Runtime-Guided Mitigation of Manufacturing Variability in Power-Constrained Multi-Socket NUMA Nodes

Dimitrios Chasapis*† Marc Casas*† Miquel Moretó*†
Martin Schulz‡ Eduard Ayguadé*† Jesus Labarta*†
Mateo Valero*†

*Barcelona Supercomputing Center
29 Jordi Girona
Barcelona, Spain
name.surname@bsc.es

†Universitat Politécnica de Catalunya
Campus Nord
Barcelona, Spain

‡Lawrence Livermore National Laboratory
7000 East Avenue
Livermore, CA, US
schulz6@llnl.gov

ABSTRACT

Current large scale systems show increasing power demands, to the point that it has become a huge strain on facilities and budgets. Researchers in academia, labs and industry are focusing on dealing with this "power wall", striving to find a balance between performance and power consumption. Some commodity processors enable power capping, which opens up new opportunities for applications to directly manage their power behavior at user level. However, while power capping ensures a system will never exceed a given power limit, it also leads to a new form of heterogeneity: natural manufacturing variability, which was previously hidden by varying power to achieve homogeneous performance, now results in heterogeneous performance caused by different CPU frequencies, potentially for each core, to enforce the power limit.

In this work we show how a parallel runtime system can be used to effectively deal with this new kind of performance heterogeneity by compensating the uneven effects of power capping. In the context of a NUMA node composed of several multi-core sockets, our system is able to optimize the energy and concurrency levels assigned to each socket to maximize performance. Applied transparently within the parallel runtime system, it does not require any programmer interaction like changing the application source code or manually reconfiguring the parallel system. We compare our novel runtime analysis with an offline approach and demonstrate that it can achieve equal performance at a fraction of the cost.

ICS '16, June 01- 03, 2016, Istanbul, Turkey

DOI: http://dx.doi.org/10.1145/2925426.2926279

CCS Concepts

•Computer systems organization → Multicore architectures; •Hardware → Power and energy;

Keywords

Parallel Architectures; Power and Energy; Manufacturing Variability; Pararallel Runtimes; Parallel Programming; High Performance Computing

1. INTRODUCTION

One major constraint of future High-Performance Computing (HPC) systems is their power consumption. Agencies have set strict targets for building an exascale machine — e.g., the US Department of Energy has set the limit to 20MW [28] — while others, like the European Union, are investing in novel approaches leveraging mobile technologies to build low-power HPC infrastructures [25]. The resulting need for reducing hardware power consumption has started to force computer architects and vendors to include power capping capabilities in their hardware designs. This allows applications to more efficiently exploit their entire power envelope of a system, while guarding the system against intermediate power spikes. Prior work has shown that this can lead to significant performance benefits [24, 23].

Manufacturing variability, however, causes processors and DRAM memories to react inhomogeneously to power constraints enforced by the system. While already present in current systems, such variability has so far been hidden by varying power consumption to achieve homogeneous performance. In fact, existing studies show a variation of up to 10% in power consumption to deliver the same performance [27]. With the ability to vary power removed by imposing a particular power limit, this variation becomes visible in realized performance [18]. Further, this uneven distribution of delivered performance is specific to each single hardware component, since two nominally identical processors can suffer from different degrees of manufacturing issues. From the HPC applications perspective, this can cause load imbalances, even if the workload is perfectly balanced, resulting in significantly degraded performance. To make this problem even worse, since such degradations are

hardware specific, it is not possible to design static or hardware agnostic techniques to mitigate this induced new type of load imbalance.

In this paper, we present a runtime guided hardware/software reconfiguration approach that effectively mitigates the effects of inhomogeneous hardware behavior in low power environments. Further, we demonstrate that classical work stealing and load balancing techniques [4, 3, 26, 36] are insufficient to mitigate this performance issue. In the context of a NUMA node composed of several multi-core sockets, our technique is able to efficiently distribute a total node power budget among the node's different sockets, while also adjusting their corresponding concurrency levels. In order to enable this, our approach dynamically selects the best power/concurrency level for each socket involved in the computation by performing a light weight initial training phase. This initial training phase selects the optimal power/concurrency level to be assigned to each socket to reduce applications' load imbalance induced by power/performance inhomogeneity and thus increase performance.

The contributions of our paper are as follows:

- We provide a precise description of the limitations of current, state of the art load balancing techniques when dealing with inhomogeneous hardware behavior under node-level power limits.
- We demonstrate how uneven power and thread assignments to sockets can mitigate the inhomogeneous hardware behavior in dual socket NUMA nodes, resulting in up to 30% increased performance for some applications.
- We describe a dynamic runtime technique to discover the optimal power/concurrency assignment for each application on a given parallel machine that provides up to 22% performance improvements for some applications.

This paper is organized in the following way: Section 2 describes the irregular behavior that multi-core architectures exhibit when operating under restricted power budgets and discusses ways to mitigate it. Section 5 shows a systematic runtime solution to mitigate this behavior. Section 6 displays a comprehensive evaluation of the techniques introduced in this work. Section 8 summarizes the contributions of this paper.

2. HOMOGENEOUS MACHINES ARE IN FACT HETEROGENEOUS

HPC systems are becoming increasingly power hungry, as we keep pushing the boundaries of performance on the road to exascale and beyond. While significant advances have been made in increasing the power efficiency of each single hardware component, i. e., flop/watt ratios continue to decrease driven by significant architecture advances, these savings are not enough to compensate for the growth in terms of computational elements required to realize the needed performance advances. Consequently, we need to build systems that use power more efficiently and ensure that any power provisioned for a system is also used and turned into realized performance, i.e., we need systems that can dynamically manage their power budgets among the available hardware components to direct power where it is needed.

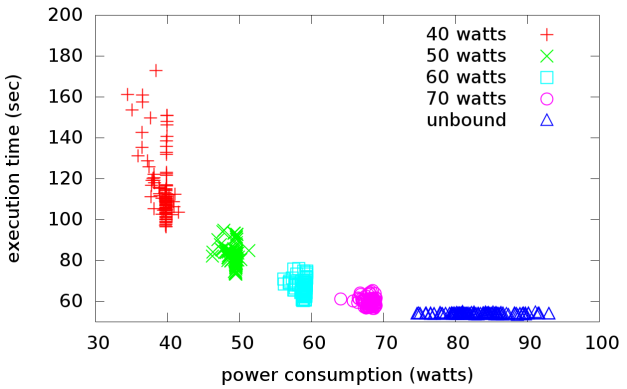


Figure 1: Performance obtained when the `freqmine` application is run on 64 different 12-core Intel Xeon E5-2695v2 sockets under different power budgets.

Current machines are "worst-case provisioned", i.e., all components of a system can be powered at the same time without reaching the system's power limit. Since applications rarely keep all components occupied[1], this conservative approach leads to "wasted power", i.e., provisioned power that is not used. Prior studies show that this wasted power can be up to 30% of a system's power rating [24, 23]. One solution is to reduce the provisioned power to the expected average power consumption, or even lower, allowing systems to exploit all available power, and, consequentially, allowing for larger systems at the same total provisioned power. In such systems, which we refer to as "overprovisioned systems", though, we must cap power to avoid power spikes caused by intermittent phases to exceed the provisioned power and with that endanger the operation of the entire system.

Many current architectures either already provide such power capping mechanisms or have them on their near term road map. However, such capping doesn't come for free: it impacts performance, as show by the results of some initial experiments in Figure 1. The graph shows timing and power consumption of multiple runs of the `freqmine` code from the PARSEC benchmark suite [2] on 64 nodes of the Lawrence Livermore National Laboratory (LLNL) Catalyst cluster [22], using 12 threads per execution. `Freqmine` has been adapted to use OpenMP-like task-based parallelism [9] and runs in the top of the Nanos++ (v0.7a) parallel runtime system [6]. Since each node is composed of two 12-core Intel Xeon E5-2695v2 sockets, our experiments involve 128 different 12-core sockets and each run is limited to one of these 128 sockets. We consider five different power bounds: 40, 50, 60, 70W and unlimited per socket[2] The x-axis shows the measured power consumption for each execution of the `Freqmine` benchmark while in the y-axis we show the corresponding execution time. We can see that running without a power bound results in almost no performance variation, but exhibits a wide spread of power consumption. Under a power bound, this power variation is no longer possible and we can see a drastic impact on performance variation instead. Further, we can see that lower bounds result in

[1]It's a well known fact that many applications only run at a fraction of peak performance — often way below 10%

[2]The TDP for each socket was 115W.

higher variations.

This behavior can be explained by natural manufacturing variability, which causes different processors to exhibit different efficiencies. The consequence of this phenomena is, though, that nominally homogeneous NUMA turn into heterogeneous systems when operated under a power cap equally applied to each socket.

3. DEALING WITH MANUFACTURING VARIABILITY

In order to mitigate this performance inhomogeneity caused by manufacturing variability, current static load balancing and scheduling mechanisms are insufficient, since they are only based on workloads and do not take into account dynamic variability. Classical dynamic work stealing and load balancing techniques may mitigate [4, 3, 26, 36] this problem under certain circumstances, but they are not enough when dealing with complex codes with frequent synchronization points. In the following, we illustrate the limitations of dynamic load balancing techniques on power limited scenarios by means of two examples: one considering a code with no barriers (Section 3.1) and another one with many (Section 3.2). In both examples, the considered applications belong to the PARSEC benchmark suite, but have been adapted to use OpenMP task-based parallelism [9] and executed on top of the Nanos++ (v0.7a) runtime system [6].

3.1 Example with no Barrier or Synchronizations

Figure 2 compares two executions of the `swaptions` benchmark [9] on a NUMA node composed of two 12-core Intel Xeon E5-2695v2 sockets, each run with 24 threads and with power capped at 40W. The x-axis of the figures show time and the y-axis the activity of each of the 24 threads involved in the parallel execution. When the activity of a particular thread i appears in red on time t, the thread is doing useful work; if it appears in white, the thread is idling. The scale of the x-axis is the same in both figures and covers 0s to 52.8s. In the run shown in Figure 2a the load is evenly distributed among all threads statically using a naive distribution. The reaction of the two sockets involved in the parallel run is different, which makes the threads running on the faster socket (Threads 1-8) finish much earlier than the threads mapped to the slow socket (Threads 9-15). As a result, threads from 1 to 8 are idle for 26% of the execution time.

In Figure 2b we show a second parallel execution of the same code, performed in the same NUMA node as above, but with dynamic scheduling. For this, we have over-decomposed the parallel execution into more tasks than cores and let the parallel runtime system assign tasks to cores once they were idle. In this way, the cores on the fast socket executed some of the tasks that were assigned to cores on the slower socket in the static case, which allows the whole parallel execution to achieve a 1.13x speedup over static scheduling. This dynamic task assignment technique is equivalent to the numerous work stealing approaches described in the literature [4, 3, 26, 36]: it is able to deal with uneven hardware responses under restricted power budgets in the absence of barriers or synchronization points. Note however, that in order for conventional work stealing to be effective, finer grain parallelism is prefered. Introducing synchronization and coarser parallel work unit limit the effectiveness of this method.

3.2 Example with Barrier Operations

Figure 3 compares the behavior of two parallel executions of the `blackscholes` benchmark [9] on the same NUMA node as the one used above, again limited to 40W per socket. In this case we show the behavior of the parallel run around a barrier operation instead of the whole execution. The x-axis represents time and the y-axis shows the threads involved in the parallel execution. Green flags mark the separation between the different pieces of sequential work in which the parallel execution is split or, in other words, the tasks.

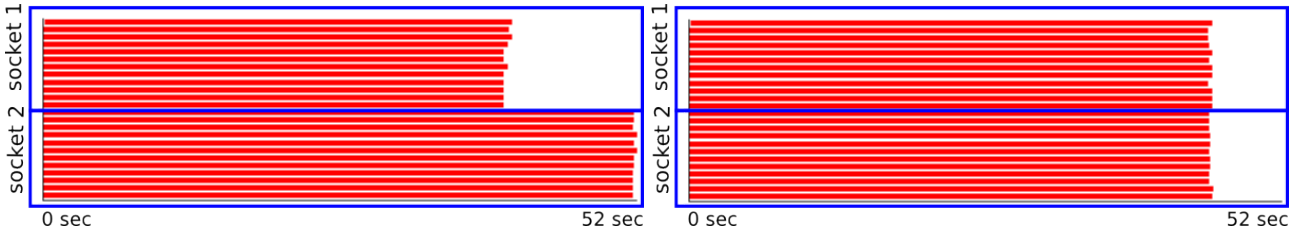
Figure 3a shows the behavior of the dynamic task scheduling using the same policy as above, with idle time shown in white. While in the absence of barriers this technique properly balances the load between the two 12-core sockets, the results in this case clearly show that they fail in case of barriers: the green flags show that the same tasks exhibit differences in execution time depending on the socket they run on: around 74μs on average when run on the slow socket and 58 when run on the fast one, despite each task executing the same computational workload.

Figure 3b shows a second execution with the number of threads per socket reduced to 10, i.e., 2 cores per socket or 4 cores total are left unused during the execution. In this example power is evenly distributed with 40W per socket. The average execution time of the tasks mapped to the slow socket gets reduced to 54μs, while the average time of those mapped to the fast socket takes 48μs to run. This improved execution time is caused by the fact that the socket power budget is now distributed among 10 cores instead of 12. More importantly, the heterogeneous character of the socket's response to the imposed power limit seems to be reduced by leaving 2 cores idle. This better balance between the two sockets significantly reduces the impact of barriers and, therefore, their idle time. Clearly, this is a much more balanced execution than the one shown in Figure 3a. Overall, the parallel run considering 10 cores per socket and dynamic task assignment shows a 1.21x speedup with respect to the execution with 12 cores per socket combined with a dynamic assignment.

This last example clearly shows that, under restricted power budgets and uneven hardware reactions, operating with the maximum possible concurrency while dynamically balancing load is insufficient since barrier points can introduce significant idling effects. In this cases, it can be better to restrict concurrency levels in order to homogenize the hardware reaction to low power budgets. Alternatively, it can also be helpful to unevenly distribute the total power budget assigned to the multi-core sockets of a NUMA node in order to compensate for varying processor efficiency, as we demonstrate in Section 4.

4. MITIGATING HETEROGENEITY

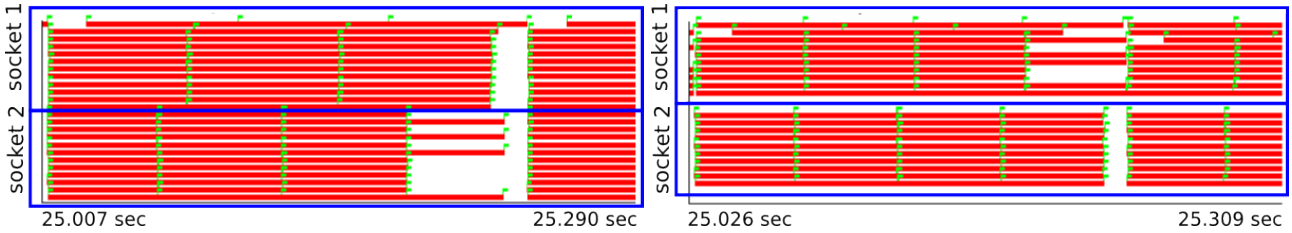
Following Sections 3.1 and 3.2, which illustrate the negative impact of heterogeneity introduced by power capping, we now provide a general evaluation of the benefits of heterogeneity mitigation. We consider a wide range of parallel applications coming from many areas and we test their performance considering a large range of power and concurrency configurations. For each application and power bound, we select the best configuration and compare its performance with the performance obtained by deploying the naive even configuration — assign half the power to each thread and



(a) Static scheduling and 12 cores enabled.

(b) Dynamic scheduling and 12 cores enabled.

Figure 2: Executions of `swaptions` under 40 W power capping.



(a) Dynamic Scheduling and 12 cores enabled.

(b) Dynamic Scheduling and 10 cores enabled.

Figure 3: Executions of `blackscholes` near a synchronization point under 40 W.

use all the available cores — combined with traditional task scheduler and balancer.

4.1 Experimental Setup

Applications: we utilize nine OpenMP codes: six of them come from the PARSEC benchmark suite [2, 9] (`blackscholes`, `ferret`, `fluidanimate`, `freqmine`, `streamcluster` and `swaptions`). Two of them are dense linear algebra routines (a cholesky matrix factorization, `cholesky`, and a QR communication-avoiding code, `qrca` [12]) and another one builds a histogram from a set of data points (`inthist`). All of these codes exploit task-based parallelism.

Hardware and System Software: NUMA nodes of the Catalyst supercomputer [22] are composed of two 12-core Intel Xeon E5-2695v2 sockets each. The applications run in top of the Nanos++ (v0.7a) parallel runtime system [6]. We map one thread per active core. To set power constraints and measure power consumption on each socket, we use Intel's RAPL [19] . These registers are accessed by our modified version of the Nanos++ runtime using the libMSR library [31].

Configurations: We consider power bounds of 80W, 100W and 120W for total node power. If we allow a power limit of 80W, we consider 5 different ways of distributing the power among the two sockets of the NUMA node: 30W:50W, 35W:45W, 40W:40W, 45W:35W and 50W:30W as well as 36 ways of specifying the maximum concurrency allowed in each 2-socket NUMA node: 2-2, 4-2, 6-2, 8-2, 10-2, 12-2, 2-4, etc. up to 12-12. In total, this leads to a total of 180 combinations. Similarly, when allowing a power limit of 100W there are 8 ways of distributing it, which combined with the 36 possible ways of distributing the concurrency, leads us to a total of 324 combinations. Similarly, when the total power budget reaches 120W, the total number of combinations is 468. Overall, for each particular application we have 972 different combinations.

Other Considerations: The results of these experiments are machine dependent since each particular 12-core socket reacts in a different way when a power limit is set. Ideally, all 972 configurations per application should be executed on many NUMA nodes to really account for many possible hardware reactions when a power limit is set. However, due to the size of our experimental campaign, we randomly chose a single 2-socket NUMA node for each considered application and run all 972 combinations on it. Although this random choice can slightly influence the relative results between the benchmarks, the general conclusions we extract from them remain unchanged.

4.2 Evaluation

In Figure 4 we show our experimental results. On the x-axis we represent all the considered applications and the three power bounds we consider: 80W, 100W and 120W. In the y-axis we represent, for each particular application, the speedup achieved over evenly distributing 80W among two sockets (40W per socket) and keeping 12 active cores per socket. On average, the optimal configurations outperforms the totally even distribution (50% of the power and 12 cores per socket) by 11.8% (80W), 7.3% (100W) and 7.6% (120W).

Not surprisingly, the more restrictive the power capping is, the more beneficial the optimal configuration becomes in terms of performance. The uneven hardware reaction gets exacerbated by restrictive power bounds, which gives more room for improvement when the hardware is rebalanced by changing the power and concurrency assignation per socket. Application-wise, the benefits are much larger for applications composed of several execution phases separated by barriers, like `fluidanimate` (24% improvement) or `cholesky` (30%). On the other side, `swaptions` does not get any benefit from our power rebalancing techniques since its lack of barriers enables simple load balancing schemes to mitigate the hardware heterogeneous response, as described in Section 3.1.

In Table 1 we list the optimal configuration for each application and power bound. As expected, for applications without barriers (`swaptions` and `inthist`) the most balanced

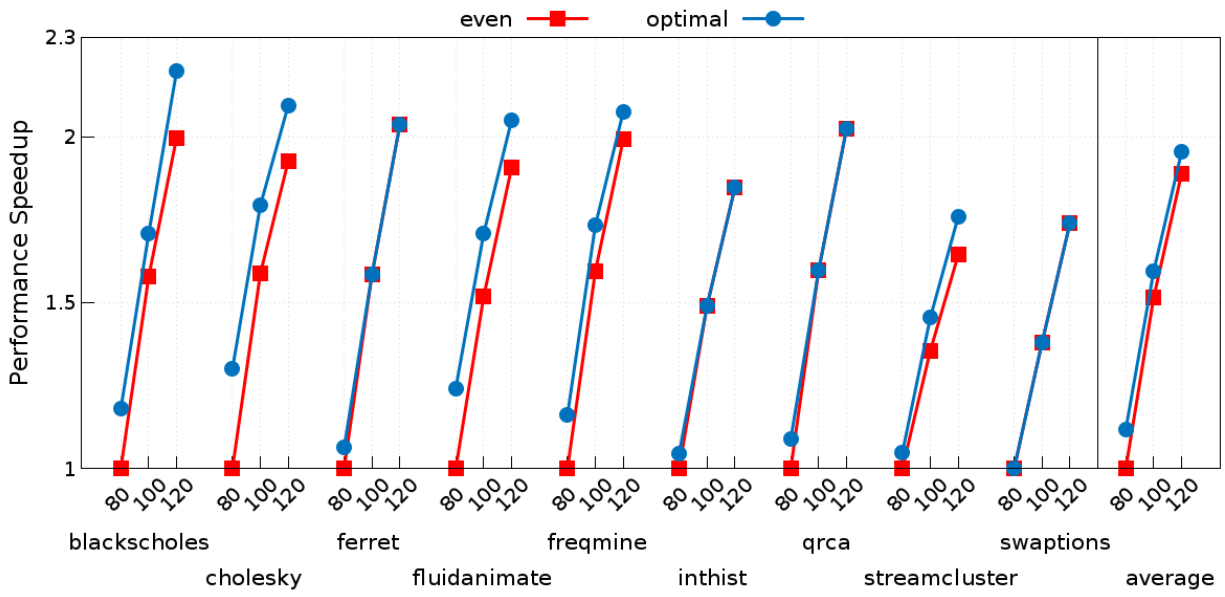


Figure 4: Comparison between the even and the best configuration observed by application profiling. The speedup is computed over the execution of the even resource distribution for 80 W.

Table 1: Optimal configurations per application and power bound in terms or Watts and active cores per socket

	80 W	100 W	120 W
`blackscholes`	40-40 W, 10-10 cores	55-45 W, 10-12 cores	70-50 W, 12-10 cores
`cholesky`	30-50 W, 2-12 cores	35-65 W, 2-12 cores	30-90 W, 10-10 cores
`ferret`	40-40 W, 10-10 cores	50-50 W, 12-12 cores	60-60 W, 12-12 cores
`fluidanimate`	45-35 W, 10-6 cores	55-45 W, 10-6 cores	65-35 W, 10-6 cores
`freqmine`	45-35 W, 12-6 cores	55-45 W, 10-12 cores	65-55 W, 12-12 cores
`inthist`	40-40 W, 10-10 cores	45-55 W, 12-12 cores	60-60 W, 12-12 cores
`qrca`	45-35 W, 12-6 cores	50-50 W, 12-12 cores	60-60 W, 12-12 cores
`streamcluster`	35-45 W, 2-12 cores	60-40 W, 12-12 cores	65-55 W, 12-12 cores
`swaptions`	40-40 W, 12-12 cores	50-50 W, 12-12 cores	60-60 W, 12-12 cores

configurations (40W:40W and 12-12 active cores, 40W:40W and 10-10 cores respectively) are optimal. Since for these applications the parallel runtime system successfully manages the load, there is no need for system balancing by means of power or concurrency reassignment among the involved threads. On the other side, applications like `cholesky` or `fluidanimate` do really benefit from leaving significant parts of the cores idle and rebalancing the power accordingly. Clearly, the hardware heterogeneity induced by setting a power bound is not compensated by load balancing schemes delivered at the parallel runtime system side and some concurrency and power rebalancing must be done to maximize the performance of these applications.

This evaluation demonstrates that classical work stealing and load balancing techniques are not able to compensate the heterogeneity induced by power capping, except for trivial situations where a parallel code has no global barriers or synchronization points and the size of the parallel work unit is small enough to allows versatile alternative scheduling scenarios. Since the potential benefits of power and concurrency rebalancing is up to 30%, there is a need for developing techniques able to figure out the optimal configuration within a single execution run.

5. RUNTIME APPROACH

While the previous experiments demonstrate the potential benefits of unevenly setting up the power caps and the number of active cores per socket in a power-constrained NUMA node, these benefits are obtained under the huge cost of running each application multiple times in the targeted NUMA node, each time with a particular power and active cores limit. Further, the results obtained using extensive search on a particular NUMA node are not applicable to another one since the hardware response to low power bounds are driven by manufacturing variability and cannot be known in advance. Also, deriving a particular performance ratio per power bound among the sockets contained in a NUMA node is not enough as different applications react in a different way to such variability. It is thus necessary to develop techniques able to quickly determine the optimal power-# of cores distribution for a particular software component and NUMA node.

We implement our method at the runtime level, since such systems offer load-balancing and can be easily extended with additional functionality. They are also widely used and expected to play a significant role in future parallel architectures [35, 8].

5.1 Exploiting Application Structure

Parallel codes often decompose loops or segments of serial code into multiple work units that run in parallel. While (at least to date) many codes follow a simple Single Program Multiple Data (SPMD) approach where multiple cores execute the same code several times, even more complex patterns, different repetitions of loops that iterate over similar sets of data several times produce similar execution patterns. As a consequence, codes almost always exhibit a certain degree of repetitive behavior that can be observed either over time or by considering the logical execution structure, which is composed of event sequences [20, 33, 7]. This iterative nature of parallel applications allows us to effectively guide the whole application behavior by observing only small but significant portions of the parallel execution. Our approach considers execution segments and associates each with a particular power and number of active core assignation per socket. This makes it possible to explore many different configurations on representatives code sections in a single run, which can the be used in an online search.

5.2 Search Algorithm

The search algorithm aims to find the optimal power and total number of active cores balance among the different sockets of a NUMA node. It starts with evenly distributing power and activating all cores and then progressively iterates over a set of power/#cores configurations and selects the best one. Per each configuration, the targeted application runs for a certain amount of time. The particular amount of time each configuration runs for is a parameter we call *monitoring window*. The smaller this parameter, the shorter the exploration, but the more chances of getting a non-optimal configuration since the amount of time it has been trained for may not be representative of the whole execution. On the other hand, large window sizes significantly increase the chance of finding the right configuration, but make the algorithmic search phase larger.

To characterize the performance achieved by each configuration we use a *throughput metric* defined as the number of tasks executed during the monitoring window each configuration runs for. This metric is well defined for all applications we consider in this paper (see Section 4.1) and is particularly well-suited since it also implicitly captures the amount of idle time spent by the active cores. Further, it does not imply a significant amount of measurement overhead if the task granularity is kept over the tens of μs threshold. Although this metric is specific for task-based codes, any other light-weight metric able to capture the amount of time spent doing useful work would provide similar results for other kinds of applications or programming models.

During the first monitoring window, the runtime system measures the throughput of evenly distributing power among the sockets and using all the available cores. This is considered the best candidate until a configuration providing larger throughput is observed. After each iteration we then compare the throughput for the current profile with the best one. If the current one is better, it becomes the new best and is used for the subsequent comparisons. This analysis continues until the search space is exhausted, which may require more than one application run. At the end of each run, if we have not yet exhausted the search space, the runtime saves a checkpoint of the analysis state and resumes it in a succeeding run.

Special care must be taken to make sure that we are considering monitoring windows that constitute representative execution segments. If two windows capture different task types comparing them is not fair since different tasks have different execution times. To address this issue, we keep a set of task types for each different window. If the task sets collected during the best and current windows are not equal, it could mean that the two configurations were run at a different stages of the application's execution. We call these incompatible profile results *mismatching windows*. When this occurs, we ignore the current configuration without comparing it to the best and continue by checking another configuration over the next monitoring window. Special care need to be taken for the first monitoring window. If the first and second windows mismatch, we discard both and retrain the first configuration. This will continue until we capture a representative segment of the application, meaning that two consecutive windows will be matching.

Note that different alternatives are available when dealing with mismatching windows. However, for this work we employ the simplest case, which is to discard it.

The search algorithm looks for the best configuration after trying several ones and measuring their throughput over their corresponding monitoring windows. As explained, the monitoring windows size is an input parameter of our search algorithm. The algorithm's sensitivity to the windows size and its optimal value are explored in detail in Section 6.1. Also, the set of configurations the search algorithm iterates over is a key choice. Large sets increase the chances of getting the optimal power/#active cores balance per socket, but also increases the cost of running the search. Alternatively, reduced sets may produce cheaper searches but also be unable to find configurations that significantly improve performance.

5.3 Training Sets

We have implemented four variations of our analysis, based on the size of the configurations sets:

Exhaustive Search: We use the different configurations defined in Section 4.1. As discussed above, in case we target a 80W power bound, the exhaustive search considers 180 different configurations. This is a conservative, but expensive analysis.

Naive Scoped Search: The scoped search does not consider extremely unbalanced configurations since they rarely produce the most optimal results. As a general rule we focus the search on a small area around the default balanced configuration. The reasoning here is that just slightly providing more power or reducing the concurrency in the slower socket will mitigate the imbalance between the sockets. The scoped search considers 80 different configurations for the 80W bound. They are composed of five different power configurations (30W:50W, 35W:45W, 40W:40W, 45W:35W and 50W:30W) deployed for each one of the 16 active core distributions: 6-6, 6-8, 6-10, 6-12, 8-6, ... , 12-12.

Scoped Search 1: This training set aims to further reduce the search space, but considers both balanced and unbalanced configurations. It avoids irrational distributions like assigning more than half of the power but less than half of the active cores to one of the sockets. This training set considers the even power configuration (40W:40W) and 9 different active cores distributions for it: 8-8, 8-10, 8-12, 8-10, 10-10, 12-10, 8-12, 10-12 and 12-12. It also takes into

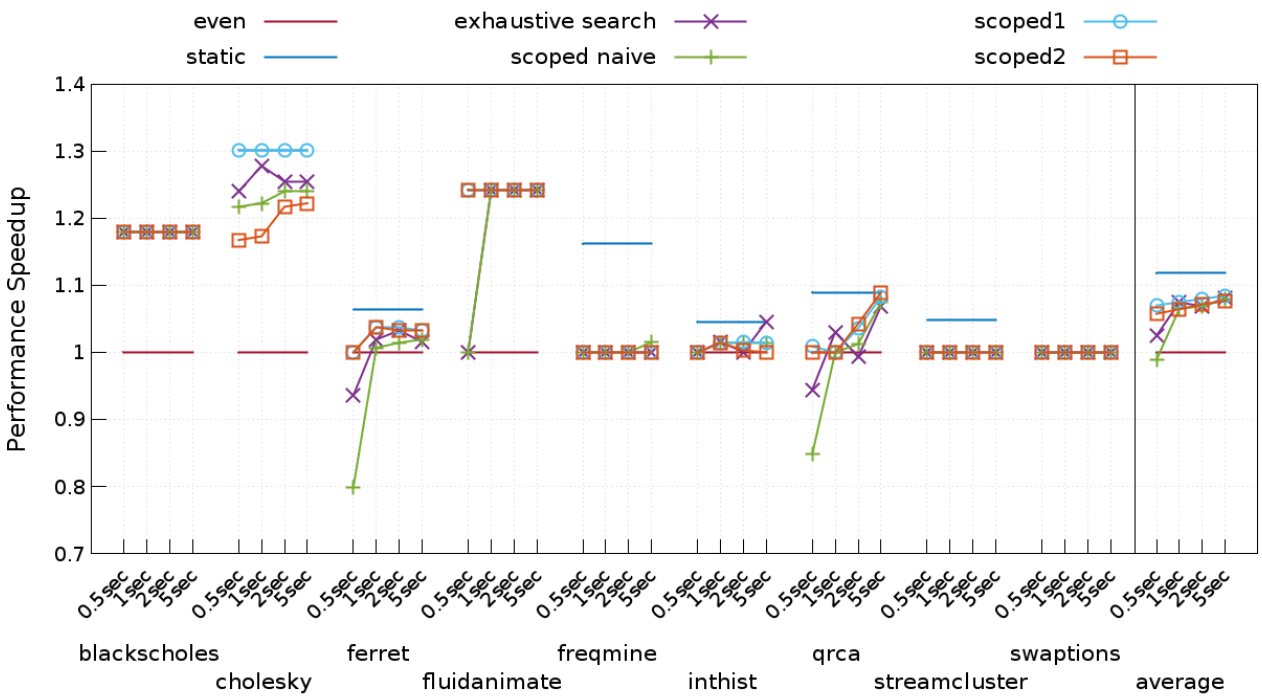


Figure 5: Comparison of best configuration found by exhaustive and scoped online analyses for different *monitoring window* size, when running under a 80W power constraint. The size of the monitoring window can influence the precision of the analysis.

account assigning 35W to the first socket and 45W to the second one with active core counts 6-10, 6-12, 8-10 and 8-12 and its counterpart, that is, 45W to the first socket and 35W to the second with active cores counts of 10-6, 12-6, 10-8 and 12-8. Finally, this training set considers two unbalanced configurations: 30W:50W assigned to the sockets and 2-12 active core counts per socket, and 50W:30W with 12-2 active cores. This leaves us with 19 configurations when operating under the 80W power bound.

Scoped Search 2: This training set contains the same distributions as Scoped Search 1 except the two unbalanced configurations 50W:30W and 30W:50W, reducing the set to 17 configurations. Very unbalanced configurations can produce large performance improvements, but also increase the search costs since they significantly slowdown the execution in certain cases. This training set avoids the dangers of such unbalanced configurations by not considering them.

6. EVALUATION

This section shows the results in applying our optimization technique. The experimental setup in terms of applications, system software and hardware is the same as in section 4.1.

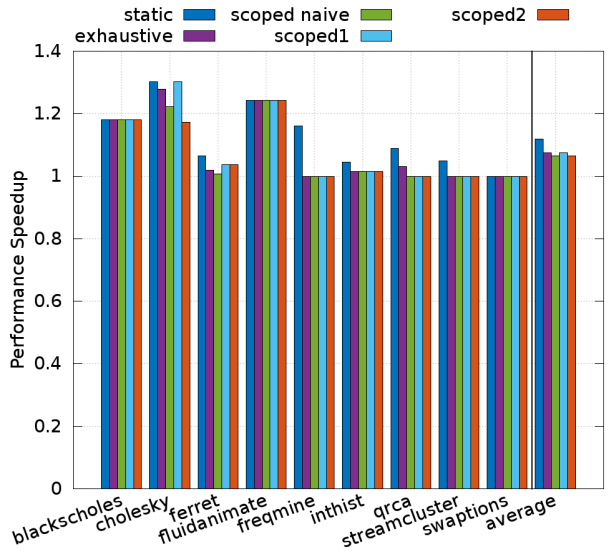
6.1 Monitoring Window Sensitivity

This first section shows how an optimal windows size is obtained by optaining a detailed sensitivity study. This optimal size is leveraged in the following general evaluation of the search algorithm in terms of its costs and benefits depending on the training set.

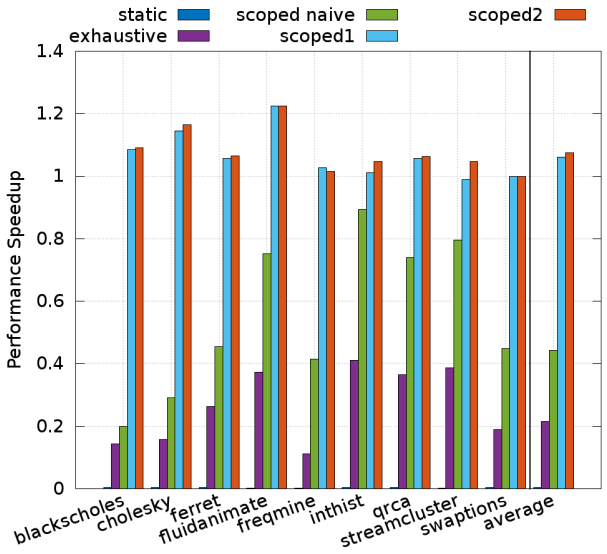
Figure 5 shows how window sizes of 0.5, 1, 2 and 5 seconds influence the effectiveness of the algorithmic search under an 80W power constraint. The x-axis represents the considered windows sizes for each application, while the y-axis shows the speedups obtained over the trivially balanced configuration (40W and 12 active cores per socket), represented in the figure with red horizontal lines. The blue horizontal line represents the speedups achieved by the optimal configuration found using the multi-execution analysis presented in Section 4 . Purple and green lines show results considering the exhaustive and scoped naive search spaces described in section 5.3, while the blue and the orange lines represent the scoped1 and scoped2 search spaces. These results do not consider the cost of the search algorithm, just the benefit of the optimal configurations found when using different windows sizes and training sets.

Results shown in Figure 5 clearly show that a windows size of 0.5 seconds on average does not provide any gain when using the scoped naive training set and only marginal gains when using the exhaustive search. Indeed, the exhaustive and scope naive searches bring significant performance degradations in cases like `ferret` and `qrca` and fail in providing a configuration that delivers the potential performance gains in case of `fluidanimate`. When considering the scoped1 and scoped2 training sets, 0.5 seconds window sizes do not provide significant benefits in case of `ferret` and `qrca`. On average, 0.5 seconds large windows provide average speedups of 1.02x, 0.98x, 1.07x and 1.06x when exhaustive, naive scope, scope1 and scope2 training sets are used.

Increasing the windows size from 0.5 to 1 second improves the quality of the configurations selected. Indeed, it provides speedups of 1.27x, 1.22x, 1.30x and 1.17x for `cholesky` or 1.24x, 1.24x, 1.24x and 1.24x for `fluidanimate` when ex-



(a) Performance benefits of the selected configurations without accounting for the cost of the search.



(b) Performance benefits of the selected configurations taking into account the cost of the search.

Figure 6: Performance benefits using monitoring windows of 1 second under 80 W power limit.

haustive, trivial scoped, scoped1 and scoped1 searches are used, respectively. On average, the 1 second window size provides benefits of 1.08x when exhaustive search is used and 1.07x when the trivial scope set is considered, 1.08x when the scope1 is considered and 1.07x for the scope2. As a reference, when running a whole execution per each configuration we get optimum power and active core distribution that bring average speedups of 1.12x. Increasing the windows sizes to 2 and 5 seconds does not significantly improve the results quality although they asymptotically get closer to the ones obtained using the multi-execution analysis. In conclusion, the 1 second window size is the optimal one since it provides similar benefits for all the considered training sets as the 2 and 5 second window sizes under a lower cost.

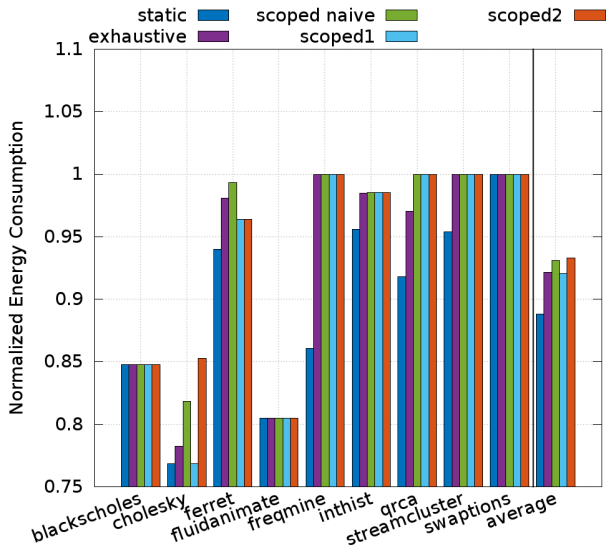
The search algorithm works very well for applications with regular computations separated by barriers (`blackscholes`, `fluidanimate` or `cholesky`) since each monitoring window can capture different iterations of the same behavior. In case of `ferret` or `qrca` the scarcity of barriers or synchronization points reduces the potential gains of our techniques. When computations are more irregular, it is more challenging to have consistent monitoring windows, which reduces the effectiveness of our scheme. In the particular case of `freqmine` the task type that accounts for more than 90% of the execution is input dependent and actually a single instance of this task type can take up to half of the total execution time. As a result the vast majority of the considered configurations are dismissed since their corresponding monitoring windows either mismatch or fail to capture any information.

6.2 Performance Improvements of the Selected Configurations

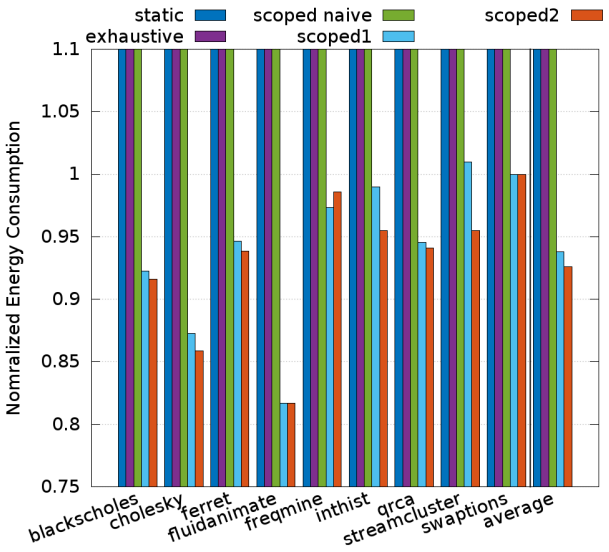
In Figure 6a we show in detail the performance benefits provided by the optimal configurations found by each one of the four training sets considering a 80W power bound and 1 second long monitoring windows. The results are expressed in terms of speedup with respect to the execution time when using the naive even distribution (40W:40W and 12-12 active cores). The static technique consists of entirely running the applications for each one of the 180 configurations defined in Section 4.1. The results of the static technique have already been presented in Section 4.2. This technique, while prohibitively expensive in practice as it requires 180 runs per application, always finds the best possible configuration and hence provides an upper bound of the speedup possible. We represent its results in Figure 6a.

In case of `blackscholes` and `fluidanimate`, all the training sets (exhaustive, naive scoped, scoped 1 and scoped 2) find configurations that provide the same speedup as the static technique, 1.17x and 1.24x respectively. In case of `cholesky`, the exhaustive and scoped 1 training sets allow the system to find configurations that provide speedups very close to 1.3x, the best possible one. The naive scope and scoped 2 techniques provide speedups close to 1.2x. Although these benefits are significant, they are far from the ones achieved by the other techniques. The reason is that the `cholesky` application benefits a lot from unbalanced distributions (Table 1), which are neither considered by the naive scope nor by the scoped 2 training sets. In case of `freqmine`, although the optimal configuration identified by the static analysis does provide significant benefits, the 4 training sets considered by the searching algorithm fail in finding this optimal configuration, since tasks are input dependent and can take up to half of the execution time, as a result most windows fail to capture task throughput since not tasks finish execution. In case of `ferret`, `inthist`, `qrca` and `swaptions`, the potential benefits of power and active cores balancing are very limited, since these applications do not have a significant number of barrier synchronizations. As we have explained in Section 3.1, when the overall number of barriers is not significant, classical load balancing mechanisms are enough to maximize performance under low power scenarios.

On average, all training sets provide benefits of around



(a) Energy reductions of the selected configurations without accounting for the cost of the search.



(b) Energy reductions of the selected configurations taking into account the cost of the search.

Figure 7: Energy consumption reduction using 1 second monitoring windows under a 80 W power bound.

1.07x, while the static technique provides an average speedup of 1.11x. The training costs of the five approaches are not considered in Figure 6a.

6.3 Performance Improvements Taking into Account Analysis Costs

All considered techniques require an analysis to find a power/active cores balance that optimally improves the trivially balanced distribution. This analysis starts once the execution of the parallel code begins and finishes when all the considered configurations have been tested. If a single application run is not sufficent to test all the configurations of the training set, the application is run again and again until the training is complete.

Figure 6b shows the speedups achieved by all considered techniques including the training phase costs. In case of the static analysis it is required to run the application multiple times, one per each of the 180 different power/active cores distributions. Consequently, the overall speedup is 0.003x, much smaller than 1x. The exhaustive training set considered 180 distributions and checks their performance over monitoring windows that are 1 second long. Therefore, more than one run is required to test all the configurations for those applications with execution times smaller than 180 seconds. Since this is the case of all the considered parallel codes, the average speedup achieved by the exhaustive training set is 0.21x. Similarly, the trivial scoped training set obtains a speedup of 0.44x. These three techniques do not improve the trivial approach which consists in just evenly distributing the total available 80W power budget and using all the cores available in the 2-socket NUMA node.

The scoped 1 and 2 training sets consider much fewer configurations than all previously mention approaches and, therefore, their training costs are significantly smaller. Indeed, they are able to test all configurations for 1 second, select the best one and then run the rest of the application using this optimal configuration. Of course, the cost of the training phase can reduce the overall benefits of the optimal configuration, as it is the case of `blackscholes`, where the benefits of the scoped 1 and 2 training sets are reduced from 1.17x to 1.08x and 1.09x respectively. `Cholesky` and `fluidanimate` have larger execution times than `blackscholes`, which allows them to compensate the cost of the training phase when scoped 1 and 2 training sets are considered and to keep almost the same performance gains as if the training costs were not considered (1.15x and 1.22x, respectively).

Finally, Figure 6b reports marginal performance benefits for some applications for which the search algorithm does not find any distribution significantly better than the trivial. For example, in case of `qrca` there are speedups of exactly 1x in Figure 6a, but of 1.05x and 1.06x for scoped1 and scoped2 in Figure6b. The explanation of this behavior is that, although the search algorithm fails to find any configuration that is significantly better than the evenly distributed, there are indeed many configurations that perform slightly faster than the even one, which accelerate the execution as they are used during the training phase.

Overall, the static technique and the exhaustive and trivial scope training sets produce an overall performance slowdown, which makes these approaches useless in practice. On the other hand, the scoped 1 and 2 training sets provide important performance benefits even when the search phase is taken into consideration, which makes these approaches very useful to maximize performance in power constrained scenarios.

6.4 Energy Consumption Reductions

Figure 7a shows the energy consumption reductions achieved when using configurations found by the five different techniques if training costs are not considered. The baseline is the energy spent by the trivially balanced configuration (12 active cores and a maximum of 40W per socket) and all results are normalized to this baseline. The configurations selected by the static analysis provide energy reductions of

11% with respect to the even distribution since the normalized energy gets reduced from 1 to 0.89. The reductions provided by the search algorithm are between 7% and 8%, depending on the training set.

Once the cost of the exploration phase is considered, the energy consumed by the static analysis, the exhaustive and the scope naive training sets are 295, 3.7 and 1.8 times larger than the energy consumed by a single run with the even configuration. In Figure 7b we just represent normalized energy consumptions below 1.1 for readability purposes. As observed before, the scoped 1 and 2 training sets are able to compensate the cost of the training set and deliver improvements over the even configurations. The energy consumption reductions are 0.94 and 0.93 for the scoped 1 and 2 training sets respectively.

7. RELATED WORK

The ability to set up power bounds in many-core systems is becoming a common feature. For example, Intel introduced a set of machine-specific registers (MSRs) [31] on their Sandy/Ivy Bridge processors to explicitly constrain on-chip power consumption. Since the release of commodity chips with such capabilities, several studies have shown the impact power capping can have. In particular, work by Rountree et al. [27] motivates the research presented in this paper on how processor performance variability due to power capping can be addressed.

Inadomi et al. [18] also study the performance variability on a number production clusters and propose a variation-aware power budgeting framework. Their approach requires specific single core executions for profiling the HPC applications plus a once-per-system profiling to build a reference table containing performance variability information for all nodes. This table and the single core profiling is used to make decisions using a model. Compared to their method, our method does not require dedicated profiling runs or system wide reference tables containing performance variations. Instead, we use profiling information obtained at runtime to adjust power distribution and concurrency levels, which reduces the analysis costs and increases its benefits.

Bailey et al. [1] propose a linear programming formulation for MPI+OpenMP programs for maximizing performance under job-level power constraints. While this approach provides a good approximation of the upper bound of possible performance in dynamic runtime systems, the use of a linear programming solver is too slow to be practical for optimizing applications at runtime. The same group also introduced Conductor [23], a dynamic runtime system that directs power to the critical path of the computation to minimize overall execution time under a power cap. Conductor, however, does not deal with the hardware manufacturing variability we describe in this paper. As such, our approach is orthogonal and can be combined to maximize parallel applications performance.

On a single node, Cochran et al. [10] classify the PARSEC benchmark suite applications for their power, temperature and performance characteristics. Using these results, they maximize performance while meeting power constraints by using thread packing and DVFS. In contrast to our approach, though, they rely on their priori characterization, while our approach can work without prior information.

There is a significant body of work focused on job scheduling for power constrained systems. Etinski et al. [14] propose an LP-based job scheduling policy; Sarood et al. [30] use performance modeling to make job scheduling decisions in power constraint system to improve job throughput; and Ellsworth et al. [13] discuss a dynamic job scheduling algorithm, which when running under a system-wide power limit, detects unused power and redistributes it to nodes that can make use of it.

The impact of manufacturing quality on power consumption variability of processor chips has been studied in a significant number of works as well. The power leakage of processors is directly connected to our work, since by setting a power limit on the socket, we impair its ability to adjust power consumption to maintain the proper frequency level. Davis et al. [11] study the effect of inter-node variability on power model characterization, in the context of homogeneous clusters. Herbert et al. [16] show that exposing the power leakage variability of processors to the DVFS control algorithm to shift work to the less leaky processors, can reduce overall system power consumption. Further, several projects study the on-die power variation to improve DVFS scheduling [21, 32, 17]. As an additional concern, modern processors require transistors to shrink to a level that introduces significant power and reliability variations among processors, a phenomena explored in detail by a variety of groups [5, 15, 34]. Overall, most studies conclude that power variation is expected to become worse in the future [29, 15].

8. CONCLUSIONS

In this work we studied how state-of-the-art parallel runtime systems can mitigate the performance imbalance between sockets on the same node when operating under strict power constraints. We establish that load-balancing, although improving performance, is not sufficient for a wide set of applications. In our study we use six applications from the PARSEC benchmark suite and three additional applications, all implemented with OpenMP 4.0 tasks. By performing profiling runs of the applications with different power distributions and active number of cores on each socket we demonstrate that it is possible to achieve speedups up to 1.30x over naively spreading the power budget and using all possible cores. We also propose and implement an online analysis that monitors only a segment of the application's execution and is able to switch between different power and concurrency configurations at runtime, reducing the overhead of profiling. Our evaluation shows that it is possible to carefully compose the configuration search space by eliminating candidates that are unlikely to give a good result (such as reducing power but increasing concurrency). The online analysis achieves speedups up to 1.22x over the naive case.

This work focused on figuring out the optimal power-concurrency balance on machines with 2 sockets per NUMA node, which is a common setup in current systems. Future configurations will have many more sockets on a single node. In respect to our method this trend will likely require increasing the training set sizes, thus the cost and its complexity. However, the benefits of our technique will also increase: more sockets imply an even more varied response to low power scenarios within the same NUMA node. Adding accelerators will further increase the number of frequency/power capping domains. By restricting our searches to well-balanced configurations, as shown in this work, we can avoid a combinatorial explosion in the training set sizes,

which keeps training costs within reasonable margins and enables larger performance improvements on multi-socket NUMA nodes. The results on 2 a socket system, as described in the paper, therefore cover the worst case scenario.

Acknowledgements

This work has been supported by the Spanish Government (Severo Ochoa grants SEV2015-0493, SEV-2011-00067), by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), by the RoMoL ERC Advanced Grant (GA 321253) and the European HiPEAC Network of Excellence. M. Moretó has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship number JCI-2012-15047. M. Casas is supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (Contract 2013 BP_B 00243). This work was also partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-689878).

Finally, the authors are grateful to the reviewers for their valuable comments, to the RoMoL team, to Xavier Teruel and Kallia Chronaki from the Programming Models group of BSC and the Computation Department of LLNL for their technical support and useful feedback.

9. REFERENCES

[1] P. E. Bailey, A. Marathe, D. K. Lowenthal, B. Rountree, and M. Schulz. Finding the limits of power-constrained application performance. In *SC*, pages 79:1–79:12, 2015.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, pages 72–81, 2008.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP*, pages 207–216, 1995.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[5] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *DAC*, pages 338–342, 2003.

[6] BSC. Programming models group. the nanos++ parallel runtime. https://pm.bsc.es/nanox, 2015.

[7] M. Casas, R. M. Badia, and J. Labarta. Automatic phase detection and structure extraction of mpi applications. *Int. J. High Perform. Comput. Appl.*, 24(3):335–360, Aug. 2010.

[8] M. Casas, M. Moreto, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar, O. Unsal, A. Cristal, E. Ayguade, J. Labarta, and M. Valero. *Euro-Par 2015*, chapter Runtime-Aware Architectures, pages 16–27. August 2015.

[9] D. Chasapis, M. Casas, M. Moretó, R. Vidal, E. Ayguadé, J. Labarta, and M. Valero. Parsecss: Evaluating the impact of task parallelism in the parsec benchmark suite. *ACM Trans. Archit. Code Optim.*, 12(4):41:1–41:22, Dec. 2015.

[10] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. Pack & cap: Adaptive dvfs and thread packing under power caps. In *MICRO*, pages 175–185, 2011.

[11] J. D. Davis, S. Rivoire, M. Goldszmidt, and E. K. Ardestani. Accounting for Variability in Large-Scale Cluster Power Models. In *EXERT*, 2011.

[12] J. W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[13] D. A. Ellsworth, A. D. Malony, B. Rountree, and M. Schulz. POW: System-wide Dynamic Reallocation of Limited Power in HPC. In *HPDC*, pages 145–148, 2015.

[14] M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Linear programming based parallel job scheduling for power constrained systems. In *HPCS*, pages 72–80, July 2011.

[15] L. R. Harriott. Limits of lithography. *Proceedings of the IEEE*, 89(3):366–374, Mar 2001.

[16] S. Herbert, S. Garg, and D. Marculescu. Exploiting process variability in voltage/frequency control. *IEEE Trans. Very Large Scale Integr. Syst.*, 20(8):1392–1404, Aug. 2012.

[17] S. Herbert and D. Marculescu. Variation-aware dynamic voltage/frequency scaling. In *HPCA*, pages 301–312, 2009.

[18] Y. Inadomi, T. Patki, K. Inoue, M. Aoyagi, B. Rountree, M. Schulz, D. Lowenthal, Y. Wada, K. Fukazawa, M. Ueda, M. Kondo, and I. Miyoshi. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *SC*, pages 78:1–78:12, 2015.

[19] Intel. *Intel-64 and IA-32 Architectures Software Developer's Manual*. Intel, December 2011.

[20] K. E. Isaacs, A. Bhatele, J. Lifflander, D. Böhme, T. Gamblin, M. Schulz, B. Hamann, and P.-T. Bremer. Recovering logical structure from charm++ event traces. In *SC*, pages 49:1–49:12, 2015.

[21] B. Lin, A. Mallik, P. Dinda, G. Memik, and R. Dick. User- and process-driven dynamic voltage and frequency scaling. In *ISPASS*, pages 11–22, April 2009.

[22] Livermore Computing. The Catalyst supercomputer. http://computation.llnl.gov/computers/catalyst, 2014.

[23] A. Marathe, P. Bailey, D. Lowenthal, B. Rountree, M. Schulz, and B. de Supinski. A run-time system for power-constrained HPC applications. In *High Performance Computing*, volume 9137 of *Lecture Notes in Computer Science*, pages 394–408. 2015.

[24] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski. Exploring hardware overprovisioning in power-constrained, high performance computing. In *ICS*, pages 173–182, 2013.

[25] N. Rajovic, P. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero. Supercomputing with commodity CPUs: Are mobile SoCs ready for HPC? In *SC*, pages 1–12, Nov 2013.

[26] K. Ravichandran, S. Lee, and S. Pande. Work stealing for multi-core hpc clusters. In *Euro-Par*, pages 205–217, 2011.

[27] B. Rountree, D. Ahn, B. de Supinski, D. Lowenthal, and M. Schulz. Beyond DVFS: A first look at performance under a hardware-enforced power bound. In *IPDPS Workshops PhD Forum*, pages 947–953, May 2012.

[28] P. B. S. Ashby and, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, T. Mezzacappa, P. Moin, M. Norman, R. Rosner, V. Sarkar, A. Siegel, F. Streitz, A. White, and M. Wright. The opportunities and challenges of exascale computing. DOE Technical Report, 2010.

[29] S. Samaan. The impact of device parameter variations on the frequency and performance of VLSI chips. In *ICCAD*, pages 343–346, Nov 2004.

[30] O. Sarood, A. Langer, A. Gupta, and L. Kale. Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *SC*, pages 807–818, 2014.

[31] K. Shoga, B. Rountree, and M. Schulz. Whitelisting MSRs with msr-safe, November 2014.

[32] R. Teodorescu and J. Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. *SIGARCH Comput. Archit. News*, 36(3):363–374, June 2008.

[33] E. Totoni, J. Torrellas, and L. V. Kale. Using an adaptive hpc runtime system to reconfigure the cache hierarchy. In *SC*, pages 1047–1058, 2014.

[34] J. Tschanz, J. Kao, S. Narendra, R. Nair, D. Antoniadis, A. Chandrakasan, and V. De. Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *Solid-State Circuits, IEEE Journal of*, 37(11):1396–1402, Nov 2002.

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

[36] G. Zheng, A. Bhatelé, E. Meneses, and L. V. Kalé. Periodic hierarchical load balancing for large supercomputers. *Int. J. High Perform. Comput. Appl.*, 25(4):371–385, Nov. 2011.