EVeREST-C: An Effective and Versatile Runtime Energy Saving Tool for CPUs

Anna Yue
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN, USA
yue00082@umn.edu

Pen-Chung Yew
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN, USA
yew@umn.edu

Sanyam Mehta
Hardware-software Codesign
Hewlett-Packard Labs
Bloomington, MN, USA
sanyam.mehta@gmail.com

Abstract

Power and energy efficiency are increasingly important challenges within HPC. However, it is still important to achieve these goals while maintaining desired/high application performance. Balancing these goals involves the challenge of precise application characterization. For successful user adoption, this must avoid modifying the application and/or extraneous application profiling, and also be portable to different processors across processor generations and vendors.

We propose EVeREST-C to solve these challenges. Everest targets the finer-grained individual application functions for exploiting power/energy saving opportunities via Dynamic Voltage Frequency Scaling (DVFS) in both the core and the uncore, without application-specific knowledge. Since Everest relies on a single standard and accurate performance event, IPS (instructions per second), for its characterization rather than on the (many) performance counters that can differ across platforms, it is portable across processors. Finally, the fine-grained approach enables Everest to additionally save power/energy for select communication (MPI) phases, where appropriate phases are chosen based on both their length and position in the application with regards to the memory/compute boundedness of surrounding user routines. We evaluate Everest using SPEC CPU 2017 and various MPI applications, on Intel and AMD platforms. We find that Everest saves on average 11% more energy for SPEC compared to the baseline and 8% more energy on MPI applications compared to a state-of-the-art solution.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725759

CCS Concepts

• **Software and its engineering → Power management**; • **Hardware → Chip-level power issues**; • **Computer systems organization → Multicore architectures**.

Keywords

Power, Energy-efficiency, CPUs, DVFS, MPI

ACM Reference Format:
Anna Yue, Pen-Chung Yew, and Sanyam Mehta. 2025. EVeREST-C: An Effective and Versatile Runtime Energy Saving Tool for CPUs. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721145.3725759

1 Introduction

Concern over power and energy efficiency in the high performance computing (HPC) industry continues to grow, especially as we enter the post-exascale era. This interest has been exacerbated by rising energy costs combined with increasing power requirements as well as by environmental sustainability goals such as reduced carbon emissions [48, 51]. However, these concerns often conflict with the ever-increasing desire for more performance, leading to contrasting demands for hardware and software to manage.

Dynamic Voltage Frequency Scaling (DVFS) lends itself as an effective mechanism for reducing dynamic power, which is proportional to CV^2f, demonstrating a cubic relationship when both voltage and frequency decrease. DVFS is thus most effective when performance loss is minimal, such as lowering *core* clock during memory bound [15, 20, 23, 47, 54] and communication bound [3, 5, 45, 52] phases or lowering *uncore* clock during compute bound [17] phases.

Although DVFS has been long around, the industry still lacks a practical and useful tool that effectively and automatically manages clocks at application runtime for power and energy savings while ensuring desired performance. This is because of various challenges:

Adapting to program behavior. Most applications are neither fully compute bound nor memory bound, i.e., applications usually have phases with different behavior. For example, HPC applications often model different sciences in succession. Thus, a natural method for separating phases is at the communication call granularity [7, 22, 45], which doubly benefits from the ability to intercept standardized communication (MPI) calls at runtime. However, this approach misses phases that occur in-between communication calls, which can lead to lost performance and/or overprovisioned power. Other works avoid this pitfall by separating phases at the function or loop granularity [1, 11, 14, 42, 55], but this comes at the cost of extra application profiling efforts such as through offline analysis or code instrumentation through compiler support [55] or manual annotations [11].

On the other hand, many approaches opt for application-oblivious profiling, where the application is sampled at a fixed instruction or time interval [15, 17, 20, 23]. While this removes dependence on extraneous application profiling and the granularity is only limited by overhead and hardware capabilities, the chosen metric(s) to sample become especially critical. This is because these approaches rely on a change in the measured values to indicate a phase change. As a result, most of these approaches measure a combination of counters which indicate compute activity and memory activity. This is problematic for reasons explained next.

Performance counters and their problems. Prior art proposes the use of various combinations of hardware performance counters for application characterization and/or phase detection [4, 13, 25, 49, 54]. However, the accuracy of these approaches varies depending on how the chosen counters detect the level of overlap between memory accesses and computation [40]. Furthermore, both the events collected and the accuracy of performance counters varies widely across processor vendors (and even processors of the same vendor) as reported previously [35, 36, 53]. As a result, a single set of performance counters identified through rigorous training/calibration for a single processor will likely fail to work for another processor or vendor.

Handling communication for parallel applications. While communication phases where communicating processes often 'wait' to send/receive data offer a significant additional power saving opportunity especially for HPC, the variety of communication calls (such as point to point vs. collective communication) combined with the varied distribution of payload size and time spent in those calls makes it difficult to extract savings. For instance, if core clock is lowered during a short 'wait' call that appears in a compute bound phase of the application, then application performance (and energy efficiency) could take a severe hit.

We propose EVeREST-C, a lightweight, effective, and versatile tool that utilizes DVFS to automatically (without user intervention) and dynamically characterize application phases and adjust clocks to save power and energy for CPUs. Specifically, Everest balances the tradeoff between adapting to program behavior and portability by utilizing a sampling approach for application characterization while separating phases at the function granularity. This enables Everest to rely on a minimal set of performance counters, measuring IPS (instructions per second) at two different CPU frequencies to determine memory and compute boundedness for a phase. Everest then determines the ideal frequency (both core as well as uncore) to run each phase at, to maximize power savings while ensuring a specified performance threshold. Additionally, Everest identifies MPI calls that commonly contribute to slack time and achieve important energy savings by selectively lowering clocks depending on slack duration.

Everest addresses the aforementioned three challenges through the following contributions:

- Everest is *effective* in maximizing power saving opportunities within applications by adapting to the finer-grained individual application functions as opposed to the (much) coarser communication calls. Everest achieves this dynamically, without application-specific knowledge (e.g., through user intervention, extraneous application profiling, compiler support).
- Everest is *versatile* as it avoids the common problems associated with performance counters across processors. This is achieved by relying only on a single default (i.e., fixed) performance event, retired instructions, for the purpose of application characterization. This event is both standard and accurate across all processors.
- Everest *effectively exploits additional opportunities* to save power in communication phases. Everest is selective in choosing power saving opportunities to avoid hurting performance by accounting for both the time spent in communication and the nature (compute/memory boundedness) of the surrounding user routines.
- We implement and evaluate our approach on the latest generation Intel and AMD server processors using the SPEC CPU 2017 benchmark suite and various MPI applications from the CORAL-2/ECP suites in single-node and multi-node runs. Everest achieves an average 11% more energy savings on SPEC CPU 2017 applications compared to the baseline and 8% more energy savings on MPI applications compared to a state-of-the-art power management solution.

2 Related Work

With the widespread adoption of DVFS, there exists extensive literature exploiting the power-peformance tradeoff from

DVFS when there is minimal performance loss, especially within HPC. These works target specific opportunities and can be broadly categorized into approaches which rely primarily on software or hardware to identify and react to program phases. We describe the limitations of these approaches, with concrete examples in Section 3.

Software approaches. Many works make use of the natural program phases present in HPC applications, which arise due to the nature of parallel (MPI) communication and synchronization [3, 6, 7, 14, 22, 45, 52]. For determining memory and compute boundedness, strategies which solely rely on communication calls to delineate phases [7, 22] miss phases that occur in-between these calls. Some approaches mitigate this by additionally accounting for load imbalance when determining the appropriate DVFS setting, and only slow ranks which are not on the critical path utilizing per-core DVFS [3, 45]. Other strategies only target the communication calls themselves [6, 52], lowering frequency as long as sufficient (slack) time is spent within the call. While this can result in significant savings, these works do not consider the surrounding compute/memory bound user routines; to our knowledge, Everest is the first of its kind to do so.

A more general and finer-grained approach is to consider program phases at the function or loop level [1, 14, 42, 55]. Freeh et al. [14] identify phases through a combination of a program trace of MPI calls and loop nests and the measured operations per miss. Acun et al. [1] in particular make use of per-core DVFS to account for overlapping kernels and dedicated I/O or communication threads common in HPC applications. However, while these approaches are highly accurate in adapting to phase behavior, they are either restricted to using a specific programming framework/compiler [1, 55] or require multiple a priori application runs to identify the optimal frequency for minimizing energy [14, 42].

Hardware approaches. Without knowledge of the underlying application, these approaches detect phase changes by measuring some combination of hardware counters. Early works explored using offline analysis (i.e., not done at runtime) to determine which set of performance counters to best use for power and performance estimation. Weissel and Bellosa [54] and Snowdon et al. [49] both use microbenchmarks to evaluate the correlation of different events with performance and energy consumption, and subsequently use those events to determine the ideal frequency at run time. Although these approaches save significant energy especially from memory-bound applications, they assume an accurate system characterization that is not prone to training error, which requires a comprehensive selection of benchmarks.

Other approaches operate completely at run time. Most of these strategies rely on a detailed set of performance counters measuring both compute and memory [15, 23, 26, 38, 44], limiting their portability due to the difference in accuracy and availability of performance counters across platforms. For instance, Ge et al. [15] achieve up to 20% energy savings with user-defined performance limits, but even admit themselves that the chosen performance events (L1 D-cache accesses, L2 D-cache accesses, memory data accesses) were chosen for their specific processors and may require adjustment for other architectures. Similarly, Uncore Power Scavenger [17] detects phase changes using DRAM power and instructions per cycle (IPC); DRAM power reporting is currently limited to Intel platforms. Hsu and Feng [20] propose an algorithm using only IPS to measure performance impact from frequency changes, but without accounting for phases. Everest avoids these hardware portability issues by relying on only IPS, while still detecting program phases fully at runtime.

Limitations. Each approach makes individual contributions towards Everest's three proposed contributions: adapting to application characteristics (e.g., memory-boundedness) at a fine granularity (*effective*), runtime characterization avoiding performance counter limitations and user intervention (*versatile*), and exploiting additional opportunities in communication phases and the uncore (*flexible*). Everest fully addresses each challenge, consolidating them all into a practical and useful tool. We categorize the most notable prior works in Table 1.

Table 1: Prior Works

Citation	Effective	Versatile	Flexible
Ge et al. [15] (CPU MISER)	✓	✗	✗
Isci et al. [23] (GPHT)	✓	✗	✗
Acun et al. [1] (Charm++ module)	✓	✗	✓
Hsu and Feng [20] (β-adaptation)	✗	✓	✗
Corbalan et al. [7] (EAR)	✓	✗	✓
Rountree et al. [45] (Adagio)	✓	✓	✓
Venkatesh et al. [52] (EAM)	✗	✗	✓
Gholkar et al. [17] (UPS)	✗	✗	✓

Only Adagio [45] also employs effectiveness (adapts to applications at MPI call granularity), versatility (only measures instructions per second), and flexibility (also targets communication phases). However, its effectiveness and flexibility are limited; Adagio does not account for phases in-between MPI calls (like EAR [7]), nor does it consider the uncore.

More importantly, the goals of Adagio and Everest are very different. Everest explicitly targets energy-saving opportunities in both load-balanced and load-imbalanced applications, while Adagio implicitly exploits opportunities in load-imbalanced applications. Specifically, Adagio's objective is to slow down processors that are not on the critical path. On the other hand, Everest handles computation and communication phases more broadly, calculating compute-boundedness to determine the frequency scaling factor (core and uncore) to meet a given performance threshold.

In summary, no other proposed DVFS-based runtime combines effectiveness, versatility, and flexibility to achieve the same goals as Everest. Additionally, Everest uniquely considers the surrounding compute/memory bound computation phases when determining communication phase frequency, ensuring the best outcome for performance and energy-efficiency for the variety of communication phases present in HPC applications.

Furthermore, in practice, modern systems have not adopted any above methods of application characterization for saving power/energy; the Linux ondemand governor for instance uses the simplest characterization, decreasing frequency when CPU utilization is low (i.e., fully idle), but otherwise running at the highest clock. To our knowledge, only EAR [7] is employed at a Top500 system.

Other Power Management Techniques. Other mechanisms for saving power (and energy) present an opportunity to extend our approach. Greater potential power savings can be achieved by incorporating memory DVFS [8] and properties of asymmetric multicores [16, 27], while still using our existing characterization. Meng et al. [37] provide a model for determining the optimal combination of power optimizations. If job throughput is of greater concern, power savings can be allocated towards hardware overprovisioning [41, 46], with greater flexibility due to our PD configurability. Moreover, different domains such as servers and data centers are often addressed with scheduling techniques [2, 24], given that applications are commonly colocated through VMs or containers. These techniques can be combined with our fine-granularity characterization for determining ideal application placement.

3 Motivation and Key Insights

For each challenge described in Section 1, we provide a detailed example and insights to overcome each challenge.

3.1 Program Behavior

We first examine PENNANT, an MPI application from CORAL-2 with a mixture of memory-bound and compute-bound functions. We measure the compute-boundedness using detailed performance counters and offline analysis and demonstrate with a code analysis in Figure 1 how separating phases at the communication call granularity (as done in contemporary solutions [7]) can lead to subpar results.

In the main loop, each iteration calculates the timestep (calcGlobalDt) followed by a series of computations (doCycle). An MPI_Allreduce call is made in this outer loop through calcGlobalDt, as is common in many HPC applications. In doCycle, multiple user functions are called, with the most time-consuming ones shown in Figure 1. The only other MPI call in the main loop is made in the middle of doCycle, in

```
// Driver.cc
while (cycle < cstop && time < tstop) { ...
  calcGlobalDt();     // **calls MPI_Allreduce**
  hydro->doCycle(dt); // memory bound (~38ms each)
... }

// Hydro.cc
void Hydro::doCycle(...) { ...
  for (int sch = 0; sch < numsch; ++sch) {
    ... // computation
    qcs->calcForce(...); // 20% runtime,
  } ...                  // partial memory bound
  mesh->sumToPoints(...); // **calls MPI_Waitall**
... }

//QCS.cc
void QCS::calcForce(...) { ...
  setCornerDiv(...);     // 10% runtime,
... }                    // compute-bound
```

Figure 1: PENNANT [12] source code excerpt.

sumToPoints. Therefore, any DVFS approach that separates phases at communication calls will consider much of the computation that occurs within doCycle to be of the same phase. Importantly, we measure setCornerDiv as compute-bound and consuming 10% of the runtime, whereas the other user functions are memory-bound. The functions within the given for loop contribute to 45% of the total runtime.

If this for loop within doCycle is considered a single phase, then it will be classified as partially memory-bound and run at a low frequency. For instance, using the numbers above, the single phase could be considered 22% compute-bound. This will cause setCornerDiv to run at a much lower than ideal frequency, resulting in performance loss (measured as execution time), while the other functions run at a higher than ideal frequency, resulting in power waste. This can be avoided by appropriately adapting to the program behavior, such as at the finer granularity of functions.

3.2 Performance Counters

The discrepancy and inaccuracy among counters in different processors is a major problem within the industry - identified both in the past [35] and also more recently [36]. A concrete example shall explain the situation, based on experience developing industry-grade performance tools (CrayPat [9]). On Intel Broadwell, Skylake, and Icelake processors, memory bandwidth utilization had to be (loosely) approximated as there was no event to report L3 writebacks (only L3 misses was reported). On the latest Intel Sapphire Rapids (abbreviated SPR), there is a new counter for total writes to memory (OCR:WRITE_ESTIMATE_MEMORY), which is helpful as L3_Writebacks need not be approximated. But, since SPR uses

chiplets, total read bandwidth is the sum of reads to the local chiplet (OCR:READS_TO_CORE_LOCAL_SOCKET_DRAM) and remote chiplets (OCR:READS_TO_CORE_REMOTE_-DRAM). However, these 2 counters both report the same values, leading to inaccurate bandwidth reporting. This inaccuracy exists for various other counters.

Memory bandwidth utilization has been even more difficult to compute on AMD Rome, Milan, and now Genoa. On Genoa, for instance, ANY_DATA_CACHE_FILLS_FROM_-SYSTEM:DRAM_IO_NEAR:DRAM_IO_FAR and DEMAND_-DATA_CACHE_FILLS_FROM_SYSTEM:DRAM_IO_NEAR:-DRAM_IO_FAR both report the same values that seriously under-reports memory bandwidth, which is crucial for accurately estimating compute/memory boundedness.

Thus, while counters are necessary for characterization, careful consideration must be made to account for the diversity in accuracy and availability of hardware counters. In fact, we can circumvent the issue of portability by relying on a 'common subset of equivalent counters' [53]. We find that the retired instructions counter meets these requirements, and are among the 3 fixed counters on all Intel processors [21] (the other 2 are typically core clocks and reference clocks). Moreover, instructions per cycle (IPC) is commonly used to describe performance and can be useful in classifying a phase's compute/memory boundedness.

3.3 Communication Phases

Parallel applications can spend a considerable amount of time during communication over the network, which is commonly split into slack time (time spent waiting to send/receive data) and copy time (time spent sending/receiving data, which can involve small computation e.g., reduction) [6, 45]. This opens up a useful opportunity for power savings, especially when slack time is entirely spent busy waiting which can lead to the phase being classified as compute-bound and thus made to (incorrectly) run at the highest clock. On the other hand, copy time is often compute-intensive (e.g., as in MPI_Allreduce) and introduces performance overhead when also run at the lowest clock.

For example, MPI_Wait is a blocking call within the popular MPI communication library that waits for a corresponding nonblocking Send or Receive request to complete and consists entirely of slack time and thus is a prime candidate for reducing clock frequency. We observe that simply lowering core clock upon seeing an MPI_Wait routine improves energy efficiency in some cases but hurts performance up to 2x in other cases.

The latter scenario is observed for `specfem3D` [56], a compute bound application with communication calls that have a small payload size (avg. 189 bytes). This leads to an average of 5 μs spent per wait call, which is much faster than the frequency transition latency (we observe on the order of tens of μs). As a result, execution quickly exits the communication calls and enters user code while the core clock is still being lowered. Moreover, these short calls are made very often (avg. 10 wait calls per ms). In fact, this is a general trend we observe across MPI applications, that there are either very many short MPI calls or much fewer long MPI calls. Therefore, the compute bound user code in this scenario ends up running at a much lower clock effectively, leading to the observed 2x performance degradation.

Thus, one solution could be to observe/predict time per MPI call and only lower frequency when the time per MPI call is large enough as done in [5, 52]. However, this can induce performance loss given that some communication calls have considerable copy time, so an additional solution is to simply raise core clock for these calls [6]. However, this can lead to an undesirable result of overprovisioning power, if for instance the surrounding user routines are memory bound. We find this to be the case for `PENNANT` [12]. For each main loop iteration, PENNANT makes a call to MPI_Allreduce, as shown earlier in Figure 1. We find that these Allreduce calls have small payload sizes (avg. 11 bytes) leading to short time spent per call (avg. 21 μs) and are called often (avg. 4 times per sec). Since the surrounding user code is overall memory bound, attempting to raise the clock for Allreduce leads to power waste (we observe 6% less savings) where the subsequent user code runs at a higher clock than necessary.

Overall, the variety of different communication calls combined with variety in payload sizes (and thus time spent) associated with those calls makes it difficult to exploit this useful opportunity. Moreover, any decision for communication phases should also account for the surrounding user code to maximize power/energy savings, which is practical when there are different phases, detailed in Section 5.

4 Application Characterization

We discuss our approach to a quantitative characterization of an application phase as compute or memory bound using a single easily-available and accurate performance metric.

4.1 Core Frequency and Application Performance

In Figure 2, we observe an example demonstrating that (fully) memory bound applications are oblivious to changes in core frequency, whereas (fully) compute bound applications witness performance improvement proportional to increase in core frequency. We find that a simple test is useful and effective for inferring compute/memory boundedness, by comparing the performance of an application at two different clock frequencies. For example, if performance decreases at

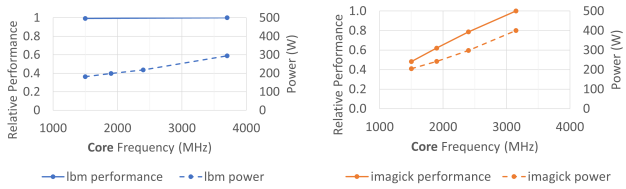

Figure 2: Core frequency vs. power and performance of a memory bound (lbm, left) and a compute bound (imagick, right) application on AMD Genoa.

Figure 3: Uncore frequency vs. power and performance of a memory bound (lbm, left) and a compute bound (imagick, right) application on Intel SPR. Core frequency set to 2.5 GHz.

the same ratio as the frequency, the application is considered 100% compute-bound (and 0% memory-bound), and if the performance does not change at all with frequency, it is then 100% memory-bound (and 0% compute-bound).

To measure performance, IPC is a well-known performance metric. Also, as discussed in Section 3, it is easily-available across processors and accurate. But, only looking at raw IPC and establishing thresholds for compute/memory boundedness is not enough, because IPC values/thresholds change across platforms/architectures and moreover IPC is affected by frequency. For example, lowering frequency for a memory-bound application can increase overall IPC because less cycles are needed to complete the same number of instructions. This is especially detrimental in a dynamic management approach. One way to avoid the dependency on clock frequency while still capturing performance is to instead use IPS, or instructions per second. Therefore, we calculate compute-boundedness (CB) and memory-boundedness (MB) using the IPS measured at two specific frequencies, high and low:

$$\%CB = 100\% * \frac{\frac{IPS_{high}}{IPS_{low}} - 1}{\frac{Freq_{high}}{Freq_{low}} - 1}, 0\% \leq \%CB \leq 100\% \quad (1)$$

$$\%MB = 100\% - \%CB \quad (2)$$

Importantly, we only need to measure the IPS at the maximum frequency and one lower frequency. Moreover, Everest should not use a frequency that is too low or all phases will appear compute-bound (which is overly conservative). We find that maintaining a frequency ratio of about 1.2 is sufficient (depending on available frequency steps), while also minimizing overhead. By default, this often amounts to using the highest non-turbo frequency for $Freq_{low}$. Also note that this covers the case where the maximum turbo frequency is not achievable (e.g., when all cores are active) and a lower $Freq_{low}$ may thus be necessary.

With %MB and %CB, we predict the relative performance at any (lower) frequency as follows:

$$\frac{Time_{low}}{Time_{high}} = \%CB * (\frac{Freq_{high}}{Freq_{low}} - 1) + 1$$

To allow the user to still control application performance, we introduce a performance degradation variable (PD) that defines a target performance to meet while maximizing power savings. PD is relative to performance at peak clock frequency. For example, 5% PD means that we should provide at least 95% relative performance. We substitute PD into the above formula to calculate the ideal frequency:

$$\frac{Time_{high}}{Time_{low}} = 1 - PD$$

$$Freq_{ideal} = \frac{Freq_{high}}{1 + \frac{PD}{\%CB(1-PD)}} \quad (3)$$

The ideal frequency is thus a function of %CB (%MB) and the acceptable performance degradation, scaled by the maximum possible frequency. This is especially useful because there are individual application phases that are neither 100% compute or 100% memory bound. In such scenarios, additional power savings could be made if some performance could be sacrificed. This makes it crucial for both the user to define and for the tool to accommodate an acceptable performance loss.

4.2 Uncore Frequency and Application Performance

Uncore frequency scaling (UFS) has been supported on Intel processors since Haswell [18]. We recognize that compute-bound applications are less likely to have L3 cache accesses and thus do not benefit from a high uncore frequency. Therefore, we use our compute-bound characterization to additionally make uncore frequency decisions on our Intel platform. Specifically, to calculate uncore frequencies, we still use Equation 3 except substituting %MB for %CB. For instance, a 0% MB phase can run at the lowest uncore frequency with minimal performance loss, whereas a 100% MB phase should run at the highest uncore frequency for no performance loss, as indicated by Figure 3.

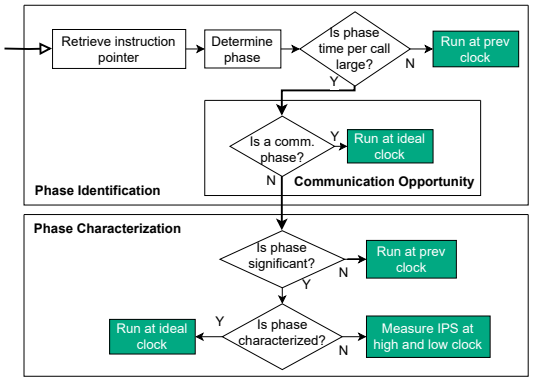

Figure 4: Everest loop flowchart.

5 Implementation

Using the characterization and additional energy-saving techniques outlined above, we propose Everest, a cross-platform runtime tool that saves power and energy while meeting a desired performance target.

In HPC, applications are typically launched as jobs through the job scheduler or workload manager, with exclusive allocation of one or multiple nodes. In this scenario, Everest operates as an external (to the application) standalone process that performs both dynamic application characterization and consequently effects appropriate frequency changes. It is important to note that only a single Everest process is launched on each allocated node and is responsible for observing the job only for that node (assuming exclusive allocation, i.e., one job per node). Each Everest process thus makes DVFS decisions independent of other nodes allocated to the job (i.e., without any global synchronization). This is made possible within Everest given its ability to observe time spent within communication routines on each node.

We integrate Everest with the job scheduler which automates the above mentioned launch of one Everest process per node of each job. This integration also directly provides the needed permissions for each launched Everest process to change clock frequencies. For our experiments, we integrate Everest with Slurm using the SPANK plugin architecture. SPANK is a low cost, low effort, generic interface that can be used to dynamically modify the runtime behavior of Slurm job launch [32]. Similar methods are available for other job schedulers (e.g., hooks for PBSPro, prolog/epilog). In particular, at job launch, our plugin determines which nodes are allocated and executes a single instance of Everest for each of those nodes. From a user's perspective, the user only needs to specify two additional parameters in addition to their normal job script: whether to enable Everest and the desired PD. For example, a user desiring to use Everest with 10% PD would submit:

```
srun -use-everest -everest-pd=10 <slurm-options> <binary>
```

Upon launching, Everest monitors a single application process on the same node and periodically loops through three stages: phase identification, phase characterization, and frequency scaling. Figure 4 presents these stages as a flowchart, which we expand on here.

5.1 Phase Identification

Everest first needs to identify application phases. We define a phase as any section of contiguous memory addresses (instructions) being executed within the application. This means a phase can be a single instruction at the smallest granularity, but this is not feasible for measuring IPS nor to perform frequency scaling. Therefore, a phase will ideally be at the smallest granularity that is still sufficiently large to both measure its IPS and make use of DVFS. From Section 3.1, we note functions to be a useful granularity. In practice, we observe that most functions consist of a single loop or loop nests where behavior remains stable. Thus, we find that using functions as phases is both practical and meets our design considerations. Moreover, our approach is still applicable at other phase granularities, assuming the above considerations are satisfied.

The instruction pointer presents a simple method for determining the current phase. For instance, we use the instruction pointer to determine which function we are in (at the leaf level, i.e., bottom of the callstack) by using information from the application's ELF binary. Even when this information is unavailable (as in stripped binaries), we still use the instruction pointer to obtain a distribution of memory addresses, which is then used to delineate phases. Everest obtains the instruction pointer by periodically sampling (every 50 us) using the Linux perf_event interface. This is similar to how the popular Linux profiling tool perf determines hot spots when sampling performance counters, which we compare to in our evaluation in Section 7.1.

Furthermore, we recognize that some phases may still end up being too short, i.e., have a small time per call (e.g., 100 μs) relative to the frequency transition latency. In such scenarios, we maintain the clock frequency of the previous phase until a new phase is entered.

5.1.1 Communication Opportunity. While the instruction pointer allows Everest to also identify MPI functions, the many different MPI implementations introduce complications in identifying the desired functions. For instance, MPI_Wait in Cray MPICH makes calls to hundreds of internal functions. Thus, we instead identify communication phases using the MPI standard profiling interface (PMPI [39]). This allows Everest to intercept MPI calls (upon entry and exit) and track payload sizes and thus target MPI functions that we expect to incur significant slack time or copy time. This includes the MPI_Wait function that is associated with non-blocking

point-to-point communication routines such as MPI_Isend and MPI_Irecv, and other blocking point-to-point and collective operations. If the payload size is large, indicating sufficient time spent in slack, we then run at the minimum frequency. For instance, we find that a payload size of 10KB corresponds to about 100 μs per call. Otherwise if the payload size is small, we keep the clock frequency of the previous compute/memory bound phase upon entering an MPI call, and adjust frequency according to the next phase upon exiting an MPI call. Similarly, for functions which can have significant copy time such as MPI_Allreduce, we run at the maximum frequency only when the payload size is large enough and otherwise maintain frequency of the prior phase.

Moreover, the use of subcommunicators is another potential source of performance loss as it allows for different processes to be assigned different tasks. This can result in processes on the same node to be in different phases, and an ideal frequency for one process will differ for another process. Everest mitigates these losses by conservatively running at the highest clock when subcommunicators are utilized.

5.2 Phase Characterization

Next, Everest must characterize the individual phases. We recognize that characterizing every single phase encountered will incur additional overhead. Thus, we require that a phase be significant before characterizing it - the phase must account for at least a certain percentage (e.g., 1%) of the application's current run time. Otherwise, we run at the highest clock permitted by PD (i.e., run at ideal clock assuming 100% CB). While this is conservative and avoids any potential loss, our offline analysis reveals that these functions are very often compute-bound.

If a phase is significant and has not been characterized, Everest measures IPS as required by Equation 1 to calculate %CB, which has two major steps. First, Everest raises the core and uncore frequency to the maximum and samples IPS for 100ms (configurable). Second, Everest lowers the core frequency to the specified low frequency and again samples IPS for 100ms. IPS is measured through the perf_event interface, which allows Everest to sample the RETIRED_INSTRUCTIONS counter at the shortest sampling period of 100,000 and also obtain an instruction pointer and timestamp associated with each sample. Everest then assigns each IPS sample to the phase matching the address, which may include phases in addition to the uncharacterized phase that we are interested in. Additionally, an uncharacterized phase might only be missing IPS_{high} or IPS_{low} but not both, so one of the two sampling steps can be skipped.

If a significant phase has already been classified, we reuse the previous (stored) characterization and thus reduce overhead from repeated characterization. However, we recognize that resampling functions may be necessary, especially for long-running applications, as their behavior may change. By default, we choose to resample functions every 60 seconds; this interval can be optionally configured.

5.3 Frequency Scaling

Finally, Everest performs DVFS for the current phase, represented by the green boxes in Figure 4.

If we are in an already characterized significant phase, the ideal core frequency is calculated using Equation 3, but the calculated frequency may not be an available frequency option. If frequencies are available at a fine granularity (e.g., 100MHz steps), we simply round up to the next frequency step. However, if available frequencies are at a coarser granularity, simply rounding up can diminish power and energy savings. In this case, we choose to emulate the ideal frequency by running a fraction of time at the next lowest and next highest frequency, at a timing ratio that results in the desired frequency if calculated using a weighted average.

The actual frequency scaling is done using the `cpufreq` driver for core frequency and the `wrmsr` utility (to write to a Model Specific Register or MSR) for uncore frequency.

Importantly, changing frequencies does not happen instantly and we must take into account switching latency. According to the specifications on our systems, Intel SPR has a 10 μs transition latency and AMD Genoa has an 8 μs transition latency. We measure our evaluation platforms to need about 50 μs for the new frequency to take into effect.

5.4 Overhead

Since Everest is using multiple interfaces and interacting with the OS in different ways, it is important to measure the amount of overhead involved. We perform a holistic measurement of overhead by running a version of Everest that only profiles and characterizes application phases (i.e., we are not doing any 'Run at X clock' decisions in Figure 4), on a single node. We find that the overhead on average causes a decrease in performance by 1%, increase in power by 1%, and increase in energy by 2%, with fully compute-bound applications impacted the most. Additionally, since each instance of Everest only characterizes the application locally, scaling to a larger number of nodes does not imply increased overhead from the use of Everest on all nodes.

Furthermore, since we are dealing with frequent clock frequency changes, we are interested in the overhead resulting from the worst case scenario of constantly switching between two frequencies, compared to only performing that switch once. For this study, we use `exchange2` from SPEC CPU2017 since it is overwhelmingly compute bound, relatively long (>200s), and expected to have stable behavior (90% runtime within the same function). We thus measure

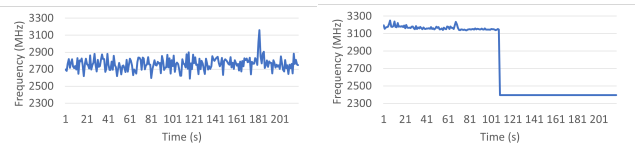

Figure 5: Core frequency for exchange2 when switching every 100μs (left) vs. once (right) on AMD Genoa.

exchange2 when attempting to switch between turbo (3.2 GHz) and 2.4 GHz every 100 μs compared to performing the switch once halfway into the runtime (Figure 5). Note that this experiment is without using Everest and does not include the sampling overhead measured in the previous experiment. We find that this switching overhead only causes a decrease in performance by <1%, increase in power by <2%, and increase in energy by <3%. Moreover, we are still able to establish the expected effective frequency of 2.8 GHz at this switching granularity.

6 Experimental Setup

We evaluate Everest on two server platforms with the latest Intel and AMD processors. The two platforms used are different not only in processor type, but also OS, size, and power measurement methodology. The setup tests Everest's effectiveness in traversing these differences.

The Intel system has 1 node, with a dual socket Xeon Platinum 8470 (SPR) with 52-cores and 8-channels of DDR5 per socket. The system uses SLES 15 SP5. SPR allows the user to set the core frequency in 100 MHz increments from 800 MHz to 3 GHz, and also 3.8 GHz (turbo). Uncore frequency is set in 100MHz increments from 800MHz to 2.5GHz. Power measurements are obtained through reading RAPL MSRs.

The AMD system consists of 8 nodes, each with a dual socket EPYC 9654 (Genoa) with 96-cores and 16-channels of DDR5 per socket. The system uses Cray OS (2.4.96) and Slingshot 11 high-speed network, with 2 Cassini 200GB NICs per node. Genoa allows the user to set the frequency to 3.7 GHz (turbo), 2.4 GHz, 1.9 GHz, and 1.5 GHz. Power measurements are obtained using PM counters available on Cray EX systems [34] and collect real power and energy data, since RAPL MSR data is only modeled in AMD [10].

We first evaluate our approach on the SPEC CPU 2017 benchmarks [50], running in 'rate' mode and using the 'reference' input set for full socket utilization. Because SPEC CPU demonstrates a range of memory and compute bound behavior, we are able to evaluate Everest's ability to both accurately characterize and adapt to different application phases. We exclude blender on AMD due to compilation issues. We then test a number of MPI applications chosen from the CORAL-2 [29] and ECP [43] suites, which are commonly used within HPC (Table 2) and allow us to evaluate how Everest exploits communication phases and scales across multiple nodes. All applications are compiled with CCE v11.0 (Cray compiler), which includes Cray MPICH.

Benchmark	Size
HPCG [19]	104^3
specfem3D [56]	NSTEP=800000, DT=0.05
SNAP [28]	nx=192, ny=228, nz=80
PENNANT [12]	sedovflat w/ nzx=7680, nzy=8640, dtinit=1.e-7
Nekbone [30]	iel0=750, ielN=750, nx0=16, nxN=16
miniAMR [33]	max_blocks=4000, num_objects=2, num_tsteps=100

Table 2: MPI Applications

We evaluate Everest with PD set to 5%, 10%, and 20%, with the specific configuration indicated as Everest ([PD]%). For the MPI applications, we compare to the current state-of-the-art Energy Aware Runtime (EAR) [7]. EAR is currently in use in SuperMUC-NG [31] (which sits at number 64 in the TOP500) and also utilizes DVFS for phase-aware energy savings. While their code is open source, their solution requires a system-wide installation, which was not feasible for our testing. We instead implemented EAR based on their published work and available code for our experiments.

We also compare results with the Linux 'ondemand' governor, disabling turbo boost, and a static oracle (Intel only). The ondemand governor is provided by the OS as a way to scale CPU frequency with CPU load, whereas disabling turbo boost is an easy measure to boost power savings. These simple baselines illustrate what can be achieved by any user today. Since Intel provides finer-grained frequency steps, we also evaluate an oracle that runs at the lowest static (i.e., unchanging) core and uncore frequencies (determined manually) that meet the performance target.

7 Results and Discussion

We first present results evaluating Everest with SPEC CPU 2017 benchmarks on a single node for both AMD and Intel platforms. We then evaluate Everest with various MPI applications from the CORAL-2 and ECP suites running on the full 8-node AMD system, in comparison to EAR. All results are relative to the Linux performance governor, or equivalently running at max (turbo enabled) frequency.

7.1 SPEC CPU 2017

AMD. Figure 6 compares the relative performance, power, and energy of Everest (5%), (10%), and (20%) on 1 node from the AMD system, with colored reference lines corresponding to the PD of the same-colored bar. The bottom graph of the figure compares the percentage energy to percentage performance ratio, indicating when the energy reduction is more than the performance reduction (i.e., when the value is <1) and vice versa. The applications are ordered by increasing overall compute-boundedness from left to right, as indicated

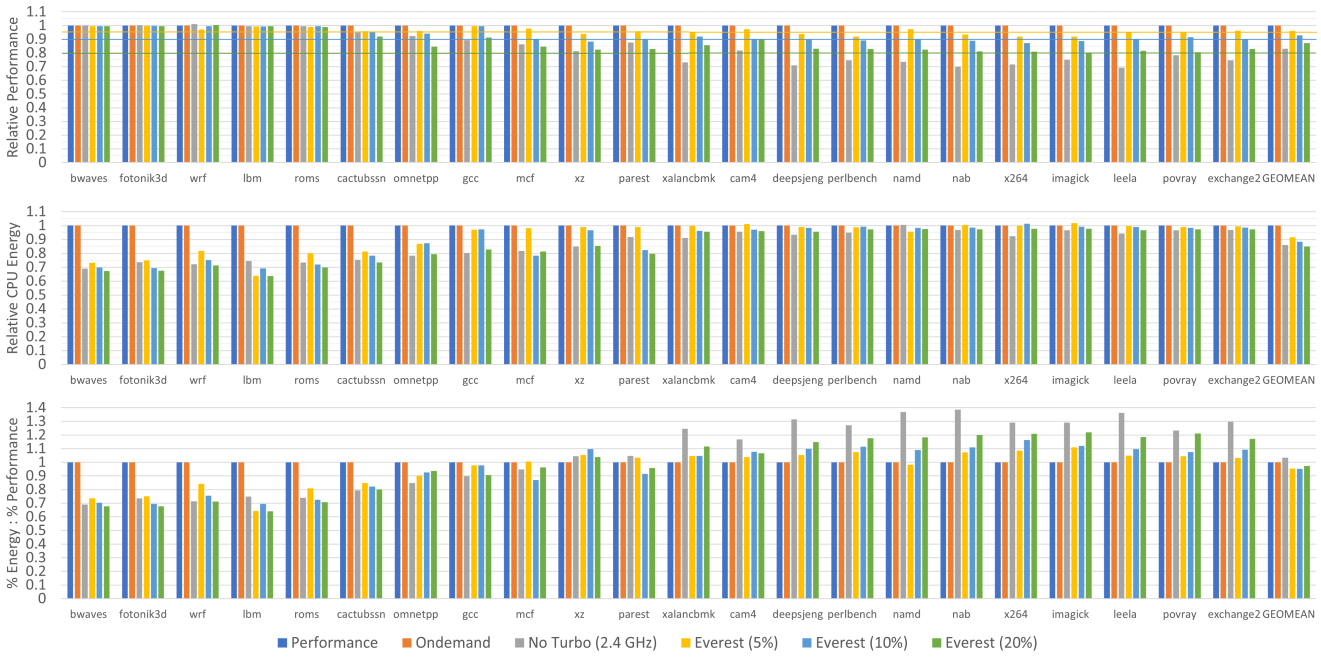

Figure 6: Relative performance and energy comparing baselines (ondemand, no turbo) to Everest on AMD Genoa. The bottom graph shows % energy to % performance ratio, where smaller (<1) is better.

in Table 3. Table 3 additionally compares Everest's phase profiling and characterization to the actual observed values measured using Linux perf and offline analysis.

	Overall	Most time-consuming function			
	%CB	%Runtime		%CB	
App	Perf	Perf	Everest	Perf	Everest
bwaves	0%	61%	70%	0%	0%
fotonik3d	0%	27%	27%	0%	11%
wrf	0%	7%	5%	0%	7%
lbm	1%	99%	95%	0%	0%
roms	1%	35%	49%	0%	0%
cactubssn	10%	40%	37%	9%	0%
omnetpp	16%	19%	14%	63%	64%
gcc	24%	3%	5%	92%	100%
mcf	33%	65%	87%	0%	10%
xz	44%	75%	79%	67%	61%
parest	57%	31%	22%	70%	71%
xalancbmk	72%	42%	33%	98%	100%
cam4	72%	9%	5%	47%	44%
deepsjeng	93%	15%	13%	99%	100%
perlbench	96%	42%	26%	100%	100%
namd	100%	12%	14%	100%	93%
nab	100%	71%	67%	100%	100%
x264	100%	19%	17%	100%	100%
imagick	100%	41%	39%	100%	100%
leela	100%	67%	65%	100%	100%
povray	100%	15%	16%	100%	96%
exchange2	100%	93%	91%	100%	100%

Table 3: SPEC 2017 most time-consuming function %runtime and %CB, for perf vs. Everest.

As expected, the overwhelmingly memory bound applications (`bwaves` to `roms`) receive minimal performance loss when lowering core frequency and save about 30% energy regardless of PD. The overwhelmingly compute bound applications (`namd` to `exchange2`) are easily characterized by Everest and lowered to the appropriate core frequency to meet desired performance within 3% while also saving power.

For the mixed compute and memory bound applications in the middle, Everest's phase awareness allows for energy savings (i.e., accounting for the memory bound phases) while still meeting the performance guarantee (i.e., accounting for the compute bound phases). For example, compared to the fully compute bound applications, these applications achieve on average similar performance while also saving 4%, 9%, and 14% energy for Everest PD = 5%, 10%, and 20%, respectively.

On the other hand, no turbo varies wildly in performance (losing up to 35% performance). Thus, despite the potential power/energy savings that can be made, users disabling turbo must also be willing to accept much more performance loss, especially if the nature of the application is unknown. This is reflected by the energy to performance ratio, where a desirable ratio is achieved for memory bound applications but is unacceptable for compute bound applications. Unfortunately, the ondemand governor is not helpful as it scales core frequency to the maximum, given the application load.

Lastly, by incorporating PD, Everest provides the flexibility to make additional power and energy savings. As seen from

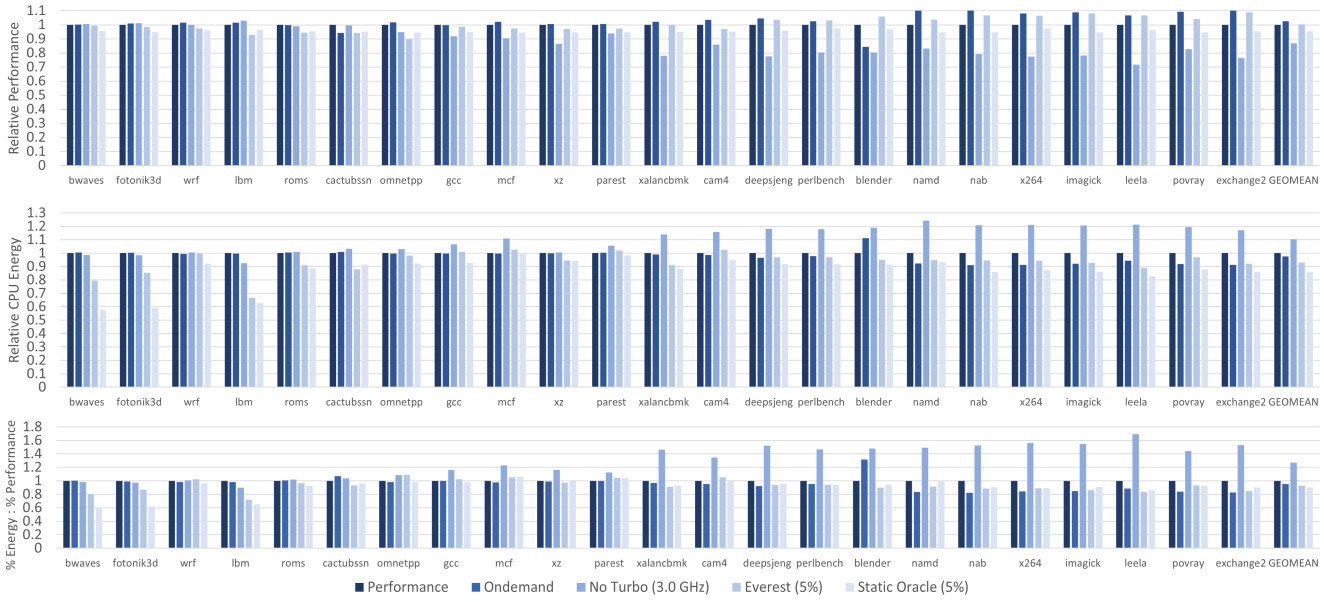

Figure 7: Relative performance and energy comparing baselines (ondemand, no turbo, static oracle) to Everest on Intel SPR. The bottom graph shows % energy to % performance ratio, where smaller (<1) is better.

the geomean, a larger PD is consistent with more power and energy savings on average, but potentially diminishing returns as indicated by the energy to performance ratio.

Intel. While on AMD the ondemand governor behaves exactly like the performance governor, it differs on Intel due to the additional uncore frequency scaling knob. Since the uncore is underutilized during compute-bound applications, the ondemand governor is able to lower the uncore frequency and essentially redistribute the power to the core, improving performance. Although Everest optimizes for minimal power consumption, we demonstrate that Everest too captures these performance gains at low PD, e.g. 5% PD (Figure 7).

For the memory bound applications, Everest achieves 10-20% energy savings with about 5% performance loss. At most, we see a 7% performance loss with `lbm`, accompanied by 33% energy and 38% power savings. Importantly, both naive strategies do not detect these available savings. We would have expected the no turbo strategy to save more energy/power given that these applications are memory-bound, but we find that the primary reason for this disparity is from uncore power consumption, which the oracle accounts for.

On the other hand, Everest is able to save 5-10% energy by *improving* performance in the compute-bound applications. This performance improvement comes from again reducing the uncore frequency. Lowering the uncore power contribution means the processor is no longer at the TDP, allowing CPU frequency to scale up and resulting in faster core computation and better performance. We see that the ondemand governor also improves in performance and energy for the compute-bound applications for the same reasons (lowering uncore frequency). Lastly, the no turbo strategy understandably reduces performance by over 20% for compute-bound applications with only about 5% power savings (no uncore frequency adjustments), so energy also increases.

For applications in the middle, Everest achieves results mostly on par with the baseline and ondemand, with about the same energy and <3% performance loss. We believe these lack of savings are again due to the uncore. For example, for `mcf`, the static oracle reveals that both core and uncore frequency must remain relatively high to achieve the desired performance. However, Everest still improves over ondemand and the baseline (and achieve close to the oracle) for mixed applications `xz` and `xalancbmk`, which see 5% and 9% energy savings respectively. Again, no turbo does the worst overall, but with results exacerbated by the uncore.

7.2 HPC

Figure 8 compares Everest to EAR for MPI applications on our AMD system (8 nodes, **1536 ranks**). Everest achieves on average 8% better energy savings compared to EAR.

The key difference between EAR and Everest is the granularity of phases. Since EAR defines phases at the communication call granularity (which often only occur in the outer loop), EAR misses phases that occur between communication calls, as discussed in Section 3. Moreover, Everest's awareness of slack within communication phases largely enhances its ability to save energy, and Everest limits performance

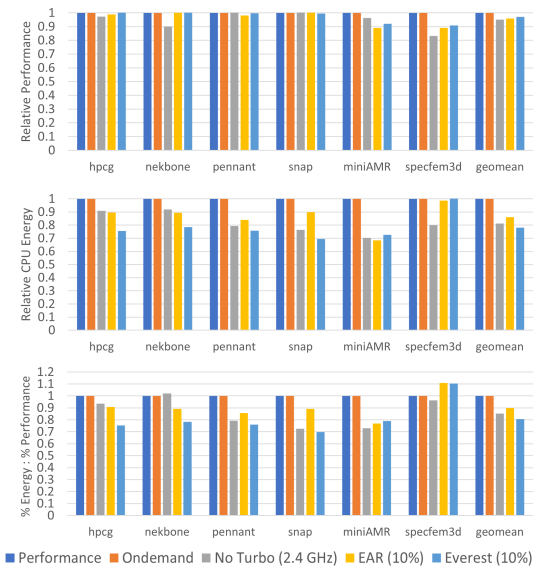

Figure 8: Relative performance, energy, and energy-performance ratio for MPI applications with EAR vs. Everest on AMD Genoa.

loss by accounting for time per call. We find that multiple applications spend 10-20% runtime within MPI; thus lowering frequency during slack time can be quite significant.

HPCG and SNAP are memory-bound and both Everest and EAR decide to lower core frequency. However, a significant portion of runtime is spent within communication, which EAR classifies as compute-bound and thus does not recognize the potential savings, whereas Everest identifies the large send/recv payload size (>20KB) and lowers frequency during the long blocking wait calls. Everest keeps frequency low even for SNAP despite its short nonblocking waits, given the surrounding memory-bound code. As a result, EAR only saves 11% energy compared to Everest's 25% energy savings.

PENNANT, Nekbone, and miniAMR have both compute and memory bound phases, so detecting these phases are again important for maximizing savings. For example, Everest gets better savings (24%) on PENNANT compared to EAR's 18% energy savings by accounting for finer-grained phases (i.e., that occur within the doCycle function which dominates the outer loop) in addition to the communication phases. Specifically, Everest identifies 6 significant phases corresponding to the functions doCycle (0% CB), setCornerDiv (84% CB), calcForce (54% CB), calcCtrs (8% CB), calcVols (5% CB), and sumToPoints (0% CB). This aligns with the offline analysis made in Section 3.1. Furthermore, Everest lowers frequency during the long MPI wait calls (also in Nekbone especially) and maintains the low frequency from surrounding memory bound user code for MPI_Allreduce.

Lastly, specfem3D is an overall compute-bound application with about 20% runtime spent in slack. As noted previously, payload sizes within specfem3D are very small leading to minimal time spent in each wait call, meaning significant performance loss will occur if frequency is lowered here, shown by no turbo. Everest recognizes this fact and thus is able to meet the performance guarantee in this case. While EAR does not have to worry about this and thus achieves the same results as Everest, this again comes at the detriment of losing out on savings in the other applications.

Overall, Everest improves considerably over EAR by exploiting opportunities from communication phases in addition to making finer-grained phase characterizations, leading to 9% better energy to performance ratio on average.

Communication scale. As applications scale, the communication expectations such as time per call may vary, potentially affecting Everest. As such, we analyzed how time per wait call changes for each application at 1, 2, 4, and 8 nodes. When problem size remains the same, payload size and time per call decrease in tandem as node count increases, with at most specfem3D seeing up to 75% reduction for both. When problem size also scales with node count, the time per call remains about the same. Therefore, despite potential changes in communication patterns at scale, we expect Everest to exhibit low sensitivity to these changes and still make use of its time per call estimates to achieve similar results.

Load imbalance. For parallel applications, load imbalance is a critical performance bottleneck. We demonstrate that Everest does not worsen load imbalance through an additional evaluation of miniAMR [33]. miniAMR automatically performs load balancing across ranks at each refinement step, but this can be disabled (which induces 71% performance loss by default). For load-imbalanced miniAMR, we measure that Everest (10%) appropriately lowers performance by 4%, for 8% power and 3% energy savings. Thus, Everest still ensures the performance target, relative to the default run (i.e., with load balancing disabled in this case).

8 Conclusion

We identify three challenges towards a runtime energy saving solution - effective adaptation to program phases for improved efficiency, versatility to operate across processors, and flexibility to handle parallel applications involving communication. We show that Everest, our proposed tool, addresses these challenges. Everest achieves versatility by relying on a single easily available (across processors) and accurate performance metric for finding energy saving opportunities. Owing to fine-granularity phase-awareness and accurate computation of MPI slack time, Everest finds enhanced energy saving opportunities and thus saves up to 20% more energy than existing solutions.

References

[1] Bilge Acun, Kavitha Chandrasekar, and Laxmikant V. Kale. 2019. Fine-Grained Energy Efficiency Using Per-Core DVFS with an Adaptive Runtime System. In *2019 Tenth International Green and Sustainable Computing Conference (IGSC)*. 1–8. doi:10.1109/IGSC48788.2019.8957174

[2] Changmin Ahn, Seungyul Lee, Chanseok Kang, and Bernhard Egger. 2020. NuPow: Managing Power on NUMA Multiprocessors with Domain-Level Voltage and Frequency Control. In *Economics of Grids, Clouds, Systems, and Services*, Karim Djemame, Jörn Altmann, José Ángel Bañares, Orna Agmon Ben-Yehuda, Vlado Stankovski, and Bruno Tuffin (Eds.). Springer International Publishing, Cham, 126–141.

[3] Sridutt Bhalachandra, Allan Porterfield, Stephen L. Olivier, and Jan F. Prins. 2017. An Adaptive Core-Specific Runtime for Energy Efficiency. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 947–956. doi:10.1109/IPDPS.2017.114

[4] W.L. Bircher, M. Valluri, J. Law, and L.K. John. 2005. Runtime identification of microprocessor energy saving opportunities. In *ISLPED '05. Proceedings of the 2005 International Symposium on Low Power Electronics and Design, 2005.* 275–280. doi:10.1145/1077603.1077668

[5] Daniele Cesarini, Andrea Bartolini, Piero Bonfà, Carlo Cavazzoni, and Luca Benini. 2018. COUNTDOWN: a run-time library for application-agnostic energy saving in MPI communication primitives. In *Proceedings of the 2nd Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems* (Limassol, Cyprus) *(ANDARE '18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. doi:10.1145/3295816.3295818

[6] Daniele Cesarini, Andrea Bartolini, Andrea Borghesi, Carlo Cavazzoni, Mathieu Luisier, and Luca Benini. 2020. Countdown Slack: A Run-Time Library to Reduce Energy Footprint in Large-Scale MPI Applications. *IEEE Transactions on Parallel and Distributed Systems* 31, 11 (2020), 2696–2709. doi:10.1109/TPDS.2020.3000418

[7] Julita Corbalan, Lluis Alonso, Jordi Aneas, and Luigi Brochard. 2020. Energy Optimization and Analysis with EAR. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 464–472. doi:10.1109/CLUSTER49012.2020.00067

[8] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. 2011. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing* (Karlsruhe, Germany) *(ICAC '11)*. Association for Computing Machinery, New York, NY, USA, 31–40. doi:10.1145/1998582.1998590

[9] Luiz DeRose, Bill Homer, Dean Johnson, Steve Kaufmann, and Heidi Poxon. 2008. Cray Performance Analysis Tools. In *Tools for High Performance Computing*, Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 191–199.

[10] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. 2016. A Validation of DRAM RAPL Power Measurements. In *Proceedings of the Second International Symposium on Memory Systems* (Alexandria, VA, USA) *(MEMSYS '16)*. Association for Computing Machinery, New York, NY, USA, 455–470. doi:10.1145/2989081.2989088

[11] Jonathan Eastep, Steve Sylvester, Christopher Cantalupo, Brad Geltz, Federico Ardanaz, Asma Al-Rawi, Kelly Livingston, Fuat Keceli, Matthias Maiterth, and Siddhartha Jana. 2017. Global Extensible Open Power Manager: A Vehicle for HPC Community Collaboration on Co-Designed Energy Management Solutions. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes (Eds.). Springer International Publishing, Cham, 394–412.

[12] Charles Ferenbaugh. [n. d.]. PENNANT. https://github.com/lanl/PENNANT. (accessed May 25, 2023).

[13] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. 2020. SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers. arXiv:2001.02505 [cs.DC]

[14] Vincent W. Freeh and David K. Lowenthal. 2005. Using Multiple Energy Gears in MPI Programs on a Power-Scalable Cluster. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 164–173. doi:10.1145/1065944.1065967

[15] Rong Ge, Xizhou Feng, Wu-chun Feng, and Kirk W. Cameron. 2007. CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters. In *Proceedings of the 2007 International Conference on Parallel Processing (ICPP '07)*. IEEE Computer Society, USA, 18. doi:10.1109/ICPP.2007.29

[16] Soraya Ghiasi. 2004. *Asymmetric Multi-core Design for Dynamic Thermal Management*. Ph. D. Dissertation. University of Colorado at Boulder.

[17] Neha Gholkar, Frank Mueller, and Barry Rountree. 2019. Uncore Power Scavenger: A Runtime for Uncore Power Conservation on HPC Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 27, 23 pages. doi:10.1145/3295500.3356150

[18] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. 2015. An Energy Efficiency Feature Survey of the Intel Haswell Processor. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 896–904. doi:10.1109/IPDPSW.2015.70

[19] HPCG. [n. d.]. The HPCG Benchmark. https://www.hpcg-benchmark.org. (accessed Dec. 1, 2022).

[20] Chung hsing Hsu and Wu chun Feng. 2005. A Power-Aware Run-Time System for High-Performance Computing. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. 1–1. doi:10.1109/SC.2005.3

[21] Intel. 2023. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System Programming Guide, Part 2* (2023). https://cdrdv2.intel.com/v1/dl/getContent/671427

[22] Nikolas Ioannou, Michael Kauschke, Matthias Gries, and Marcelo Cintra. 2011. Phase-Based Application-Driven Hierarchical Power Management on the Single-chip Cloud Computer. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 131–142. doi:10.1109/PACT.2011.19

[23] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. 2006. Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 359–370. doi:10.1109/MICRO.2006.30

[24] Chanseok Kang, Seungyul Lee, Yong-Jun Lee, Jaejin Lee, and Bernhard Egger. 2017. Scheduling for Better Energy Efficiency on Many-Core Chips. In *Job Scheduling Strategies for Parallel Processing*, Narayan Desai and Walfredo Cirne (Eds.). Springer International Publishing, Cham, 46–68.

[25] Jeffrey O. Kephart, Hoi Chan, Rajarshi Das, David W. Levine, Gerald Tesauro, Freeman Rawson, and Charles Lefurgy. 2007. Coordinating Multiple Autonomic Managers to Achieve Specified Power-Performance Tradeoffs. In *Fourth International Conference on Autonomic Computing (ICAC'07)*. 24–24. doi:10.1109/ICAC.2007.12

[26] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. 2010. Interval-based models for run-time DVFS orchestration in superscalar processors. In *Proceedings of the 7th ACM International Conference on Computing Frontiers* (Bertinoro, Italy) *(CF '10)*. Association for Computing Machinery, New York, NY, USA, 287–296. doi:10.1145/1787275.1787338

[27] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, and K.I. Farkas. 2004. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* 64–75. doi:10.1109/ISCA.2004.1310764

[28] Los Alamos National Laboratory. [n. d.]. SNAP. https://github.com/lanl/SNAP. (accessed May 25, 2023).

[29] Lawrence Livermore National Laboratory. [n. d.]. CORAL-2 Benchmarks. https://asc.llnl.gov/coral-2-benchmarks. (accessed Jul. 1, 2023).

[30] Lawrence Livermore National Laboratory. [n. d.]. Nekbone. https://asc.llnl.gov/sites/asc/files/2021-02/Nekbone_Summary_2_3_6.pdf. (accessed May 25, 2023).

[31] Leibniz-Rechenzentrum. [n. d.]. Energy Aware Runtime. https://doku.lrz.de/energy-aware-runtime-10746191.html. (accessed November 16, 2023).

[32] Slurm Workload Manager. [n. d.]. SPANK. https://slurm.schedmd.com/spank.html. (accessed February 28, 2024).

[33] Mantevo. [n. d.]. miniAMR. https://github.com/Mantevo/miniAMR. (accessed September 10, 2024).

[34] S Martin, G Koprowski, and S Wallace. 2018. Cray XC Advanced Power Management Updates. *Proc. Cray Users Group* (2018).

[35] Margaret Martonosi, David Brooks, and Pradip Bose. 2001. Modeling and Analyzing CPU Power and Performance: Metrics, Methods, and Abstractions. https://www.princeton.edu/~mrm/tutorial/hpca2001_tutorial.pdf International Symposium on High-Performance Computer Architecture (HPCA).

[36] Sanyam Mehta. 2022. Performance Analysis and Optimization with Little's Law. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 12–23. doi:10.1109/ISPASS55109.2022.00002

[37] Ke Meng, Russ Joseph, Robert P. Dick, and Li Shang. 2008. Multi-Optimization power management for chip multiprocessors. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 177–186.

[38] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. 2012. Predicting Performance Impact of DVFS for Realistic Memory Systems. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 155–165. doi:10.1109/MICRO.2012.23

[39] Sava Mintchev and Vladimir Getov. 1997. PMPI: High-level message passing in Fortran77 and C. In *High-Performance Computing and Networking: International Conference and Exhibition Vienna, Austria, April 28–30, 1997 Proceedings 5*. Springer, 601–614.

[40] Rajib Nath and Dean Tullsen. 2015. The CRISP performance model for dynamic voltage and frequency scaling in a GPGPU. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 281–293. doi:10.1145/2830772.2830826

[41] Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. 2013. Exploring hardware overprovisioning in power-constrained, high performance computing. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (Eugene, Oregon, USA) *(ICS '13)*. Association for Computing Machinery, New York, NY, USA, 173–182. doi:10.1145/2464996.2465009

[42] Joshua Peraza, Ananta Tiwari, Michael Laurenzano, Laura Carrington, and Allan Snavely. 2016. PMaC's green queue: a framework for selecting energy optimal DVFS configurations in large scale MPI applications. *Concurrency and Computation: Practice and Experience* 28, 2 (2016), 211–231. doi:10.1002/cpe.3184 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.3184

[43] Exascale Computing Project. [n. d.]. Application Development. https://www.exascaleproject.org/research/#application. (accessed Jul. 1, 2023).

[44] Barry Rountree, David K. Lowenthal, Martin Schulz, and Bronis R. de Supinski. 2011. Practical performance prediction under Dynamic Voltage Frequency Scaling. In *2011 International Green Computing Conference and Workshops*. 1–8. doi:10.1109/IGCC.2011.6008553

[45] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. 2009. Adagio: Making DVS Practical for Complex HPC Applications. In *Proceedings of the 23rd International Conference on Supercomputing* (Yorktown Heights, NY, USA) *(ICS '09)*. Association for Computing Machinery, New York, NY, USA, 460–469. doi:10.1145/1542275.1542340

[46] Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant Kale. 2014. Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 807–818. doi:10.1109/SC.2014.71

[47] Robert Schöne, Daniel Hackenberg, and Daniel Molka. 2012. Memory Performance at Reduced CPU Clock Speeds: An Analysis of Current X86_64 Processors. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems* (Hollywood, CA) *(HotPower'12)*. USENIX Association, USA, 9.

[48] Woong Shin, Vladyslav Oles, Ahmad Maroof Karimi, J. Austin Ellis, and Feiyi Wang. 2021. Revealing power, energy and thermal dynamics of a 200PF pre-exascale supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 12, 14 pages. doi:10.1145/3458817.3476188

[49] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. 2009. Koala: A Platform for OS-Level Power Management. In *Proceedings of the 4th ACM European Conference on Computer Systems* (Nuremberg, Germany) *(EuroSys '09)*. Association for Computing Machinery, New York, NY, USA, 289–302. doi:10.1145/1519065.1519097

[50] SPEC. [n. d.]. SPEC CPU 2017. https://www.spec.org/cpu2017/. (accessed Dec. 1, 2022).

[51] John Thiede, Noman Bashir, David Irwin, and Prashant Shenoy. 2023. Carbon Containers: A System-level Facility for Managing Application-level Carbon Emissions. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 17–31. doi:10.1145/3620678.3624644

[52] Akshay Venkatesh, Abhinav Vishnu, Khaled Hamidouche, Nathan Tallent, Dhabaleswar Panda, Darren Kerbyson, and Adolfy Hoisie. 2015. A case for application-oblivious energy-efficient MPI runtime. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. doi:10.1145/2807591.2807658

[53] Vincent M. Weaver and Sally A. McKee. 2008. Can hardware performance counters be trusted?. In *2008 IEEE International Symposium on Workload Characterization*. 141–150. doi:10.1109/IISWC.2008.4636099

[54] Andreas Weissel and Frank Bellosa. 2002. Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (Grenoble, France) *(CASES '02)*. Association for Computing Machinery, New York, NY, USA, 238–246. doi:10.1145/581630.581668

[55] Qiang Wu, V.J. Reddi, Youfeng Wu, Jin Lee, D. Connors, D. Brooks, M. Martonosi, and D.W. Clark. 2005. A dynamic compilation framework for controlling microprocessor energy and performance. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. 12 pp.–282. doi:10.1109/MICRO.2005.7

[56] ETH Zurich. [n. d.]. SPECFEM3D. https://seg.ethz.ch/software/specfem3d.html. (accessed Aug. 4, 2023).