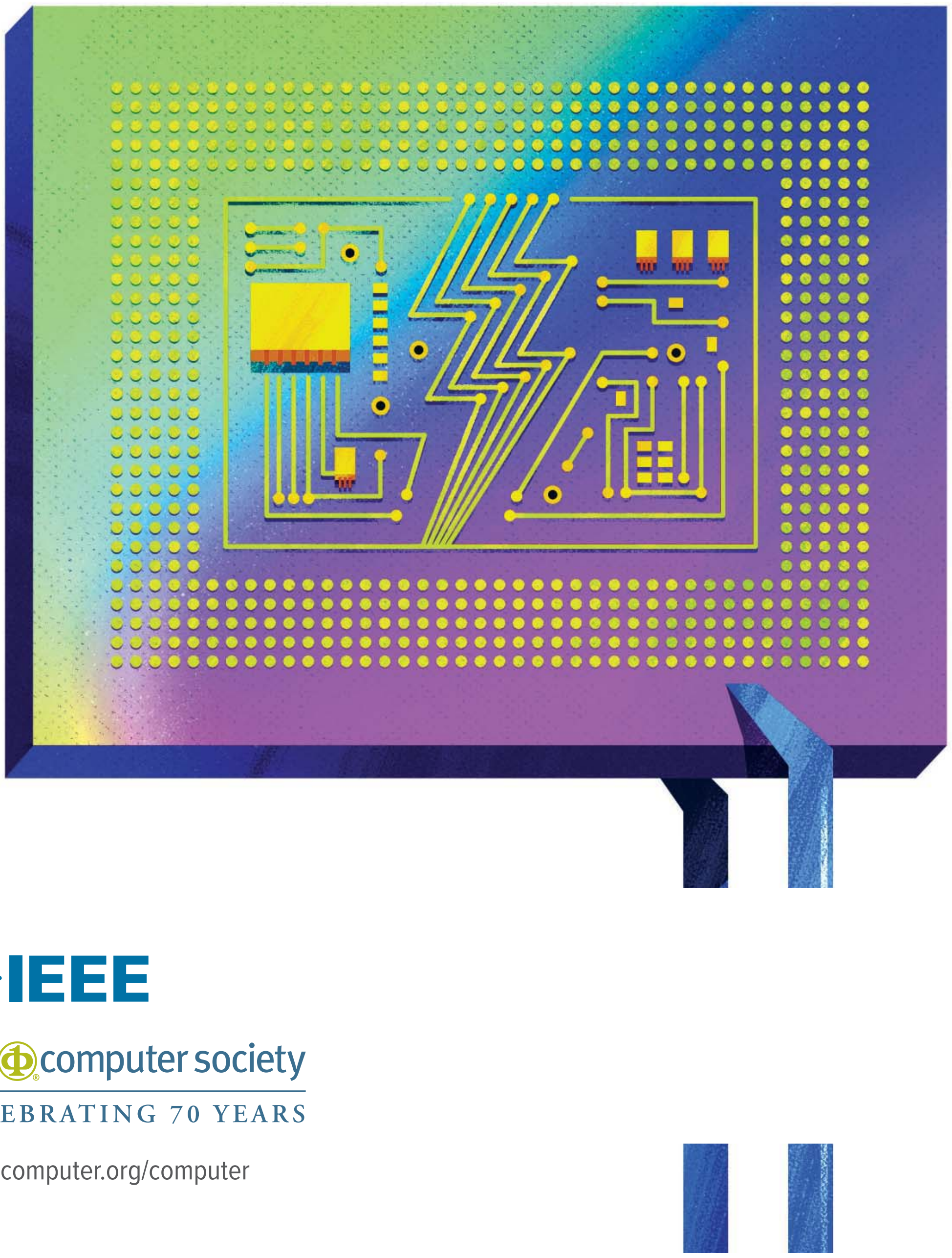
10.16

Computer

ENERGY-EFFICIENT COMPUTING



IEEE

IEEE computer society

CELEBRATING 70 YEARS

www.computer.org/computer

MULTIMEDIA from Computer's October 2016 Issue

X

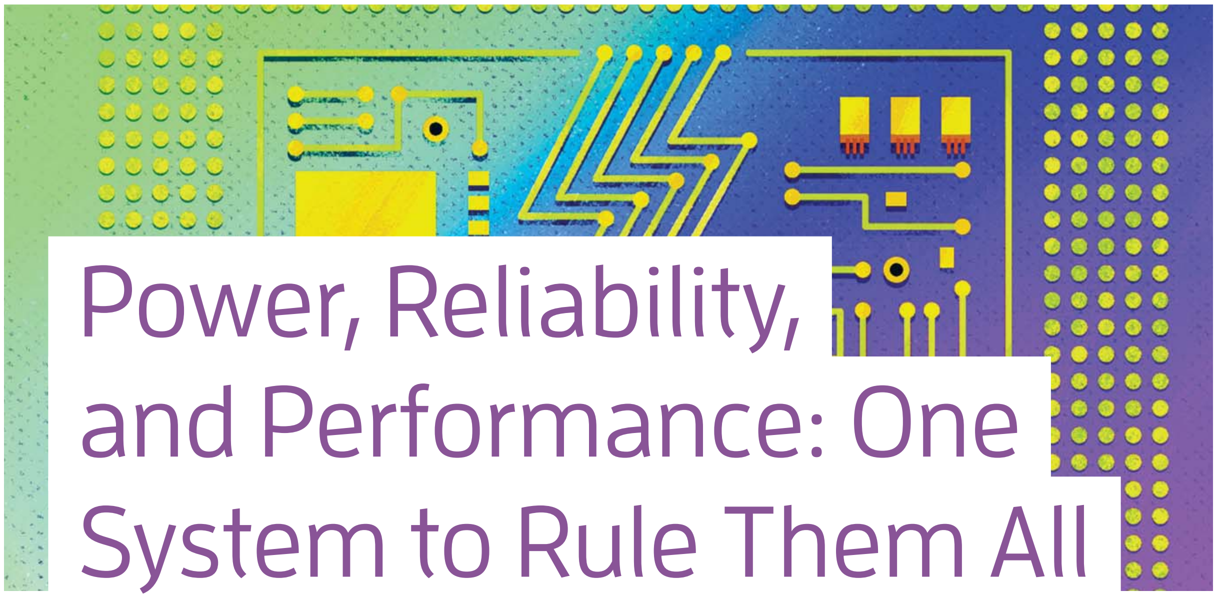
pg. 12 Author Charles Severance goes behind the scenes at the Living Computer Museum. View the full-resolution video at https://youtu.be/ZHU4nzIsaIM. Subscribe to the Computing Conversations podcast on iTunes at https://itunes.apple.com/us/podcast/computing-conversations/id731495760.

pg. 12 Author Charles Severance provides an audio recording of his Computing Conversations column in which he recounts his visit to the Living Computer Museum. Subscribe to the Computing Conversations podcast on iTunes at https://itunes.apple.com/us/podcast/computing-conversations/id731495760.

pg. 14 In this video at https://youtu.be/35PaR65EYa8, Karlheinz Meier from Ruprecht-Karls University in Heidelberg speaks about neuromorphic computing's concepts, achievements, and challenges at the 2016 International Supercomputing Conference.

pg. 112 Author David Alan Grier reads his Errant Hashtag column, in which he traces the influence of the American industrial experience on Silicon Valley's innovation system. Subscribe to the Errant Hashtag podcast on iTunes at https://itunes.apple.com/us/podcast/errant-hashtag/id893229126.

VLADIMIR GETOV, ADOLFY HOISIE, AND PRADIP BOSE



Power, Reliability, and Performance: One System to Rule Them All

Bilge Acun, University of Illinois at Urbana–Champaign
Akhil Langer, Intel
Esteban Meneses, Costa Rica Institute of Technology and Costa Rica National High Technology Center
Harshitha Menon, University of Illinois at Urbana–Champaign
Osman Sarood, Yelp
Ehsan Totoni, Intel Labs
Laxmikant V. Kalé, University of Illinois at Urbana–Champaign

In a design based on the Charm++ parallel programming framework, an adaptive runtime system dynamically interacts with a datacenter's resource manager to control power by intelligently scheduling jobs, reallocating resources, and reconfiguring hardware. It simultaneously manages reliability by cooling the system to the running application's optimal level and maintains performance through load balancing.

High-performance computing (HPC) datacenters and applications are facing major challenges in reliability, power management, and thermal variations that will require disruptive solutions to optimize performance. A unified system design with a smart runtime system that interacts with the system resource manager will be imperative in addressing these challenges. The system would be part of each job, but interact with an adaptive resource manager for the whole machine. Studies have shown that a smart, adaptive runtime system can improve efficiency in a power-constrained environment,[1] increase performance with load-balancing algorithms,[2] better control the reliability of supercomputers with substantial thermal variations,[3] and configure hardware components to operate within power constraints to save energy.[4,5]

Although smart runtime systems are a promising way to overcome exascale computing barriers, there is

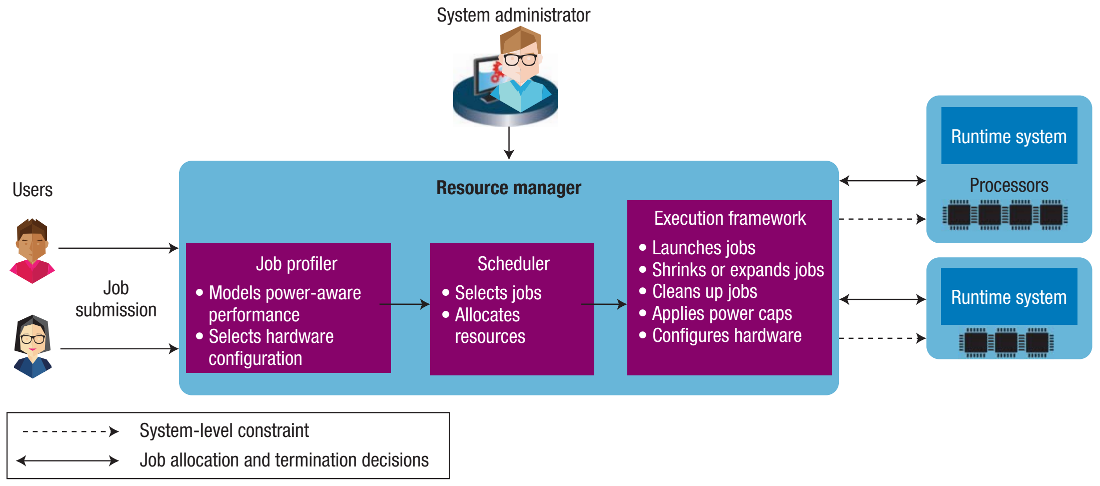


FIGURE 1. Interacting components in our design based on the Charm++ framework. The two major interacting components are the resource manager and an adaptive runtime system. Our design achieves the objectives of both datacenter users and system administrators by allowing dynamic interaction between the system resource manager or scheduler and the job runtime system to meet system-level constraints such as power caps and hardware configurations.

no integrated solution that combines past research into a single system for optimizing performance across multiple dimensions, such as load balancing and power use. To fill that need, we have developed a comprehensive design based on Charm++, a parallel programming framework in C++ that has been in use since 2001. Our design

- lets the datacenter resource manager dynamically interact with the individual job's runtime system,
- optimizes for both performance and power consumption, and
- enables operation in an environment with system failures under constraints supplied by users or administrators.

At Charm++'s core is an adaptive runtime system that enables the dynamic collection of performance data, dynamic task migration (load balancing), and temperature restraint and power capping with optimal performance. The ability to migrate tasks and data from one processor to any other available processor is critical to solving conflicting requirements, such as application load imbalances across processors, high fault rates, power and energy constraints, and thermal variations. Balance is essential in addressing these concerns because of potential conflicts among them—for example, applying power and temperature constraints can compromise performance and lead to a load imbalance.

Research has shown that Charm++ enhances user productivity and allows programs to run portably from small multicore computers (laptops, phones) to the largest supercomputers.[6] Users can easily expose and express much of the parallelism in their algorithms while automating many of the requirements for high performance and scalability. Because of these strengths, Charm++ has thousands of users across a wide variety of computing disciplines with multiple large-scale applications, including simulation programs such as Nanoscale Molecular Dynamics—formerly Not (just) Another Molecular Dynamics program—for molecular dynamics, ChaNGa for cosmology, and OpenAtom for quantum chemistry.[6]

In our design, the Charm++ adaptive runtime system intelligently schedules jobs and reallocates job resources when utilization changes, controls reliability by a temperature-aware module that cools the system to an application-based optimal level, and reconfigures the hardware without sacrificing performance.

To evaluate our solution, we conducted several efficiency tests. Our results show that the adaptive runtime system's capabilities result in greater power efficiency for common HPC applications.

INTERACTION FLOW

Figure 1 shows how Charm++'s adaptive runtime system interacts with the resource manager to address power, reliability, and performance issues. Datacenter users are focused on job performance. System administrators are tasked with guaranteeing good performance to individual jobs, yet ensuring that total power consumption does not exceed the center's allocated budget and that overall job throughput remains high despite node failures and thermal variations. Charm++ addresses both concerns. The job scheduler strives to allocate system resources to jobs optimally according to their power and performance characteristics, and the runtime system implements the scheduler's decision by shrinking or expanding the number of nodes assigned and dynamically balancing load as needed. The runtime system can turn on or off or reconfigure

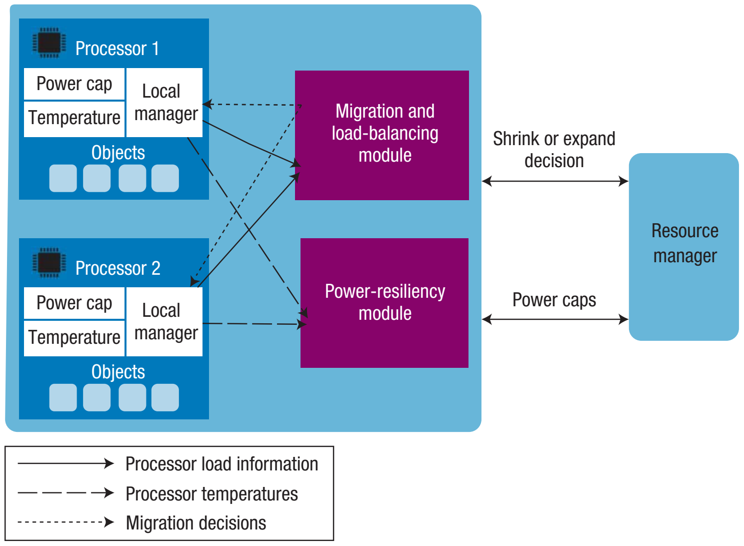


FIGURE 2. Components of the adaptive runtime system and their interaction with the resource manager. Charm++ has three main components: the local manager and the load-balancing and power-resiliency modules.

hardware components without impacting application performance, as long as vendors provide adequate hardware control.

CHARM++ ATTRIBUTES

Charm++ has three main attributes. The first is *overdecomposition*, in which the programmer divides an application's computation requirements into small work and data units so that the number of units greatly exceeds the number of processors. The second attribute is *message-driven execution*, which involves scheduling work units on the basis of when a message is received for them. The third attribute, *migratability*, is the ability to move data and work units among processors. These three attributes enable Charm++ to provide many useful features including dynamic load balancing, fault tolerance, and job malleability (shrinking or expanding the number of processors the application is running on).

The framework collects information about the application and system in a distributed database including processor loads, each object's load, communication patterns, and core temperatures. With a large number of processors, centralized data collection becomes a performance bottleneck, so data collection and decision making are hierarchical. The adaptive runtime system's modules use this information to make decisions such as improving load balance, handling faults, and enforcing power constraints.

Load balancing

One critical Charm++ feature is a measurement-based mechanism for load balancing that relies on the principle-of-persistence heuristic. This principle states that, for overdecomposed iterative applications, the task's or object's computation load and communication pattern tend to persist over time. The heuristic uses the application's load statistics collected by the runtime system, which provides an automatic, application-independent way of obtaining load statistics without any user input.

If desired, the user can specify predicted loads and thus override system predictions. Using the collected load statistics, Charm++ executes a chosen load-balancing strategy to determine a possible objects-to-processors mapping and then carries out migrations on the basis of this mapping. A suite of load balancers includes several centralized, distributed, and hierarchical strategies.

Charm++ can also automate the decision of when to call the load balancer,[7] predict future load, and make load-balancing decisions. Load balancing is automatically triggered when Charm++ detects an imbalance and load-balancing benefits are likely to exceed its overhead.

Fault tolerance

Charm++ implements both proactive and reactive techniques for ensuring reliability.[8] In a proactive strategy, the runtime system evacuates all objects from a node that a monitoring system predicts will soon crash. Because failure prediction is not completely accurate, reactive techniques recover the information lost after a failure brings down a system node. Because recovery techniques are based primarily on checkpoint and restart, Charm++ routinely stores the application's global state so that it can retrieve a prior global state during recovery.

Job malleability

The ability of Charm++ objects to migrate enables job malleability, in which a job can shrink (decrease) or expand (increase) the number of nodes it is running on. Job malleability does not require any additional code from the application developer;[9] rather, an external command or internal decision by the runtime system can trigger shrink or expand operations.

During a shrink operation, the runtime system moves objects from processors that will no longer be used and returns them to the resource manager.

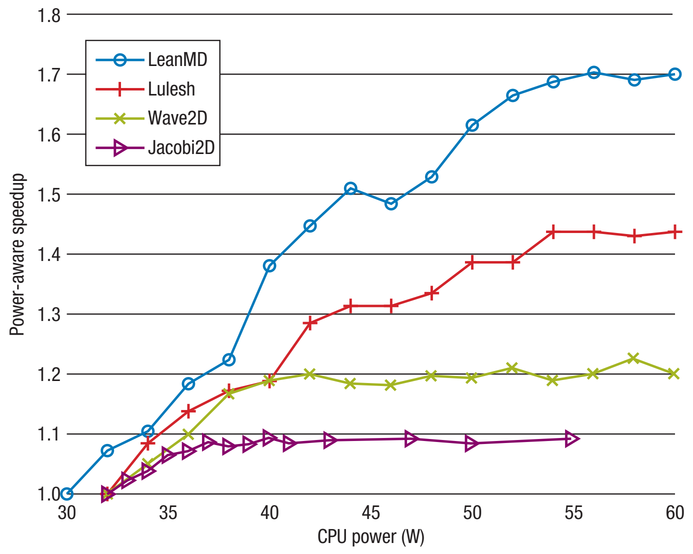


FIGURE 3. Power-aware speedups of four applications running on 20 nodes. LeanMD, a molecular dynamics application, has the highest power-aware speedup because it is the most CPU-intensive application. Jacobi2D, a stencil application, has the lowest speedup because it is memory intensive. The many minor deviations from monotonic behavior (lack of a smooth curve) are likely due to external factors such as OS jitter and network delays.

In an expand operation, the runtime system launches new processes on the additional processors and distributes objects from current processors to the newly allocated ones. In addition to moving objects, the runtime system must adjust its distributed data structures, such as spanning trees and location managers.

Figure 2 shows Charm++'s internal components and functions. Each processor has a local manager (LM) that controls objects residing on that processor and interaction with other Charm++ components. The LM sends its total computational load and the computational load of each of its objects to the load-balancing module (LBM) and sends the CPU temperature to the power-resiliency module (PRM). The LBM makes load-balancing decisions and redistributes load in response to shrink or expand commands from the resource manager. It also communicates object-migration decisions to the respective LMs.

The PRM ensures that the CPU temperatures remain below the temperature threshold specific to that job, controlling temperature by adjusting the CPU's power cap. When a processor's temperature is above the threshold, the PRM lowers its power cap; when the temperature is well below the desired threshold, it increases the cap, thus ensuring that the total power the job consumes remains below that job's allocated power budget. Jobs do not have administrator rights to constrain their CPUs' power consumption, so the PRM must communicate any new power caps to the resource manager, which then applies them to the CPUs.

HANDLING POWER LIMITS

The US Department of Energy has set a power limit of 20 MW for an exascale supercomputer that it is willing to purchase and deploy.[10] Such a limit underlines the need for power-aware resource management along with job malleability.

With recent advances in processor hardware design, users can employ software such as Intel's Running Average Power Limit (RAPL) driver to control the power the processor consumes. Power capping forces the processors to run below their *thermal design power* (TDP)—the maximum power a processor can consume. A node's TDP also determines the maximum number of nodes that a datacenter with a power budget can use.

Overprovisioning

With power capping, the datacenter can control the nodes' power consumption and thus have additional nodes while remaining within the power budget—a practice referred to as *overprovisioning*.[11] Research shows that an increase in the power allocated to a processor does not yield a proportional increase in job performance because jobs react differently to an increase in power allocated to the CPU.[1] The idiosyncrasies in a job's performance based on allocated CPU power points to the possibility of running different applications at different power levels. Overprovisioned systems can significantly improve the performance of applications that are not sensitive to CPU power by capping CPU power to values well below their TDP and adding more nodes to get the scaling benefits.

Power-aware speedup

An application's response to a CPU power limit can be captured by the application's power-aware speedup[12]—the ratio of the job's execution time on a CPU capped at a certain power level compared to the execution time of the same job when running on the lowest power level the CPU allows.[1] The higher the power-aware speedup, the more the sensitive the application is to changes in the power allocated to the CPU.

Figure 3 shows the power-aware speedups for four HPC applications with different characteristics running under different CPU power caps.[1]

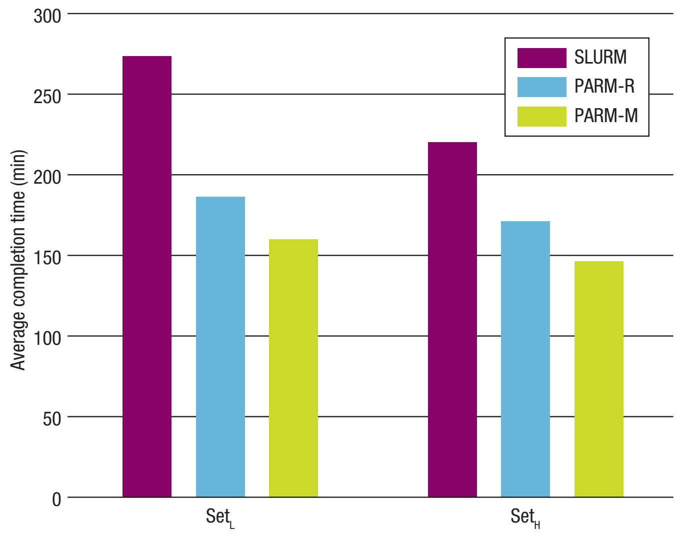


FIGURE 4. Comparison of average job-completion times with the Simple Linux Utility for Resource Management (SLURM; now the Slurm Workload Manager) and the Power-Aware Resource Manager (PARM) in rigid (R) and malleable (M) versions. Each bar represents average completion time as a percentage of the average completion time using SLURM. In PARM-R, the node allocation decision cannot be changed once the job starts running. With PARM-M, in contrast, the nodes allocated to a running job can either decrease or increase (shrink or expand). Set$_L$ has jobs with low sensitivity to CPU power, and Set$_H$ has jobs with high sensitivity.

Power-Aware Resource Manager

The Power-Aware Resource Manager (PARM),[1] developed at the Parallel Programming Laboratory of the University of Illinois at Urbana–Champaign, is an essential part of our Charm++-based design. PARM makes scheduling decisions by selecting jobs and their resource configurations (power budget and compute nodes) such that the total power-aware speedup of running jobs is maximized. It dynamically interacts with the runtime system, hardware, user, and system administrator to optimally distribute available resources to the jobs while staying under the datacenter's total power budget and not exceeding the allowable number of computational nodes. Similar to the resource manager in Figure 1, PARM has three components.

Job profiler. Before a job is added to the scheduler queue, PARM profiles it to develop a power-aware model with strong scaling that is the basis for calculating power-aware speedups. This profiling mechanism has negligible overhead; the application needs to run only for a few iterations for PARM to get the necessary data points.

Scheduler. PARM implements its strategy for optimizing resource allocation as an integer linear program (ILP) that aims to maximize the power-aware speedup of jobs running under power constraints. The arrival of a new job or termination of a running job triggers PARM's scheduler and reoptimizes scheduling and resource-allocation decisions. The ILP is fast enough to run frequently with negligible overhead.[1]

Execution framework. This component implements the scheduler decisions by launching jobs, sending shrink/expand decisions to the runtime system of the jobs, and by applying power caps on compute nodes. Job runtime systems interact with the execution framework to convey job termination, completion of shrink/expand operations, and any changes to CPU power caps as determined by the runtime system's PRM module.

Comparison with other resource managers. As Figure 4 shows, PARM outperforms the Simple Linux Utility for Resource Management (SLURM; now the Slurm Workload Manager)—an open source power-unaware utility that many supercomputers use. The malleable version of PARM (PARM-M) reduced average job completion time by up to 41 percent relative to SLURM. PARM gives less performance benefit over SLURM with Set$_H$ as compared to Set$_L$ because reducing CPU power greatly affects performance for jobs with high power sensitivity (Set$_H$).

IMPROVING RELIABILITY

Checkpoint and restart is the most popular mechanism for providing fault tolerance in high-performance computers and is used to recover work from the stored checkpoint before the failure. An application's total execution time T on an unreliable system is given by

$$T = T_{solve} + T_{checkpoint} + T_{recover} + T_{restart},$$

where T_{solve} represents the total effort required to solve the problem, $T_{checkpoint}$ accumulates all the time spent on saving the system checkpoints, $T_{recover}$ represents the total lost work from system failures that must be recovered, and $T_{restart}$ represents the amount of time required to resume execution after a crash (usually a constant value).

A system using checkpoint and restart must choose an appropriate checkpoint period, denoted by τ. The value of τ has a delicate balance. A long value (low checkpoint frequency) decreases $T_{checkpoint}$, but might increase $T_{recover}$. Conversely, a short value (high checkpoint frequency) could reduce $T_{recover}$, but might enlarge $T_{checkpoint}$. The optimum value of τ strongly depends on the system's mean time between failures (MTBF).

An electronic component's MTBF is directly affected by the component's temperature. That relation is usually exponential, and there is some experimental evidence that a 10°C increase in a processor's temperature decreases its MTBF by half.[3] Therefore, system reliability can be controlled by restraining the temperature of its components. The cooler the system runs, the more reliable it is, but the slower it runs because CPU power is constrained to keep temperature down. This interplay between power management and reliability has been studied in other contexts.[13]

The runtime system allows each core to consume the maximum allowable power as long as it is within the maximum temperature threshold. If the temperature any core exceeds the maximum threshold, power is decreased, causing the core's temperature to fall. However, this can degrade the performance of tightly coupled applications because of thermal variations. The LBM will automatically detect any load imbalance and make the load-balancing decisions.[2]

The runtime system must strike a balance in the temperature at which each component should be restrained. Moreover, that balance depends on the application. Different codes generate different thermal profiles on the system at different stages of the application. Some codes are more compute intensive and tend to heat up the processors more quickly.

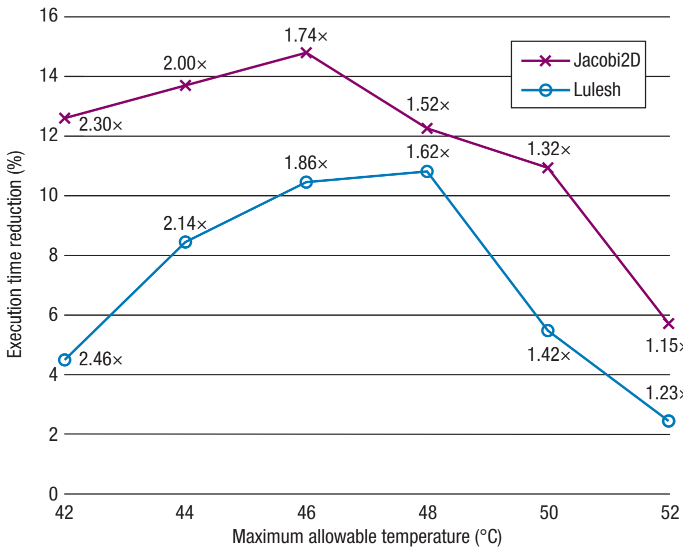


FIGURE 5. Reduction in execution time and change in mean time between failures (MTBF) for different temperature thresholds. Comparison is relative to a baseline case in which core temperature is unconstrained.

Appropriate, application-based temperature thresholds are stored as part of the job profiler in Figure 1. In the end, the runtime system aims to reduce the application's total execution time in light of the system's MTBF and power limitations.[3]

Figure 5 shows the percentage reduction in execution time after constraining core temperatures to different thresholds for two different applications. The reduction is calculated relative to the baseline case in which processor temperature is unconstrained. Figure 5 also shows the ratio of MTBF for the machine with our design versus the MTBF for the baseline case. For example, by restraining core temperatures to 42°C for Jacobi2D, the machine's MTBF increased 2.3 times while the execution time decreased by 12 percent relative to the baseline case. The inverted U shape of both curves strongly suggests a tradeoff between reliability (MTBF) and the slowdown induced by the temperature restraint.

The resource manager sends the runtime system's PRM the upper bounds of the temperatures that honor the system's power envelope (as shown in Figure 2). These temperature values are input to an internal resilience component in PRM and are changed according to algorithms that optimize performance and consider the system's MTBF and the running application's characteristics. The output will be propagated to additional PRM components, which will consolidate the final power limits and communicate them back to the resource manager.

As this interaction sequence implies, a dynamic runtime system is fundamental to controlling system reliability while simultaneously adhering to power limits. Because thermal variations are dynamic, a reactive runtime system efficiently responds to those changes and provides a healthy balance between system performance and reliability. It also allows users to address a broader range of reliability concerns, including how to correct soft errors (such as bit flips caused by high-energy particles) in tandem with hard errors.[14] Circuits using near-threshold voltage will introduce a more complex scenario with higher performance variability and a higher transient-failure rate.

DYNAMIC CONFIGURATION

The runtime system can exploit available hardware controls for power

ABOUT THE AUTHORS

BILGE ACUN is a doctoral student in the Department of Computer Science at the University of Illinois at Urbana–Champaign (UIUC). Her research interests include power-aware software design, malleability, and variability in large-scale applications. Acun received a BS in computer science from Bilkent University. She is a member of ACM. Contact her at acun2@illinois.edu.

AKHIL LANGER is a software engineer at Intel. His research interests include scalable distributed algorithms and power and communication optimizations in high-performance computing (HPC). Langer received a PhD in computer science from UIUC. Contact him at akhil.langer@intel.com.

ESTEBAN MENESES is an assistant professor in the School of Computing at the Costa Rica Institute of Technology and the Costa Rica National High Technology Center. His research interests include reliability in HPC systems, parallel-objects application design, and accelerator programming. Meneses received a PhD in computer science from UIUC. He is a member of ACM. Contact him at esmeneses@tec.ac.cr.

HARSHITHA MENON is a doctoral student in the Department of Computer Science at UIUC. Her research interests include scalable load-balancing algorithms and adaptive runtime techniques to improve large-scale application performance. Menon received an MS in computer science from UIUC. Contact her at gplkrsh2@illinois.edu.

OSMAN SAROOD is a software engineer at Yelp. His research interests include performance optimization for parallel and distributed computing under power and energy constraints. Sarood received a PhD in computer science from UIUC. Contact him at osarood@yelp.com.

EHSAN TOTONI is a research scientist at Intel Labs. His research interests include programming systems for HPC and big-data analytics. Totoni received a PhD in computer science from UIUC. He is a member of IEEE and ACM. Contact him at ehsan.totoni@intel.com.

LAXMIKANT V. KALÉ is a professor in the Department of Computer Science at UIUC. His research interests include aspects of parallel computing, with a focus on enhancing performance and productivity through adaptive runtime systems. Kalé received a PhD in computer science from Stony Brook University. He is a Fellow of IEEE and a member of ACM. Contact him at kale@illinois.edu.

management, such as frequency scaling and power capping. However, greater power savings are possible with more runtime control over hardware components. By taking advantage of the running application's properties, the runtime system can turn off or reconfigure many components without a significant performance penalty.

Ideally, high-performance computing systems should be energy proportional; the hardware components should consume power and energy only when their functionality is being used. However, network links are always on, whether or not they are actually being used. Moreover, processor caches consume large amounts of power, even when they are not improving the application's performance. The runtime system approach can save this wasted energy by dynamically reconfiguring hardware as the application requires.

Caches consume up to 40 percent of a processor's power.[4] Much of that consumption can be avoided by turning off some cache banks when doing so would not degrade application performance. Many common HPC applications do not use caches effectively. For example, molecular dynamics applications typically have small working data sets and do not need the large last-level caches. Grid-based physical simulation applications typically have very large data sets that do not fit in caches, and the data reuse in cache is minimal. The hardware cannot predict this application behavior.

To address this concern, we developed an approach in which the runtime system uses profiling data to reconfigure the cache to save power without significant performance loss. Using a set of representative HPC applications, we demonstrated that,