David H. Albonesi, and Christine A. Shoemaker. "Flicker: A Dynamically Adaptive Architecture for Power Limited Multicore Systems". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. Tel-Aviv, Israel: Association for Computing Machinery, 2013, pp. 13–23. ISBN: 9781450320795. DOI: 10.1145/2485922.2485924. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/2485922.2485924 (cit. on p. 14).

[125] Mihail Popov, Alexandra Jimborean, and David Black-Schaffer. "Efficient Thread/Page/Parallelism Autotuning for NUMA Systems". In: *Proceedings of the ACM International Conference on Supercomputing*. ICS '19. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 342–353. ISBN: 9781450360791. DOI: 10.1145/3330345.3330376. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3330345.3330376 (cit. on pp. 3, 10, 11).

[126] R. P. Pothukuchi, A. Ansari, P. Voulgaris, and J. Torrellas. "Using Multiple Input, Multiple Output Formal Control to Maximize Resource Efficiency in Architectures". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 658–670. DOI: 10.1109/ISCA.2016.63 (cit. on p. 15).

[127] Raghavendra Pradyumna Pothukuchi, Joseph L. Greathouse, Karthik Rao, Christopher Erb, Leonardo Piga, Petros G. Voulgaris, and Josep Torrellas. "Tangram: Integrated Control of Heterogeneous Computers". In: *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus, OH, USA: ACM, 2019, pp. 384–398. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358285. URL: http://doi.acm.org.recursos.

biblioteca.upc.edu/10.1145/3352460.3358285 (cit. on pp. 4, 18, 85, 86, 89, 94, 103, 104).

[128] Raghavendra Pradyumna Pothukuchi, Sweta Yamini Pothukuchi, Petros Voulgaris, and Josep Torrellas. "Yukta: Multilayer Resource Controllers to Maximize Efficiency". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 505–518. DOI: 10.1109/ISCA.2018.00049 (cit. on pp. 15, 89).

[129] Petar Radojković, Vladimir Čakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. "Optimal Task Assignment in Multithreaded Processors: A Statistical Approach". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. London, England, UK: Association for Computing Machinery, 2012, pp. 235–248. ISBN: 9781450307598. DOI: 10.1145/2150976.2151002. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/2150976.2151002 (cit. on pp. 3, 11).

[130] S. Rahman, M. Burtscher, Z. Zong, and A. Qasem. "Maximizing Hardware Prefetch Effectiveness with Machine Learning". In: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. 2015, pp. 383–389. DOI: 10.1109/HPCC-CSS-ICESS.2015.175 (cit. on pp. 14, 16).

[131] N. Rajovic, A. Rico, F. Mantovani, D. Ruiz, J. O. Vilarrubi, C. Gomez, L. Backes, D. Nieto, H. Servat, X. Martorell, J. Labarta, E. Ayguade, C. Adeniyi-Jones, S. Derradji, H. Gloaguen, P. Lanucara, N. Sanna, J. F. Mehaut, K. Pouget, B. Videau, E. Boyer, M. Allalen, A. Auweter, D. Brayford, D. Tafani, V. Weinberg, D. Brömmel, R. Halver, J. H. Meinke, R. Beivide, M. Benito, E. Vallejo, M. Valero, and A. Ramirez. "The Mont-Blanc Prototype: An Alternative Approach for HPC Systems". In: *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, pp. 444–455. DOI: 10.1109/SC.2016.37 (cit. on p. 71).

[132] Nikola Rajovic, Alejandro Rico, Nikola Puzovic, Chris Adeniyi-Jones, and Alex Ramirez. "Tibidabo11Tibidabo is a mountain overlooking Barcelona.: Making the case for an ARM-based HPC system". In: *Future Generation Computer Systems* 36 (2014). Special Section: Intelligent Big Data Processing Special Section: Behavior Data Security Issues in Network Information Propagation Special Section: Energy-efficiency in Large Distributed Computing Architectures Special Section: eScience Infrastructure and Applications, pp. 322–334. ISSN: 0167-739X. DOI: http://dx.doi.org/10.1016/j.future.2013.

07.013. URL: http://www.sciencedirect.com/science/article/pii/S0167739X13001581 (cit. on p. 71).

[133] Karthik Rao, Jun Wang, Sudhakar Yalamanchili, Yorai Wardi, and Handong Ye. "Application-Specific Performance-Aware Energy Optimization on Android Mobile Devices". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 169–180. DOI: 10.1109/HPCA.2017.32 (cit. on p. 90).

[134] S. Reda, R. Cochran, and A. K. Coskun. "Adaptive Power Capping for Servers with Multithreaded Workloads". In: *IEEE Micro* 32.5 (2012), pp. 64–75. ISSN: 0272-1732. DOI: 10.1109/MM.2012.59 (cit. on pp. 16, 17, 85).

[135] Haris Ribic and Yu David Liu. "AEQUITAS: Coordinated Energy Management Across Parallel Applications". In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS '16. Istanbul, Turkey: ACM, 2016, 4:1–4:12. ISBN: 978-1-4503-4361-9. DOI: 10.1145/2925426.2926260. URL: http://doi.acm.org.recursos.biblioteca.upc.edu/10.1145/2925426.2926260 (cit. on p. 85).

[136] Todd Rosedahl, Martha Broyles, Charles Lefurgy, Bjorn Christensen, and Wu Feng. "Power/Performance Controlling Techniques in OpenPOWER". In: *High Performance Computing*. Ed. by Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf. Cham: Springer International Publishing, 2017, pp. 275–289. ISBN: 978-3-319-67630-2 (cit. on pp. 16, 28, 96).

[137] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. "Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge". In: *IEEE Micro* 32.2 (2012), pp. 20–27. ISSN: 0272-1732. DOI: 10.1109/MM.2012.12 (cit. on p. 16).

[138] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y (cit. on p. 27).

[139] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke. "IBM Power9 Processor Architecture". In: *IEEE Micro* 37.2 (2017), pp. 40–51. ISSN: 0272-1732. DOI: 10.1109/MM.2017.40 (cit. on pp. 16, 23).

[140] Isaac Sánchez Barrera, David Black-Schaffer, Marc Casas, Miquel Moretó, Anastasiia Stupnikova, and Mihail Popov. "Modeling and Optimizing NUMA Effects and Prefetching with Machine Learning". In: *Proceedings of the 34th ACM International Conference on Supercomputing*. ICS '20. Barcelona, Spain: Association for Computing Machinery, 2020. ISBN: 9781450379830.

DOI: 10.1145/3392717.3392765. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3392717.3392765 (cit. on pp. 10, 11, 15).

[141] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. "Applying Deep Learning to the Cache Replacement Problem". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Association for Computing Machinery, 2019, pp. 413–425. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358319. URL: http://doi.org/10.1145/3352460.3358319 (cit. on p. 16).

[142] A. J. Smith. "Sequential Program Prefetching in Memory Hierarchies". In: *Computer* 11.12 (1978), pp. 7–21. ISSN: 0018-9162. DOI: 10.1109/C-M.1978.218016 (cit. on p. 13).

[143] Allan Snavely and Dean M. Tullsen. "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor". In: *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IX. Cambridge, Massachusetts, USA: Association for Computing Machinery, 2000, pp. 234–244. ISBN: 1581133170. DOI: 10.1145/378993.379244. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/378993.379244 (cit. on pp. 10, 11).

[144] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers". In: *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. HPCA '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 63–74. ISBN: 1-4244-0804-0. DOI: 10.1109/HPCA.2007.346185. URL: http://dx.doi.org/10.1109/HPCA.2007.346185 (cit. on pp. 14, 72).

[145] G. Sun, J. Shen, and A. V. Veidenbaum. "Combining Prefetch Control and Cache Partitioning to Improve Multicore Performance". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 953–962. DOI: 10.1109/IPDPS.2019.00103 (cit. on p. 15).

[146] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. "Rethinking the Inception Architecture for Computer Vision". In: *CoRR* abs/1512.00567 (2015). arXiv: 1512.00567 (cit. on p. 27).

[147] Stephen J. Tarsa, Rangeen Basu Roy Chowdhury, Julien Sebot, Gautham Chinya, Jayesh Gaur, Karthik Sankaranarayanan, Chit-Kwan Lin, Robert Chappell, Ronak Singhal, and Hong Wang. "Post-Silicon CPU Adaptation Made Practical Using Machine Learning". In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA '19. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 14–26. ISBN: 9781450366694. DOI:

10.1145/3307650.3322267. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3307650.3322267 (cit. on p. 16).

[148] Priyanka Tembey, Augusto Vega, Alper Buyuktosunoglu, Dilma Da Silva, and Pradip Bose. "SMT Switch: Software Mechanisms for Power Shifting". In: vol. 12. 2. 2013, pp. 67–70. DOI: 10.1109/L-CA.2012.26 (cit. on p. 11).

[149] Zeynep Toprak-Deniz, Michael Sperling, John Bulzacchelli, Gregory Still, Ryan Kruse, Seongwon Kim, David Boerstler, Tilman Gloekler, Raphael Robertazzi, Kevin Stawiasz, Timothy Diemoz, George English, David Hui, Paul Muench, and Joshua Friedrich. "5.2 Distributed system of digitally controlled microregulators enabling per-core DVFS for the POWER8^{TM} microprocessor". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 98–99. DOI: 10.1109/ISSCC.2014.6757354 (cit. on p. 16).

[150] François Trahay, Manuel Selva, Lionel Morel, and Kevin Marquet. "NumaMMA: NUMA MeMory Analyzer". In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP 2018. Eugene, OR, USA: Association for Computing Machinery, 2018. ISBN: 9781450365109. DOI: 10.1145/3225058.3225094. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3225058.3225094 (cit. on p. 19).

[151] "Trinity benchmarks. http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/". In: 2016 (cit. on p. 25).

[152] Mateo Valero, Miquel Moreto, Marc Casas, Eduard Ayguade, and Jesus Labarta. "Runtime-Aware Architectures: A First Approach". In: *Supercomput. Front. Innov.: Int. J.* 1.1 (2014), pp. 29–44. ISSN: 2409-6008. DOI: 10.14529/jsfi140102. URL: https://doi-org.recursos.biblioteca.upc.edu/10.14529/jsfi140102 (cit. on pp. 18, 19).

[153] Augusto Vega, Alper Buyuktosunoglu, Heather Hanson, Pradip Bose, and Srinivasan Ramani. "Crank It Up or Dial It Down: Coordinated Multiprocessor Frequency and Folding Control". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 210–221. ISBN: 978-1-4503-2638-4. DOI: 10.1145/2540708.2540727. URL: http://doi.acm.org.recursos.biblioteca.upc.edu/10.1145/2540708.2540727 (cit. on pp. 3, 10, 16, 17, 29, 85).

[154] H. Wang, V. Sathish, R. Singh, M. J. Schulte, and N. S. Kim. "Workload and power budget partitioning for single-chip heterogeneous processors". In: *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2012, pp. 401–410. DOI: 10.1145/2370816.2370873 (cit. on pp. 85, 86).

[155] Wei Wang and Edgar Leon. "Evaluating DVFS and Concurrency Throttling on IBM's Power8 Architecture". In: Nov. 2015 (cit. on p. 9).

[156] Xiaodong Wang and José F. Martínez. "ReBudget: Trading Off Efficiency vs. Fairness in Market-Based Multicore Resource Allocation via Runtime Budget Reassignment". In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*. Ed. by Tom Conte and Yuanyuan Zhou. ACM, 2016, pp. 19–32. DOI: 10.1145/2872362.2872382. URL: https://doi.org/10.1145/2872362.2872382 (cit. on p. 89).

[157] Xiaodong Wang and Jose F. Martínez. "XChange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 113–125. DOI: 10.1109/HPCA.2015.7056026 (cit. on pp. 17, 85, 86, 103).

[158] Zheng Wang and Michael F.P. O'Boyle. "Mapping Parallelism to Multi-Cores: A Machine Learning Based Approach". In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '09. Raleigh, NC, USA: Association for Computing Machinery, 2009, pp. 75–84. ISBN: 9781605583976. DOI: 10.1145/1504176.1504189. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/1504176.1504189 (cit. on p. 11).

[159] Zhenlin Wang, D. Burger, K.S. McKinley, S.K. Reinhardt, and C.C. Weems. "Guided region prefetching: a cooperative hardware/software approach". In: *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* 2003, pp. 388–398. DOI: 10.1109/ISCA.2003.1207016 (cit. on p. 14).

[160] J. Weinberg, M.O. McCracken, E. Strohmaier, and A. Snavely. "Quantifying Locality In The Memory Access Patterns of HPC Applications". In: *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. 2005, pp. 50–50. DOI: 10.1109/SC.2005.59 (cit. on p. 19).

[161] Jonathan A. Winter, David H. Albonesi, and Christine A. Shoemaker. "Scalable thread scheduling and global power management for heterogeneous many-core architectures". In: *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2010, pp. 29–39 (cit. on pp. 17, 85).

[162] Margaret Wright and al. "The opportunities and challenges of exascale computing". In: http://science.energy.gov/ (2010). URL: https://science.energy.gov/~/media/ascr/ascac/pdf/reports/exascale_subcommittee_report.pdf (cit. on p. 71).

[163] C. Wu and M. Martonosi. "Characterization and dynamic mitigation of intra-application cache interference". In: *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. 2011, pp. 2–11. DOI: 10.1109/ISPASS.2011.5762710 (cit. on pp. 3, 11).

[164] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely, and Joel Emer. "PACMan: Prefetch-Aware Cache Management for High Performance Caching". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. Porto Alegre, Brazil: Association for Computing Machinery, 2011, pp. 442–453. ISBN: 9781450310536. DOI: 10.1145/2155620.2155672. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/2155620.2155672 (cit. on p. 13).

[165] Wm. A. Wulf and Sally A. McKee. "Hitting the Memory Wall: Implications of the Obvious". In: *SIGARCH Comput. Archit. News* 23.1 (1995), 20–24. ISSN: 0163-5964. DOI: 10.1145/216585.216588. URL: https://doi.org/10.1145/216585.216588 (cit. on p. 1).

[166] Huazhe Zhang and Henry Hoffmann. "Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 545–559. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872375 (cit. on pp. 16, 17, 85, 89).

[167] Y. Zhang, M. Voss, and E. S. Rogers. "Runtime Empirical Selection of Loop Schedulers on Hyperthreaded SMPs". In: 19th IEEE International Parallel and Distributed Processing Symposium. 2005, 44b–44b. DOI: 10.1109/IPDPS.2005.386 (cit. on pp. 3, 11, 29).

[168] Yun Zhang, Mihai Burcea, Victor Cheng, Ron Ho, and Michael Voss. "An adaptive OpenMP loop scheduler for Hyperthreaded SMPs". In: Jan. 2004, pp. 256–263 (cit. on pp. 3, 11, 29).

[169] Xiaotong Zhuang and Hsien-hsin S. Lee. "Reducing Cache Pollution via Dynamic Data Prefetch Filtering". In: *IEEE Transactions on Computers* 56.1 (2007), pp. 18–31. DOI: 10.1109/TC.2007.250620 (cit. on p. 13).