A Run-Time System for Power-Constrained HPC Applications

Aniruddha Marathe[1] , Peter E. Bailey[1] , David K. Lowenthal[1] , Barry Rountree[2] , Martin Schulz[2], and Bronis R. de Supinski[2]

[1] Department of Computer Science
The University of Arizona
{amarathe, pbailey, dkl}@cs.arizona.edu

[2] Lawrence Livermore National Laboratory
{rountree, schulzm, bronis}@llnl.gov

Abstract. As the HPC community attempts to reach exascale performance, power will be one of the most critical constrained resources. Achieving practical exascale computing will therefore rely on optimizing performance subject to a power constraint. However, this additional complication should not add to the burden of application developers; optimizing the runtime environment given restricted power will primarily be the job of high-performance system software.

This paper introduces *Conductor*, a run-time system that intelligently distributes available power to nodes and cores to improve performance. The key techniques used are *configuration space exploration* and *adaptive power balancing*. Configuration exploration dynamically selects the optimal thread concurrency level and DVFS state subject to a hardware-enforced power bound. Adaptive power balancing efficiently determines where critical paths are likely to occur so that more power is distributed to those paths. Greater power, in turn, allows increased thread concurrency levels, the DVFS states, or both. We describe these techniques in detail and show that, compared to the state-of-the-art technique of using statically predetermined, per-node power caps, *Conductor* leads to a best-case performance improvement of up to 30%, and average improvement of 19.1%.

1 Motivation

The US government, as well as European and Asian agencies, have set a goal to reach exascale computing in less than 10 years. However, if we were to build an exascale machine out of today's hardware, it would consume half of a gigawatt of power [21, 13] and effectively require a dedicated power plant. In reality, there is a practical power bound, which is much tighter, and one such bound commonly used by both the research as well as the industrial high-performance computing (HPC) community is 20 megawatts [2]. It is clear that future HPC systems will have a whole-system power constraint that will filter down to job-level power constraints. The goal at the job-level will be to optimize performance subject to a prescribed power bound.

HPC users have enough to handle with ensuring correctness and maintaining sufficient performance, so the task of enforcing the job level power bound should be left to

HPC system software. More importantly, system software is in an ideal position to dynamically configure applications for the best performance subject to a power constraint. We define a processor's *configuration* as: (1) a value for *c*, the number of active cores, and (2) the DVFS state. The power constraint states that the total job power consumption must always be no more than the job-level power bound *P*, and the goal is to minimize application runtime. We use Intel's Running Average Power Limit (RAPL) [14], introduced in the SandyBridge microarchitecture, to enforce the power constraint.

This paper describes the *Conductor* run-time system, which efficiently chooses an initial configuration resulting in near-optimal application performance for a given job power bound, then adapts this configuration as necessary during application execution according to changing application behavior and power constraints. The fundamental ideas behind *Conductor* are twofold. First, *Conductor* performs *configuration space exploration*, which dynamically selects the optimal thread concurrency level (DCT) and dynamic voltage frequency state (DVFS) subject to a RAPL-enforced power bound. Second, *Conductor* performs *adaptive power balancing*, which locates noncritical parts of the application (i.e., off the critical path), reduces their power consumption, and uses that excess power to speed up the parts on the critical path.

In *Conductor*, adaptive power balancing itself is done in three stages. First, *Conductor* monitors an application timestep to gauge representative application behavior. Second, *Conductor* continually applies a local, adaptive algorithm to select task configurations to reduce power consumption without increasing task execution time where possible. Third, *Conductor* improves performance by reallocating power at the MPI process level, using a global algorithm that is periodically executed at the end of timesteps. *Conductor* uses RAPL to enforce the chosen power allocation on each MPI process.

Specifically, this paper makes the following contributions.

– We design *Conductor*, the first run-time system that utilizes nonuniform power distribution, RAPL, DVFS, and DCT for optimizing HPC application performance under a power constraint.
– *Conductor* is fully automatic and chooses configurations with no involvement of the application programmer other than marking the end of an application timestep.
– *Conductor* chooses and adapts configurations dynamically based on application characteristics, resulting in efficient execution on a number of applications.

We implement *Conductor* on a large scale cluster at Lawrence Livermore National Laboratory, which has infrastructure for constraining power on a per-processor basis. Our results on up to 64 processors (512 cores) show that over five applications, *Conductor* achieves an average performance improvement of 19.1% in a range of power limits over the state-of-the-art method, which is statically selected, per-node power caps. Moreover, we observe that *Conductor* achieves best-case performance improvement of 30% over the static allocation scheme.

2 Optimizing Overprovisioned Systems

Traditionally, achieving maximum throughput in HPC clusters has been constrained by the available (fixed) hardware. However, as we move towards exascale, *power*, rather

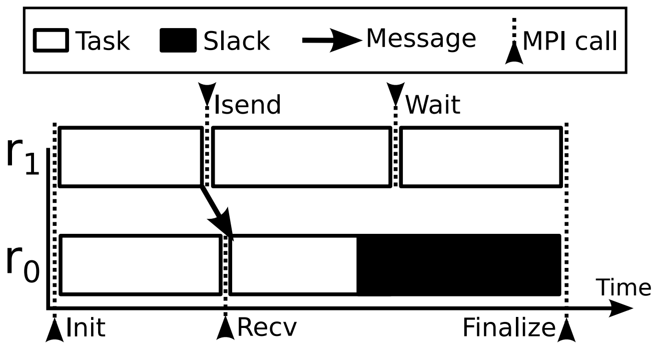

Fig. 1. Execution model on two MPI processes. The process with rank 1 sends a message to the process on rank 0. The process on rank 0 arrives first at `MPI_Finalize`, inducing slack time.

than hardware, will become a limiting factor in achieving optimal performance. With power consumption in a supercomputing cluster becoming critical, the allocation of power to individual jobs must comply to strict power constraints.

Under the given power constraint, which is typically imposed at the facility level, a supercomputing system may run fewer nodes at maximum power or more nodes at a lower power (known as *overprovisioning*). The system-level power constraint translates into job-level power constraints, which motivate the design of techniques to optimize performance under job-level power constraints. The primary objective is to run each job at a configuration that is power-efficient and allocate power to the critical path of the application.

2.1 Execution Model

In order to reason about optimization approaches in overprovisioned systems, we adopt the task based execution model (Figure 1) that we used in our work on the Adagio runtime system [23]. A *task* is the basic unit of scheduling, comprising total communication and computation that takes place on an MPI process between two consecutive MPI calls. Note that a process must block at a communication call if there is an inbound receive edge and the data has not yet arrived, or if a process arrives late to a collective. This can lead to what we refer to as *slack* time. As an example of the first case, the MPI process with rank 0 arrives before the process with rank 1 at `MPI_Finalize` in Figure 1, inducing slack time.

2.2 Assumptions

We use the execution model discussed above to design our optimizing runtime system, which we introduce in the next section. Additionally, we currently focus on applications implemented with the SPMD (Single-Program, Multiple-Data [7]) model and use OpenMP for intra-node parallelism and MPI for inter-node parallelism. We assume that programs use `MPI_THREAD_SINGLE`, so there are no MPI calls within an OpenMP region. There is nothing that prevents us from conceptually supporting pure MPI programs (i.e., one MPI rank per core), but because our system chooses a given number of cores per processor dynamically, a pure MPI approach would require expensive data redistribution or core oversubscription when the number of cores per processor changes. We assume that an application is composed of several timesteps, and that the programmer identifies the end of a timestep (currently accomplished with inserting `MPI_Pcontrol` into the application code).

Following the work of Li et al. [18], we also restrict each MPI process (and therefore each OpenMP parallel region), to a single CPU socket/NUMA node, and we assume that we use the same number of active cores for OpenMP regions between consecutive MPI calls. This avoids increasing the number of cache misses due to a change in the number of active cores between two OpenMP regions.

2.3 Challenges

Previous work in the area of power-constrained performance optimization outlines the following challenges in developing a run-time system to adaptively select the best processor configuration for an application. First, the configuration space from which to select the optimal configuration is large, because modern day processors have over a dozen frequency steps and provide 16 or more physical cores. Thus finding the optimal configuration for each processor in a job becomes a large combinatorial problem. Combined with the fact that different configurations can result in vastly different performance [20], the quality of current techniques is unknown. Second, allocating the optimal amount of power to individual processors in a job is complicated by the fact that the critical path may move through multiple processes in a time step. Finally, efficiently monitoring power usage for individual processors allocated to a job and re-allocating power with acceptable overhead is a challenge. *Conductor* addresses all three challenges and provides a novel approach for effective use of large, power-limited systems.

3 *Conductor*: Power-Constrained Runtime Scheduling

Conductor continuously monitors the execution behavior of an application and adjusts its configuration parameters to stay within the job-level power limit while optimizing performance. In particular, we use two knobs in the configuration of an application, on a per-processor basis: the number of active threads in a computation phase and the voltage/frequency setting. For the latter, we use two mechanisms: we use DVFS to control an application's speed and power usage based on observational data (predictive control), while we use RAPL (Runtime Average Power Limit) to set hard power caps for each processor in case our predicted configurations would violate the power constraint (prescriptive control).

The algorithm used in *Conductor* can be split into four steps: initialization, configuration exploration, adaptive reconfiguration, and power reallocation.

3.1 Initialization

During the first timestep of the application, each MPI process is assigned an equal amount of power derived from the job-level power constraint. This is the timestep before *Conductor* starts the configuration exploration step. The process-level power constraint is enforced using RAPL. The execution time and power usage of each application task (i.e., unit of computation between communication events) in the timestep is recorded and stored in a task graph. At the end of the initialization step, the power constraint per process is (temporarily) removed to facilitate the configuration exploration step.

3.2 Configuration Exploration

The next step taken by *Conductor* is to choose the configuration, or combination of thread concurrency level and the DVFS state for each application task. The choice of configuration has a significant impact on program execution time (as much as an 30.9% difference over our five applications). There are a number of ways to choose the ideal thread concurrency level given a power bound. One way is to profile the code beforehand, which has the distinct disadvantage that it requires at least one extra program execution. Another is to build offline models based on program executions, but the disadvantage is that the model could lack accuracy and generality, especially in the case that a given program differs from the set of programs used to build the model.

In *Conductor*, we take a simpler approach: given n MPI processes executing a given application, we use a small number of application iterations to perform a parallel exploration of the configuration space by selecting a different thread/DVFS configuration on each MPI process. There are k such configurations that we consider, and given n processes, we simply test all of them and choose the best-performing configuration depending on the current process-level power constraint. We retain the set of power-efficient configurations for each task, yielding a per-task power/time Pareto frontier. As mentioned above, we disable the power bound during this step in our current prototype; this can be fixed—at the expense of more overhead—by more carefully executing the configuration exploration.

This clearly adds overhead while we are searching for efficient configurations. In general, it takes $m = \lceil n/k \rceil$ timesteps to finish testing all configurations. Assume an example application with a single task per process in each iteration. Suppose that during a timestep, the optimal configuration of thread concurrency level and DVFS state takes time t_{opt}, and the process with the slowest configuration on timestep i (during the search phase) takes time t^i_{worst}. Then, the upper bound on the execution time is $T = \sum_{i=1}^{m}(t^i_{worst}) + \sum_{i=m+1}^{n}(t^i_{opt})$, assuming that there are n timesteps in total and m timesteps in the search phase.

Given that high-performance computing applications generally execute many timesteps ($n \gg m$), the overhead in *Conductor*, compared to an oracle that could choose the optimal thread/power configuration a priori, will be generally small because it is amortized over the lifetime of the computation. Because *Conductor* potentially selects the optimal configuration, this overhead can be expressed as $\frac{T}{\sum_{i=1}^{n}(t^i_{opt})}$.

3.3 Adaptive Reconfiguration

The configuration exploration phase makes the assumption that the optimal configuration does not change, which is not true in general. Further, for dynamic applications with load imbalance, this can lead to wasted power during unnecessary wait operations (slack time). To handle both of these issues, we additionally introduce a novel adaptive power-balancing algorithm that changes configurations when appropriate due to application behavior. In addition to application behavior, *Conductor* takes into account the current power constraint, processor DVFS state and thread concurrency level.

***Conductor* Monitoring** After the thread/DVFS relationships are characterized for each task during the configuration exploration phase, the per-process power constraint is re-enforced using RAPL. *Conductor* monitors application execution during each individual timestep and uses this information to predict the behavior of following, similar tasks. During each timestep, *Conductor* records the elapsed time and power usage for each task in a statically selected configuration. *Conductor* also measures the slack by observing time spent within the MPI library, if any, for each task. This makes the assumption that the significant portion of the time in the MPI library is spent blocking waiting for messages, which works as a useful predictor in the absence of the ability to directly measure slack. This measurement step is borrowed from our previous run-time system, Adagio [23]. It distinguishes tasks based on callstacks and uses a threshold to differentiate slack time from MPI processing time.

Adjusting Task Execution Times The previous step simply identifies tasks that contain slack. In the following timesteps, *Conductor* adjusts task execution times in such a way that overall execution time will decrease. *Conductor* avoids adding to existing application inter-process communication where possible. Accordingly, *Conductor* handles each task completely locally via the following method.

First, for any task that contains slack, *Conductor* can guarantee that it is *not* on the critical path; by definition, any task that contains slack can be slowed down by some nonzero amount without slowing down overall application execution. Consequently, *Conductor* attempts to fill as much of the slack as possible with computation time without affecting the completion time of the task. For this purpose, *Conductor* leverages both DVFS and thread concurrency levels to fill slack. In other words, *Conductor* will not allow any (non-critical) path through any task with slack to become the critical path. Note that the reason that we adjust DVFS and thread concurrency levels and not the RAPL bound itself is (1) the task granularity is too small to use RAPL, and (2) RAPL does not adjust thread concurrency.

Second, for any task that has no slack, *Conductor* conservatively changes its configuration to the one with next fastest thread/DVFS on the Pareto frontier, which was determined (and saved) as part of the configuration exploration phase. The intuition here is that *Conductor* knows that such a task *may be* critical. Therefore, *Conductor* treats it as a task that should decrease its execution time, because the critical path could potentially decrease. Note that this decision is made locally because the overhead of determining the exact critical path is prohibitive.

3.4 Reallocation of Per-Process Power

While the above step adjusts power consumption using DVFS and thread concurrency selection, the overall power cap per process is as yet unchanged; this cap ensures that the power constraint is not violated. Consequently, even after *Conductor* has adjusted configurations for individual tasks to fill slack, critical tasks may continue to run at or near their process's power constraint, while processes with no critical tasks do not use all of their power allocation. Such a situation may be caused by load imbalance inherent to the application or differences in power efficiency between individual processors [22]. Regardless of the cause, *Conductor* takes advantage of the opportunity to speed up the application by using a global algorithm to reallocate power between processes.

As an example, Figure 2(a) shows the power consumption profile of an MPI process in an iterative MPI application with repeating computation and communication phases. Since the tasks on this process are off the critical path, there is slack in the communication phase following the computation phase. Figure 2(b) shows that the computation tasks can be slowed down within the slack boundaries using DVFS and thread concurrency selection without affecting the overall execution time.

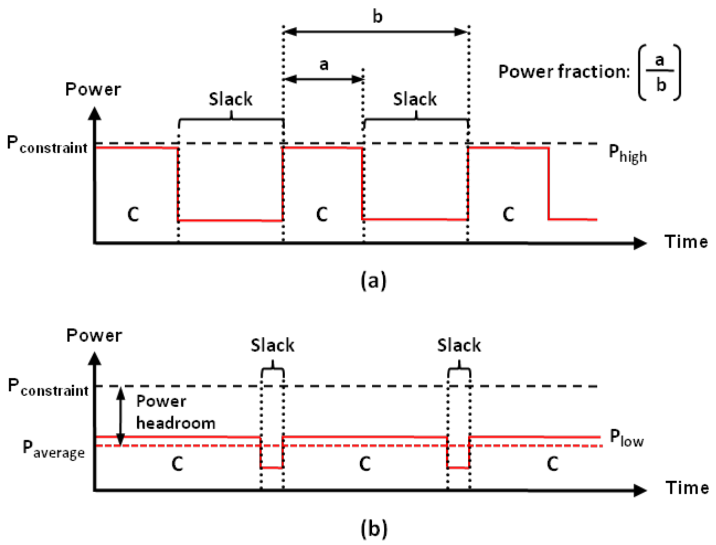

Fig. 2. Opportunity for re-scheduling excess power in an MPI process that runs off the critical path with Adagio. Plot (a) shows near-cap power consumption for computation task (C) at the highest processor DVFS state (highest voltage and frequency). Plot (b) shows lower power consumption for computation task (C) at a lower DVFS/thread concurrency set by Adagio.

We define the *power fraction* for a process as the fraction of time between power reallocations that an MPI process spends within a small tolerance of its power constraint. Figure 2(a) shows the fraction of time the process spends operating at power P_{high}, which is essentially at $P_{constraint}$. The process consumes P_{low} power after *Conductor* has slowed down the computation operation as shown in Figure 2(b). We define the *power headroom* for a processor as the difference between the processor's power constraint and processor's average power consumption. In Figure 2(b), the power headroom is the difference between $P_{constraint}$ and $P_{average}$.

Conductor gathers power headroom information (which is computed based on *all* tasks) from all processes after a configurable number of timesteps of the application. Using process-level power headroom information, *Conductor* calculates job-level power headroom and reallocates process-level power constraints based on the power fraction. While this technique has the potential disadvantage that the critical path could, for a pathological situation, move through a process that "donates" power to another process, this situation is rare. Moreover, to address it would require power reallocation on task granularity, which is quite complex.

4 Experimental Setup

We performed all experiments on Cab, a 1200-node Xeon E5-2670 cluster at LLNL with an InfiniBand QDR interconnect. Each cab node is composed of two 8-core processors and 32 GB of DRAM.

4.1 Benchmarks and Tools

The codes we use for comparison are CoMD, LULESH 2.0, SP and BT from NAS-MZ, ParaDiS and a synthetic benchmark. These benchmarks were selected because they exhibit performance and scaling behavior typical for a wide range of HPC applications. We note that the most interesting behavior for these benchmarks occurs between 30 and 60 watts per processor.

CoMD [1] is a molecular dynamics benchmark. CoMD is unique among our tested benchmarks in that all of its MPI communication is in the form of collectives. As a result, the only tasks that remain for the power-balancing algorithm are to minimize load imbalance by reallocating power between processes at every collective call and to select efficient configurations under processor-level power constraints. We use the input problem size of 20x40x40 with 100 timesteps.

LULESH 2.0 [17] is a shock hydrodynamics benchmark. In terms of MPI communication, LULESH differs from CoMD in that it relies on a multitude of point-to-point messages between collective calls. This behavior complicates analysis of opportunities to balance power, but we show in Section 5.1 that *Conductor* improves performance over state-of-the-art methods for running under a job-level power constraint. We use an input problem size of 32, with 100 timesteps.

ParaDiS [5] is a production dislocation dynamics simulations application that operates on dynamically changing, unbalanced data set sizes across MPI processes. The random nature of data set sizes results in varying computational load, introducing load-imbalance across MPI processes. We use the "Copper" input set provided with ParaDiS with 600 timesteps.

NAS Multi-Zone [27] is an extension of the NAS Parallel Benchmark suite [3]. It involves solving the application benchmarks BT, LU and SP on collections of loosely coupled discretization meshes. In our work, we use Block Tri-diagonal (BT-MZ) and Scalar Penta-diagonal (SP-MZ) algorithms. Both applications use OpenMP for intra-node computation and MPI for inter-node communication. We use a custom class D input size with 500 timesteps.

To quantify how a load-imbalanced application can benefit from power reallocation in *Conductor*, we developed a synthetic benchmark. The synthetic benchmark has two properties. First, it is written in such a way that the best configurations under various power limits use six (out of eight) threads per socket. Second, half of MPI processes execute nearly six times more computation load than the other half, which leads to process-level load imbalance. This synthetic program focuses on the opportunity for *Conductor* to improve performance through power re-allocation for process-level load imbalance.

4.2 Overheads

As we instrument every instance of any potentially blocking MPI call in order to capture slack time and select configurations, our profiler incurs some overhead. The median measurement overhead is 34 microseconds per MPI call and adds less than 0.05% time to the tested applications. We use 60 configurations that consist of thread concurrency levels of 5 to 8 threads per socket and 15 discrete DVFS states. For SP-MZ, BT-MZ, CoMD and Lulesh, the configuration exploration phase took up to 3 timesteps. For ParaDiS we run the configuration exploration phase locally over each MPI process due to non-repeatability in computation phase across MPI processes. For the configuration exploration phase, we observe an overhead of 1.96 seconds in the worst case (recall this is amortized over the entire application execution). For the runtime power re-allocation algorithms, all power allocation decisions are coordinated within existing application collective calls, with an average overhead of 566 microseconds per invocation. For the job sizes tested, we consider this an acceptable trade-off. For larger jobs, a hierarchical power-balancing strategy would be required.

5 Experimental Evaluation

In this section, we compare the performance of *Conductor* to two alternate policies: *Static*, and *Config-Only*. The *Static* algorithm, which is the current state of the art, chooses the largest number of threads possible under the power bound assuming that the frequency is the lowest possible. Then, it increases the frequency as high as possible subject to the power bound. *Config-Only* executes the configuration exploration part of *Conductor* and performs one-time configuration selection for each task, but does *not* execute the adaptive reconfiguration or power reallocation steps.

5.1 Load-Balanced Applications

Figure 3 shows execution times of our three load-balanced applications for each of the three policies. This includes Lulesh on 27 (3^3) MPI processes and BT-MZ and CoMD

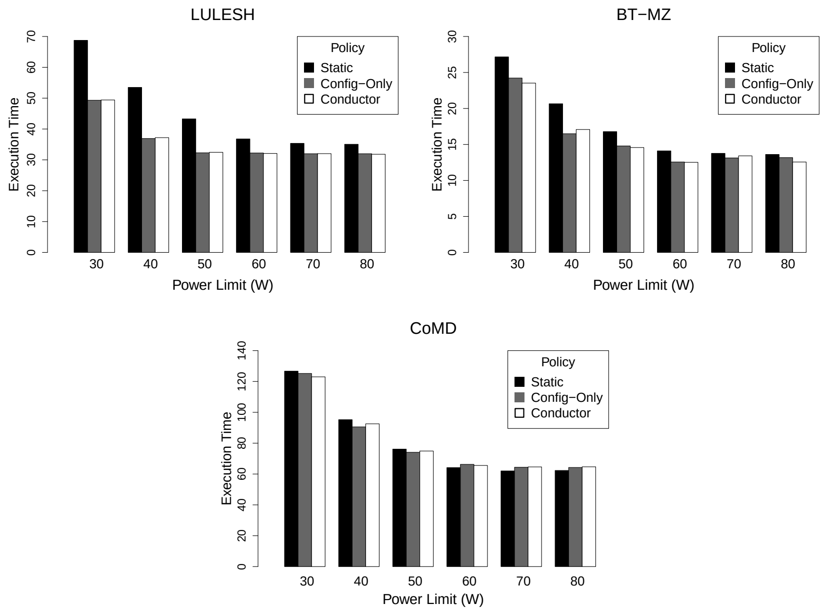

Fig. 3. Comparison of power allocation policies for our three load-balanced applications: Lulesh, BT-MZ, and CoMD. We use 27, 32, and 32 processes, respectively.

on 32 processes. The execution times are grouped by process-level power limits on the x-axis ranging 30 watts to 80 watts in steps of 10 watts; each process is confined to a singled processor. We make the following important general observations. First, for lower power limits (30 watts and 40 watts per processor), *Conductor* performs significantly better than *Static*; the difference is as much as 30.4% in the case of Lulesh at 27 tasks. The improvement in performance is due to the power-efficient configurations selected by *Conductor* under the MPI process-level power constraint. Second, at higher power limits (60-80 watts), the difference between *Static* and *Conductor* remains constant, because the default configuration selected by *Static* and the optimal configuration selected by *Conductor* remain constant. Also, *Conductor* performs almost identically compared to *Config-Only* for the three load-balanced applications. This is because there is virtually no load imbalance. In general, load-balanced applications will execute slightly faster when using *Config-Only* if there is opportunity to select a better configuration over the default configuration chosen by *Static*. This is because both *Config-Only* and *Conductor* choose the same configuration, and *Conductor* does not change the power allocation per MPI process (but does have a slight overhead for monitoring).

5.2 Load-Imbalanced Applications

Figure 4 shows several runs of ParaDiS, SP-MZ, and our synthetic load imbalanced application at 32 (for SP and the synthetic benchmark) and 64 (for ParaDiS) processes

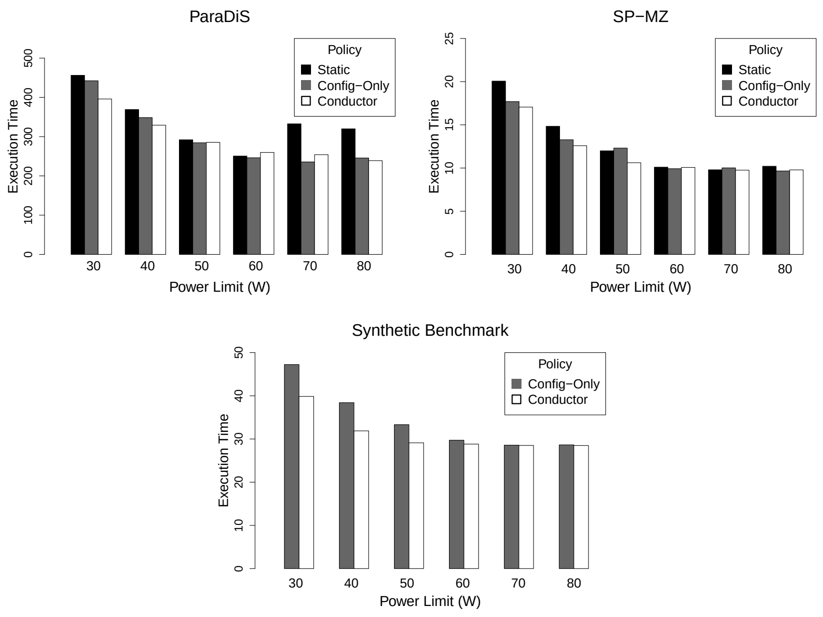

Fig. 4. Comparison of power allocation policies in our three load-imbalanced applications: ParaDis, SP-MZ, and a synthetic microbenchmark. We use 64, 32 and 32 processes, respectively.

under power limits of 30 watts to 80 watts. For load-imbalanced applications, *Conductor* has each process record power and execution times for all configurations because of potential non-repeatability in the characteristics of computation tasks across MPI processes.

For lower power limits (30 watts to 50 watts), *Config-Only* benefits from selecting the best configuration for individual computation tasks. However, the benefits of our power re-allocation policy in *Conductor* are more pronounced. The difference in power usage across different MPI processes is up to 15% and 20% for process-level power limits of 30 watts and 40 watts respectively, indicating load-imbalance and potential benefit through power re-allocation.

Compared to *Config-Only*, *Conductor* achieves an improvement of up to 10.4% (for ParaDiS) and 3.6% (for SP-MZ) improvement at the power limit of 30 watts. For the same power limit, compared to *Static*, *Conductor* achieves an improvement of up to 13.2% (for ParaDiS) and 14.9% (for SP-MZ). Compared to *Config-Only*, *Conductor* achieves an improvement of 5.5% (for ParaDiS) and 5.2% (for SP-MZ) for a power limit of 40 watts.

Compared to *Static*, *Conductor* achieves an improvement of 10.8% (for ParaDiS) and 15.1% (for SP-MZ). For a 50-watt power limit with ParaDiS, *Conductor* performs similar to *Config-Only* and marginally better than *Static*. However, in case of SP-MZ, *Config-Only* performs marginally worse than *Static* due to non-repeatability in performance at the thread concurrency/DVFS configuration selected at 50 watts. At the same time, *Conductor* benefits from load-imbalance and performs better than *Static*.

At higher power limits (60 watts to 80 watts), *Conductor* generally performs slightly worse than *Config-Only* because the computation tasks consume lower power than the process-level power limit and do not benefit from the power re-allocation scheme. The performance impact of power re-allocation scheme is affected by how accurately it can shuffle power between MPI processes without changing the critical path of the application. We observe that for ParaDiS for power limits of 60 watts and 70 watts, the power re-allocation scheme alters the critical path of the application before shifting it back during some time steps. As expected, the higher execution times shown by *Static* are due to the choice of sub-optimal configuration of maximum thread concurrency and DVFS state (8 threads per socket and 2.6GHz on our test system).

For the synthetic program, we show only *Conductor* and *Config-Only*; We leave out the execution times for *Static*, which are inferior to *Config-Only* due to the way we program the synthetic benchmark. For a process-level power limit of 30 watts, the computation times in the load-imbalanced synthetic benchmark are 181 ms and 122 ms for *Config-Only* and *Conductor* respectively (*Conductor* is 32% faster). The corresponding change in frequency was from 1.2 GHz to 1.8GHz. This improvement in computation time results in an overall execution time improvement of 25% with *Conductor*. For the power limit of 40 watts, the corresponding execution times for computation tasks are 123 ms and 86 ms (30% faster with *Conductor*), and the corresponding change in frequency was from 1.9GHz to 2.6 GHz. For higher power limits (60 watts to 80 watts), the difference between the two policies diminishes as power is not a limiting factor on any process, regardless of the load imbalance.

6 Related Work

The closest work to *Conductor* is our own on Adagio [23] and Jitter [16]. In fact, *Conductor* can be thought of as fusing modified versions of Adagio and Jitter together.

Adagio saves energy in HPC programs with a negligible increase in execution time. *Conductor* differs from Adagio in three important ways. First, *Conductor* optimizes performance under a power bound, which is a completely different goal. Second, *Conductor* determines an efficient thread/frequency configuration, while Adagio assumed single-threaded programs. Third, while *Conductor* and Adagio both decrease frequency of tasks that block (and are therefore off of the critical path), *Conductor* (but not Adagio) simultaneously chooses a faster thread/frequency configuration of tasks that may be on the critical path.

Conductor differs from Jitter in two ways. First, *Conductor* shifts power to improve performance, while Jitter lowers frequency to save energy. Second, *Conductor* measures power and makes some power decisions at the task level, while Jitter *solely* operates at timestep granularity.

There is other work in optimizing performance under a power bound, especially on overprovisioned HPC clusters. This includes an empirical study on the effect of different configurations [20] and choosing configurations via interpolation [26]. In addition, Isci et al. optimized performance under a power bound on a single multicore node [15], and Bailey et al. did the same for a CPU+GPU node [4]. There has also been work on scheduling algorithms to improve performance under a power bound [8, 9, 25]. Finally, there has been work on overprovisioning for commercial applications in a datacenter, where the goal is increased throughput [10], as well as in improving performance under power constraints in virtualized clusters in datacenters [19].

Other related work is focused on saving power/energy under a time bound in HPC. There has been work using linear programming to find near-optimal energy savings with zero time increase [24]. Other run-time approaches to save energy include those that trade off power/energy saving for (hopefully minimized) performance degradation [6, 12, 11, 18].

7 Conclusion

Current run time systems are leaving performance on the table and wasting power, and these problems will only become more costly with future generations of supercomputers. *Conductor* effectively allocates power to the parts of the application that primarily impact application performance. In our experiments, we found that selecting the optimal configuration and adaptively re-allocating power to the critical path can result in up to a 30% performance improvement compared to the state-of-the-art algorithm for the same power constraint. In theory, our system can adapt to the job-level power constraint, which may vary during application execution time because of external factors, and adaptively select application configuration and power allocation. Our results also highlight that incorporating OpenMP (or other configurable node-level parallelism) in addition to MPI goes a long way toward the goal of flexible power and performance management.

Acknowledgements

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-667408).

References

1. CoMD. `https://github.com/exmatex/CoMD`, 2013.
2. S. Ashby, P. Beckman, 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, 2010.
3. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, et al. The NAS parallel benchmarks summary and preliminary results. In *Supercomputing*, pages 158–165, 1991.
4. P. E. Bailey, D. K. Lowenthal, V. Ravi, B. Rountree, M. Schulz, and B. R. de Supinski. Adaptive configuration selection for power-constrained heterogeneous systems. In *ICPP*, 2014.

5. V. Bulatov, W. Cai, J. Fier, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis. Scalable line dynamics in ParaDiS. In *Supercomputing*, 2004.
6. K. W. Cameron, X. Feng, and R. Ge. Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In *Supercomputing*, 2005.
7. F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, pages 11–24, 1988.
8. M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Optimizing job performance under a given power constraint in HPC centers. In *IGCC*, 2010.
9. M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Linear programming based parallel job scheduling for power constrained systems. In *HPCS*, 2011.
10. M. E. Femal and V. W. Freeh. Safe overprovisioning: using power limits to increase aggregate throughput. In *PACS*, Dec 2005.
11. R. Ge, X. Feng, W. Feng, and K. W. Cameron. CPU Miser: A performance-directed, run-time system for power-aware clusters. In *ICPP*, 2007.
12. C.-H. Hsu and W.-C. Feng. A power-aware run-time system for high-performance computing. In *Supercomputing*, Nov. 2005.
13. InsideHPC. Power consumption is the exascale gorilla in the room. `http://insidehpc.com/2010/12/10/power-consumption-is-the-exascale-gorilla-in-the-room/`.
14. Intel. Intel-64 and IA-32 Architectures Software Developer's Manual, Volumes 3A and 3B: System Programming Guide, December 2011.
15. 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 *IEEE/ACM International Symposium on Microarchitecture*, pages 347–358, 2006.
16. N. Kappiah, V. W. Freeh, and D. K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. In *Supercomputing*, Nov. 2005.
17. I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.
18. D. Li, B. de Supinski, M. Schulz, K. Cameron, and D. Nikolopoulos. Hybrid MPI/OpenMP power-aware computing. In *IPDPS*, 2010.
19. R. Nathuji, K. Schwan, A. Somani, and Y. Joshi. Vpm tokens: virtual machine-aware power budgeting in datacenters. *Cluster computing*, 12(2):189–203, 2009.
20. 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*, 2013.
21. S. S. Pawlowski. Exascale science: the next frontier in high performance computing. In *International Conference on Supercomputing*, June 2010.
22. B. Rountree, D. H. Ahn, B. R. de Supinski, D. K. Lowenthal, and M. Schulz. Beyond DVFS: A first look at performance under a hardware-enforced power bound. In *HPPAC*, 2012.
23. B. Rountree, D. K. Lowenthal, B. de Supinski, M. Schulz, and V. W. Freeh. Adagio: Making DVS practical for complex HPC applications. In *ICS*, 2009.
24. B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. de Supinski, and M. Schulz. Bounding energy consumption in large-scale MPI programs. In *Supercomputing*, Nov. 2007.
25. O. Sarood, A. Langer, A. Gupta, and L. Kale. Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *Supercomputing*, 2014.
26. O. Sarood, A. Langer, L. Kalé, B. Rountree, and B. De Supinski. Optimizing power allocation to cpu and memory subsystems in overprovisioned hpc systems. In *CLUSTER*, 2013.
27. R. F. vanderWijngaart and J. Haopiang. NAS parallel multi-zone benchmarks. 2003.