UC Riverside

UC Riverside Electronic Theses and Dissertations

Title

The Interplay Between Energy Efficiency and Resilience for Scalable High Performance Computing Systems

Permalink

https://escholarship.org/uc/item/2ms6f538

Author

Tan, Li

Publication Date

2015

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

The Interplay Between Energy Efficiency and Resilience for Scalable High Performance Computing Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Li Tan

December 2015

Dissertation Committee:

Dr. Zizhong Chen, Chairperson
Dr. Nael Abu-Ghazaleh
Dr. Laxmi N. Bhuyan
Dr. Sheldon Tan

Copyright by
Li Tan
2015

The Dissertation of Li Tan is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I feel indebted to my advisor, Dr. Zizhong Chen, without whose help, I would not have been here. Back to five years ago, I came to my dreamland of doing research in Computer Science. It took me until now to realize what the real research should look like. Dr. Chen is the one who leads me to the zen of research in a graceful way. I would have left my research career but for the all-the-way-through support from Dr. Chen. It is him who enlightens me all the time with a holistic view of what High Performance Computing (HPC) is. It is him who inspires me to find the right new research topics during the Ph.D. study. It is him who opens my mind to see a self-fulfilling career path after the pursuit of a Ph.D. degree.

I am also grateful to our collaborators who help me in different manners during my research efforts. Dr. Rong Ge, as my academic model of doing energy efficiency research, motivated me a lot on how to see the big picture comprehensively and professionally, brainstormed with us actively for a potential reserach direction, and provided me great hardware support, without which my research could not be evaluated. In particular, I highly appreciate the technical writing help from Dr. Ge, from which I do learn a lot. Dr. Shuaiwen Leon Song offered me a chance to revive the research idea that has been floating around the mind of Dr. Chen and me, and made it come true a solid and meaningful work for my career. Dr. Darren J. Kerbyson enrolled me and checked me out for the wonderful 4-month internship at Pacific Northwest National Laboratory in Summer 2014. Dr. Dong Li showed me promising research directions of HPC, and Dr. Ziliang Zong helped me learn more interesting interdisciplinary research topics in HPC.

I would like to thank my colleagues in the supercomputing laboratory, Hongbo Li, Panruo Wu, Longxiang Chen, Dingwen Tao, Jieyang Chen, Xin Liang, Teresa Davies, and Sihuan Li, and numerous friends I made within my life in Riverside, my conference trips, and my internship. They really helped me, encouraged me, and relaxed me a lot during the persistent Ph.D. grind. My heartfelt appreciation goes to our departmental Graduate Student Affairs Officer Amy S. Ricks's help on my questions of all kinds.

Particularly, I am thankful to other professors in my Ph.D. dissertation committee, Dr. Nael Abu-Ghazaleh, Dr. Laxmi N. Bhuyan, and Dr. Sheldon Tan. They guided me to a well-structured and highly-refined thesis, and paid unique efforts in my Ph.D. proposal and final defense, with insightful comments and valuable suggestions.

Last but not least, I express my sincere gratitude for the love from my family. My parents always recognize my talent of doing what I like, and back me up without any complaints throughout the five years. My wife gives me the earnest and endless love to endorse most of my personal decisions. "*What do you learn from this?*" – I will keep this in mind and really learn a lot from your everlasting and true love in the world.

Funding Acknowledgments

The research work in this dissertation is partly supported by the US National Science Foundation grants CCF-1305622, ACI-1305624, and CCF-1513201, and the SZSTI basic research program JCYJ20150630114942313.

To my parents and my wife for all the support. Without you, I cannot survive my PhD.

ABSTRACT OF THE DISSERTATION

The Interplay Between Energy Efficiency and Resilience for Scalable High Performance Computing Systems

by

Li Tan

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2015
Dr. Zizhong Chen, Chairperson

As the exascale supercomputers are expected to embark around 2020, supercomputers nowadays expand rapidly in size and duration in use, which brings demanding requirements of energy efficiency and resilience. These requirements are becoming prevalent and challenging, considering the crucial facts that: (a) The costs of powering a supercomputer grow greatly together with its expanding scale, and (b) failure rates of large-scale High Performance Computing (HPC) systems are dramatically shortened due to a large amount of compute nodes interconnected as a whole. It is thus desirable to consider both crucial dimensions for building scalable, cost-efficient, and robust HPC systems in this era. Specifically, our goal is to fulfill the optimal performance-power-failure ratio while exploiting parallelism during HPC runs.

Within a wide range of HPC applications, numerical linear algebra matrix operations including matrix multiplication, Cholesky, LU, and QR factorizations are fundamental and have been extensively used for science and engineering fields. For some scientific applications, these matrix operations are the core component and dominate the total execution

time. Saving energy for the matrix operations thus significantly contributes to the energy efficiency of scientific computing nowadays. Typically, when processors are experiencing idle time during HPC runs, i.e., slack, energy savings can be achieved by leveraging techniques to appropriately scale down processor frequency and voltage during underused execution phases. Although with high generality, existing OS level energy efficient solutions can effectively save energy for some applications in a black-box fashion, they are however defective for applications with variable workloads such as the matrix operations – the optimal energy savings cannot be achieved due to potentially inaccurate and high-cost workload prediction they rely on. Therefore, we propose to utilize algorithmic characteristics of the matrix operations to maximize potential energy savings. Specifically, we achieve the maximum of energy savings in two ways: (a) reducing the overhead of processor frequency switches during the slack, and (b) accurately predicting slack of processors via algorithm-based slack prediction, and eliminating the slack accordingly by respecting the critical path of an HPC run.

While energy efficiency and resilience issues have been extensively studied individually, little has been done to understand the interplay between them for HPC systems. We propose to quantitatively analyze the trade-offs between energy efficiency and resilience in the large-scale HPC environment. Firstly, we observe that existing energy saving solutions via slack reclamation are essentially frequency-directed, and thus fail to fully exploit more energy saving opportunities. In our approach, we decrease the supply voltage associated with a given operating frequency for processors to further reduce power consumption at the cost of increased failure rates. We leverage the mainstream resilience techniques to

tolerate the increased failures caused by the undervolting technique. Our strategy is theoretically validated and empirically evaluated to save more energy than a state-of-the-art frequency-directed energy saving solution, with the guarantee of correctness. Secondly, for capturing the impacts of frequency-directed solutions and undervolting, we also develop analytic models that investigate the trade-offs among resilience, energy efficiency, and scalability for large-scale HPC systems. We discuss various HPC parameters that inherently affect each other, and also determine the optimal energy savings at scale, in terms of the number of floating-point operations per Watt, in the presence of undervolting and fault tolerance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the expected embarking time of around 2020 for the exascale supercomputers [76], i.e., High Performance Computing (HPC) systems that are able to deliver the performance up to ExaFLOPS (10^{18} FLoating-point OPerations per Second), HPC architects become aware of two crucial facts: (a) The costs of powering a supercomputer are rapidly increasing nowadays due to expansion of its size and duration in use, and (b) although small on a single node, failure rates of large-scale computing systems are dramatically shortened – can be of the order of magnitude of hours [72] due to a large amount of compute nodes interconnected as a whole. The ever-growing and demanding requirements for improving energy efficiency and resilience of HPC systems have been regarded as a pressing issue. It is highly desirable to consider both critical dimensions for building scalable, cost-efficient, and robust HPC systems nowadays. HPC architects start to concern not only compute capability of a supercomputer, ranked by the TOP500 list [27], but also energy efficiency

and fault tolerance of the computing system. The Green500 list [10], ranks the top 500 supercomputers worldwide by performance-power ratio in six-month cycles.

Solutions of curtailing energy consumption of HPC runs without sacrificing performance via preserving parallelism (e.g., respecting the critical path of an HPC run) have been widely studied. With different focuses of studying, holistic hardware and software approaches for reducing energy costs of running high performance scientific applications have been extensively proposed. Software-controlled hardware solutions such as DVFS-directed (Dynamic Voltage and Frequency Scaling [141]) energy efficient scheduling are deemed to be effective and lightweight [46] [67] [69] [86] [124] [116] [115]. Performance and memory constraints have been considered as trade-offs for energy savings [92] [70] [139] [127] [132]. Generally, energy savings can be achieved by leveraging power-aware techniques that strategically switch power-scalable hardware components to low-power states, when the peak performance of the components is not necessary.

DVFS is a runtime technique that is able to switch operating frequency and supply voltage of a hardware component (CPU, GPU, memory, etc.) to different *levels* (also known as *gears* or *operating points*). Energy efficient approaches employ DVFS strategically per workload characteristics of HPC runs to exploit energy saving opportunities dynamically. CPU and GPU are the most widely applied hardware components for saving energy via DVFS due to two primary reasons: (a) Compared to other components such as memory, CPU/GPU DVFS is easier to implement [54] – various handy DVFS APIs have been industrialized for CPU/GPU DVFS such as CPUFreq kernel infrastructure [5] incorporated into the Linux kernel and NVIDIA System Management Interface (nvidia-smi) [21] for NVIDIA

GPU; (b) CPU energy costs dominate the total system energy consumption [71] (CPU and GPU energy costs dominate if heterogeneous architectures are considered), and thus saving CPU and GPU energy greatly improves energy efficiency of the whole system. In our work, we focus on distributed-memory HPC systems with or without accelerators such as GPU, where CPU DVFS or GPU DVFS is conducted for energy saving purposes.

For instance, energy can be saved by reducing CPU frequency and voltage during non-CPU-bound operations such as large-message MPI communication, since generally execution time of such operations barely increases at a low-power state of CPU, in contrast to original runs at a high-power state. Given the fact that energy consumption equals product of average power consumption and execution time ($E = \overline{P} \times T$), and the assumption that dynamic power consumption of a CMOS-based processor is proportional to product of operating frequency and square of supply voltage ($P \propto fV^2$) [108] [80], energy savings can be effectively achieved using DVFS-directed strategical scheduling approaches with little performance loss.

1.1 Background Knowledge: High Performance Numerical Linear Algebra

Improving energy efficiency of commonly used libraries is a promising solution to building energy efficient HPC systems. Linear algebra has been widely used in almost all science and engineering fields, and has been considered as the core component of high performance scientific computing nowadays. While many linear algebra libraries have been developed to optimize their performance, no linear algebra libraries optimize their energy efficiency at the library design time.

As classic dense numerical linear algebra operations for solving systems of linear equations, such as $Ax = b$ where A is a given coefficient matrix and b is a given vector, Cholesky factorization applies to the case that A is a symmetric positive definite matrix, while LU and QR factorizations apply to any general $M \times N$ matrices. The goal of these operations is to factorize A into the form LL^T where L is lower triangular and L^T is the transpose of L, the form LU where L is unit lower triangular and U is upper triangular, and the form QR where Q is orthogonal and R is upper triangular, respectively. Thus from $LL^Tx = b$, $LUx = b$, $QRx = b$, x can be easily solved via forward substitution and back substitution. In practice, Cholesky, LU, and QR factorizations are widely employed in extensive areas of high performance scientific computing. Various software libraries of numerical linear algebra for distributed multicore scientific computing such as ScaLAPACK [26], DPLASMA [8], and MAGMA [18] provide routines of these matrix factorizations as standard functionality. Matrix multiplication is the core computation workload within Cholesky, LU, and QR factorizations, and it is also extensively used in many other areas, including quantum mechanics and economics. Therefore, saving energy for numerical linear algebra matrix operations such as matrix multiplication, Cholesky, LU, and QR factorizations contributes significantly to the energy efficiency of high performance scientific computing nowadays. Next we present basics of Cholesky, LU, and QR factorizations. Details of matrix multiplication is introduced in Chapter 2. We also describe an effective graph representation for parallelism present during the execution of high performance matrix operations.

A well-designed partitioning and highly-efficient parallel algorithms of computation and communication substantially determine energy and performance efficiency of

task-parallel applications. For such purposes, classic implementations of high performance Cholesky, LU, and QR factorizations are as follows: (a) Partition the global matrix into a number of computing cores as a process grid using 2-D block cyclic data distribution [134] for load balancing; (b) perform local *diagonal matrix* factorizations in each core individually and communicate factorized local matrices to other cores for *panel matrix* solving and *trailing matrix* updating, as shown in Figure 4.2, a stepwise LU factorization without pivoting. Due to frequently-arising data dependencies, parallel execution of the matrix operations can be characterized using Directed Acyclic Graph (DAG), where data dependencies among parallel tasks are appropriately represented. DAG for the matrix operations is formally defined below:

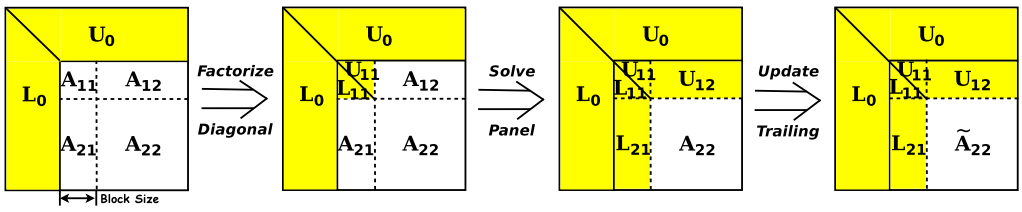


Figure 1.1: Stepwise Illustration of LU Factorization without Pivoting.

Definition 1. Data dependencies among parallel tasks of high performance matrix operations running on a HPC system are modeled by a Directed Acyclic Graph (DAG) $G = (V, E)$, where each node $v \in V$ denotes a task of the matrix operations, and each directed edge $e \in E$ represents a dynamic data dependency from task t_j to task t_i that both tasks manipulate on either different local matrices (i.e., an *explicit* dependency) or the same local matrix (i.e., an *implicit* dependency), denoted by $t_i \rightarrow t_j$.

$$\begin{pmatrix} A_{11} & A_{21}^T & A_{31}^T & A_{41}^T \\ A_{21} & A_{22} & A_{32}^T & A_{42}^T \\ A_{31} & A_{32} & A_{33} & A_{43}^T \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix} \times \begin{pmatrix} L_{11}^T & L_{21}^T & L_{31}^T & L_{41}^T \\ 0 & L_{22}^T & L_{32}^T & L_{42}^T \\ 0 & 0 & L_{33}^T & L_{43}^T \\ 0 & 0 & 0 & L_{44}^T \end{pmatrix}$$

$$= \begin{pmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T & L_{11}L_{31}^T & L_{11}L_{41}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T & L_{21}L_{31}^T + L_{22}L_{32}^T & L_{21}L_{41}^T + L_{22}L_{42}^T \\ L_{31}L_{11}^T & L_{31}L_{21}^T + L_{32}L_{22}^T & L_{31}L_{31}^T + L_{32}L_{32}^T + L_{33}L_{33}^T & L_{31}L_{41}^T + L_{32}L_{42}^T + L_{33}L_{43}^T \\ L_{41}L_{11}^T & L_{41}L_{21}^T + L_{42}L_{22}^T & L_{41}L_{31}^T + L_{42}L_{32}^T + L_{43}L_{33}^T & L_{41}L_{41}^T + L_{42}L_{42}^T + L_{43}L_{43}^T + L_{44}L_{44}^T \end{pmatrix}$$

Figure 1.2: Matrix Representation of a 4 × 4 Blocked Cholesky Factorization.

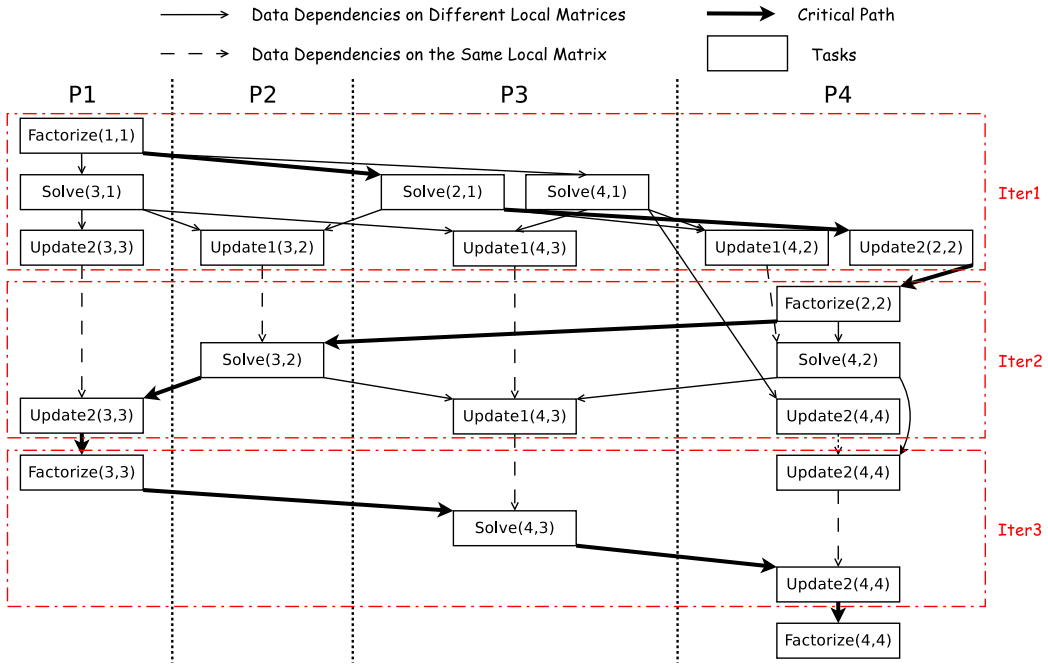


Figure 1.3: DAG Representation of Task Scheduling for the 4 × 4 Blocked Cholesky Factorization in Figure 4.3 on a 2 × 2 Process Grid.

Example. Due to similarity among the three matrix factorizations and space limitation, we henceforth take Cholesky factorization for example to elaborate our approach. Consider a 4×4 blocked Cholesky factorization as given in Figure 4.3. The outcome of the task factorizing A_{11}, i.e., L_{11}, is used in the tasks solving local matrices L_{21}, L_{31}, and L_{41} in the same column as L_{11}, i.e., the tasks calculating the *panel matrix*. In other words, there exist three data dependencies from the tasks solving L_{21}, L_{31}, and L_{41} to the task factorizing A_{11}, denoted by three *solid* directed edges from the task Factorize(1,1) to the tasks Solve(2,1), Solve(3,1), and Solve(4,1) individually as shown in Figure 4.4. Besides the above *explicit* dependencies, there exists an *implicit* dependency between the task updating local matrix A_{32} and the task subsequently solving L_{32} on the same local matrix, denoted by the *dashed* directed edge from the task Update1(3,2) to the task Solve(3,2) in Figure 4.4. Note that communication among tasks is not shown in Figure 4.4, and updating diagonal local matrices and updating non-diagonal local matrices are distinguished as Update2() and Update1() respectively due to different computation time complexity.

1.2 Algorithm-Based Energy Saving for Numerical Linear Algebra Operations

Based on the task-parallel DAG representation, we can effectively pinpoint potential energy saving opportunities in terms of slack among the tasks, via identifying Critical Path (CP) in runs of the matrix operations.

Although load balancing techniques are leveraged for distributing workloads into a number of computing cores as evenly as possible, assuming that all cores have the same

hardware configuration and thus the same computation and communication capability, slack can result from the fact that different cores can be utilized unfairly due to three primary reasons: (a) imbalanced computation delay due to data dependencies among tasks, (b) imbalanced task partitioning, and (c) imbalanced communication delay. Difference in CPU utilization results in different amount of computation slack. For instance, constrained by data dependencies, the start time of processes running on different cores differs from each other, as shown in Figure 4.4 (see page 15) where P1 starts earlier than the other three processes. Moreover, since the location of local matrices in the global matrix determines what types of computation are performed on the local matrices, load imbalancing from difference in task types and task amount allocated to different processes cannot be eliminated completely by the 2-D block cyclic data distribution, as shown in Figure 4.4 where P2 has lighter workloads compared to the other three processes. Imbalanced communication time due to different task amount among the processes further extends the difference in slack length for different processes.

Definition 2. Critical path is one particular task trace from the beginning task of one run of a task-parallel application to the ending one with the total slack of zero.

Per the definition, any delay on tasks on the CP increases the total execution time of the application, while dilating tasks off the CP into their slack individually without further delay does not cause performance loss as a whole. Energy savings can be achieved by appropriately reducing frequency to dilate tasks off the CP into their slack as much as possible, which is referred to as the *CP* approach. Numerous existing OS level solutions

effectively save energy via *CP-aware* analysis [46] [101] [116] [115] [33] [30]. Figure 4.4 highlights one CP for the provided parallel Cholesky factorization with bold edges.

We can generate a CP for high performance matrix operations via their algorithmic characteristics. Consider the same Cholesky factorization above. The heuristic of the CP generation algorithm is as follows: (a) Each task of factorizing is included in the CP, since the local matrices to factorize are always updated last, compared to other local matrices in the same row of the global matrix, and the outcome of factorizing is required in future computation. In other words, the task of factorizing serves as a transitive step that cannot be executed in parallel together with other tasks; (b) each task of Update1() is excluded from the CP, since it does not have direct dependency relationship with any tasks of factorizing, which are already included in the CP; (c) regarding Update2(), we select the ones that are directly depended by the tasks of factorizing on the same local matrix into the CP; (d) we choose the tasks of solving that are directly depended by Update2() (or directly depends on Factorize(), not shown in the algorithm) into the CP. Note that CP can also be identified using methods other than TDS analysis [46] [33] [30].

Although effective, existing energy efficient approaches for slack reclamation can be defective and thus cannot achieve the optimal energy savings. Thus we propose Algorithm-Based Energy Saving (ABES) for the widely used matrix operations to fully exploit energy saving opportunities. Firstly, the DVFS-based solutions may bring disadvantageous impacts to performance and energy efficiency: introducing DVFS itself in HPC runs can incur non-negligible overhead that may offset energy savings gained. Regardless of the efforts that keep only necessary processor frequency switches when CPU or GPU is idle, we propose to

further reduce the DVFS overhead by utilizing algorithmic characteristics of distributed matrix multiplication. In a loop-fashion interleaving of computation and communication, there exist a large number of frequency switches issued by DVFS. We minimize the number of DVFS switches via aggregating blocked computation and blocked communication as groups and employ DVFS individually at group level. Moreover, we leverage a high performance communication scheme for fully exploiting network bandwidth via pipeline broadcast. Secondly, the existing OS level workload prediction approaches are costly and inaccurate to predict slack for applications with variable workloads, such as the matrix operations. Therefore, the maximum of energy savings cannot be obtained due to the workload prediction strategies OS level approaches rely on. We utilize algorithmic characteristics of the matrix operations to accurately predict slack and maximize energy savings accordingly by respecting the CP of an HPC run. With negligible overhead, our optimized matrix operations are able to deliver *high performance* – comparable performance to other highly optimized linear algebra kernels, *energy efficiency* – energy savings achieved is close to the theoretical upper bound, and *high portability* – the optimized matrix operations are standardized as a numerical linear algebra library, with which it can benefit a wide range of scientific applications with mininal programming efforts, and the equipped energy saving techniques is hardware-transparent, making it outperform others across different architectures.

1.3 Entangled Effects: Energy Efficiency and Resilience in Scalable Systems

Energy efficiency and resilience are two crucial challenges for HPC systems to reach exascale. While energy efficiency and resilience issues have been extensively studied individually, little has been done to understand the interplay between energy efficiency and resilience for HPC systems. The average power of the top 5 supercomputers worldwide has been inevitably growing to 10.1 MW according to the latest TOP500 list [27], as opposed to the power consumed by a city with a population of 20,000 is about 11 MW [25]. The 20 MW exascale computing power-wall prediction by the US Department of Energy [9] indicates the severity of constrained energy budget against the necessity of performance requirements due to ever-growing computational complexity. Empirically, running HPC applications on supercomputers can be interrupted by failures including hardware breakdowns and soft errors. Although small on a single node, failure rates of large-scale computing systems can be of the order of magnitude of hours [72] due to a large amount of nodes interconnected as a whole. Greatly shortened Mean Time To Failures (MTTF) of such systems at large scales entails less reliability. For instance, a compute node in a cluster of 692 nodes (22,144 cores in total) at Pacific Northwest National Laboratory can experience up to 1,700 ECC (Error-Correcting Code) errors in a two-month period [24]. K computer, ranked the first on the TOP500 list in June/Nov. 2011, held a hardware failure rate of up to 3% and affected by up to 70 soft errors in a month [14]. Larger forthcoming exascale systems are expected to suffer from more errors of different types in a fixed time period [118].

Both energy efficiency and resilience are considered to fulfill an optimal performance-

cost ratio with a given amount of resources, as the trend of future exascaling computing implies. Widely studied individually, energy saving techniques and resilience techniques may restrict each other to attain the optimal effectiveness if employed jointly. In HPC runs, different forms of slack can result from numerous factors such as load imbalance, network latency, commmunication delay, and memory and disk access stalls. Such slack can be exploited either as energy saving opportunities, since during the slack the peak performance of processors (e.g., CPU, GPU, and even memory) is generally not necessary for meeting time requirements of the applications, and thus slack reclamation techniques can be employed to save energy [115] [102] [132]; or exploited as fault tolerance opportunities, since resilience techniques such as Checkpoint/Restart (CR) [110] [53] and Algorithm-Based Fault Tolerance (ABFT) [58] [48] can be performed during the slack without incurring performance loss overall. Therefore, given a specific schedule of an HPC application, energy saving techniques and resilience techniques can compete for the fixed amount of slack, which can restrict each other to attain the best extent.

During the slack where the peak performance of processors is not necessary, energy savings can be achieved with negligible performance loss, by fine-grained slack utilization via DVFS in two fashions: CP-aware slack reclamation [115] and race-to-halt [64]. For detected slack, both DVFS techniques can effectively save energy by reducing frequency and voltage to different extents, due to the facts that dynamic power consumption of a CMOS-based processor is proportional to product of operation frequency and square of supply voltage, and that overall runtime is barely increased. For non-slack durations, e.g., computation, frequency reduction generally incurs a proportional reduction in runtime, which is usually

not acceptable for HPC runs. Therefore, for such durations, both DVFS techniques employ the highest frequency and voltage to guarantee the performance overall.

However, for both slack and non-slack, existing DVFS techniques do not fully exploit potential energy savings, since they are essentially frequency-directed: Voltage is only lowered together with frequency when frequency reduction is necessary in the presence of slack. There still exist further energy saving opportunities from *undervolting* [143] [29] [39] regardless of frequency scaling, i.e., lowering only supply voltage of a chip without corresponding reduction of its operating frequency, when frequency cannot be further lowered during either slack or non-slack due to performance requirements. Per the previous definition of power consumption, decreasing the supply voltage associated with a given operating frequency for processors and other CMOS-based components can significantly reduce power consumption. Nevertheless, this often raises system failure rates and consequently increases application execution time and energy consumption.

Energy efficiency and resilience are by nature two correlated but mutually constrained goals to achieve, from both theoretical and experimental perspectives. For energy saving purposes, reducing operating frequency and/or supply voltage of processors such as CPU will increase their failure rate, assuming that failures of combinational logic circuits follow a Poisson distribution (i.e., an exponential failure model) [119] [154], and thus incur more overhead on fault tolerance that may lead to performance loss and less energy savings. For reliability improving purposes, keeping operating frequency and/or supply voltage of the hardware at a conservatively high scale will lead to unnecessary energy costs without performance gain [39]. There exists an optimal trade-off between system reliability and

energy costs. It is a challenging issue to accomplish the most balanced energy efficiency and resilience on large-scale systems nowadays without performance degradation, which is especially substantial to be addressed for the forthcoming exascale systems.

As a technique of high generality, undervolting is advantageous to save extra energy for any phases of HPC runs since required performance of processors is not degraded, at the cost of increased failure rates. Unlike traditional simulation-based approaches [143] [29], Bacha *et al.* [39] first implemented an empirical undervolting system on Intel Itanium II processors, which is intended for reducing voltage margins and thus saving power, with ECC memory correcting arising faults. Their work indeed maximized potential power savings since they used pre-production processors that allows thorough undervolting until the voltages lower than the lowest voltage corresponding to the lowest frequency supported. In general, production processors are locked for reliability purposes, and they will typically shut down when the voltage is lowered below the one corresponding to the minimum frequency. For *generality* purposes, we propose a scheme that works for general production processors and thus can be deployed on large-scale HPC clusters nowadays.

Moreover, their work requires ECC memory to correct greatly-increased faults. Conventional supercomputer ECC memory extensively uses a Single Error Correcting, Double Error Detecting (SECDED) code [91] [142] [14], which relies on hardware support of ECC memory and can be limited to handle real-world hard and soft failures. For further achieving energy savings, a general and/or specialized lightweight resilience technique is expected to tolerate more complicated failures in the scenario of undervolting on HPC systems. We investigate the interplay between energy efficiency and resilience in HPC systems,

and demonstrate theoretically and empirically that the two significant goals in HPC nowadays can be traded-off in different scenarios. In this work, we consider two popular fault tolerance schemes: the classic CR approach, and the state-of-the-art ABFT approach.

For further analyzing the trade-offs between energy efficiency and resilience in the HPC environment, we quantitatively model the impacts of various HPC paramenters that inherently affect each other, and also determine the maximum of balanced performance and energy efficiency at scale, with the guarantee of resilience in the scenario of undervolting. Specifically, we look into typical parameters in scalable HPC runs including operating frequency and supply voltage of processors, number of used cores, problem size, checkpoint and restart overhead, and checkpoint interval. We discuss the optimal values of these parameters that contribute to the most balanced integrated performance and energy efficiency for large-scale HPC systems, in terms of the performance-power ratio, i.e., the number of floating-point operations per Watt, in the presence of energy saving DVFS, undervolting, and fault tolerance techniques.

1.4 Contributions

In summary, in this dissertation, the following contributions are achieved:

- We propose an adaptively aggressive DVFS scheduling strategy to achieve energy efficiency for data intensive applications, and further save energy via speculation to mitigate DVFS overhead for imbalanced branches. We implemented and evaluated our approach using five memory and disk access intensive benchmarks with imbalanced branches against another two energy saving approaches. The experimental

results indicate an average of 32.6% energy savings were achieved with 6.2% average performance loss compared to the original executions on a power-aware 64-core cluster.

- We propose a strategy that gains the optimal energy savings for distributed matrix multiplication via algorithmically trading more computation and communication at a time adaptively with user-specified memory costs for less DVFS switches. Combining a high performance pipeline broadcast communication scheme, the integrated approach achieves substantial energy savings (up to 51.4%) and performance gain (28.6% on average) compared to ScaLAPACK matrix multiplication on a cluster with an Ethernet switch, and outperforms ScaLAPACK and DPLASMA counterparts respectively by 33.3% and 32.7% on average on a cluster with an Infiniband switch;

- we propose `TX`, a library level race-to-halt DVFS scheduling approach that analyzes Task Dependency Set of each task in parallel Cholesky, LU, and QR factorizations to achieve substantial energy savings OS level solutions cannot fulfill. Partially giving up the generality of OS level solutions per requiring library level source modification, `TX` leverages algorithmic characteristics of the applications to gain greater energy savings. Experimental results on two power-aware clusters indicate that `TX` can save up to 17.8% more energy than state-of-the-art OS level solutions with negligible 3.5% on average performance loss;

- We present an energy saving undervolting approach that leverages the mainstream resilience techniques to tolerate the increased failures caused by undervolting. Our strategy is directed by analytic models, which capture the impacts of undervolting

and the interplay between energy efficiency and resilience. Experimental results on a power-aware cluster demonstrate that our approach can save up to 12.1% energy compared to the baseline, and conserve up to 9.1% more energy than a state-of-the-art DVFS solution;

- By extending the Amdahls Law and the Karp-Flatt Metric, taking resilience into consideration, we quantitatively model the integrated energy efficiency in terms of performance per Watt, and showcase the trade-offs among typical HPC parameters, such as number of cores, frequency/voltage, and failure rates. Experimental results for a wide spectrum of HPC benchmarks on two HPC systems show that the proposed models are accurate in extrapolating resilience-aware performance and energy efficiency, and capable of capturing the interplay among various energy saving and resilience factors. Moreover, the models can help find the optimal HPC configuration for the highest integrated energy efficiency, in the presence of failures and applied resilience techniques.

The rest of this dissertation is organized as follows. Chapter 2 introduces adaptive and aggressive energy saving for data intensive workloads. Chapter 3 and Chapter 4 presents algorithm-based energy savings for high performance matrix multiplication, Cholesky, LU, and QR factorizations. We discuss the interplay between energy efficiency and resilience for HPC systems in Chapter 5, and the quantitative relationship among HPC parameters for achieving the integrated optimal energy efficiency and resilience for scalable HPC systems in Chapter 6. Chapter 7 discusses related work and Chapter 8 concludes.

Chapter 2

A2E: *A*daptively *A*ggressive *E*nergy Efficient DVFS Scheduling for Data Intensive Applications

With the growing severity of power and energy consumption on high performance distributed-memory computing systems nowadays in terms of operating costs and system reliability [69] [92], reducing power and energy costs has been considered as a critical issue in high performance computing, in particular in this big data era. Featured by high portability and programmability, Dynamic Voltage and Frequency Scaling (DVFS) [141] [5] techniques have been empirically applied for scaling down power and energy costs with little or limited performance loss [108] [46] [86] [80] [67] [124] [93] [70] [116] [115] [128]. Generally, energy efficiency can be achieved during runs of high performance applications by scaling down operating voltage and frequency of CPU, where peak CPU performance is not necessary

such as slack from load imbalance, communication delay, memory and disk access latency, etc., given the assumption that CPU dominates the total system-wise energy consumption. DVFS is thus deemed an effective approach to address the concerns of operating costs and system reliability for high performance applications nowadays.

Per the functionality of an application, types of workloads within the application consist of computation, communication, memory accesses, and disk accesses, etc. For communication intensive applications, an effective way of improving energy efficiency, referred to as basic DVFS scheduling strategy, is to scale down CPU voltage and frequency during communication, while keep peak CPU performance when CPU is fully loaded during computation. This approach can be easily fulfilled, since at source code level the boundary between communication and computation is explicit. Appropriate CPU frequency can be assigned via DVFS techniques at the boundary between communication and computation. Since the execution of communication is not CPU-bound, communication time will barely increase due to low CPU performance. Moreover, computation time will not grow since CPU performance during computation is kept the same as the orignal by not altering CPU frequency. Since generally voltage is proportional to frequency, energy savings can be achieved using basic DVFS scheduling strategy with negligible performance loss due to lower CPU voltage and frequency on average compared to the original execution.

Similarly, peak CPU performance is not needed when CPU is waiting for data from memory and disk. Typically, for memory and disk access intensive applications, memory and disk access latency are performance bottleneck of the applications. According to the fundamental memory hierachy of moderm computer architectures, compared to CPU,

main memory access takes hundreds of clock cycles while local disk access time is of the order of magnitude of millisecond, 10^6 greater than memory access time in general. As for memory and disk access intensive applications, energy efficiency can be intuitively achieved by reducing CPU frequency when memory and disk accesses are performed and CPU is waiting for data.

Despite the straightforward deployment of DVFS for communication intensive applications, it is however not intuitive to achieve energy efficiency for other types of data intensive applications such as memory and disk access intensive applications due to two reasons: Firstly, employing DVFS in our approach is implemented at source code level within the application via system calls for modifying CPU frequency configuration files at runtime. Empirically, memory and disk accesses are generally accompanied by CPU-bound operations at source code level, which causes the boundary between memory and disk accesses and computation implicit. As a consequence, it is difficult to separate memory and disk accesses from computation and then apply DVFS for energy savings. Secondly, the overhead on employing DVFS can be high: Given the iterative nature of many high performance applications, the time and energy costs on employing fine-grained DVFS scheduling can be non-negligible due to a large number of CPU frequency switches [93] [126] [132]. A lightweight DVFS scheduling strategy is thus desirable.

In this chapter, we introduce an adaptively aggressive DVFS scheduling strategy (A2E) for energy efficient memory and disk access intensive applications with imbalanced branches, where memory and disk accesses are mixed with minor computation. Instead of separating memory and disk accesses from computation for an Energy Saving Block

(ESB) with different types of workloads, and then performing fine-grained DVFS scheduling accordingly, we aggressively apply DVFS to the hybrid ESB holistically, and adaptively set an appropriate CPU frequency to the hybrid ESB according to the computation time proportion within the total execution time of the ESB. In summary, the contributions of this chapter are as follows:

- We analyze the impact of factors such as CPU frequency and execution time on energy consumption of applications consisting of different dominant workloads, which motivates our idea of A2E;
- We demonstrate the significance of code boundary for achieving energy efficiency via DVFS, and thus define ESB to refine energy saving opportunities and model energy and performance efficiency of our approach;
- We propose A2E to improve energy efficiency for memory and disk access intensive applications with mixed minor computation, and further save energy using speculation to mitigate DVFS overhead for imbalanced branches. Our approach is evaluated to achieve considerable energy savings (32.6% on average) and incur minor performance loss (6.2% on average) compared to the original runs of five benchmarks.

The rest of this chapter is organized as follows. Section 2.1 motivates and Section 2.2 introduces three energy saving approaches for data intensive applications. We provide implementation details and evaluate our approach in Section 2.3, and Section 2.4 concludes.

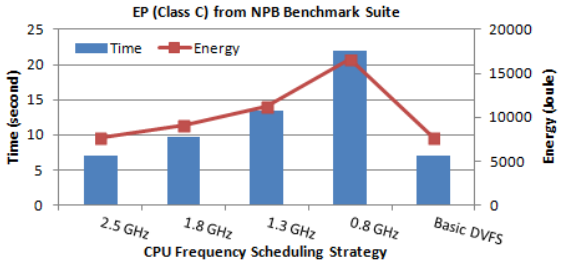


Figure 2.1: DVFS Scheduling for Compute Intensive Application.

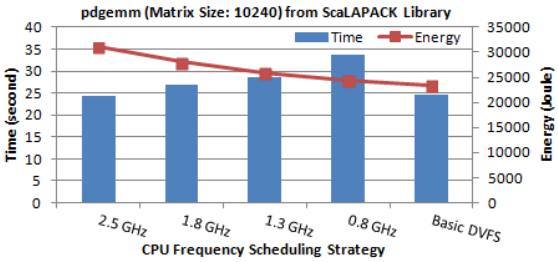


Figure 2.2: DVFS Scheduling for Compute/Non-Compute Comparable Application.

2.1 Motivation: DVFS Scheduling for Different Workload Intensive Applications

In order to learn the impact of factors such as CPU frequency and execution time that may affect energy consumption of applications with different dominant workloads, we conducted some experiments and the results are plotted in Figures 2.1 and 2.2. Motivated by the experimental results on DVFS scheduling for compute intensive and compute/non-compute comparable applications, we observe that the proportion of non-compute operations in an application determines whether energy consumption of the application is time-directed or frequency-directed. In other words, energy consumption is affected more by exectuion time in a compute intensive application, and is affected more by CPU frequency in a compute/non-compute comparable application, given the fact that energy consumption equals product of average power and time, where power is proportional to frequency and voltage.

As shown in Figure 2.1, CPU performance degradation for the compute intensive application EP from NPB [20] leads to more energy costs due to the longer execution time that is the more dominant factor compared to CPU frequency. On the other hand, for

an application with comparable proportion of computation and non-computation such as pdgemm() routine from ScaLAPACK [132] shown in Figure 2.2, there exists even a slight decrease of energy costs as CPU frequency goes down, despite the increasing execution time due to low CPU performance. We can further infer from the experimental results that if computation only takes a small proportion of the total execution time of an application as in the case of data intensive applications such as memory and disk access intensive applications, performance loss at a low CPU frequency is comparatively limited, and the less computation exists, the less performance loss is incurred from reducing CPU frequency. Therefore, the resulting reduction of power from lowering frequency dominates the ultimate energy costs. Compared to the compute/non-compute comparable application, more energy savings can be achieved by aggressively reducing CPU frequency for a non-compute intensive application.

Non-compute intensive applications can be any applications with a dominant proportion of non-compute workloads such as communication, memory accesses, and disk accesses, etc., where memory and disk access intensive applications are commonly regarded as data intensive applications. Different from communication intensive applications, it is challenging to employ DVFS on memory and disk access intensive applications for achieving energy efficiency, since data operations such as memory and disk accesses generally mix with minor computation at source code level. As we know, energy savings can be achieved by applying DVFS at source code level by lowering CPU frequency for data intensive operations where peak CPU performance is not necessary. It is however difficult to separate non-computation from computation for later assignment of appropriate CPU frequency to

```
 1:  while (caseA) {
 2:    ...
 3:    buffer = (char*)malloc(num*sizeof(char));
 4:    /* MPI communication routine call I */
 5:    MPI_Bcast(&buffer, count, type, root, comm);
 6:    /* Independent computation code */
 7:    computation();
 8:    /* MPI communication routine call II */
 9:    MPI_Alltoall(&sb, sc, st, &rb, rc, rt, comm);
10:    ...
11:  }
```

Figure 2.3: Typical Kernel Pattern of Communication Intensive Code.

```
 1:  while (caseA) {
 2:    ...
 3:    /* Memory accesses mixed with computation */
 4:    valueA = arrayA[baseA+offset];
 5:    arrayB[baseB] += valueB;
 6:    arrayC[baseC++] = arrayB[baseB++]+valueC;
 7:    ...
 8:    /* Disk accesses mixed with computation */
 9:    buffer = (char*)malloc(num*sizeof(char));
10:    fread(buffer, size, count, read_file_stream);
11:    fwrite(buffer, size, count, write_file_stream);
12:    ...
13:  }
```

Figure 2.4: Typical Kernel Pattern of Memory and Disk Access Intensive Code.

different workloads. To fulfill energy efficiency for data intensive applications, our goals include: (a) Reducing the performance loss from computation accompanying data intensive operations due to low CPU frequency, i.e., low-performance trade-off; (b) reducing the number of CPU frequency switches by DVFS, i.e., DVFS overhead. Both low-performance trade-off and DVFS overhead result in higher execution time and thus greater energy costs.

2.2 Energy Efficient DVFS Scheduling Strategies for Data Intensive Applications

In this section, we present our adaptively aggressive energy efficient DVFS scheduling strategy (A2E) for data intensive applications, e.g., memory and disk access intensive applications. Leveraging speculation, A2E can also handle conditional statements with imbalanced branches whose possibilities of occurrence are significantly different. Next we first introduce the concept of Energy Saving Blocks at source code level.

2.2.1 Energy Saving Blocks

Similarly as the common term *basic block* in the area of compilers, from the perspective of energy, an Energy Saving Block (ESB) is defined as a statement block of one specific type of workload such as computation, communication, memory accesses and disk accesses, etc., where runtime energy savings may be achieved by different means. For simplicity, such ESBs are referred to as *Comp*-ESB, *Comm*-ESB, *Mem*-ESB, and *Disk*-ESB respectively in the later text. For instance, in the code example shown in Figure 2.5 (a), there exist six ESBs located at Lines 5, 7, 8, 9, 11, and 17, respectively, i.e., two *Comp*-ESBs, two *Comm*-ESBs, one *Mem*-ESB, and one *Disk*-ESB, each of which can be assigned an appropriate CPU frequency accordingly via DVFS for energy saving purposes.

2.2.2 Basic DVFS Scheduling for Comp-ESB and Comm-ESB

We can apply a basic DVFS scheduling strategy for *Comp*-ESB and *Comm*-ESB that simply sets CPU frequency to as high as possible for *Comp*-ESB and sets CPU fre-

quency to as low as possible for *Comm*-ESB, which can be easily fulfilled since the boundary of *Comm*-ESB is explicit as shown in Figure 2.3: Little computation is involved in the MPI communication routine calls at Lines 5 and 9 respectively, and computation independent of communication at Line 7 is conducted after the communication code. The basic DVFS scheduling strategy is shown in Figure 2.5 (a), where a low-high CPU frequency pair is assigned around the communication code, since CPU is barely utilized in the communication and peak CPU performance is thus not necessary. Yet, the basic DVFS scheduling strategy suffers from two disadvantages: (a) It can only work at inter-ESB level but fail at intra-ESB level, i.e., towards single ESB with mixed workloads as shown in Figure 2.4 (we discuss it next); (b) the number of CPU frequency switches can be considerably large if the number of *Comm*-ESBs and the number of iterations of the loop are large, which incurs non-negligible overhead on time and energy [132].

2.2.3 Aggressive DVFS Scheduling for Mem-ESB and Disk-ESB

Figure 2.4 depicts typical kernel of memory and disk access intensive applications. Lines 4, 5, and 6 give three typical memory accesses mixed with computation. At Line 4, `valueA` is assigned until the finish of calculating the array index and accessing the content of corresponding memory location. Lines 5 and 6 show how array values are involved in computation after and before addressing, respectively. Likewise, for disk accesses given at Lines 10 and 11 that read and write blocks of data from and into local disk files individually, the value of input/output buffer pointer is frequently accessed and updated for current and next reading/writing position as the file reading and writing operations proceed. If the

```
 1:  while (caseA) {
 2:    if (caseB) { P1
 3:      ...
 4:      SetFreq(L_DVFS);
 5:      communication();
 6:      SetFreq(H_DVFS);
 7:      memory_access();
 8:      disk_access();
 9:      computation();
10:      SetFreq(L_DVFS);
11:      communication();
12:      SetFreq(H_DVFS);
13:      ...
14:    }
15:    else { P2 (P2 ≪ P1)
16:      ...
17:      computation();
18:      ...
19:    }
20:  }
```

(a) Basic DVFS Scheduling

```
 1:  SetFreq(L_DVFS);
 2:  while (caseA) {
 3:    if (caseB) { P1
 4:      ...
 5:      communication();
 6:      memory_access();
 7:      disk_access();
 8:      computation();
 9:      communication();
10:      ...
11:    }
12:    else { P2 (P2 ≪ P1)
13:      ...
14:      SetFreq(H_DVFS);
15:      computation();
16:      SetFreq(L_DVFS);
17:      ...
18:    }
19:  }
20:  SetFreq(H_DVFS);
```

(b) Aggressive DVFS Scheduling with Speculation (AGGREE)

Figure 2.5: Basic and Aggressive DVFS Scheduling for Typical Communication, Memory Access, and Disk Access Mixed Code with Imbalanced Branches.

CPU-bound computation time is significant among the total execution time of the *Mem*-ESB/*Disk*-ESB, i.e., in the case of compute intensive applications, considerable slowdown will be incurred from reducing CPU frequency for the ESB as a whole, and thus energy consumption grows as the trend shown in Figure 2.1.

Yet for applications with a small proportion of computation mixed with memory and disk accesses depicted in Figure 2.4, aggressively reducing CPU frequency for the whole ESB only causes minor performance loss while obtains considerable energy savings from low CPU frequency and voltage during waiting for memory and disk data, since memory and disk access time dominate the total execution time. Basic DVFS scheduling strategy fails to achieve energy savings for such applications since it is difficult to separate non-computation from computation and then apply DVFS accordingly. Even if the programmer manages to rewrite the source code for categorizing ESBs with explicit boundary between each other via the use of temporary variables, etc. (we use this method to calculate the proportion/percentage of different types of workloads within a hybrid ESB), performance and energy loss can be caused by numerous CPU frequency switches within the loop of ESBs, as shown in Figure 2.5 (a), the kernel of an application with different types of workloads including computation, communication, memory accesses and disk accesses. The basic DVFS scheduling strategy sets CPU frequency to low before the *Comm*-ESBs at Lines 5 and 11 respectively and sets it back to high after the *Comm*-ESBs. It keeps CPU frequency high for all *Mem*-ESB, *Disk*-ESB, and *Comp*-ESB if the *Mem*-ESB and the *Disk*-ESB are accompanied by minor computation as shown in Figure 2.4. Potential energy saving opportunities can be leveraged by aggressive DVFS scheduling (AGGREE) as presented in

Algorithm 1 *Adaptively Aggressive DVFS Scheduling Algo.*

SetDVFS(ESB, p_{comp}) /*Assume $f_0 < f_1 < \cdots < f_{N_f-1}$*/

```
 1:  Bcast(p_comp)
 2:  N_f ← GetNumFreq()
 3:  p'_comp ← Max(p_comp of all ESBs)
 4:  pSet_{0,...,N_f-1} ← GetRange(p'_comp, N_f)
 5:  while 0 ≤ i < N_f − 1 do
 6:    if (0 ≤ p_comp < pSet_i) then
 7:      SetFreq(f_0)
 8:    else if (pSet_i ≤ p_comp < pSet_{i+1}) then
 9:      SetFreq(f_i)
10:    else if (p_comp ≥ pSet_{N_f-1}) then
11:      SetFreq(f_{N_f-1})
12:    end if
13:    i ← i + 1
14:  end while
```

Figure 2.5 (b). Instead of fine-grained deployment of DVFS for setting appropriate CPU frequency to *Comm*-ESBs without exploiting energy saving opportunities from *Mem*-ESBs and *Disk*-ESBs, AGGREE aggressively sets CPU frequency to low once for the whole loop given that the loop is data intensive, which achieves higher energy efficiency than the basic DVFS scheduling strategy due to lower CPU power at the cost of minor performance and energy loss from the small proportion of computation. Moreover, AGGREE overcomes the excessive number of CPU frequency switches by croase-grained DVFS scheduling outside the loop.

2.2.4 Adaptively Aggressive DVFS Scheduling for Mem-ESB and Disk-ESB

Recall one of our goal is to reduce the performance loss from minor computation accompanying data intensive operations at low CPU frequency. One effective way

to moderate the low-performance trade-off from AGGREE for data intensive applications is to set an intermediate CPU frequency adaptively on case-by-case basis for *Mem*-ESBs and *Disk*-ESBs within such applications, instead of always employing the lowest CPU frequency during executions. We refer to this adaptively aggressive DVFS scheduling strategy as A2E. The heuristic of A2E is similar to AGGREE: For those ESBs with implicit boundaries, we specify an appropriate CPU frequency for them as a whole, since fine-grained DVFS scheduling upon the finish of separating non-computation from computation is difficult. Considering the code example shown in Figure 2.5 (b), AGGREE aggressively sets CPU frequency to the lowest possible value once outside the data intensive loop, while A2E calculates an intermediate CPU frequency adaptively according to the proportion of computation time among the total execution time of an ESB, and also aggressively sets the calculated frequency once for the ESB with mixed workloads. Algorithm 1 details the steps of employing A2E. For each ESB in the application, we empirically obtain in advance the proportion of computation time p_{comp} among the total execution time of the ESB. The A2E algorithm first broadcasts the p_{comp} of current ESB to all other ESBs and thus the highest p_{comp}, p'_{comp} can be used as a threshold for future reference. Given a set of CPU frequencies defined for DVFS, we divide the range of possible p_{comp} $[0, p'_{comp}]$ into N_f sub-ranges, where N_f is the number of available CPU frequencies. Which sub-range the p_{comp} of an ESB sits determines which CPU frequency to apply for the ESB. Figure 2.6 contrasts AGGREE and A2E using the same code example shown in Figure 2.5.

Example. Consider a data intensive application with 10 ESBs, among which the highest proportion of computation time within the total execution time is 20%, and there are four

gears of CPU frequency available for DVFS. According to Algorithm 1, the range of CPU frequency for adaptively aggressive DVFS scheduling consists of four individual sub-ranges from 0 to 20%, i.e., [0, 5%), [5%, 10%), [10%, 15%), and [15%, 20%]. If the proportion of computation time for an ESB is within the range of [0, 5%), we set CPU frequency to f_0, i.e., the lowest frequency; if the proportion falls into the range of [5%, 10%), we set CPU frequency to f_1, i.e., the second lowest frequency, and so on. Consequently for each ESB, we can assign a fitting frequency based on the amount of computation within the ESB.

Although the low-performance trade-off is moderated by A2E, the overhead on employing DVFS increase a bit due to more CPU frequency switches issued by A2E. From Figure 2.6, we can see that the number of CPU frequency switches approximates the number of ESBs in the `if` branch, since for each ESB, we at least set an appropriate CPU frequency for it once. For the code example shown in Figure 2.6, we do not need to switch CPU frequency for the ESB within the `else` branch, because we guarantee at the end of the `if` branch CPU frequency is set to high. Overall, the number of CPU frequency switches for A2E approximates NN_iP_1, comparable to that for the basic DVFS scheduling $2N_iN_m$, where N is the number of ESBs in the loop, N_i is the number of iterations of the loop, and N_m is the number of *Comm*-ESBs in the loop. Note that different types of workloads do not necessarily appear in a loop, we let $N_i = 1$ when hybrid workloads are not present in a loop, but in a code segment without loops. In this case, the number of CPU frequency switches for A2E dramatically decreases to NP_1 that is of the same order of magnitude as that for AGGREE. In other words, the DVFS overhead of A2E and AGGREE are comparable when different types of workloads are present in a code segement without loops.

```
1:  SetFreq(L_DVFS);
2:  while (caseA) {
3:    if (caseB) { P1
4:      ...
5:      communication();
6:      memory_access();
7:      disk_access();
8:      computation();
9:      communication();
10:     ...
11:   }
12:   else { P2 (P2 ≪ P1)
13:     ...
14:     SetFreq(H_DVFS);
15:     computation();
16:     SetFreq(L_DVFS);
17:     ...
18:   }
19: }
20: SetFreq(H_DVFS);
```

(a) Aggressive DVFS Scheduling with Speculation (AGGREE)

```
1:  SetFreq(L_DVFS);
2:  while (caseA) {
3:    if (caseB) { P1
4:      ...
5:      communication();
6:      SetFreq(M_DVFS);
7:      memory_access();
8:      SetFreq(M'_DVFS);
9:      disk_access();
10:     SetFreq(H_DVFS);
11:     computation();
12:     SetFreq(L_DVFS);
13:     communication();
14:     ...
15:   }
16:   else { P2 (P2 ≪ P1)
17:     ...
18:     SetFreq(H_DVFS);
19:     computation();
20:     SetFreq(L_DVFS);
21:     ...
22:   }
23: }
24: SetFreq(H_DVFS);
```

(b) Adaptively Aggressive DVFS Scheduling with Speculation (A2E)

Figure 2.6: AGGREE and A2E DVFS Scheduling for Typical Communication, Memory Access, and Disk Access Mixed Code with Imbalanced Branches.

2.2.5 Speculative DVFS Scheduling for Imbalanced Branches

Speculation is a technique that allows a compiler or a processor to predict the execution of an instruction so that an earlier execution of other instructions depending on the speculated instruction may be enabled. In our case, we speculate the outcome of a branching statement for energy saving purposes. If the application consists of conditional statements with significantly different possibility of occurrence (i.e., imbalanced branches) such as the `if-then-else` construct shown at Lines 2 and 15 in the kernel of an application with different workloads with imbalanced branches as depicted in Figure 2.5 (a). There are two branches to take where the taken possibility $\mathtt{P}_1$ of the `if` branch is much greater than that of the `else` branch $\mathtt{P}_2$, which is a real case for the benchmark DT from NPB. As shown in Figures 2.5 (b) and 2.6, we can speculatively set CPU frequency to low outside the rarely taken `else` branch inside the loop, and set CPU frequency to high for computation within the `else` branch, as a recovery mechanism used for incorrect speculation, so that the overall performance is not compromised even if the `else` branch is taken empirically.

Speculation can be applied to both AGGREE and A2E to reduce the number of CPU frequency switches for less DVFS overhead. Although in comparison to AGGREE with no speculation and A2E with no speculation, the use of speculation within both approaches slightly increases the number of CPU frequency switches by additional $2N_iP_2$ times, respectively. Overall, the speculative DVFS scheduling together with AGGREE and A2E effectively reduce the number of CPU frequency switches from $2N_iN_m$ of the basic DVFS scheduling strategy to $2 + 2N_iP_2$ and $NN_iP_1 + 2N_iP_2$ individually. Following the

constraint $P_2 \ll P_1$ due to the imbalanced branches, AGGREE with speculation is more effective on reducing DVFS overhead against A2E with speculation.

2.2.6 Performance Model

Next we model the performance efficiency of the three approaches (Basic DVFS, AGGREE, and A2E) at ESB level for a data intensive application. Since the application consists of different types of ESBs, performance efficiency of each ESB reflects the overall performance efficiency of the application. Table 2.1 lists the notation used in the formalization of performance. Given an application with different types of workloads comprised of computation (CPU-bound), communication (network-bound), memory accesses (memory-bound), and disk accesses (disk-bound), we model the performance of the original application without any DVFS scheduling strategies as sum of execution time of different components:

$$T = T_{comp} + T_{comm} + T_{mem} + T_{disk} \tag{2.1}$$

Let us assume the application is executed with the optimal efficiency (100%), T_{comp} can be represented as:

$$T_{comp} = \frac{O_{comp}}{f N_c N_F} \tag{2.2}$$

As we know, only CPU frequency f in calculating T_{comp} is affected by DVFS, while execution time of operations other than computation is bounded by non-CPU hardware factors such as network bandwith and disk data transfer rate, and is not related to CPU frequency. Therefore we separate the computation accompanying memory and disk accesses

Table 2.1: Notation in Performance Efficiency Formalization.

T	Total execution time of the application
T_{comp}	Computation time of the application
T_{comm}	Communication time of the application
T_{mem}	Average memory access time of the application
T_{disk}	Average disk access time of the application
O_{comp}	Time complexity of computation of the application
f	Current CPU operating frequency
f_h	A high CPU frequency set by DVFS
f_m	A medium CPU frequency set by DVFS adaptively
f_l	A low CPU frequency set by DVFS
N_c	Number of cores within one node of the cluster
N_F	Floating Point Unit of one core divided by 64-bit
T_{DVFS}	Time consumed by a DVFS CPU frequency switch
P_1	Taken possibility of the likely taken imblanced branch
P_2	Taken possibility of the rarely taken imblanced branch
N_i	Number of iterations of a loop with hybrid workloads
N	Number of ESBs in a hybrid loop/application
N_m	Number of *Comm*-ESBs in a hybrid loop/application

from the actual memory and disk accesses for each approach below, where T'_{mem} and T'_{disk} denote the actual memory and disk access time respectively, and the impact of DVFS on execution time is shown by setting different CPU frequencies. Note that each approach employs the same heuristic for energy efficient computation and communication: Keeping the highest CPU performance for computation and applying the lowest CPU performance for communication.

$$\begin{aligned} T_{orig} =& \frac{O_{comp}}{f_h N_c N_F} + T_{comm} + T'_{mem} + \frac{O_{comp\_mem}}{f_h N_c N_F} \\ &+ T'_{disk} + \frac{O_{comp\_disk}}{f_h N_c N_F} \end{aligned} \tag{2.3}$$

$$T_{basic} = \frac{O_{comp}}{f_h N_c N_F} + T_{comm} + T'_{mem} + \frac{O_{comp\_mem}}{f_h N_c N_F} + T'_{disk} + \frac{O_{comp\_disk}}{f_h N_c N_F} + T_{DVFS} \times 2N_i N_m \quad (2.4)$$

$$T_{aggree} = \frac{O_{comp}}{f_l N_c N_F} + T_{comm} + T'_{mem} + \frac{O_{comp\_mem}}{f_l N_c N_F} + T'_{disk} + \frac{O_{comp\_disk}}{f_l N_c N_F} + T_{DVFS} \times (2 + 2N_i P_2) \quad (2.5)$$

$$T_{a2e} = \frac{O_{comp}}{f_h N_c N_F} + T_{comm} + T'_{mem} + \frac{O_{comp\_mem}}{f_m N_c N_F} + T'_{disk} + \frac{O_{comp\_disk}}{f'_m N_c N_F} + T_{DVFS} \times (NN_i P_1 + 2N_i P_2) \quad (2.6)$$

Without loss of generality, given a data intensive application with different types of workloads and imblanced branches, since computation only takes a small proportion of the total execution time of the application, we assume $T_{comm} + T'_{mem} + T'_{disk} = n \times T_{comp} = \frac{nO_{comp}}{f_h N_c N_F}$, where $n > 1$. The last added items in Equations 2.4, 2.5, 2.6 are the overhead on employing DVFS. We know P_2 approximates to 0 since this branch is rarely taken, so the DVFS overhead is negligible for AGGREE. Additionally, from Table 5.2 we can see that in our experimental platform $f_h \approx 3f_l$ if we adopt Gear 0 as f_h and Gear 3 as f_l for AGGREE, and we assume $f_m = mf_l$ and $f'_m = m'f_l$. Thus we obtain the simplified formulae of performance for the three approaches as:

$$T_{orig} \approx \frac{(n+1)O_{comp} + O_{comp\_mem} + O_{comp\_disk}}{3f_l N_c N_F} \quad (2.7)$$

Table 2.2: Notation in Energy Efficiency Formalization.

E_{sys}	Total energy consumption of the whole cluster
E_{node}	Total energy consumption of all components in a node
P_{node}	Total power consumption of all components in a node
P_{CPU_d}	CPU dynamic power consumption in the busy state
P_{CPU_s}	CPU static/leakage power consumption in any states
P_{other}	Power consumption of components other than CPU
A	Percentage of active gates in the CMOS-based chip
C	Total capacitive load in the CMOS-based chip
V	Current CPU supply voltage
V_h	A high supply voltage set using DVFS
V_l	A low supply voltage set using DVFS
n	Time ratio between non-computation and computation

$$T_{basic} \approx \frac{(n+1)O_{comp} + O_{comp\_mem} + O_{comp\_disk}}{3f_l N_c N_F} + T_{DVFS} \times 2N_i N_m \tag{2.8}$$

$$T_{aggree} \approx \frac{(\frac{n}{3}+1)O_{comp} + O_{comp\_mem} + O_{comp\_disk}}{f_l N_c N_F} + T_{DVFS} \times (2 + 2N_i P_2) \tag{2.9}$$

$$T_{a2e} = \frac{(\frac{n+1}{3})O_{comp} + \frac{1}{m}O_{comp\_mem} + \frac{1}{m'}O_{comp\_disk}}{f_l N_c N_F} + T_{DVFS} \times (NN_i P_1 + 2N_i P_2) \tag{2.10}$$

From the comparison between Equations 2.7, 2.8, 2.9, and 2.10, we can see that against the original application without any DVFS strategies, the basic DVFS scheduling strategy only results in performance loss due to additional DVFS overhead, while both

Table 2.3: Frequency-voltage Pairs for the AMD Opteron 2380 Processor.

Gear	Frequency (GHz)	Voltage (V)
0	2.5	1.35
1	1.8	1.2
2	1.3	1.1
3	0.8	1.025

AGGREE and A2E incur performance loss from reducing CPU performance during computation. Compared to AGGREE, performance loss from A2E is moderated, since each coefficient of computation time complexity of A2E is smaller than that of AGGREE. Moreover, A2E suffers from DVFS overhead comparable to the basic DVFS scheduling strategy, while AGGREE has the minimal overhead on using DVFS due to the constraint $P_2 \ll P_1$ for imbalanced branches.

2.2.7 Energy Model and Energy Efficiency Analysis

We next formalize energy saving opportunities provided by the three energy efficient approaches individually using the notation in Table 2.2. Within a given time interval (t_1, t_2), the total energy costs of a distributed-memory computing system consisting of multiple computing nodes can be formulated as below, where we denote the execution time as $T = t_2 - t_1$ and the nodal average power consumption as $\overline{P_{node}}$:

$$E_{sys} = \sum_{1}^{\#nodes} E_{node} = \sum_{1}^{\#nodes} \int_{t_1}^{t_2} P_{node} dt = \sum_{1}^{\#nodes} \overline{P_{node}} \times T \tag{2.11}$$

Assuming each node in the computing system has the same hardware configuration and local energy efficiency results in global energy efficiency according to Equation 2.11, we only consider nodal energy consumption in the later discussion. Generally, we break down nodal power consumption as:

$$P_{node} = P_{CPU\_d} + P_{CPU\_s} + P_{other};\ P_{CPU\_d} \approx ACfV^2 \tag{2.12}$$

In (4.3), we categorize the nodel power consumption by power consumption of CPU and other components. By substituting P_{CPU_d}, we obtain the ultimate nodal power consumption formula with DVFS-dependent parameters f and V as:

$$P_{node} \approx ACfV^2 + P_{CPU\_s} + P_{other} \tag{2.13}$$

In our case, P_{CPU_s} and P_{other} barely change during the execution and thus we denote $P_{CPU\_s} + P_{other}$ as a constant P_c. From Equation 2.9, we know that the DVFS overhead of AGGREE is negligible due to the presence of P_2. Following the constraints of Equations 2.11, 4.3, and 4.6, we can calculate energy costs of running a data intensive application with different DVFS scheduling strategies respectively. Further, we model the energy savings achieved by AGGREE and A2E in contrast to the original application individually as below:

$$\begin{aligned}\Delta E_{aggree} &= E_{node}^{orig} - E_{node}^{aggree} = \overline{P_{node}^{orig}} \times T_{orig} - \overline{P_{node}^{aggree}} \times T_{aggree} \\ &\approx (ACf_hV_h^2 + P_c)T_{orig} - (ACf_lV_l^2 + P_c)T_{aggree}\end{aligned} \tag{2.14}$$

$$\begin{aligned}\Delta E_{a2e} &= E_{node}^{orig} - E_{node}^{a2e} = \overline{P_{node}^{orig}} \times T_{orig} - \overline{P_{node}^{a2e}} \times T_{a2e} \\ &\approx (ACf_hV_h^2 + P_c)T_{orig} - (\overline{ACfV^2 + P_c})T_{a2e}\end{aligned} \tag{2.15}$$

From Equations 2.14 and 2.15, we observe that there exists a performance-energy trade-off that should be considered to determine the optimal CPU frequency to employ

in different requirements. In our scenario, achieving the maximal energy savings with minor performance loss is the goal. For evaluating if the energy efficiency achieved and the performance degradation incurred are balanced, we adopt an integrated metric to quantify the energy-performance efficiency: Energy-Delay Product (EDP), a widely used metric to weigh the comprehensive effects of energy and performance for a given application under different configurations [71]. Therefore, we leverage the EDP metric and its variant ED2P to evaluate among the three energy efficient approaches, which one is able to achieve the optimal energy-performance efficiency (the smaller value, the better efficiency) for data intensive applications. Details of the implementation and evaluation of all three energy efficient approaches are illustrated next.

2.3 Implementation and Evaluation

We have implemented all three energy efficient DVFS scheduling strategies and evaluated their effectiveness towards five high performance data intensive applications with different dominant workloads such as memory and disk accesses with imbalanced branches. Instead of assigning appropriate CPU frequencies to an ESB with differet types of workloads in a fine-grained fashion, we aggressively schedule CPU frequency to an intermediate value for memory and disk accesses mixed with minor computation adaptively according to the proportion of computation time among the total execution time, in order to achieve considerable energy savings at the cost of minor performance loss. As for imbalanced branches, we adopt speculative DVFS scheduling to reduce the number of CPU frequency switches to minimize the overhead on employing DVFS. The DVFS technique in our approach through

modifying CPU frequency configuration files dynamically at system level enables us to scale CPU voltage and frequency up and down if necessary for energy efficiency. Benchmarks used consist of various sources of memory and disk access intensive programs with imbalanced branches, such as DT and MG from NPB and ASC benchmark suites [20] [1], an MPI version of the high-quality data compressor bzip2 [23], and an self-written MPI version of the Linux standard file copy command cp [16]. Table 6.1 shows the details of benchmarks.

2.3.1 Experimental Setup

We applied the three DVFS scheduling approaches individually to the five benchmarks to assess their effectiveness of energy savings and performance loss trade-off. Experiments were performed on a computing cluster with an Ethernet switch consisting of 8 computing nodes with two Quad-core 2.5 GHz AMD Opteron 2380 processors (totalling 64 cores) and 8 GB RAM running 64-bit Linux kernel 2.6.32, The power-aware and DVFS-enabled cluster was equipped with power sensors and meters for energy measurement. In our experiments, time was measured using the MPI_Wtime() routine. Energy consumption was measured using PowerPack [71], a comprehensive software and hardware framework for energy profiling and analysis of high performance systems and applications. The range of CPU frequency on HPCL was {0.8, 1.3, 1.8, 2.5} GHz. PowerPack was deployed and running at a meter node within the cluster to collect energy costs on all involved components such as CPU, memory, disk, motherboard, etc. on all 8 computing nodes of the cluster. The collected energy information was recorded into a log file in the local disk and accessed after execution of these benchmarks.

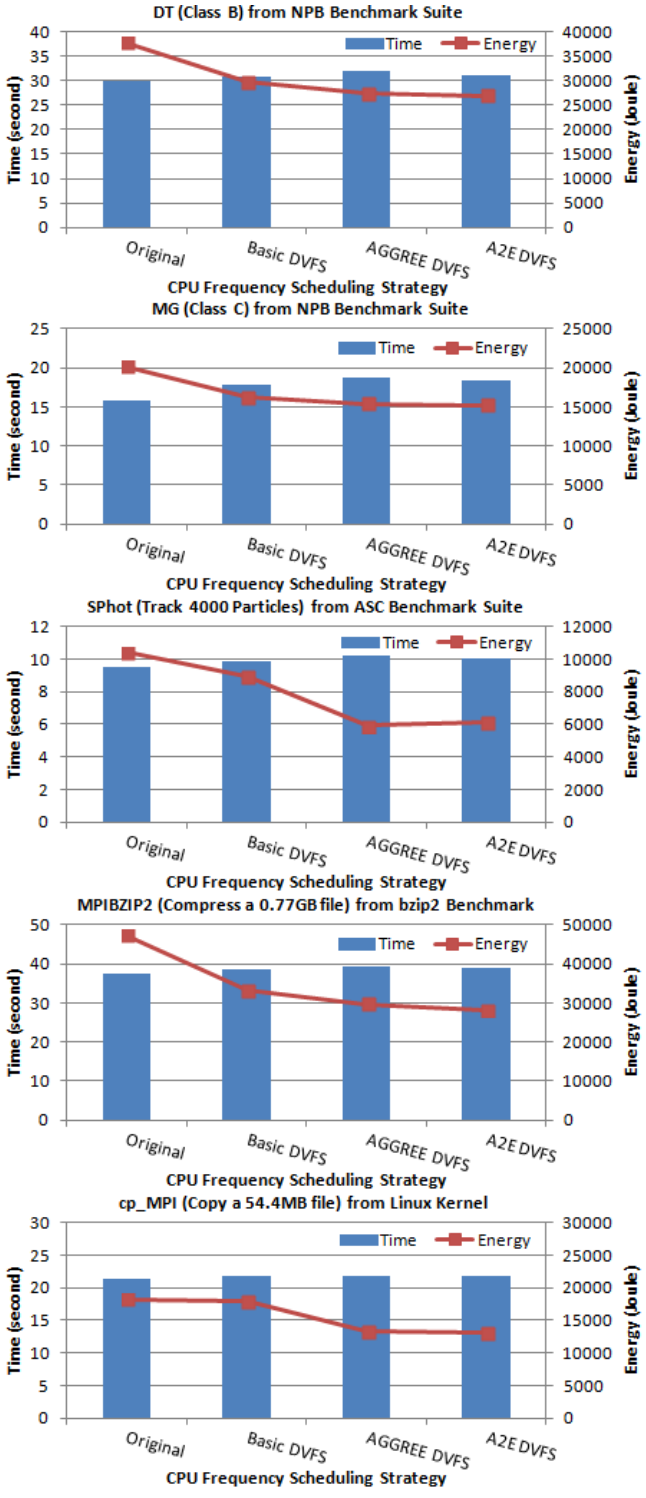


Figure 2.7: Performance Loss and Energy Savings on a Cluster with 8 Nodes, 64 Cores of {0.8, 1.3, 1.8, 2.5} GHz CPU frequencies, and 8 GB Memory/Node.

Table 2.4: Benchmark Details.

Benchmark	Source	Test Case	Category
DT	NPB	Class B	Memory Access Intensive and Imbalanced Branches
MG	NPB	Class C	Memory and Disk Access Intensive
SPhot	ASC	Track 4000 particles	Memory Access Intensive
MPIBZIP2	bzip2	Compress a 0.77GB file	Disk Access Intensive
cp_MPI	Linux	Copy a file of 54.4MB	Disk Access Intensive

2.3.2 Performance Degradation

All three DVFS scheduling approaches improve energy efficiency for data intensive applications at the cost of minor performance loss as shown in Figure 2.7, where the x axis label Original denotes the original application without any DVFS scheduling strategies, and Basic DVFS, AGGREE DVFS, and A2E represent the basic, AGGREE, and A2E DVFS scheduling strategies introduced in the last section, respectively, where speculation is applied to both AGGREE and A2E. We can see that in general A2E incurs similar performance loss as the basic DVFS scheduling strategy compared to the original no-DVFS executions: 6.2% and 4.7% on average, respectively, while AGGREE degrades performance more (8.1% on average) due to aggressively lowering down CPU performance regardless of minor computation within the data intensive application.

The overhead on employing DVFS in the basic DVFS scheduling strategy primarily results from two factors: (a) The additional time spent on modifying CPU frequency configuration files dynamically at system level and (b) CPU frequency transition latency.

Thus the number of CPU frequency switches by DVFS determines the DVFS overhead. Some application such as MG incurs up to 13.0% performance loss due to applying DVFS, since there exist a great amount of alternate *Comm*-ESBs and *Comp*-ESBs as shown in Figure 2.5 (a), which requires a large amount of CPU frequency switches by DVFS as well. The communication time for some application like cp_MPI is negligible and the amount of *Comm*-ESBs is limited. Therefore constrained by both factors, the overhead on employing DVFS for cp_MPI is also negligible (1.5%).

As discussed before, performance loss from AGGREE is attributed to low performance of the small proportion of computation mixed with memory and disk accesses. According to Equation 2.5, reducing CPU frequency aggressively results in longer execution time for the CPU-bound computation and thus incurs overall performance degradation, although performance of memory and disk accesses is barely affected. Since the ratio between computation and non-computation is significantly low in memory and disk access intensive applications, the impact of performance loss from computation is limited on the total execution time. A2E further decreases the performance loss by adaptively scheduling an appropriate CPU frequency to an ESB according to the computation time proportion instead of always setting the lowest CPU frequency. On the other hand, AGGREE and A2E successfully reduce the number of CPU frequency switches by applying DVFS outside of a loop of ESBs and inside a rarely taken branch, respectively. The DVFS overhead is reduced accordingly compared to the basic DVFS scheduling strategy where there exist a larger number of CPU frequency switches due to fine-grained DVFS scheduling before and after each *Comm*-ESB within the loop. Consequently, with less DVFS overhead, AGGREE

and A2E only suffer from 3.4% and 1.5% more performance loss than that of the basic DVFS scheduling strategy, respectively.

2.3.3 Energy Savings for Memory Access Intensive Applications

Figure 2.7 also reflects energy efficiency achieved by the three approaches. Compared to the basic and AGGREE DVFS scheduling strategies, A2E is able to achieve more energy savings, since energy saving opportunities from memory and disk accesses that the basic DVFS scheduling strategy fails to leverage are exploited by AGGREE and A2E as depicted in Figures 2.5 (b) and 2.6 (b) respectively, and further moderation of low-performance trade-off is performed by A2E against AGGREE. Specifically, considering energy consumption of the original executions as the baseline, 32.6% on average energy savings are fulfilled by A2E, in contrast to 17.3% and 31.7% energy savings on average achieved by the basic and AGGREE DVFS scheduling strategies individually.

The most energy savings 43.4% AGGREE achieves is for the memory access intensive application SPhot, while A2E manages to achieve less energy savings 40.9%. We applied AGGREE and A2E to a code segment within SPhot, where a great amount of memory accesses mixed with calculating array indices before accessing corresponding memory locations are present in a double-loop. The basic DVFS scheduling strategy only obtains 13.9% energy savings, since it fails to handle *Mem*-ESBs accompanied by computation but only saves energy for *Comm*-ESBs. With AGGREE employed, performance of SPhot is degraded by 7.2% due to low performance of memory address calculation interleaved in memory accesses as a consequence of aggressively scaling down CPU frequency. Performance loss is moderated by A2E to 4.9% at the cost of less energy savings, since the *Mem*-ESBs of SPhot have

similar proportion of computation time and thus most CPU frequencies adaptively assigned are close to the highest one.

2.3.4 Energy Savings for Disk Access Intensive Applications

Besides memory access intensive applications, A2E performs better than the other two approaches in gaining energy efficiency for disk access intensive applications. Regarding the disk access dominant application cp_MPI, the basic DVFS scheduling strategy saves a limited amount of energy (2.1%) since the communication time is significant low compared to the disk access time. AGGREE and A2E can obtain more energy savings for this type of applications, since aggressively reducing CPU frequency barely affects performance of the application. As for cp_MPI, most execution time is spent on non-CPU-bound opertions, disk accesses, whose execution time is constrained by non-CPU hardware factors such as average seek time and disk data transfer rate. Low CPU performance brings in considerable energy savings from CPU during data waiting time without significant performance loss as a whole. Another disk access intensive application MPIBZIP2 also benefits from the moderation of low-performance loss by A2E with 40.5% energy savings achieved compared to 37.1% from AGGREE.

Note that although similar percentage of energy savings are fulfilled for memory access intensive and disk access intensive applications, performance degradation for employing aggressive DVFS scheduling strategies like AGGREE and A2E towards the two types of applications differ: Despite MG, an application with comparable memory and disk accesses, memory access intensive applications (DT and SPhot) suffer from average performance loss

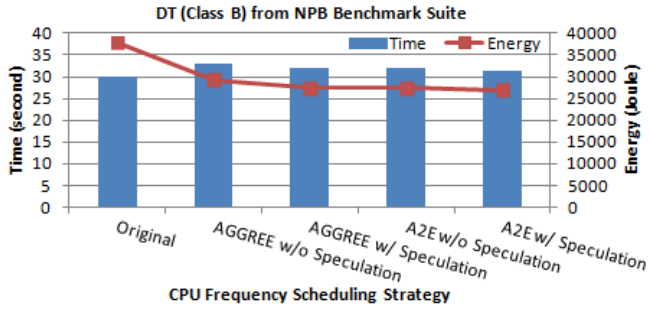


Figure 2.8: Performance and Energy Efficiency upon Employing Speculation in AGGREE and A2E for the DT Benchmark with Imbalanced Branches.

of 7.2% for AGGREE and 4.7% for A2E, while disk access intensive applications (MPIBZIP2 and cp_MPI) only sacrifice minor performance loss on average of 3.4% for AGGREE and 2.7% for A2E. This is attributed to two causes: (a) Memory access time is much smaller than disk access time (typically with a ratio of the order of magnitude $1/10^6$) and thus is closer to CPU clock cycles; (b) The amount of computation mixed with memory accesses is generally more than that with disk accesses. Both reasons make the impact of CPU performance degradation on the total execution time of memory access intensive applications greater than that of disk access intensive applications. It is notable that moderating performance loss from A2E shrinks the gap.

2.3.5 Energy Savings for Imbalanced Branches

AGGREE and A2E adopt speculation to further gain energy efficiency for code with imbalanced branches by reducing the DVFS overhead. DT is a memory intensive graph application where a great amount of imbalanced branches exist. Figure 2.8 shows energy consumption and execution time of DT using AGGREE and A2E with and without

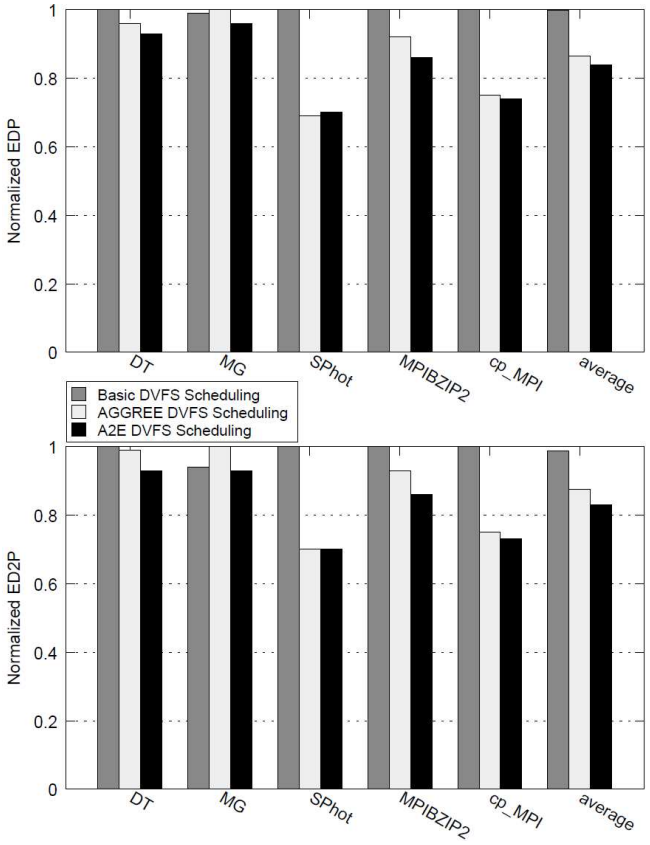


Figure 2.9: Energy-Performance Efficiency Trade-off in Terms of EDP and ED2P.

speculation individually. We can see that employing speculation within AGGREE and A2E mitigates performance degradation and thus saves energy: Performance loss from AGGREE drops from 10.8% to 7.1%, while energy savings increase from 22.7% to 27.4%; performance loss from A2E drops from 6.8% to 4.4%, while energy savings increase from 27.3% to 28.6%. The effectiveness of speculation for saving time and energy results from aggressively reducing CPU frequency for the frequently taken branch while keeping CPU frequency high for computation within the rarely taken branch as the recovery mechanism used for incorrect speculation, as shown in Figures 2.5 (b) and 2.6. Note that A2E is empirically less effective than AGGREE in reducing the DVFS overhead upon the use of speculation, which is consistent with the performance loss from employing DVFS calculated formally in Section 2.2.

2.3.6 Energy and Performance Efficiency Trade-off

From Equations 2.14 and 2.15, we observe there exists an energy-performance efficiency trade-off for AGGREE and A2E. In general, moderating CPU performance degradation by adaptively scheduling an intermediate rather than always the lowest CPU frequency for a *Mem*-ESB or a *Disk*-ESB decreases performance loss at the cost of higher average power, since power is proportional to CPU frequency and voltage. Variation of performance and energy efficiency at different operating points can be quantified by an integrated metric that both impacts of performance and energy are considered. We adopt the EDP (Energy-Delay Product) metric and its variant ED2P (Energy-Delay-Squared Product) to evaluate the balance between energy and performance efficiency for the three energy saving approaches, as presented in Figure 2.9.

Since a smaller value in the EDP and ED2P metrics represents higher energy and performance efficiency as a whole, we can see that for data intensive applications, the basic DVFS scheduling strategy is not the optimal approach since it fails to exploit the energy saving opportunities present in the operations other than communication. Except that for SPhot, A2E and AGGREE have similar EDP and ED2P values, A2E is superior to AGGREE for all other applications in terms of the balance of energy-performance efficiency. The average values of EDP and ED2P for A2E and AGGREE over all five benchmark consolidate this observation.

2.4 Summary

Driven by the growing energy concerns, DVFS techniques have been widely applied to improve energy efficiency for high performance applications on distributed-memory computing systems nowadays. Energy saving opportunities from slack in terms of load imbalance, network latency, communication dalay, memory and disk access stalls, etc. are exploited to save energy through scaling up and down CPU voltage and frequency via DVFS, since peak CPU performance is not necessary during the slack. We propose an adaptively aggressive energy efficient DVFS scheduling strategy (A2E) for data intensive applications such as memory and disk access intensive applications with imbalanced branches. Instead of assigning CPU frequency in a fine-grained fashion towards an Energy Saving Block (ESB) with different types of workloads, A2E adaptively schedules an appropriate CPU frequency for the hybrid ESB aggressively as a whole and reduces the overhead on employing DVFS via speculation to save energy with minor performance loss. The experimental results in-

dicate the effectiveness of A2E for saving energy of running target applications with minor performance loss.

Chapter 3

HP-DAEMON: *H*igh *P*erformance *D*istributed *A*daptive *E*nergy-efficient *M*atrix-multiplicati*ON*

With the trend of increasing number of interconnected processors providing the unprecedented capability for large-scale computation, despite exploiting parallelism for boosting performance of applications running on high performance computing systems, the demands of improving their energy efficiency arise crucially, which motivates holistic approaches from hardware to software. Software-controlled hardware solutions [69] [67] [124] [116] of improving energy efficiency for high performance applications have been rec-

ognized as effective potential approaches, which leverage different forms of slack in terms of non-overlapped latency [127] to save energy.

For applications running on distributed-memory architectures, Dynamic Voltage and Frequency Scaling (DVFS) [141] has been leveraged extensively to save energy for components such as CPU, GPU, and memory, where the performance of the components is modified by altering its supply voltage and operating frequency. Reduction on supply voltage generally results in decrease of operating frequency as a consequence, and vice versa. Given the assumption that dynamic power consumption P of a CMOS-based processor is proportional to product of working frequency f and square of supply voltage V, i.e., $P \propto fV^2$ [108] [86], and also existing work that indicates energy costs on CPU dominate the total system energy consumption [71], DVFS is deemed an effective software-based dynamic technique to reduce energy consumption on a high performance computing system. For instance, DVFS can be leveraged to reduce CPU frequency if the current operation is not CPU-bound. In other words, execution time of the operation will barely increase if CPU frequency is scaled down. CPU frequency is kept at the highest scale if performance of the operation is harmed by decreasing CPU frequency. Energy savings can thus be achieved due to lower CPU frequency as wells as supply voltage with negligible performance loss. As a fundamental component of most numerical linear algebra algorithms [50] employed in high performance scientific computing, state-of-the-art algorithms of matrix multiplication on a distributed-memory computing system perform alternating matrix broadcast and matrix multiplication on local computing nodes with a local view [49]. Given that the communication in distributed matrix multiplication is not bound by CPU frequency

while the computation is, one classic way to achieve energy efficiency for distributed matrix multiplication is to set CPU frequency to low for broadcast while set it back to high for multiplication [46] [93]. In general, considerable energy savings can be achieved from the low-power communication phase.

Although employing DVFS is beneficial to saving energy via software-controlled power-aware execution, introducing DVFS itself can cost non-negligible energy and time overhead from two aspects. Firstly, using DVFS in our approach is via dynamically modifying CPU frequency configuration files that are essentially in-memory temporary system files, for setting up appropriate frequencies at OS level. It incurs considerable memory access overhead if there exist a large number of such virtual file read and write operations for switching CPU frequency. Secondly, CPU frequency (a.k.a., gear [67]) transition latency required for taking effect (on average 100μs for AMD Athlon processors and 38μs for AMD Opteron 2380 processors employed in this work) results in additional energy costs, since a processor has to stay in use of the old frequency while switching to the newly-set frequency is not complete. We need to either minimize the time spent on memory accesses for switching frequency, i.e., the latency required for changing the gears successfully, or reduce the number of frequency switches to save energy. It is challenging to reduce the first type of time costs, since it depends on hardware-related factors such as memory accessing rate and gear transition latency. Alternatively, a software-controlled energy efficient DVFS scheduling strategy to reduce frequency switches is thus desirable.

Numerous efforts have been conducted on devising energy efficient DVFS-directed solutions, but few of them concern possible non-negligible overhead incurred from employing

DVFS. In this work, we propose an adaptive DVFS scheduling strategy with a high performance communication scheme via pipeline broadcast to achieve the optimal energy and performance efficiency for distributed matrix multiplication, named *HP-DAEMON*. Firstly, we propose a memory-aware mechanism to reduce DVFS overhead, which adaptively limits the number of frequency switches by grouping communication and computation respectively, at the cost of memory overhead within a certain user-specified threshold. Further, we take advantage of a pipeline broadcast scheme with tuned chunk size to boost performance of communication, with which network bandwidth is exploited thoroughly compared to binomial tree broadcast.

In summary, the contributions of this chapter are as follows:

- We propose an adaptive DVFS scheduling strategy aware of memory overhead in distributed matrix multiplication for reducing the number of CPU frequency switches to minimize DVFS overhead, i.e., algorithmically performing more computation and communication at a time for less DVFS switches, which saves 7.5% extra average energy compared to a classic strategy;

- We analyze the impact on performance of chunk size in pipeline broadcast to achieve communication speed-up in distributed matrix multiplication;

- Our integrated approach is evaluated to achieve up to 51.4% energy savings and on average 28.6% speed-up compared to ScaLAPACK [26] on an Ethernet-switched cluster, and achieve on average 33.3% and 32.7% speed-up compared to ScaLAPACK and DPLASMA [8] on an Infiniband-switched cluster, respectively.

The rest of this chapter is organized as follows. Section 3.2 introduces distributed matrix multiplication. We present an adaptive DVFS scheduling strategy in Section 3.3 and a high performance pipeline broadcast communication scheme in Section 3.4. We provide details of implementation and evaluation in Section 3.5. Section 3.6 summarizes.

3.1 Distributed Matrix Multiplication

Matrix multiplication is one fundamental operation of most numerical linear algebra algorithms for solving a system of linear equations, such as Cholesky, LU, and QR factorizations [50], where runtime percentage of matrix multiplication can be up to 92% [126]. Moreover, nowadays matrix multiplication has been widely used in many areas other than scientific computing, including computer graphics, quantum mechanics, game theory, and economics. In scientific computing, various software libraries of numerical linear algebra for distributed multi-core high performance scientific computing (ScaLAPACK [26] and DPLASMA [8]) have routines of various functionality where matrix multiplication is involved. In this work, we aim to achieve the optimal energy and performance efficiency for distributed matrix multiplication in general. Our approach works at library level and thus serves as a cornerstone of saving energy and time for other numerical linear algebra operations where matrix multiplication is intensively employed.

3.1.1 Algorithmic Details

The matrix multiplication routines from ScaLAPACK/DPLASMA are essentially derived from DIMMA (Distribution-Independent Matrix Multiplication Algorithm), an ad-

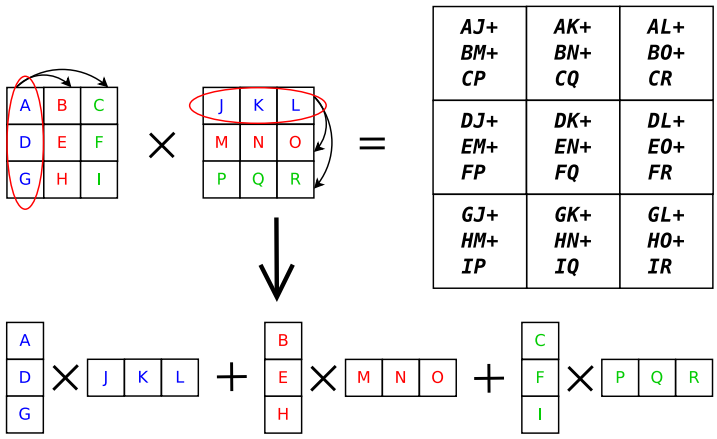
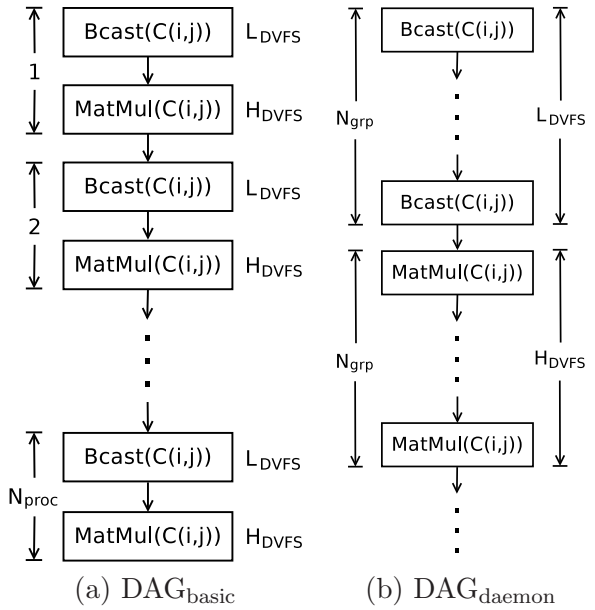


Figure 3.1: A Distributed Matrix Multiplication Algorithm with a Global View.

vanced version of SUMMA (Scalable Universal Matrix Multiplication Algorithm) [49]. The core algorithm consists of three steps: (a) Partition the global matrix into the process grid using load balancing techniques, (b) broadcast local sub-matrices in a row-/column-wise way as a logical ring, and (c) perform local sub-matrix multiplication. Applying an optimized communication scheme, DIMMA outperforms SUMMA by eliminating slack from overlapping computation and communication effectively. Next we illustrate DIMMA using Directed Acyclic Graph (DAG) representation.

3.1.2 DAG Representation

In general, a task-parallel application such as distributed matrix multiplication can be partitioned into a cluster of computing nodes for parallel execution. The method of partitioning greatly affects the outcomes of energy and performance efficiency. During task partitioning, data dependencies between tasks must be respected for correctness. When the application is partitioned, data dependencies between distributed tasks can be represented



(a) $\text{DAG}_{\text{basic}}$ (b) $\text{DAG}_{\text{daemon}}$

Figure 3.2: Matrix Multiplication DAGs with Two DVFS Scheduling Strategies.

using DAGs, which characterize parallel executions of tasks effectively. A formal definition of DAGs is given below:

Definition 1. Data dependencies between tasks from partitioning a task-parallel application are modeled by a Directed Acyclic Graph (DAG) $G = (V, E)$, where each node $v \in V$ denotes a task, and each directed edge $e \in E$ denotes a dynamic data dependency between the tasks.

Next we show how a task-parallel application is partitioned to achieve parallelism and represented in DAG, taking distributed matrix multiplication for example. Consider multiplying of a $m \times k$ matrix A and a $k \times n$ matrix B, to produce a $m \times n$ matrix C. For calculating a matrix element (i, j) in C, denoted as $C(i, j)$, we apply a cache efficient blocking method, where columns of A multiply rows of B to reduce cache misses, as shown in Figure 3.1. Recall that distributed matrix multiplication requires alternating

matrix broadcast and matrix multiplication, as two DAGs shown in Figure 3.2. Each DAG represents an execution of calculating $C(i, j)$ with a DVFS scheduling strategy, where Figure 3.2 (a) gives the DVFS scheduling strategy employed in [46] [93], and the adaptive DVFS scheduling strategy proposed in this work is shown in Figure 3.2 (b). Given matrices A and B, each matrix row needs to be broadcasted to other rows located in different nodes and likewise each matrix column needs to be broadcasted to other nodes, such that sub-matrices of the resulting matrix C are calculated locally and accumulated to C globally. As the strategy shown in Figure 3.2 (a), each broadcast step is followed by a multiplication step alternatingly until all sub-matrices of A and B involved in calculating $C(i, j)$ are broadcasted and computed, where $\mathsf{Bcast}(C(i, j))$ denotes step-wise broadcasting sub-matrices of A and B that are involved in calculating $C(i, j)$, and $\mathsf{MatMul}(C(i, j))$ denotes step-wise multiplying and accumulating these sub-matrices of A and B that are broadcasted.

3.2 Adaptive Memory-aware DVFS Scheduling Strategy

Next we present an adaptive memory-aware DVFS scheduling strategy (referred to as *DAEMON* henceforth) that limits the overhead on employing DVFS, at the cost of user-specified memory overhead threshold, which determines the extent of DVFS overhead. The heuristic of *DAEMON* is straightforward: Combine multiple broadcasts/multiplications as a group to reduce the number of frequency switches by DVFS, i.e., trade more computation and communication at a time with the memory cost trade-off for less DVFS switches, as shown in Figure 3.2 (b). Instead of performing a multiplication immediately after a broadcast, we keep broadcasting several times as a group, followed by corresponding mul-

tiplications as a group as well. Note that the number of broadcasts in a broadcast group must equal the number of multiplications in a multiplication group to guarantee the correctness. The number of DVFS switches is thus decreased since we only need to perform one DVFS switch for a group rather than individual broadcast/multiplication. Table 4.1 lists the notation used in Figure 3.2 and the later text.

As shown in Figure 3.2 (a), the basic DVFS scheduling strategy sets CPU frequency to low during broadcast while sets it back to high during matrix multiplication for energy efficiency [46] [93]. A primary defect of this strategy is that it requires two DVFS switches in one iteration, totally $2 \times N_{proc}$ DVFS switches for one process. For performance purposes in high performance computing, block-partitioned algorithms are widely adopted to maximize data reuse opportunities by reducing cache misses. In a blocked distributed matrix multiplication such as the pdgemm() routine provided by ScaLAPACK, if the basic strategy is applied, the total number of DVFS switches is $2 \times N_{proc} \times \frac{N/N_{proc}}{BS} \times N_{proc}^2$, since there are $\frac{N/N_{proc}}{BS}$ pairs of local communication and computation for each process and totally N_{proc}^2 processes in the process grid, where N/N_{proc} is the local matrix size. Given a huge number of DVFS switches, the accumulated time and energy overhead on employing DVFS can be considerable. Next we introduce details of *DAEMON* to minimize the DVFS overhead and thus achieve the optimal energy savings.

3.2.1 Memory-aware Grouping Mechanism

Intuitively, the heuristic of *DAEMON* for grouping broadcasts/multiplications requires for each process, keeping several sub-matrices of A and B received from broadcasts of

Table 3.1: Notation in the Adaptive Memory-aware DVFS Scheduling Strategy.

N	Global matrix size in blocked distributed matrix multiplication
BS	Block size in blocked distributed matrix multiplication
H_{DVFS}	The highest CPU frequency set by DVFS
L_{DVFS}	The lowest CPU frequency set by DVFS
N_{proc}	Square root of the total number of processes in a process grid
N_{grp}	Number of broadcasts/multiplications executed at a time as a group
S_{mem}	The total system memory size for one node
T_{mem}	A user-specified memory cost threshold for grouping, in terms of a % of S_{mem}
e_{DVFS}^{unit}	Energy consumption from one frequency switch
e_{DVFS}^{basic}	Energy consumption from the basic strategy employed in [46] [93]
e_{DVFS}^{daemon}	Energy consumption from *DAEMON* proposed in this work

other processes in memory for later multiplications at a time. *DAEMON* restricts the memory costs from holding matrices in memory for future computation to a certain threshold, which can be modeled as:

$$\begin{gathered} 8 \times \left(\frac{N}{N_{proc}}\right)^2 \times 2 \times N_{grp} \times N_{proc} \leq S_{mem} \times T_{mem} \\ \text{subject to } 1 \leq N_{grp} \leq N_{proc} \end{gathered} \tag{3.1}$$

where 8 is the number of bytes used by a double-precision floating-point number and $\frac{N}{N_{proc}}$ is the local matrix size. For each process, we need to keep totally $2 \times N_{grp}$ submatrices of A and B in the memory of one node for N_{grp} broadcasts and N_{grp} multiplications performed at a time, and there are N_{proc} processes for one node. Following the constraints of Equation 3.1, we can calculate the optimal N_{grp} from a memory cost threshold value T_{mem} for specific hardware.

3.2.2 DAEMON Algorithm

We next show how *DAEMON* reduces DVFS overhead via grouping. In accordance with Equation 3.1, given a memory cost threshold T_{mem} and a specific hardware configuration, the optimal N_{grp} that determines the extent of grouping can be calculated. At group level, CPU frequency is then set to low for N_{grp} times grouped broadcasts and set back to high for N_{grp} times grouped multiplications at a time, instead of being switched for individual broadcast/multiplication. Consequently, the number of CPU frequency switches are greatly decreased by *DAEMON*.

Algorithm 1 presents detailed steps of calculating N_{grp} and then employ DVFS at group level. It first calculates N_{grp} using the user-specified threshold T_{mem}, and then set frequency accordingly for grouped broadcasts/multiplications, where the number of DVFS switches is minimized. Variable $freq$ denotes current operating CPU frequency, and functions GetSysMem(), GetMemTshd(), SetFreq(), IsBcast(), and IsMatMul() were implemented to get the total system memory size, get memory cost threshold specified by the user, set specific frequencies using DVFS, and determine if the current operation is either a broadcast or a multiplication, respectively.

3.2.3 Energy Efficiency Analysis

DAEMON minimizes DVFS overhead by reducing the number of frequency switches, at the cost of memory overhead within a user-specified threshold T_{mem}. The optimal value of N_{grp} for minimizing DVFS overhead can be calculated from the threshold, which determines the extent of grouped broadcasts/multiplications for less frequency switches. Per Equation

Algorithm 1 *Adaptive Memory-aware DVFS Scheduling Strategy*

SetDVFS(N, N_{proc})
1: $S_{mem} \leftarrow$ GetSysMem()
2: $T_{mem} \leftarrow$ GetMemTshd()
3: $unit \leftarrow N/N_{proc}$
4: $N_{grp} \leftarrow S_{mem} \times T_{mem}/(8 \times unit^2 \times 2)$
5: $nb \leftarrow N_{proc}/N_{grp}$
6: **foreach** $i < nb$ **do**
7: **if** (IsBcast() && $freq$!= L_{DVFS}) **then**
8: SetFreq(L_{DVFS})
9: **end if**
10: **if** (IsMatMul() && $freq$!= H_{DVFS}) **then**
11: SetFreq(H_{DVFS})
12: **end if**
13: **end for**

3.1, for blocked distributed matrix multiplication, energy costs on employing DVFS in the basic strategy and in our *DAEMON* strategy shown in Figure 3.2 are modeled respectively, in terms of the product of unit energy costs on one DVFS switch and the number of such switches as follows:

$$e_{DVFS}^{basic} = e_{DVFS}^{unit} \times 2 \times N_{proc} \times \frac{N/N_{proc}}{BS} \times N_{proc}^2 \tag{3.2}$$

$$e_{DVFS}^{daemon} = e_{DVFS}^{unit} \times 2 \times \frac{N_{proc}}{N_{grp}} \times \frac{N/N_{proc}}{BS} \times N_{proc}^2 \tag{3.3}$$

According to Equations 3.2 and 3.3, we derive energy deflation (i.e., *ratio*) and energy savings (i.e., *difference*) from employing *DAEMON* against the basic strategy as below:

$$E_{def} = \frac{e_{DVFS}^{basic}}{e_{DVFS}^{daemon}} = N_{grp} \tag{3.4}$$

$$\begin{aligned} E_{sav} &= e_{DVFS}^{basic} - e_{DVFS}^{daemon} \\ &= e_{DVFS}^{unit} \times 2 \times \frac{N}{BS} \times N_{proc}^2 \times \left(1 - \frac{1}{N_{grp}}\right) \end{aligned} \tag{3.5}$$

From Equations 3.4 and 3.5, we can see that both E_{def} and E_{sav} greatly depend on the value of N_{grp}. The greater N_{grp} is, the more energy efficient *DAEMON* is. Following the constraints of Equation 3.1, we know in the best case that $N_{grp} = N_{proc}$, $E_{def} = N_{proc}$, while in the worst case that $N_{grp} = 1$, $E_{def} = 1$ as well, since *DAEMON* regresses to the basic strategy in this case. Further, we know that the energy saved from *DAEMON* can be huge given a large value of $\frac{N}{BS}$. In general, *DAEMON* is more energy efficient in contrast to the basic strategy, if $N_{grp} > 1$.

3.3 High Performance Communication Scheme

In addition to applying *DAEMON* to minimize DVFS overhead in distributed matrix multiplication for energy efficiency, we also aim to achieve performance efficiency and thus additional energy savings can be achieved from less execution time. Generally, performance gain of distributed matrix multiplication can be fulfilled in terms of high performance computation and communication. Given that the optimal computation implementation of local matrix multiplication routine provided by ATLAS [2] is employed, we propose a high

performance communication scheme for fully exploiting network bandwidth. Specifically, since the global matrix is evenly distributed into the process grid for load balancing, we need to broadcast each matrix row/column to all other rows/columns located in different nodes individually for later performing local matrix multiplication in parallel. A high performance broadcast algorithm is thus desirable.

3.3.1 Binomial Tree and Pipeline Broadcast

There exist a large body of distributed broadcast algorithms for high performance communication, where binomial tree and pipeline broadcast generally outperform other algorithms for different system configurations. In the original pdgemm() routine from ScaLAPACK on top of different MPI implementations, different communication schemes like ring-based, binomial tree and pipeline broadcast are adopted depending on message size and other factors [26]. Table 4.2 lists the notation used in this section. Figure 3.3 (a) depicts how the binomial tree broadcast algorithm works using a simple example with a 3-round iteration on a 8-process cluster. We can see that in each round, a process sends messages in accordance with the following pattern:

- In Round 0, process P_0 (sender) sends a message to the next available process P_1 (receiver);
- In Round j ($j > 0$), process P_i ($i \leq j,\ i = 0, 1, 2, \dots$) that is a sender/receiver in the precedent round sends a message to the next available process, until the last one receives.

Table 3.2: Notation in Binomial Tree and Pipeline Broadcast.

P	The total number of processes in the communication
S_{msg}	Message size in one broadcast
BD	Network bandwidth in the communication
T_B	The total time consumed by all binomial tree broadcasts
T_P	The total time consumed by all pipeline broadcasts
T_b	Time consumed by one binomial tree broadcast
T_p	Time consumed by one pipeline broadcast
T_s	Network latency of starting up a communication link
T_d	Time consumed by transmitting messages
n	Number of chunks from dividing a message

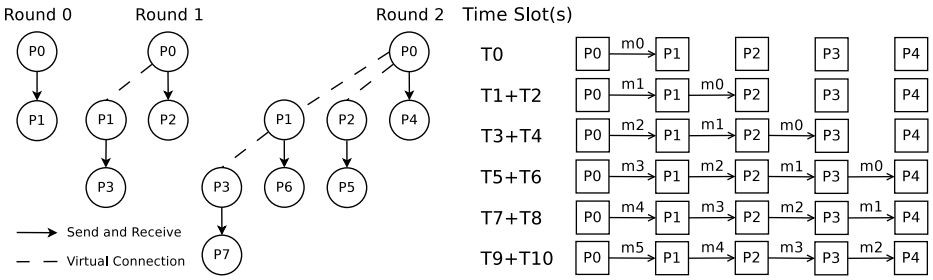
In other words, in Round j, the number of senders/receivers is 2^j and thus the algorithm takes $\log P$ rounds for the P^{th} process to receive a message. The communication time complexity can be modeled as:

$$T_B = T_b \times \log P, \text{ where } T_b = T_s + \frac{S_{msg}}{BD} \tag{3.6}$$

By substituting T_b, we obtain the final time complexity formula of binomial tree broadcast:

$$T_B = \left(T_s + \frac{S_{msg}}{BD} \right) \times \log P \tag{3.7}$$

Pipeline broadcast works in a time-sliced fashion so that different processes simultaneously broadcast different message chunks as stages in pipelining, as shown in Figure 3.3 (b). Assume a message in the pipeline broadcast is divided into n chunks. when the pipeline is not saturated, i.e., in the worst case, it takes $n + P - 1$ steps for the P^{th} process to receive a whole message of n chunks. We can model the time complexity of pipeline broadcast as:



(a) Binomial Tree Broadcast (b) Pipeline Broadcast

Figure 3.3: Binomial Tree and Pipeline Broadcast Algorithm Illustration.

$$T_P = T_p \times (n + P - 1), \text{ where } T_p = T_s + \frac{S_{msg}/n}{BD} \tag{3.8}$$

Similarly, substituting T_p into T_P in Equation 3.8 yields:

$$T_P = \left(T_s + \frac{S_{msg}/n}{BD}\right) \times (n + P - 1) \tag{3.9}$$

From Equations 3.7 and 3.9, despite the steps needed to receive a message, we can see that both T_B and T_P are essentially the sum of T_s and T_d. In a cluster connected by an Ethernet/Infiniband switch, T_s is of the order of magnitude of μs, so T_s is negligible when S_{msg} is comparatively large. Therefore, Equations 3.7 and 3.9 can be further simplified as follows:

$$T_B \approx \frac{S_{msg}}{BD} \times \log P \quad \text{and} \quad T_P \approx \frac{S_{msg}}{BD} \times \left(1 + \frac{P-1}{n}\right) \tag{3.10}$$

It is clear that both T_B and T_P scale up as P increases, with fixed message size and fixed number of message chunks. However the pipeline broadcast outperforms the binomial tree broadcast with given P and message size by increasing n, since the ratio of binomial tree broadcast to pipeline broadcast approximates to $\log P$ when n is large enough and $\frac{P-1}{n}$ becomes thus negligible. We experimentally observed the communication schemes in ScaLAPACK and DPLASMA pdgemm() routines are not optimal in two clusters. Energy and time saving opportunities can be exploited by leveraging slack arising from the communication. Thus a high performance pipeline broadcast scheme with tuned chunk size according to system characteristics is desirable.

3.4 Implementation and Evaluation

We have implemented our high performance adaptive memory-aware DVFS scheduling strategy with highly tuned pipeline broadcast (referred to as *HP-DAEMON* in the later text) to achieve the optimal energy and performance efficiency for distributed matrix multiplication. Our implementation was accomplished by rewriting pdgemm() from ScaLAPACK [26] and DPLASMA [8], two widely used high performance and scalable numerical linear algebra libraries for distributed-memory multi-core systems nowadays. In our implementation, instead of using binomial tree broadcast for communication, we tune chunk size of pipeline broadcast to fully exploit possible slack during communication [126]. We apply the core tiling topology similarly as proposed in [87] to exploit more parallelism in communication. For achieving the maximal performance of computation, we employ the dgemm() routine provided by ATLAS [2], a linear algebra software library that auto-

matically tunes performance according to configurations of the hardware. The rewritten pdgemm() has the same interface and is able to produce the same results as the original ScaLAPACK/DPLASMA pdgemm() routines, with total normalized differences between the range of 10^{-17} and 10^{-15} in our experiments. For comparison purposes, we also implemented the basic DVFS scheduling strategy (referred to as *Basic DVFS* later) employed in [46] [93].

Specifically, *HP-DAEMON* was implemented from two aspects as an integrated energy and performance efficient approach. Given a memory cost threshold specified by the user as a trade-off for saving energy, *HP-DAEMON* adaptively calculates N_{grp}, following the constraints of Equation 3.1. Then N_{grp} is applied to determine the extent of grouping, which reduces the number of DVFS switches at the cost of user-specified memory overhead. For performance efficiency, during computation, *HP-DAEMON* employs the optimal implementation of local matrix multiplication; during communication, *HP-DAEMON* leverages a non-blocking version [126] of the pipeline broadcast with tuned chunk size to maximize network bandwidth utilization.

3.4.1 Experimental Setup

We applied *HP-DAEMON* to five distributed matrix multiplications with different global matrix sizes to assess energy savings and performance gain achieved by our integrated approach. Experiments were performed on a small-scale cluster (HPCL) with an Ethernet switch consisting of 8 computing nodes with two Quad-core 2.5 GHz AMD Opteron 2380 processors (totalling 64 cores) and 8 GB RAM running 64-bit Linux kernel 2.6.32, and a large-scale cluster (Tardis) with an Infiniband switch consisting of 16 computing nodes with

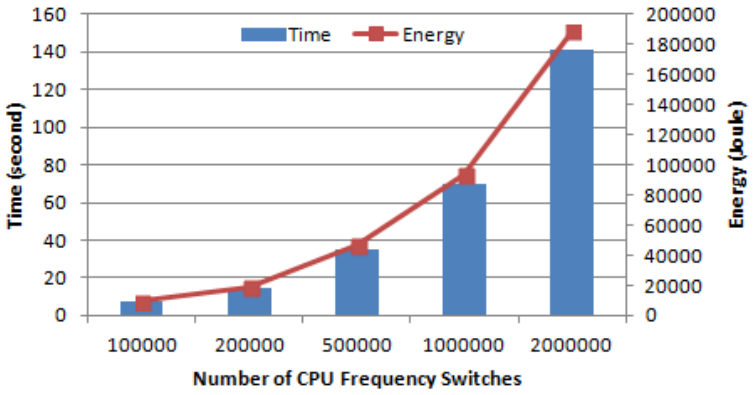


Figure 3.4: DVFS Energy and Time Overhead.

two 16-core 2.1 GHz AMD Opteron 6272 processors (totalling 512 cores) and 64 GB RAM running the same Linux kernel. All energy-related experiments were conducted only on HPCL while all performance-related experiments were performed on both clusters, since only HPCL was equipped with power sensors and meters for energy measurement. In our experiments, energy consumption was measured using PowerPack [71], a comprehensive software and hardware framework for energy profiling and analysis of high performance systems and applications. The range of CPU frequencies on HPCL was {0.8, 1.3, 1.8, 2.5} GHz. PowerPack was deployed and running on a meter node to collect energy costs on all involved components such as CPU, memory, disk, motherboard, etc. of all 8 computing nodes within HPCL. The collected energy information was recorded into a log file in the local disk and accessed after executing these distributed matrix multiplications.

3.4.2 Overhead on Employing DVFS

The introduction of DVFS may incur non-negligible energy and time costs on in-memory file read/write operations and gear transitions, if fine-grained DVFS scheduling is

Table 3.3: Memory Overhead Thresholds for Different Matrices and N_{grp}.

Global Matrix Size	N_{grp}	**Theoretical Additional Memory Overhead**	**Observed Total Memory Overhead**
7680	2	3.2%	6.4%
	4	6.4%	8.8%
	8	12.8%	14.4%
10240	2	4.8%	10.4%
	4	9.6%	16.0%
	8	19.2%	25.6%
12800	2	8.0%	16.0%
	4	16.0%	24.0%
	8	32.0%	40.0%
15360	2	11.2%	23.2%
	4	22.4%	35.2%
	8	44.8%	57.6%
17920	2	16.0%	28.0%
	4	32.0%	43.2%
	8	64.0%	78.4%

employed as in the case of *Basic DVFS*. In order to obtain accurate DVFS overhead, we measured energy and time costs on CPU frequency switches separately from the running application on the HPCL cluster, as shown in Figure 3.4. We can see that both energy and time costs increase monotonically as the number of DVFS switches increases. On average it takes about 70μs for one DVFS switch to complete and take effect, and about every 10 DVFS switches incur extra one Joule energy consumption. The additional energy costs from DVFS can be considerably large if CPU frequency switches occur frequently. A smart way of reducing the number of DVFS switches like *HP-DAEMON* can thus save energy and time of running the application.

3.4.3 Memory Cost Trade-off from HP-DAEMON

Once specifying a memory overhead threshold using command line parameters, in the form of the original command of executing the application followed by an optional parameter "`-t`" with a percentage, "`-t 0.2`" for instance, the user is afterwards not involved for an energy saving execution, e.g., dynamically modifying the threshold. *HP-DAEMON* adaptively calculates the optimal value for grouping according to the custom threshold. Essentially the total memory overhead consists of the memory costs from the execution of the original application, and the additional memory costs from *HP-DAEMON*. Using the calculated grouping value, *HP-DAEMON* performs grouped computation/communication accordingly to minimize the number of DVFS switches. Table 3.3 lists N_{grp} values for five different matrices, corresponding theoretical values of extra memory costs from *HP-DAEMON* (i.e., the left hand side of Equation 3.1), and observed memory costs overall. For simplicity, in our implementation all calculated grouping values are rounded to multiples of 2. As Table 3.3 shows, the empirical observed total memory costs generally increase more than the theoretical extra memory costs from *HP-DAEMON* as N_{grp} doubles. This is because atop the original application, besides extra memory footprint for grouped broadcasts/multiplications, *HP-DAEMON* incurs more memory costs on stacks of grouped function calls involved in grouping that cannot be freed and re-allocated immediately.

3.4.4 Performance Gain via Tuned Pipeline Broadcast

From Equations 3.10, we have two inferences: (a) Performance of pipeline broadcast is strongly tied to chunk size, and (b) performance of pipeline broadcast can be better

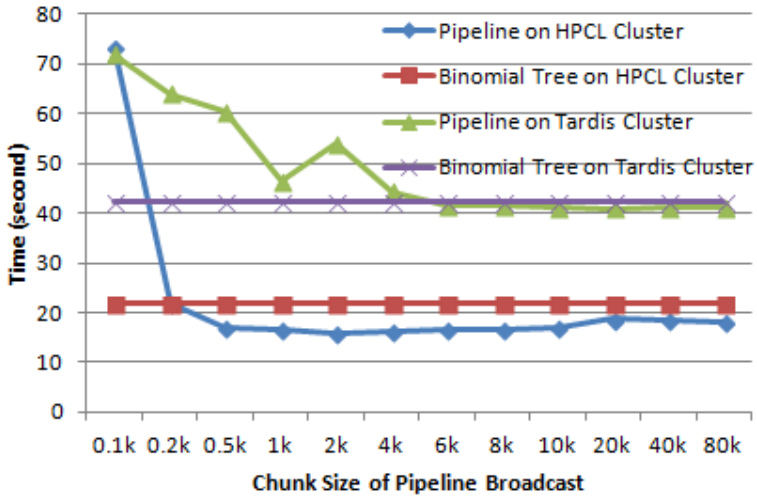


Figure 3.5: Performance Efficiency of Binomial Tree Broadcast and Pipeline Broadcast.

than binomial tree broadcast, since the ratio of binomial tree broadcast to pipeline broadcast is $logP$ when $\frac{P-1}{n}$ is negligible. Next we evaluate performance gain from the use of pipeline broadcast via tuning chunk size, in contrast to binomial tree broadcast, where global matrix sizes of distributed matrix multiplication on HPCL and Tardis are 10240 and 20480, individually.

In accordance with Equations 3.10, we can see in Figure 3.5, performance gain is achieved with the increase of chunk size (thus the decrease of the number of chunks) of pipeline broadcast, and pipeline broadcast performance converges reasonably well on both clusters as chunk size grows. The convergence point arises earlier on HPCL (0.5k) than Tardis (6k) due to the difference of network bandwidth between two clusters. This is because in order to reach the maximal network utilization, required chunk size of messages broadcasted on the cluster with slower network bandwidth is smaller compared to the cluster with faster network bandwidth. Further, we see that similarly on both clusters, pipeline

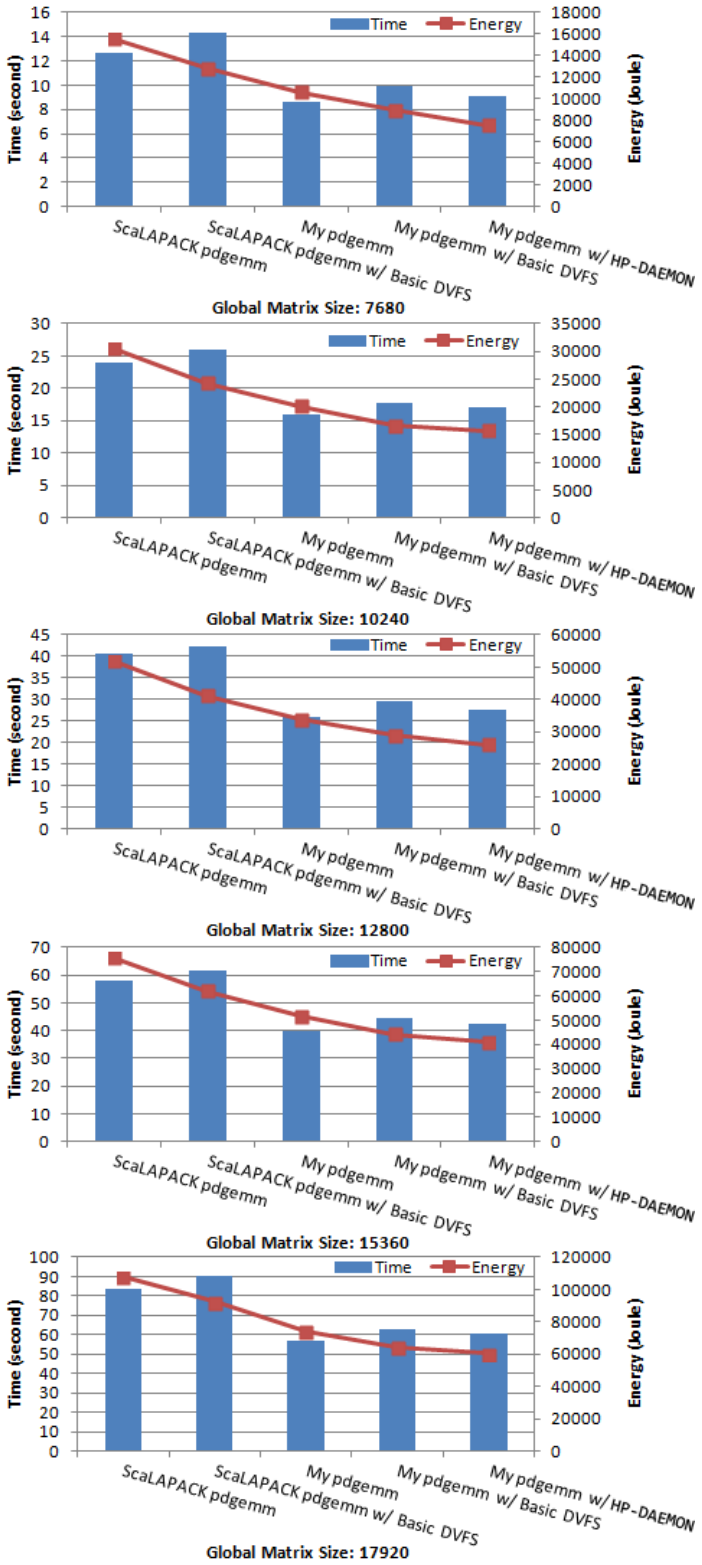


Figure 3.6: Energy Savings and Performance Gain on the HPCL Cluster (64-core, Ethernet).

broadcast outperforms binomial tree broadcast after the individual convergence point of pipelining, which complies with Equations 3.10 as well. Noteworthily, the performance between two types of broadcast differs greater on the cluster with slower network bandwidth (HPCL) than the cluster with faster network bandwidth (Tardis).

3.4.5 Overall Energy and Performance Efficiency of HP-DAEMON

Experimental results indicate that the optimal energy savings and performance gain can be achieved by applying *HP-DAEMON* in distributed matrix multiplication. We performed two types of experiments to evaluate the effectiveness of *HP-DAEMON* individually: (a) On the energy measurement enabled HPCL cluster, we evaluated energy and performance efficiency of *HP-DAEMON* by comparing ScaLAPACK pdgemm() to our implementation of pdgemm() with and without *HP-DAEMON* (we did not present the data of DPLASMA pdgemm() on HPCL, because DPLASMA pdgemm() did not manifest better performance than ScaLAPACK pdgemm() on Ethernet-switched HPCL); (b) on the Tardis cluster with faster network bandwidth but no tools for energy measurement, we evaluated performance efficiency of our pdgemm() implementation with tuned pipeline broadcast, by comparing against ScaLAPACK, DPLASMA, and our pdgemm() with binomial tree broadcast. The default block size 32 in ScaLAPACK/DPLASMA pdgemm() and the maximal value of N_{grp} (8 in our case) were adopted in our experiments.

Figure 3.6 shows energy costs and execution time of five different matrix multiplications on the HPCL cluster with two DVFS scheduling strategies, where in our implementation pipeline broadcast with tuned chunk size was employed for achieving the maximal performance of communication. We observe that the time overhead on employing DVFS is

non-negligible: 8.1% for ScaLAPACK pdgemm() and 12.4% for our pdgemm() on average. As discussed before, performance loss is attributed to time costs on virtual file read and write operations that are necessary in CPU frequency switching by DVFS. Thus extra energy consumption is incurred during the extra time on frequency switching. Although performance degrades using *Basic DVFS*, overall energy savings are achieved due to the scheduled low-power communication when CPU is idle. Compared to the original versions without DVFS, the energy savings from *Basic DVFS* enabled version of ScaLAPACK pdgemm() and our pdgemm() are considerable 18.1% and 15.1% on average, individually. Compared to *Basic DVFS*, employing *HP-DAEMON* effectively achieves more energy savings and reduces performance loss since the number of DVFS switches is minimized in accordance with the user-specified memory cost threshold. As shown in Figure 3.6, compared to our pdgemm() without DVFS, more energy savings are fulfilled (22.6% on average and up to 28.8%, 7.5% additional average energy savings compared to *Basic DVFS*) while performance loss is lowered to 6.4% on average (6.0% performance loss is eliminated compared to *Basic DVFS*). The heuristic of reducing frequency switches by grouping is thus evaluated to be beneficial to energy and performance efficiency for distributed matrix multiplication.

As the other integrated part of *HP-DAEMON*, employing pipeline broadcast with tuned chunk size further brings performance gain and thus energy savings. Comparing ScaLAPACK pdgemm() and our pdgemm() both without using DVFS, 32.9% on average and up to 35.6% performance gain is observed. Overall, compared to ScaLAPACK pdgemm(), our pdgemm() with *HP-DAEMON* achieves 47.8% on average and up to 51.4% energy savings, and 28.6% on average and up to 31.5% performance gain, due to the integrated

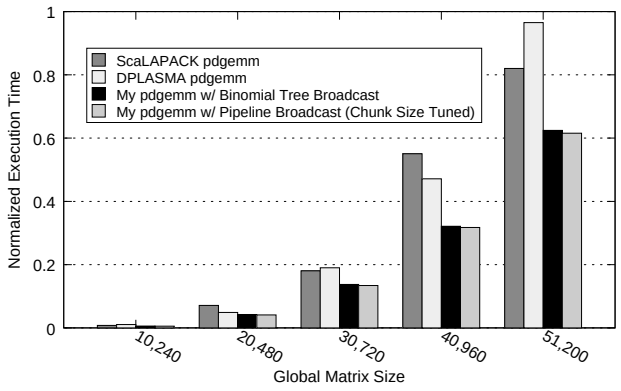


Figure 3.7: Performance Gain on the Tardis Cluster (512-core, Infiniband).

energy and performance efficiency from *HP-DAEMON*. Next we further evaluate overall performance gain achieved by our pdgemm() with *HP-DAEMON* on another cluster with more cores and faster network bandwidth.

Figure 3.7 shows performance comparison of five other distributed matrix multiplications in different implementations on the Tardis cluster. First we see that DPLASMA and ScaLAPACK pdgemm() perform similarly on average. Further the performance difference between binomial tree broadcast and pipeline broadcast narrows down to negligible extent due to the faster communication rate on Tardis. Overall on average, our pdgemm() with pipeline broadcast with tuned chunk size significantly outperforms ScaLAPACK pdgemm() by 33.3% and DPLASMA pdgemm() by 32.7%, respectively. It is thus evaluated that our pdgemm() with *HP-DAEMON* can also be superior in performance efficiency on large-scale clusters with fast network bandwidth.

3.5 Summary

Increasing requirements of exploiting parallelism in high performance applications pose great challenges on improving energy efficiency for these applications. Among potential software-controlled hardware solutions, DVFS has been leveraged to provide substantial energy savings. This work proposes an adaptive memory-aware DVFS scheduling strategy that reduces the number of CPU frequency switches to minimize the overhead on employing DVFS. A user-specified memory overhead threshold is used for grouping broadcasts/multiplications in distributed matrix multiplication to achieve the optimal energy savings. Further, a pipeline broadcast scheme with tuned chunk size for high performance communication is leveraged to fully exploit network bandwidth. The experimental results on two clusters indicate the effectiveness of the proposed integrated approach in both energy and performance efficiency, compared to ScaLAPACK and DPLASMA matrix multiplication with a basic DVFS scheduling strategy.

Chapter 4

Algorithm-Based Energy Saving for Cholesky, LU, and QR Factorizations

4.1 Introduction

4.1.1 Motivation

With the growing prevalence of distributed-memory architectures, high performance scientific computing has been widely employed on supercomputers around the world ranked by the TOP500 list [27]. Considering the crucial fact that the costs of powering a supercomputer are rapidly increasing nowadays due to expansion of its size and duration in use, improving energy efficiency of high performance scientific applications has been regarded as a pressing issue to solve. The Green500 list [10], ranks the top 500 supercomputers

worldwide by performance-power ratio in six-month cycles. Consequently, root causes of high energy consumption while achieving performance efficiency in parallelism have been widely studied. With different focuses of studying, holistic hardware and software approaches for reducing energy costs of running high performance scientific applications have been extensively proposed. Software-controlled hardware solutions such as DVFS-directed (Dynamic Voltage and Frequency Scaling) energy efficient scheduling are deemed to be effective and lightweight [46] [67] [69] [86] [124] [116] [115]. Performance and memory constraints have been considered as trade-offs for energy savings [92] [70] [139] [127] [132].

DVFS is a runtime technique that is able to switch operating frequency and supply voltage of a hardware component (CPU, GPU, memory, etc.) to different *levels* (also known as *gears* or *operating points*) per workload characteristics of applications to gain energy savings dynamically. CPU and GPU are the most widely applied hardware components for saving energy via DVFS due to two primary reasons: (a) Compared to other components such as memory, CPU/GPU DVFS is easier to implement [54] – various handy DVFS APIs have been industrialized for CPU/GPU DVFS such as CPUFreq kernel infrastructure [5] incorporated into the Linux kernel and NVIDIA System Management Interface (nvidia-smi) [21] for NVIDIA GPU; (b) CPU energy costs dominate the total system energy consumption [71] (CPU and GPU energy costs dominate if heterogeneous architectures are considered), and thus saving CPU and GPU energy greatly improves energy efficiency of the whole system. In this work, we focus on distributed-memory systems without GPU. For instance, energy saving opportunities can be exploited by reducing CPU frequency and voltage during non-CPU-bound operations such as large-message MPI communication, since

generally execution time of such operations barely increases at a low-power state of CPU, in contrast to original runs at a high-power state. Given the fact that energy consumption equals product of average power consumption and execution time ($E = \overline{P} \times T$), and the assumption that dynamic power consumption of a CMOS-based processor is proportional to product of operating frequency and square of supply voltage ($P \propto fV^2$) [108] [80], energy savings can be effectively achieved using DVFS-directed strategical scheduling approaches with little performance loss.

Running on distributed-memory architectures, HPC applications can be organized and scheduled in the unit of task, a set of operations that are functionally executed as a whole. Different tasks within one process or across multiple processes may depend on each other due to *intra-process* and *inter-process* data dependencies. Parallelism of task-parallel algorithms and applications can be characterized by graph representations such as Directed Acyclic Graph (DAG), where data dependencies among parallel tasks are appropriately denoted by directed edges. DAG can be effective for analyzing parallelism present in HPC runs, which is greatly beneficial to achieving energy efficiency. As typical task-parallel algorithms for scientific computing, dense matrix factorizations in numerical linear algebra such as Cholesky, LU, and QR factorizations have been widely adopted to solve systems of linear equations. Empirically, as standard functionality, routines of dense matrix factorizations are provided by various software libraries of numerical linear algebra for distributed-memory multicore architectures such as ScaLAPACK [26], DPLASMA [8], and MAGMA [18]. Therefore, saving energy for parallel Cholesky, LU, and QR factorizations contributes significantly to the greenness of high performance scientific computing nowadays.

4.1.2 Limitations of Existing Solutions

Most existing energy saving solutions for high performance scientific applications are (combination of) variants of two classic approaches: (a) A Scheduled Communication (*SC*) approach [46] [93] [101] [116] that keeps low CPU performance during communication and high CPU performance during computation, as large-message communication is not CPU-bound while computation is, and (b) a Critical Path (*CP*) approach [116] [115] [33] [30] that guarantees that tasks on the CP run at the highest CPU frequency while reduces frequency *appropriately* (i.e., without further delay to incur overall performance loss) for tasks off the CP to minimize slack.

Per the operating layer, existing solutions can be categorized into two types: OS level and application level. In general, OS level solutions feature two properties: (a) Working aside running applications and thus requiring no application-specific knowledge and no source modification, and (b) making online energy efficient scheduling decisions via dynamic monitoring and analysis. However, application level solutions statically utilize application-specific knowledge to perform specialized scheduling for saving energy, generally with source modification and recompilation (i.e., *generality*) trade-offs. Although with high generality, OS level solutions may suffer from critical disadvantages as follows, and consequently are far from a sound and complete solution, for applications such as parallel Cholesky, LU, and QR factorizations in particular:

Effectiveness. Although intended to be effective for general applications, OS level approaches rely heavily on the underlying workload prediction mechanism, due to lack of

knowledge of application characteristics. One prediction mechanism can work well for a specific type of applications sharing similar characteristics, but can be error-prone for other applications, in particular, applications with random/variable execution characteristics where the prediction mechanism performs poorly. Algorithms presented in [101] [116] [115] predict execution characteristics of the upcoming interval (i.e., a fixed time slice) according to recent intervals. This prediction mechanism is based on a simple assumption that task behavior is identical every time a task is executed [115]. However, it can be defective for applications with random/variable execution characteristics, such as matrix factorizations, where the remaining unfinished matrices become smaller as the factorizations proceed. In other words, length variation of iterations of the core loop for matrix factorizations can make the above prediction mechanism inaccurate, which invalidates potential energy savings. Moreover, since new technologies such as Hyper-Threading [12] have emerged for exploiting thread level parallelism on distributed-memory multicore systems, e.g., MPI programs can be parallelized on local multicore processors using OpenMP, behavior among parallel tasks can vary due to more non-deterministic events occurred in the multithreaded environment.

Further, the OS level prediction can be costly and thus energy savings are diminished. Recall that OS level solutions must predict execution details in the next interval using prior execution information. However, execution history in some cases may not necessarily be a reliable source for workload prediction, e.g., for applications with fluctuating runtime patterns at the beginning of the execution. As such, it can be time-consuming for obtaining accurate prediction results. Since during the prediction phase no energy savings

can be fulfilled, considerable potential energy savings can be wasted for accurate but lengthy prediction.

Completeness. OS level solutions only work when tasks such as computation and communication are being executed, energy saving opportunities are untapped during the time otherwise. Empirically, even though load balancing techniques have been leveraged, due to data dependencies among tasks and load imbalance that is not completely eliminated, not all tasks in different processes across nodes can start to work and finish at the same time. More energy can be saved for tasks waiting at the beginning of an execution, and for the last task of one process finishing earlier than that of other processes across nodes (details are illustrated in Figure 4.4). Restricted by the daemon-based nature of working aside real running tasks, OS level solutions cannot attain energy savings for such tasks.

4.1.3 Our Contributions

In this chapter, we propose a library level *race-to-halt* DVFS scheduling approach via Task Dependency Set (TDS) analysis based on algorithmic characteristics, namely `TX` (TDS-based race-to-halt), to save energy for task-parallel applications running on distributed-memory architectures, taking parallel Cholesky, LU, and QR factorizations for example. The idea of library level *race-to-halt* scheduling is intended for any task-parallel programming models where data flow analysis can be applied. The use of TDS analysis as a compiler technique allows possible extension of this work to a general compiler-based approach based on static analysis only. In summary, the contributions of this chapter are as follows:

- Compared to application level solutions, for widely used software libraries such as numerical linear algebra libraries, `TX` restricts source modification and recompilation at library level, and replacement of the energy efficient version of the libraries is allowed at link time (i.e., with *partial loss of generality*). No further source modification and recompilation are needed for applications where the libraries are called;

- Compared to OS level solutions, `TX` is able to achieve substantial energy savings for parallel Cholesky, LU, and QR factorizations (i.e., with *higher energy efficiency*), since via algorithmic TDS analysis, `TX` circumvents the defective prediction mechanism at OS level, and also manages to save more energy from possible load imbalance;

- We formally model that `TX` is comparable to the *CP* approach in energy saving capability under the circumstance of current CMOS technologies that allow insignificant variation of supply voltage as operating frequency scales via DVFS;

- With negligible performance loss (3.5% on average), on two power-aware clusters, `TX` is evaluated to achieve up to 33.8% energy savings compared to original runs of different-scale matrix factorizations, and save up to 17.8% and 15.9% more energy than state-of-the-art OS level *SC* and *CP* approaches, respectively.

The rest of this chapter is organized as follows. Section 4.2 introduces basics of parallel Cholesky, LU, and QR factorizations. We present TDS and CP in Section 4.3, and our `TX` approach in Section 4.4. Implementation details and experimental results are provided in Section 4.5. Section 4.6 discusses related work and Section 4.7 concludes.

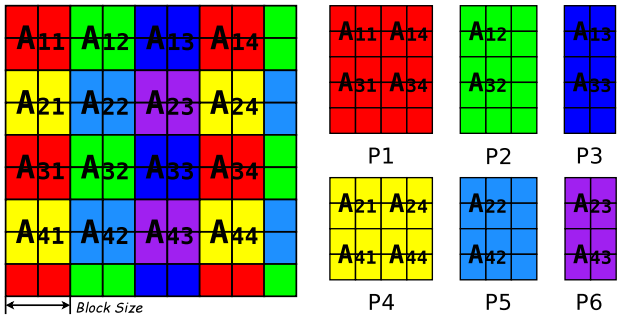
4.2 Background: Parallel Cholesky, LU, and QR Factorizations

As classic dense numerical linear algebra operations for solving systems of linear equations, such as $Ax = b$ where A is a given coefficient matrix and b is a given vector, Cholesky factorization applies to the case that A is a symmetric positive definite matrix, while LU and QR factorizations apply to any general $M \times N$ matrices. The goal of these operations is to factorize A into the form LL^T where L is lower triangular and L^T is the transpose of L, the form LU where L is unit lower triangular and U is upper triangular, and the form QR where Q is orthogonal and R is upper triangular, respectively. Thus from $LL^Tx = b$, $LUx = b$, $QRx = b$, x can be easily solved via forward substitution and back substitution. In practice, parallel Cholesky, LU, and QR factorizations are widely employed in extensive areas of high performance scientific computing. Various software libraries of numerical linear algebra for distributed multicore scientific computing such as ScaLAPACK [26], DPLASMA [8], and MAGMA [18] provide routines of these matrix factorizations as standard functionality. In this section, we present basics of parallel Cholesky, LU, and QR factorizations, and introduce an effective graph representation for parallelism present during the execution of these matrix factorizations.

4.2.1 2-D Block Cyclic Data Distribution

Regardless of the fact that nowadays matrix involved in many parallel numerical linear algebra operations is too large to fit into the memory of single node, the essence of a parallel algorithm is how it partitions workloads into a cluster of computing nodes as

balanced as possible to better exploit parallelism, which is also referred to as load balancing. In numerical linear algebra operations, distributing a global matrix into a process grid in a linear fashion does not benefit a lot from parallelism, since although the data allocated into each computing node are balanced in terms of amount, parallel execution of computation and communication are restricted by frequently arising data dependencies among tasks performed by different processes on different nodes.



(a) Global View (b) Local View

Figure 4.1: 2-D Block Cyclic Data Distribution on a 2 × 3 Process Grid in Global View and Local View.

As shown in Figure 4.1, *2-D block cyclic data distribution*, an effective way for load balancing, has been widely used in various numerical linear algebra libraries, such as HPL [11], ScaLAPACK [26], DPLASMA [8], and MAGMA [18]. Specifically, the global matrix is partitioned in a two-dimensional block cyclic fashion, and blocks are mapped into different nodes cyclically along both rows and columns of the process grid, so that tasks without data dependencies are able to be executed by multiple nodes simultaneously to achieve parallelism. Figure 4.1 (a) shows the partitioned global matrix in a global view,

while Figure 4.1 (b) depicts local matrices residing in different nodes individually, where the global matrix elements on one node are accessed periodically throughout the execution to balance the workloads in different nodes, instead of all at once as in the linear data distribution. The granularity of partitioning is determined by a *block size*, either specified by the user, or automatically tuned according to hardware configuration or set by default by the application itself. In practice, the global matrix size may not necessarily be multiples of the chosen block size. Typically, due to uneven division, the existence of remainder matrices, i.e., the last row and the last column in Figure 4.1 (a), barely affects parallelism empirically given a fine-grained partition.

4.2.2 DAG Representation of Parallel Cholesky, LU, and QR Factorizations

A well-designed partitioning and highly-efficient parallel algorithms of computation and communication substantially determine energy and performance efficiency of task-parallel applications. For such purposes, classic implementations of parallel Cholesky, LU, and QR factorizations are as follows: (a) Partition the global matrix into a cluster of computing nodes as a process grid using 2-D block cyclic data distribution [55] for load balancing; (b) perform local *diagonal matrix* factorizations in each node individually and communicate factorized local matrices to other nodes for *panel matrix* solving and *trailing matrix* updating, as shown in Figure 4.2, a stepwise LU factorization without pivoting. Due to frequently-arising data dependencies, parallel execution of parallel Cholesky, LU, and QR factorizations can be characterized using Directed Acyclic Graph (DAG), where

data dependencies among parallel tasks are appropriately represented. DAG for parallel Cholesky, LU, and QR factorizations is formally defined below:

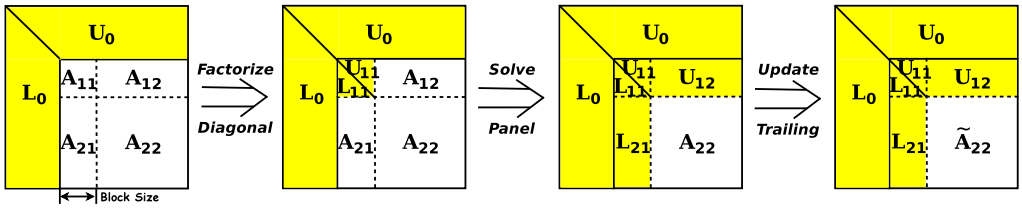


Figure 4.2: Stepwise Illustration of LU Factorization without Pivoting.

Definition 1. Data dependencies among parallel tasks of parallel Cholesky, LU, and QR factorizations running on a distributed-memory computing system are modeled by a Directed Acyclic Graph (DAG) $G = (V, E)$, where each node $v \in V$ denotes a task of Cholesky/LU/QR factorization, and each directed edge $e \in E$ represents a dynamic data dependency from task t_j to task t_i that both tasks manipulate on either different *intra-process* or *inter-process* local matrices (i.e., an *explicit* dependency) or the same *intra-process* local matrix (i.e., an *implicit* dependency), denoted by $t_i \rightarrow t_j$.

Example. Due to similarity among the three matrix factorizations and space limitation, we henceforth take parallel Cholesky factorization for example to elaborate our approach. Consider a 4×4 blocked Cholesky factorization as given in Figure 4.3. The outcome of the task factorizing A_{11}, i.e., L_{11}, is used in the tasks solving local matrices L_{21}, L_{31}, and L_{41} in the same column as L_{11}, i.e., the tasks calculating the *panel matrix*. In other words, there exist three data dependencies from the tasks solving L_{21}, L_{31}, and L_{41} to the task

$$\begin{pmatrix} A_{11} & A_{21}^T & A_{31}^T & A_{41}^T \\ A_{21} & A_{22} & A_{32}^T & A_{42}^T \\ A_{31} & A_{32} & A_{33} & A_{43}^T \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix} \times \begin{pmatrix} L_{11}^T & L_{21}^T & L_{31}^T & L_{41}^T \\ 0 & L_{22}^T & L_{32}^T & L_{42}^T \\ 0 & 0 & L_{33}^T & L_{43}^T \\ 0 & 0 & 0 & L_{44}^T \end{pmatrix}$$

$$= \begin{pmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T & L_{11}L_{31}^T & L_{11}L_{41}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T & L_{21}L_{31}^T + L_{22}L_{32}^T & L_{21}L_{41}^T + L_{22}L_{42}^T \\ L_{31}L_{11}^T & L_{31}L_{21}^T + L_{32}L_{22}^T & L_{31}L_{31}^T + L_{32}L_{32}^T + L_{33}L_{33}^T & L_{31}L_{41}^T + L_{32}L_{42}^T + L_{33}L_{43}^T \\ L_{41}L_{11}^T & L_{41}L_{21}^T + L_{42}L_{22}^T & L_{41}L_{31}^T + L_{42}L_{32}^T + L_{43}L_{33}^T & L_{41}L_{41}^T + L_{42}L_{42}^T + L_{43}L_{43}^T + L_{44}L_{44}^T \end{pmatrix}$$

Figure 4.3: Matrix Representation of a 4×4 Blocked Cholesky Factorization (We henceforth take parallel Cholesky factorization for example due to algorithmic similarity among three types of matrix factorizations).

factorizing A_{11}, denoted by three *solid* directed edges from the task Factorize(1,1) to the tasks Solve(2,1), Solve(3,1), and Solve(4,1) individually as shown in Figure 4.4 (see page 15). Besides the above *explicit* dependencies, there exists an *implicit* dependency between the task updating local matrix A_{32} and the task subsequently solving L_{32} on the same local matrix, denoted by the *dashed* directed edge from the task Update1(3,2) to the task Solve(3,2) in Figure 4.4. Note that communication among tasks is not shown in Figure 4.4, and updating diagonal local matrices and updating non-diagonal local matrices are distinguished as Update2() and Update1() respectively due to different computation time complexity.

4.3 Fundamentals: Task Dependency Set and Critical Path

Based on the task-parallel DAG representation, next we present Task Dependency Set (TDS) and Critical Path (CP) of running parallel Cholesky, LU, and QR factorizations,

Table 4.1: Notation in Algorithms 1, 2, 3, and 4 and Henceforth.

$task, t_1, t_2$	One task of matrix factorizations
N_{proc}	Square root of the total number of processes
f_l	The lowest CPU frequency set by DVFS
f_h	The highest CPU frequency set by DVFS
f_{opt}	Optimal ideal frequency to finish a task with its slack eliminated
$ratio$	Ratio between split durations for optimal frequency approximation
$\mathsf{TDS}_{in}(task)$	TDS consisting of tasks that are depended by $task$ as the input
$\mathsf{TDS}_{out}(task)$	TDS consisting of tasks that depend on $task$ as the input
$CritPath$	One task trace to finish matrix factorizations with zero total slack
$slack$	Time that a task can be delayed by with no overall performance loss
$CurFreq$	Current CPU frequency in use
$DoneFlag$	Indicator of the finish of a task

where TDS contains static dependency information of parallel tasks to utilize at runtime, and CP pinpoints potential energy saving opportunities in terms of slack among the tasks.

4.3.1 Task Dependency Set

For determining the appropriate timing for exploiting potential energy saving opportunities via DVFS, we leverage TDS analysis in our `TX` approach. Next we first formally define TDS, and then showcase how to generate two types of TDS for each task in Cholesky factorization using Algorithm 1. Producing TDS for LU and QR factorizations is similar with minor changes in Algorithm 1 per algorithmic characteristics. Table 4.1 lists the notation used in the algorithms and discussion in Sections 4.3 and 4.4.

Definition 2. Given a task t of a parallel Cholesky/LU/QR factorization, data dependencies related to a data block manipulated by the task t are classified as elements of two types of TDS: $\mathsf{TDS}_{in}(t)$ and $\mathsf{TDS}_{out}(t)$, where dependencies from the data block to other tasks

Algorithm 1 *Task Dependency Set Generation Algorithm*

```
GenTDS(task, N_proc)
 1: switch (task)
 2:   case Factorize:
 3:     foreach i < j, 1 ≤ i, j ≤ N_proc do
 4:       insert(TDS_in(S(j,i)), F(i,i));
 5:       insert(TDS_out(F(i,i)), S(j,i));
 6:   case Update1:
 7:     foreach i < j, 1 ≤ i, j ≤ N_proc do
 8:       if (IsLastInstance(U_1(j,i))) then
 9:         insert(TDS_in(U_1(j,i)), S(j,i));
10:         insert(TDS_out(S(j,i)), U_1(j,i));
11:   case Update2:
12:     foreach 1 ≤ i ≤ N_proc do
13:       if (IsLastInstance(U_2(i,i))) then
14:         insert(TDS_in(F(i,i)), U_2(i,i));
15:         insert(TDS_out(U_2(i,i)), F(i,i));
16:   case Solve:
17:     foreach 1 ≤ i < j ≤ N_proc do
18:       foreach j < k ≤ N_proc do
19:         insert(TDS_in(U_1(k,j)), S(j,i));
20:         insert(TDS_out(S(j,i)), U_1(k,j));
21:       foreach i < k < j ≤ N_proc do
22:         insert(TDS_in(U_1(j,k)), S(j,i));
23:         insert(TDS_out(S(j,i)), U_1(j,k));
24:       insert(TDS_in(U_2(j,j)), S(j,i));
25:       insert(TDS_out(S(j,i)), U_2(j,j));
26: end switch
```

t_i are categorized into $\mathsf{TDS}_{out}(t)$ and denoted as t_i for short, and dependencies from other tasks t_j to the data block are categorized into $\mathsf{TDS}_{in}(t)$ and denoted as t_j for short.

Example. Consider the Cholesky factorization in Figure 4.3. Two TDS of each task can be generated statically per algorithmic characteristics as shown in Algorithm 1: Since the resulting local matrices of factorization tasks (e.g., L_{11}) are used in column-wise *panel matrix* solving (e.g., solving L_{21}, L_{31}, and L_{41}), data dependencies from *panel matrices* to factorized *diagonal matrices* are included in TDS_{in} of tasks solving *panel matrices* (e.g.,

$\mathsf{TDS}_{in}(\mathsf{S}(2,1))$, $\mathsf{TDS}_{in}(\mathsf{S}(3,1))$, and $\mathsf{TDS}_{in}(\mathsf{S}(4,1))$), and TDS_{out} of tasks factorizing *diagonal matrices* (e.g., $\mathsf{TDS}_{out}(\mathsf{F}(1,1))$). Likewise TDS_{in} and TDS_{out} of other tasks holding different dependencies can be produced following Algorithm 1.

4.3.2 Critical Path

Although load balancing techniques are leveraged for distributing workloads into a cluster of computing nodes as evenly as possible, assuming that all nodes have the same hardware configuration and thus the same computation and communication capability, slack can result from the fact that different processes can be utilized unfairly due to three primary reasons: (a) imbalanced computation delay due to data dependencies among tasks, (b) imbalanced task partitioning, and (c) imbalanced communication delay. Difference in CPU utilization results in different amount of computation slack. For instance, constrained by data dependencies, the start time of processes running on different nodes differs from each other, as shown in Figure 4.4 (see page 15) where P1 starts earlier than the other three processes. Moreover, since the location of local matrices in the global matrix determines what types of computation are performed on the local matrices, load imbalancing from difference in task types and task amount allocated to different processes cannot be eliminated completely by the 2-D block cyclic data distribution, as shown in Figure 4.4 where P2 has lighter workloads compared to the other three processes. Imbalanced communication time due to different task amount among the processes further extends the difference in slack length for different processes.

Critical Path (CP) is one particular task trace from the beginning task of one run of a task-parallel application to the ending one with the total slack of zero. Any delay on

Algorithm 2 *Critical Path Generation Algorithm via TDS Analysis*

GenCritPath($CritPath$, $task$, N_{proc})
1: $CritPath \leftarrow \varnothing$
2: **switch** $(task)$
3: **case** Factorize:
4: insert($CritPath$, $\mathsf{F}(i,i)$)
5: **case** Update1:
6: Do Nothing
7: **case** Update2:
8: **if** $(t_1 \in CritPath \;\&\&\; t_1 \in \mathsf{TDS}_{out}(\mathsf{U}_2(i,i)))$ **then**
9: insert($CritPath$, $\mathsf{U}_2(i,i)$)
10: **case** Solve:
11: **foreach** $1 \leq i < j \leq N_{proc}$ **do**
12: **if** $(t_2 \in CritPath \;\&\&\; t_2 \in \mathsf{TDS}_{out}(\mathsf{S}(j,i)))$ **then**
13: insert($CritPath$, $\mathsf{S}(j,i)$)
14: **end switch**

tasks on the CP increases the total execution time of the application, while dilating tasks off the CP into their slack individually without further delay does not cause performance loss as a whole. Energy savings can be achieved by appropriately reducing frequency to dilate tasks off the CP into their slack as much as possible, which is referred to as the *CP* approach. Numerous existing OS level solutions effectively save energy via *CP-aware* analysis [46] [101] [116] [115] [33] [30]. Figure 4.4 highlights one CP for the provided parallel Cholesky factorization with bold edges. We next present a feasible algorithm to generate a CP in parallel Cholesky, LU, and QR factorizations via TDS analysis.

4.3.3 Critical Path Generation via TDS

We can generate a CP for parallel Cholesky, LU, and QR factorizations as the basis of the *CP* approach using Algorithm 2. Consider the same parallel Cholesky factorization above. The heuristic of the CP generation algorithm is as follows: (a) Each task of factor-

izing is included in the CP, since the local matrices to factorize are always updated last, compared to other local matrices in the same row of the global matrix, and the outcome of factorizing is required in future computation. In other words, the task of factorizing serves as a transitive step that cannot be executed in parallel together with other tasks; (b) each task of Update1() is excluded from the CP, since it does not have direct dependency relationship with any tasks of factorizing, which are already included in the CP; (c) regarding Update2(), we select the ones that are directly depended by the tasks of factorizing on the same local matrix into the CP; (d) we choose the tasks of solving that are directly depended by Update2() (or directly depends on Factorize(), not shown in the algorithm) into the CP. Note that CP can also be identified using methods other than TDS analysis [46] [33] [30].

4.4 TX: Energy Efficient Race-to-halt DVFS Scheduling

In this section, we present in detail three energy efficient DVFS scheduling approaches for parallel Cholesky, LU, and QR factorizations individually: the *SC* approach, the *CP* approach, and our `TX` approach. We further demonstrate that `TX` is able to save energy substantially compared to the other two solutions, since via *TDS-based race-to-halt*, it circumvents the defective prediction mechanism employed by the *CP* approach at OS level, and further saves energy from possible load imbalance. Moreover, we formally prove that `TX` is comparable to the *CP* approach in energy saving capability under the circumstance of current CMOS technologies.

4.4.1 Custom Functions

In Algorithms 1, 2, 3, and 4, nine custom functions are introduced for readability: insert(TDS(t_1), t_2), delete(TDS(t_1), t_2), SetFreq(), IsLastInstance(), Send(), Recv(), IsFinished(), GetSlack(), and GetOptFreq(). The implementation of insert() and delete() is straightforward: Add task t_2 into the TDS of task t_1, and remove t_2 from the TDS of t_1. SetFreq() is a wrapper of handy DVFS APIs that set specific frequencies, and Send() and Recv() are wrappers of MPI communication routines that send and receive flag messages among tasks respectively. IsLastInstance() is employed to determine if the current task is the last instance of the same type of tasks manipulating the same data block, and IsFinished() is employed to determine if the current task is finished: Both are easy to implement at library level. GetSlack() and GetOptFreq() are used to get slack of a task, and calculate the optimal CPU frequency to dilate a task into its slack as much as possible, respectively. Implementing GetSlack() and GetOptFreq() can be highly non-trivial. Specifically, GetSlack() calculates slack of a task off the CP from the difference between the latest and the earliest end time of the task. GetOptFreq() calculates the optimal ideal frequency to eliminate slack of a task from the mapping between frequency and execution time for each type of tasks.

4.4.2 Scheduled Communication Approach

One effective and straightforward solution to save energy for task-parallel applications is to set CPU frequency to high during computation, while set it to low during communication, given the fact that large-message communication is not bound by CPU performance while the computation is, so the peak CPU performance is not necessary dur-

ing the communication. Although substantial energy savings can be achieved from this Scheduled Communication (*SC*) approach [101] [116], it leaves potential energy saving opportunities from other types of slack (e.g., see slack shown in Figure 4.4 on page 15) untapped. More energy savings can be fulfilled via fine-grained analysis of execution characteristics of HPC applications, in particular during non-communication. Next we present two well-designed approaches that take advantage of computation slack to further gain energy savings. Note since the *SC* approach does not conflict with solutions exploiting slack from non-communication, it thus can be incorporated with the next two solutions seamlessly to maximize energy savings.

4.4.3 Critical Path Approach vs. TX Approach

Given a detected CP (e.g., via static analysis [46] or local information analysis [115]) for task-parallel applications, the Critical Path (*CP*) approach saves energy as shown in Algorithm 3: For all tasks on the CP, the CPU operating frequency is set to the highest for attaining the peak CPU performance, while for tasks not on the CP with the total slack larger than zero (e.g., tasks with no outgoing *explicit* data dependencies in Figure 4.4), lowering frequency appropriately is performed to dilate the tasks into their slack as much as possible, without incurring performance loss of the applications. Due to the discrete domain of available CPU frequencies defined for DVFS, if the calculated optimal frequency that can eliminate slack lies in between two available neighboring frequencies, the two frequencies can be employed to approximate it by calculating a ratio of durations operating at the two frequencies. The two frequencies are then assigned to the durations separately

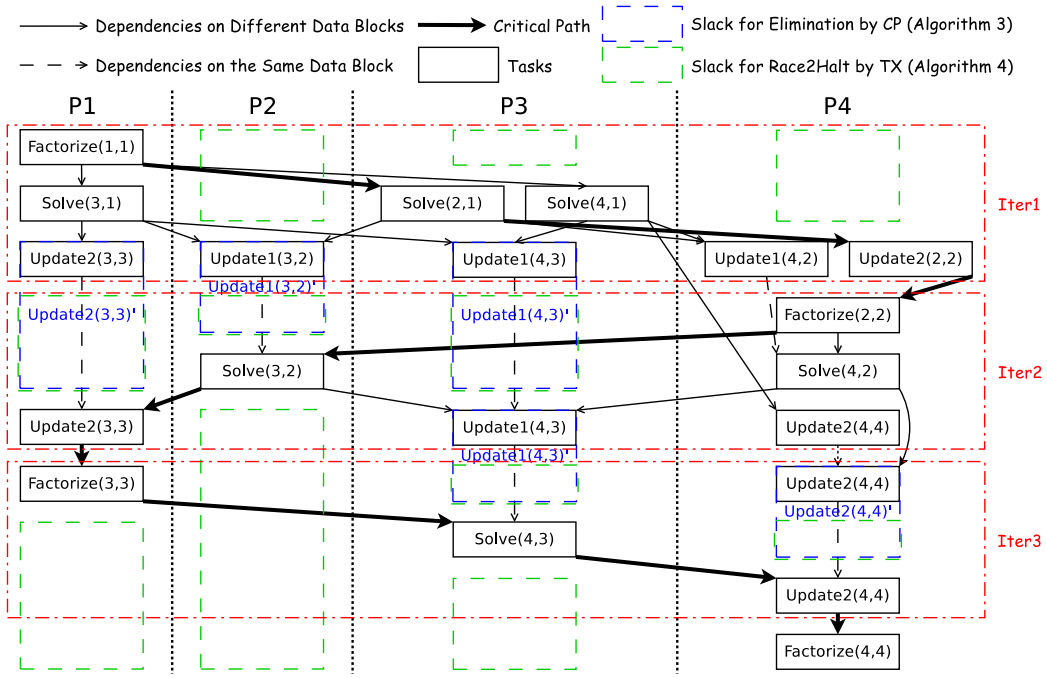


Figure 4.4: DAG Representation of Task and Slack Scheduling of CP and TX Approaches for the 4 × 4 Blocked Cholesky Factorization in Figure 4.3 on a 2 × 2 Process Grid Using 2-D Block Cyclic Data Distribution.

Algorithm 3 *DVFS Scheduling Algorithm Using CP*

DVFS_CP($CritPath$, $task$, $FreqSet$)
1: **if** ($task \in CritPath$ || $\mathsf{TDS}_{out}(task)$!= $\varnothing$) **then**
2: SetFreq(f_h)
3: **else**
4: $slack \leftarrow$ GetSlack($task$)
5: **if** ($slack > 0$) **then**
6: $f_{opt} \leftarrow$ GetOptFreq($task$, $slack$)
7: **if** ($f_l \leq f_{opt} \leq f_h$) **then**
8: **if** ($f_{opt} \notin FreqSet$) **then**
9: SetFreq($\lfloor f_{opt} \rfloor$, $\lceil f_{opt} \rceil$, $ratio$)
10: **else** SetFreq(f_{opt})
11: **else if** ($f_{opt} < f_l$) **then**
12: SetFreq(f_l)
13: **end if**

based on the ratio. Lines 7-9 in Algorithm 3 sketch the frequency approximation method [145] [115]. The ratio of split frequencies is calculated via prior knowledge of the mapping between frequency and execution time of different types of tasks. Note that we denote the two neighboring available frequencies of f_{opt} as $\lfloor f_{opt} \rfloor$ and $\lceil f_{opt} \rceil$.

Different from the *CP* approach that reduces CPU frequency for tasks off the CP to eliminate slack for saving energy without performance loss, `TX` employs the *race-to-halt* mechanism that leverages two TDS of each task (TDS_{in} and TDS_{out}) to determine the timing of *race* and *halt*, as shown in Algorithm 4. Respecting data dependencies, one *dependent* task cannot start until the finish of its *depended* task. `TX` keeps the *dependent* task staying at the lowest frequency, i.e., *halt*, until all its *depended* tasks have finished when it may start, and then allows the *dependent* task to work at the highest frequency to complete as soon as possible, i.e., *race*, before being switched back to the low-power state. Upon completion, a task sends a *DoneFlag* to all its *dependent* tasks to notify them that

Algorithm 4 *DVFS Scheduling Algorithm Using TX*

```
DVFS_TX(task, CurFreq)
 1: while (TDS_in(task) != ∅) do
 2:    if (CurFreq != f_l) then
 3:       SetFreq(f_l)
 4:    if (Recv(DoneFlag, t_1)) then
 5:       delete(TDS_in(task), t_1)
 6: end while
 7: SetFreq(f_h)
 8: if (IsFinished(task)) then
 9:    foreach t_2 ∈ TDS_out(task) do
10:       Send(DoneFlag, t_2)
11:    SetFreq(f_l)
12: end if
```

data needed has been processed and ready for use. A *dependent* task is retained at the lowest frequency while waiting for *DoneFlags* from all its *depended* tasks, and removes the dependency to a *depended* task from its TDS_{in}, once a *DoneFlag* from the *depended* task is received.

Defective Prediction Mechanism. Although effective, the OS level *CP* approach essentially depends on the workload prediction mechanism: Execution characteristics in the upcoming interval can be predicted using prior execution information, e.g., execution traces in recent intervals. However, this prediction mechanism may not necessarily be reliable and lightweight: (a) For applications with variable execution characteristics, such as dense matrix factorizations. Execution time of iterations of the core loop shrinks as the remaining unfinished matrices become smaller. Consequently dynamic prediction on execution characteristics such as task runtime and workload distribution can be inaccurate, which thus leads to error-prone slack estimation; (b) for applications with random execution patterns, such as applications relying on random numbers that could lead to variation of control flow

at runtime, which can be difficult to capture. Since the predictor needs to determine reproducible execution patterns at the beginning of one execution, it can be costly for obtaining accurate prediction results in both cases above. Given the fact that no energy savings can be fulfilled until the prediction phase is finished, considerable potential energy savings may be wasted for accurate but lengthy prediction as such at OS level.

There exist numerous studies on history-based workload prediction, which can be generally considered as two variants: the simplest and mostly commonly-used prediction algorithm PAST [141] and its enhanced algorithms [137] [51] [80] [70], where PAST works as follows:

$$W'_{i+1} = W_i \tag{4.1}$$

where W'_{i+1} is the next executed workload to predict, and W_i is the current measured workload. For applications with stable or slowly-varying execution patterns, the straightforward PAST algorithm can work well with little performance loss and high accuracy. It is however not appropriate for handling a considerable amount of variation in execution patterns. Many enhanced algorithms have been proposed to produce more accurate workload prediction for such applications. For instance, the RELAX algorithm employed in CPU MISER [70] exploits both prior predicted profiles and current runtime measured profiles as follows:

$$W'_{i+1} = (1 - \lambda)W'_i + \lambda W_i \tag{4.2}$$

where λ is a relaxation factor for adjusting the extent of dependent information of the current measurement. This enhanced prediction mechanism can also be error-prone and costly for parallel Cholesky, LU, and QR factorizations due to the use of 2-D block cyclic

data distribution. As shown in Figure 4.4, across loop iterations (highlighted by red dashed boxes) different types of tasks can be distributed to a process. For instance, in the first iteration, P_1 is assigned three tasks Factorize(1,1), Solve(3,1), and Update2(3,3), while in the second iteration, it is only assigned one task Update2(3,3). Although empirically for a large global matrix, tasks are distributed to one specific process alternatingly (e.g., Factorize(i,i) ($\mathsf{i} = 1, 3, 5, \ldots, 2\mathsf{s}+1, \mathsf{s} \geq 0$) can be all distributed to P1), the RELAX algorithm needs to adjust the value of λ alternatingly as well, which can be very costly and thus diminish potential energy savings. Length variation of iterations due to the shrinking remaining unfinished matrices further brings complexity to workload prediction.

`TX` successfully circumvents the shortcomings from the OS level workload prediction, by leveraging the TDS-based *race-to-halt* strategy. `TX` essentially differs from the *CP* approach since it allows tasks to complete as soon as possible before entering the low-power state, and thus does not require any workload prediction mechanism. `TX` works at library level and thus benefits from: (a) obtaining application-specific knowledge easily that can be utilized to determine accurate timing for DVFS, and (b) restricting the scope of source modification and recompilation to library level, compared to application level solutions.

Potential Energy Savings from Load Imbalance. Due to the existence of load imbalance regardless of load balancing techniques applied, and also data dependencies among inter-process parallel tasks, not all tasks can start to work and finish at the same time, as shown in Figure 4.4. At OS level, the *SC* approach and the *CP* approach only work when tasks are being executed, which leaves potential energy savings untapped during the time otherwise. Specifically, the *SC* approach switches frequency to high and low at the

time when computation and communication tasks start respectively, and do nothing during the time other than computation and communication. Likewise, the *CP* approach assigns appropriate frequencies for tasks on/off the CP individually. Therefore due to its nature of keeping the peak CPU performance for the tasks on the CP and dilating the tasks off the CP into its slack via frequency reduction, switching frequency is not feasible when for one process no tasks have started or all tasks have already finished.

Due to its nature of *race-to-halt*, `TX` can start to save energy even before any tasks in a process are executed, and after all tasks in a process have finished while there exist unfinished tasks in other processes. In Figure 4.4, the durations only covered by green dashed boxes highlights the additional energy savings fulfilled by `TX`, where due to data dependencies, a task cannot start yet while waiting for its depended task to finish first, or all tasks in one process have already finished while some tasks in other processes are still running.

Energy Saving Capability Analysis. Next we formally prove that compared to the classic *CP* approach, `TX` is comparable to it in energy saving capability. Given the following two energy saving strategies, towards a task t with an execution time T and slack T' at the peak CPU performance, we calculate the total nodal system energy consumption for both strategies, i.e., $E(\mathsf{S_1})$ and $E(\mathsf{S_2})$ formally below:

- Strategy I (Race-to-halt): Execute t at the highest frequency f_h until the finish of t and then switch to the lowest frequency f_l, i.e., run in T at f_h and then run in T' at f_l;

Table 4.2: Notation in Energy Saving Analysis.

E	The total nodal energy consumption of all components
P	The total nodal power consumption of all components
$P_{dynamic}$	Dynamic power consumption in the running state
$P_{leakage}$	Static/leakage power consumption in any states
T	Execution time of a task at the peak CPU performance
T'	Slack of executing a task at the peak CPU performance
A	Percentage of active gates in a CMOS-based chip
C	The total capacitive load in a CMOS-based chip
f	Current CPU operating frequency
V	Current CPU supply voltage
V'	Supply voltage of components other than CPU
I_{sub}	CPU subthreshold leakage current
I'_{sub}	non-CPU component subthreshold leakage current
f_m	Available optimal frequency assumed to eliminate T'
V_h	The highest supply voltage corresponding to f_h set by DVFS
V_l	The lowest supply voltage corresponding to f_l set by DVFS
V_m	Supply voltage corresponding to f_m set by DVFS
n	Ratio between original runtime and slack of a task

- Strategy II (CP-aware): Execute t at the optimal frequency f_m with which T' is eliminated, i.e., run in $T+T'$ at f_m (without loss of generality, assume T' can be eliminated using an available frequency f_m without frequency approximation).

For simplicity, let us assume the tasks for the use of DVFS are compute-intensive (memory-intensive tasks can be discussed with minor changes in the model), i.e., $T + T' = nT$, when $f_m = \frac{1}{n} f_h$, where $1 \leq n \leq \frac{f_h}{f_l}$. Considering the nodal power consumption P, we model it formally as follows:

$$P = P^{CPU}_{dynamic} + P^{CPU}_{leakage} + P^{other}_{leakage} \tag{4.3}$$

$$P_{dynamic} = ACfV^2 \tag{4.4}$$

$$P_{leakage} = I_{sub}V \tag{4.5}$$

Then, substituting Equations 4.4 and 4.5 into Equation 4.3 yields:

$$P = ACfV^2 + I_{sub}V + I'_{sub}V' \tag{4.6}$$

In our scenario, $P^{other}_{leakage} = I'_{sub}V'$ is independent of CPU frequency and voltage scaling, and thus can be regarded as a constant in Equation 4.6, so we denote $P^{other}_{leakage}$ as P_c for short. Further, although subthreshold leakage current I_{sub} has an exponential relationship with threshold voltage, results presented in [135] indicate that I_{sub} converges to a constant after a certain threshold voltage value. Without loss of generality, we treat $P^{CPU}_{leakage} = I_{sub}V$ as a function of supply voltage V only. Thus, we model the nodal energy consumption E_{node} for both strategies individually below:

$$\begin{aligned} E(\mathsf{S_1}) &= \overline{P(\mathsf{S_1})} \times T + \overline{P'(\mathsf{S_1})} \times T' \\ &= (ACf_hV_h^2 + I_{sub}V_h + P_c)T + (ACf_lV_l^2 + I_{sub}V_l + P_c)T' \\ &= AC(f_hV_h^2T + f_lV_l^2T') + I_{sub}(V_hT + V_lT') + P_c(T + T') \end{aligned} \tag{4.7}$$

$$\begin{aligned} E(\mathsf{S_2}) &= \overline{P(\mathsf{S_2})} \times (T + T') \\ &= (ACf_mV_m^2 + I_{sub}V_m + P_c)(T + T') \\ &= ACf_mV_m^2(T + T') + I_{sub}V_m(T + T') + P_c(T + T') \end{aligned} \tag{4.8}$$

We obtain the difference between energy costs of both strategies by dividing Equation 4.8 by Equation 4.7:

$$\frac{E(\mathsf{S_2})}{E(\mathsf{S_1})} = \frac{ACf_mV_m^2(T+T') + I_{sub}V_m(T+T') + P_c(T+T')}{AC(f_hV_h^2T + f_lV_l^2T') + I_{sub}(V_hT + V_lT') + P_c(T+T')} \tag{4.9}$$

Substituting the assumption that $T' = (n-1)T$ and $f_m = \frac{1}{n}f_h$ into both numerator and denominator yields the following simplified formula:

$$\begin{aligned}\frac{E(\mathsf{S_2})}{E(\mathsf{S_1})} &= \frac{AC\frac{1}{n}f_hV_m^2\left(1+(n-1)\right) + I_{sub}V_m\left(1+(n-1)\right) + P_c\left(1+(n-1)\right)}{AC\left(f_hV_h^2 + f_lV_l^2\left(n-1\right)\right) + I_{sub}\left(V_h + V_l\left(n-1\right)\right) + P_c\left(1+(n-1)\right)} \\ &= \frac{ACf_hV_m^2 + nI_{sub}V_m + nP_c}{AC\left(f_hV_h^2 + f_lV_l^2\left(n-1\right)\right) + I_{sub}\left(V_h + V_l\left(n-1\right)\right) + nP_c}\end{aligned} \tag{4.10}$$

In Equation 4.10, the denominator is a function of the variable n only. It is clear that it is a monotonically increasing function for n, whose minimum value is attained when $n = 1$, i.e., when slack T' equals 0. Given the fact that supply voltage has a positive correlation with (not strictly proportional to) operating frequency, scaling up/down frequency results in voltage up/down accordingly as shown in Table 4.3. Therefore for the numerator of Equation 4.10, the greater n is, the smaller f_m and V_m are, provided $f_m = \frac{1}{n}f_h$ and the above fact. It is thus complicated to determine the monotonicity of the numerator. As a matter of fact, state-of-the-art CMOS technologies allow insignificant variation of voltage as frequency scales (see Table 4.3). Consequently the term $ACf_hV_m^2$ within the numerator does not decrease much together with the increase of n. Moreover, the ratio between the highest and the lowest frequencies determines the upper bound of n ($1 \leq n \leq \frac{f_h}{f_l}$), so the other two terms within the numerator cannot increase significantly as n goes up.

Example. From the operating points of various processors shown in Table 4.3, we can calculate numerical energy savings for different values of n and a specific processor configuration

Table 4.3: Frequency-Voltage Pairs for Different Processors (Unit: Frequency (GHz), Voltage (V)).

Gear	AMD Opteron 2380		AMD Opteron 846 and AMD Athlon64 3200+		AMD Opteron 2218		Intel Pentium M		Intel Core i7-2760QM	
	Freq.	Volt.	Freq.	Volt.	Freq.	Volt.	Freq.	Volt.	Freq.	Volt.
0	2.5	1.300	2.0	1.500	2.4	1.250	1.4	1.484	2.4	1.060
1	1.8	1.200	1.8	1.400	2.2	1.200	1.2	1.436	2.0	0.970
2	1.3	1.100	1.6	1.300	1.8	1.150	1.0	1.308	1.6	0.890
3	0.8	1.025	0.8	0.900	1.0	1.100	0.8	1.180	0.8	0.760

that quantify the energy efficiency of both energy saving strategies. For the AMD Opteron 2218 processor, given a task with the execution time T and slack $0.25T$, i.e., $n = 1.25$, for eliminating the slack, 1.8 GHz is adopted as the operating frequency for running the task, and thus the numerator of Equation 4.10 equals $AC\left(2.4 \times 1.25^2 + (1.25-1) \times 1.0 \times 1.1^2\right) + I_{sub}\left(1.25 + (1.25-1) \times 1.1\right) + 1.25P_c = 4.0525AC + 1.525I_{sub} + 1.25P_c$, while the denominator of Equation 4.10 equals $AC \times 2.4 \times 1.15^2 + 1.25 \times 1.15I_{sub} + 1.25P_c = 3.174AC + 1.4375I_{sub} + 1.25P_c$. We can see that the coefficients of the corresponding term P_c between the numerator and the denominator are the same, and the coefficients of both corresponding terms AC and I_{sub} between the numerator and the denominator do not differ much. Therefore the result of $\frac{E(\mathsf{S}_2)}{E(\mathsf{S}_1)}$ would be a value close to 1, which means Strategy I is comparable to Strategy II in energy efficiency, regardless of the defective prediction mechanism.

4.5 Implementation and Evaluation

We have implemented `TX`, and for comparison purposes, the library level *SC* approach to evaluate the effectiveness of `TX` to save energy during non-communication slack.

For comparing with the OS level *SC* and *CP* approaches, we communicated with the authors of Adagio [115] and Fermata [116] and received the latest version of the implementation of both. We also compare with another OS level solution CPUSpeed [6], an interval-based DVFS scheduler that scales CPU performance according to runtime CPU utilization during the past interval. Regarding future workload prediction, essentially Adagio and Fermata leverage the PAST algorithm [141], and CPUSpeed uses a prediction algorithm similar to the RELAX algorithm employed in CPU MISER [70]. With application-specific knowledge known, all library level solutions do not need the workload prediction mechanism. In the later discussion, we denote the above approaches as follows:

- `Orig`: The original runs of different-scale parallel Cholesky, LU, and QR factorizations without any energy saving approaches;
- `SC_lib`: The library level implementation of the *SC* approach;
- `Fermata`: The OS level implementation of the *SC* approach based on the PAST workload prediction algorithm;
- `Adagio`: The OS level implementation of the *CP* approach based on the PAST workload prediction algorithm, where `Fermata` is incorporated;
- `CPUSpeed`: The OS level implementation of the *SC* approach based on a workload prediction algorithm similar to the RELAX algorithm;
- `TX`: The library level implementation of the *race-to-halt* approach based on TDS analysis, where `SC_lib` is incorporated.

The goals of the evaluation are to demonstrate that: (a) `TX` is able to save energy effectively and efficiently for applications with variable execution characteristics such as parallel Cholesky, LU, and QR factorizations, while OS level prediction-based solutions cannot maximize energy savings, and (b) `TX` only incurs negligible performance loss, similar as the compared OS level solutions. We did not compare with application level solutions, since they essentially fulfill the same energy efficiency as library level solutions, with source modification and recompilation at application level. Working at library level, `TX` was deployed in a distributed manner to each core/process. As the additional functionality of saving energy, the implementation of `TX` was embedded into a rewritten version of ScaLAPACK [26], a widely used high performance and scalable numerical linear algebra library for distributed-memory architectures. In particular, library level source modification and recompilation were conducted to the pdpotrf(), pdgetrf(), and pdgeqrf() routines, which perform parallel Cholesky, LU, and QR factorizations, respectively.

Table 4.4: Hardware Configuration for All Experiments.

Cluster	HPCL	ARC
System Size (# of Nodes)	8	108
Processor	2×Quad-core AMD Opteron 2380	2×8-core AMD Opteron 6128
CPU Freq.	0.8, 1.3, 1.8, 2.5 GHz	0.8, 1.0, 1.2, 1.5, 2.0 GHz
Memory	8 GB RAM	32 GB RAM
Cache	128 KB L1, 512 KB L2, 6 MB L3	128 KB L1, 512 KB L2, 12 MB L3
Network	1 GB/s Ethernet	40 GB/s InfiniBand
OS	CentOS 6.2, 64-bit Linux kernel 2.6.32	CentOS 5.7, 64-bit Linux kernel 2.6.32
Power Meter	PowerPack	Watts up? PRO

4.5.1 Experimental Setup

We applied all five energy efficient approaches to the parallel Cholesky, LU, and QR factorizations with five different global matrix sizes each to assess our goals. Experiments were performed on two power-aware clusters: HPCL and ARC. Table 6.2 lists the hardware configuration of the two clusters. Note that we measured the total dynamic and leakage energy consumption of distributed runs using PowerPack [71], a comprehensive software and hardware framework for energy profiling and analysis of high performance systems and applications. The total of static and dynamic power consumption was measured using Watts up? PRO [28]. Both energy and power consumption are the total energy and power costs respectively on all involved components of one compute node, such as CPU, memory, disk, motherboard, etc. Since each set of three nodes of the ARC cluster share one power meter, power consumption measured is for the total power consumption of three nodes, while energy consumption measured is for all energy costs collected from all eight nodes of the HPCL cluster. CPU DVFS was implemented via the CPUFreq infrastructure [5] that directly modifies CPU frequency system configuration files.

4.5.2 Results

In this section, we present experimental results on power, energy, and performance efficiency and trade-offs individually by comparing `TX` with the other energy saving approaches.

Power Savings. First we evaluate the capability of power savings from the five energy saving approaches for parallel Cholesky, LU, and QR factorizations on the ARC cluster (due

to the similarity of results, data for parallel LU and QR factorizations is not shown), where the power consumption was measured by sampling at a constant rate through the execution of the applications. Figure 4.5 depicts the total power consumption of three nodes (out of sixteen nodes in use) running parallel Cholesky factorization with different approaches using a 160000 $\times$ 160000 global matrix, where we select time durations of the first few iterations. Note that parallel Cholesky factorization (the core loop) performs alternating computation and communication with decreasing execution time of each iteration, as the remaining unfinished matrix shrinks. Thus we can see that for all approaches, from left to right, the durations of computation (the peak power values) decrease as the factorization proceeds.

Among the seven runs (including the theoretical one `CP_theo`), there exist four power variation patterns: (a) `Orig` and `CPUSpeed` – employed the same highest frequency for both computation and communication, resulting almost constant power consumption around 950 Watts; (b) `SC_lib`, `Fermata`, and `Adagio` – lowered down frequency during the communication, i.e., the five low-power durations around 700 Watts, and resumed the peak CPU performance during the computation; (c) `CP_theo` – not only scheduled low power states for communication, but also slowed down computation to eliminate computation slack – this is a theoretical value curve instead of real measurement, which is how OS level approaches such as `Adagio` is supposed to save more power as a *CP-aware* approach based on accurate workload prediction; and (d) `TX` – employed the *race-to-halt* strategy to lower down CPU performance for all durations other than computation.

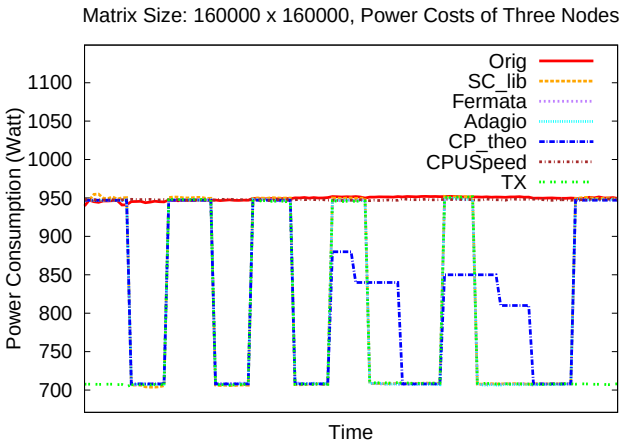


Figure 4.5: Power Consumption of Parallel Cholesky Factorization with Different Energy Saving Approaches on the ARC Cluster using 16 × 16 Process Grid.

Specifically, upon the prediction algorithm that inspects dynamic prior CPU utilization periodically for workload prediction, `CPUSpeed` failed to produce accurate prediction and scale CPU power states accordingly: It kept the peak CPU performance all the time. Either relying on knowing application characteristics (`SC_lib`) or detecting MPI communication calls (`Fermata` and `Adagio`), all three approaches can identify durations of communication and apply DVFS decisions accordingly. As discussed earlier, solutions that only slow down CPU during communication are semi-optimal. `Adagio` and `TX` are expected to utilize computation slack for achieving additional energy savings. Due to the defective OS level prediction mechanism, `Adagio` failed to predict behavior of future tasks and calculate computation slack accurately. Consequently no low-power states were switched to during computation for `Adagio`. In contrast, we provide a theoretical value curve `CP_theo` that calculates computation slack effectively, and lower power states were switched to eliminate the slack, i.e., the four medium-power durations around 850 Watts during the third and the

fourth computation as the blue line shows. Different from the solutions that save energy via slack reclamation, `TX` relies on the *race-to-halt* mechanism where computation is conducted at the peak CPU performance and the lowest CPU frequency is employed immediately after the computation. Therefore during computation slack, we can observe low-power states were switched to by `TX`. Moreover, the nature of *race-to-halt* also guarantees no high-power states are employed during the waiting durations resulting from data dependencies and load imbalance, i.e., the two low-power durations in green where the application starts and ends. This indicates that `TX` is able to gain additional energy savings that all other approaches cannot exploit: Processes have to stay at the high-power state at the beginning/ending of the execution.

Energy Savings. Next we compare energy savings achieved by all five approaches (not including `CP_theo`) on parallel Cholesky, LU, and QR factorizations on the HPCL cluster as shown in Figure 4.6, where the energy consumption was measured by recording on/off the collection of power and time costs when an application starts/ends. For eliminating errors from scalability, we collected energy and time data of five matrix factorizations with different global matrix sizes ranging from 5120 to 25600, respectively. Considerable energy savings are achieved by all approaches except for `CPUSpeed` on all Cholesky, LU, and QR with similar energy saving trend: `TX` prevails over all other approaches with higher energy efficiency, while `SC_lib`, `Fermata`, and `Adagio` have similar energy efficiency. Overall, for Cholesky, `TX` can save energy 30.2% on average and up to 33.8%; for LU and QR, `TX` can achieve 16.0% and 20.0% on average and up to 20.4% and 23.4% energy savings, respectively. Due to the reasons discussed for power savings, `Adagio` only achieves similar energy savings

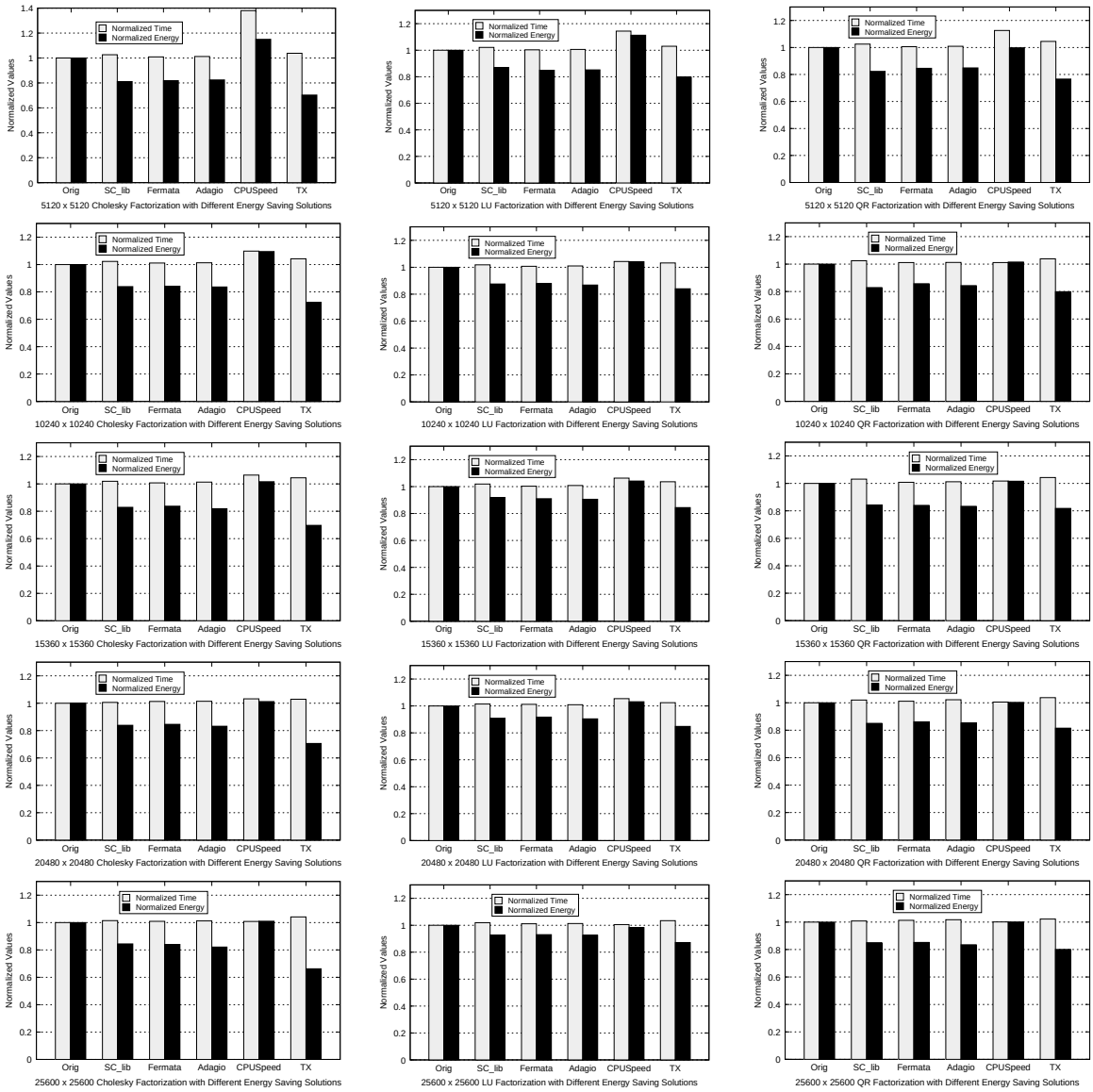


Figure 4.6: Energy and Performance Efficiency of Parallel Cholesky, LU, and QR Factorizations on the HPCL Cluster with Different Global Matrix Sizes and Energy Saving Approaches using 8×8 Process Grid.

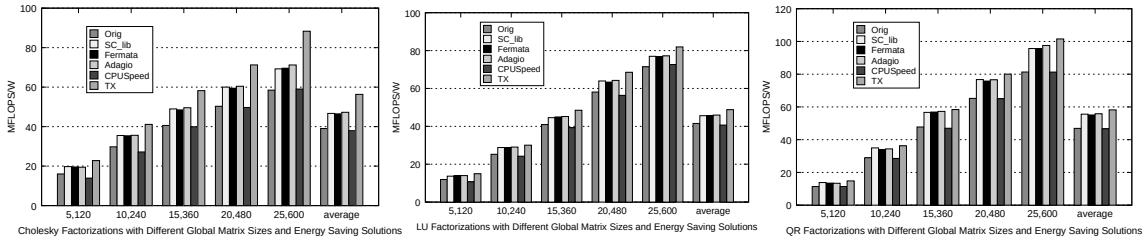


Figure 4.7: Energy and Performance Trade-off of Parallel Cholesky, LU, and QR Factorizations on the HPCL Cluster with Different Global Matrix Sizes and Energy Saving Approaches using 8×8 Process Grid.

as `SC_lib` and `Fermata`, without fulfilling additional energy savings from slack reclamation of computation. With application-specific knowledge instead of workload prediction, `TX` manages to achieve energy savings during both computation and communication slack. Moreover, `TX` benefits from the advantage of saving additional energy during possible load imbalance other approaches cannot exploit. Next we further evaluate such energy savings by increasing load imbalance in the applications.

Effects of Block Size. As discussed earlier, the additional energy savings can be achieved from load imbalance, i.e., the durations only covered by green dashed boxes as shown in Figure 4.4. Empirically, regardless of the workload partition techniques employed, load imbalance can grow due to larger tasks, longer communication, etc. For manifesting the strength of `TX` in achieving additional energy savings for *completeness*, we deliberately imbalance the workload through expanding tasks by using greater block sizes for Cholesky, while keeping the default block size for LU and QR. As shown in Figure 4.6, the average energy savings fulfilled by `TX` for Cholesky (30.2%) are consequently greater than LU and

QR (16.0% and 20.0%). Compared to the second most effective approach `Adagio`, `TX` can save Cholesky 12.8% more energy on average.

Performance Loss. Figure 4.6 also illustrates performance loss from different energy saving approaches against the original runs. We can see `TX` only incurs negligible time overhead: 3.8%, 3.1%, 3.7% on average for Cholesky, LU, and QR individually, similar to the time overhead of all other approaches except for `CPUSpeed`. The minor performance loss on employing these solutions is primarily originated from three aspects: (a) Although large-message communication is not CPU-bound, pre-computation required for starting up a communication link before any data transmission is necessary and is affected by CPU performance, so the low-power state during communication can slightly degrade performance; (b) switching CPU frequency via DVFS is essentially implemented by modifying CPU frequency system configuration files, and thus slight time overhead is incurred from the in-memory file read/write operations [132]; and (c) CPU frequency transition latency is required for the newly-set frequency to take effect. Further, `TX` suffers from minor performance loss from TDS analysis, including TDS generation and maintaining TDS for each task. The high time overhead of `CPUSpeed` is another reason for its little and even negative energy savings besides the defective prediction mechanism at OS level.

Energy/Performance Trade-off. An optimal energy saving approach requires to achieve the maximal energy savings with the minimal performance loss. Per this requirement, energy-performance integrated metrics are widely employed to quantify if the energy efficiency achieved and the performance loss incurred meanwhile are well-balanced. We adopt Energy-Delay Product (EDP) to evaluate the overall energy and performance trade-off of

the five approaches, in terms of MFLOPS/W, which equals the amount of floating-point operations per second within the unit of one Watt, i.e., the greater value it is, the better efficiency is fulfilled. As shown in Figure 4.7, compared to other approaches, `TX` is able to fulfill the most balanced trade-off between the energy and performance efficiency achieved. Specifically, `TX` has higher MFLOPS/W values for Cholesky compared to LU and QR, due to the higher energy savings achieved from the more imbalanced workload in Cholesky without additional performance loss.

4.6 Summary

The looming overloaded energy consumption of high performance scientific computing brings significant challenges to green computing in this era of ever-growing power costs for large-scale HPC systems. DVFS techniques have been widely employed to improve energy efficiency for task-parallel applications. With high generality, OS level solutions are regarded as feasible energy saving approaches for such applications. We observe for applications with variable execution characteristics such as parallel Cholesky, LU, and QR factorizations, OS level solutions suffer from the defective prediction mechanism and untapped potential energy savings from possible load imbalance, and thus cannot optimize the energy efficiency. Giving up partial generality, the proposed library level approach `TX` is evaluated to save more energy with little performance loss for parallel Cholesky, LU, and QR factorizations on two power-aware clusters compared to classic OS level solutions.

Chapter 5

Investigating the Interplay between Energy Efficiency and Resilience in High Performance Computing

Energy efficiency is one of the crucial challenges that must be addressed for High Performance Computing (HPC) systems to achieve ExaFLOPS (10^{18} FLOPS). The average power of the top five supercomputers worldwide has been inevitably growing to 10.1 MW today according to the latest TOP500 list [27], sufficient to power a city with a population of 20,000 people [25]. The 20 MW power-wall, set by the US Department of Energy [9] for exascale computers, indicates the severity of how energy budget constrains the performance improvement required by the ever-growing computational complexity of HPC applications.

Empirically, running HPC applications on supercomputers can be interrupted by failures including hardware breakdowns and soft errors. Although small on a single node, failure rates of large-scale computing systems can be of the order of magnitude of hours [72] due to a large amount of nodes interconnected as a whole. Greatly shrinked Mean Time To Failures (MTTF) of such systems at large scales entails less reliability. For instance, a compute node in a cluster of 692 nodes (22,144 cores in total) at Pacific Northwest National Laboratory can experience up to 1,700 ECC (Error-Correcting Code) errors in a two-month period [24]. K computer, ranked the first on the TOP500 list in June/Nov. 2011, held a hardware failure rate of up to 3% and affected by up to 70 soft errors in a month [14]. Larger forthcoming exascale systems are expected to suffer from more errors of different types in a fixed time period [118].

For HPC systems nowadays, both energy efficiency and resilience are considered to fulfill an optimal performance-cost ratio with a given amount of resources, as the trend of future exascaling computing implies. Widely studied individually, energy saving techniques and resilience techniques may restrict each other to attain the optimal ratio if employed jointly. In HPC runs, different forms of slack (discussed in detail later) can be exploited either as energy saving opportunities or as fault tolerance opportunities, since: (a) During the slack the peak performance of processors (e.g., CPU, GPU, and even memory) is generally not necessary for meeting time requirements of the applications and thus slack reclamation techniques can be employed to save energy [115] [102] [132], and (b) resilience techniques such as Checkpoint/Restart (CR) [110] [53] and Algorithm-Based Fault Tolerance (ABFT) [58] [48] can be performed during the slack without incurring performance loss overall.

Therefore, given a specific schedule of an HPC application, energy saving techniques and resilience techniques can compete for the fixed amount of slack, which can restrict each other to attain the best extent.

Regardless of the impacts of slack competition, energy efficiency and resilience are by nature two correlated but mutually constrained goals to achieve, from both theoretical and experimental perspectives. For energy saving purposes, reducing operating frequency and/or supply voltage of processors such as CPU will increase their failure rate, assuming that failures of combinational logic circuits follow a Poisson distribution (i.e., an exponential failure model) [119] [154], and thus incur more overhead on fault tolerance that may lead to performance loss and less energy savings. For reliability improving purposes, keeping operating frequency and/or supply voltage of the hardware at a conservatively high scale will lead to unnecessary energy costs without performance gain [39]. There exists an optimal trade-off between system reliability and energy costs. It is a challenging issue to accomplish the most balanced energy efficiency and resilience on large-scale systems nowadays without performance degradation, which is especially substantial to be addressed for the forthcoming exascale systems.

Lowering operating frequency and/or supply voltage of hardware components, i.e., DVFS (Dynamic Voltage and Frequency Scaling [141]), is one important approach to reduce power and energy consumption of a computing system for two primary reasons: First, CMOS-based components (e.g., CPU, GPU, and memory) are the dominant power consumers in the system. Second, power costs of these components are proportional to the product of operating frequency and supply voltage squared (shown in Figure 5.1). In gen-

eral, supply voltage has a positive correlation with (not strictly proportional/linear to) the operating frequency for DVFS-capable components [108], i.e., scaling up/down frequency results in voltage raise/drop accordingly. For HPC applications running on distributed-memory architectures, different forms of slack, i.e., idle time from hardware components (typically processors) during an HPC run, can result from numerous factors such as load imbalance, network latency, commmunication delay, and memory and disk access stalls. During the slack where the peak performance of processors is not necessary, energy savings can be achieved with negligible performance loss, by fine-grained slack utilization via DVFS (Dynamic Voltage and Frequency Scaling [141]) in two fashions: CP-aware (Critical Path) slack reclamation [115] and race-to-halt [64].

For detected slack, both DVFS techniques can effectively save energy by reducing frequency and voltage to different extents, due to the facts that dynamic power consumption of a CMOS-based processor is proportional to product of operation frequency and square of supply voltage, and that overall runtime is barely increased. For non-slack durations, e.g., computation, frequency reduction generally incurs a proportional reduction in runtime, which is usually not acceptable for HPC applications. Therefore, for such durations, both DVFS techniques employ the highest frequency and voltage to guarantee the performance overall.

Nevertheless, existing DVFS techniques are essentially frequency-directed and fail to fully exploit the potential power reduction and energy savings. With DVFS, voltage is only lowered to comply with the frequency reduction in the presence of slack [115]. For a given frequency, cutting-edge hardware components can be supplied with a voltage that is

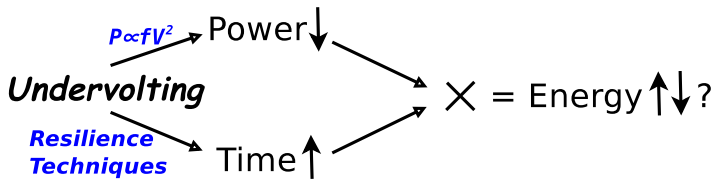


Figure 5.1: Entangled Effects of Undervolting on Performance, Energy, and Resilience for HPC Systems in General.

lower than the one paired with the given frequency. The enabling technique, *undervolting* [143] [29] [39] is independent of frequency scaling, i.e., lowering only supply voltage of a chip without reducing its operating frequency. Undervolting is advantageous in the sense that: (a) It can keep the component frequency unchanged such that the computation throughput is well maintained, and (b) it can be uniformly applied to both slack and non-slack phases of HPC runs for power reduction.

As a technique of high generality, undervolting is advantageous to save extra energy for any phases of HPC runs since required performance of processors is not degraded, at the cost of increased failure rates. Unlike traditional simulation-based approaches [143] [29], Bacha *et al.* [39] first implemented an empirical undervolting system on Intel Itanium II processors, which is intended for reducing voltage margins and thus saving power, with ECC memory correcting arising faults. Their work indeed maximized potential power savings since they used pre-production processors that allows thorough undervolting until the voltages lower than the lowest voltage corresponding to the lowest frequency supported. In general, production processors are locked for reliability purposes, and they will typically shut down when the voltage is lowered below the one corresponding to the minimum fre-

quency. For *generality* purposes, we propose a scheme that works for general production processors and thus can be deployed on large-scale HPC clusters nowadays.

Moreover, their work requires ECC memory to correct greatly-increased faults. Conventional supercomputer ECC memory extensively uses a Single Error Correcting, Double Error Detecting (SECDED) code [91] [142] [14], which relies on hardware support of ECC memory and can be limited to handle real-world hard and soft failures. For further achieving energy savings, a general and/or specialized lightweight resilience technique is expected to tolerate more complicated failures in the scenario of undervolting on HPC systems.

In conclusion, the challenge of employing undervolting as a general power saving technique in HPC lies in efficiently addressing the increasing failure rates caused by it. Both hard and soft errors may occur if components undergo undervolting. Several studies have investigated architectural solutions to support reliable undervolting with simulation. The study by Bacha *et al.* presented an empirical undervolting system on Intel Itanium II processors that resolves the arising Error-Correcting Code (ECC) memory faults yet improves the overall energy savings. While this work aims to maximize the power reduction and energy savings, it relies on pre-production processors that allow such thorough exploration on the undervolting schemes, and also requires additional hardware support for the ECC memory.

In this work, we investigate the interplay (shown in Figure 5.1) between energy efficiency and resilience for large-scale HPC systems, and demonstrate theoretically and empirically that significant energy savings can be obtained using a combination of undervolting

and mainstream software-level resilience techniques on today's HPC systems, without requiring hardware redesign. We aim to explore if the future exascale systems are going towards the direction of low-voltage embedded architectures in order to guarantee energy efficiency, or they can rely on advanced software-level techniques to achieve high system resilience and efficiency. In summary, the contributions of this chapter include:

- We propose an energy saving undervolting approach for HPC systems by leveraging resilience techniques;
- Our technique does not require pre-production machines and makes no modification to the hardware;
- We formulate the impacts of undervolting on failure rates and energy savings for mainstream resilience techniques, and model the conditions for energy savings;
- Our approach is experimentally evaluated to save up to 12.1% energy compared to the baseline runs of the 8 HPC benchmarks, and conserve up to 9.1% more energy than a state-of-the-art frequency-directed energy saving solution.

The remainder of the chapter is organized as follows: We theoretically model the problem in Section 5.1, and Section 5.2 presents the details of our experimental methodology. Results and their evaluation are provided in Section 5.3 and Section 5.4 concludes.

5.1 Problem Description and Modeling

5.1.1 Failure Rate Modeling with Undervolting

Failures in a computing system can have multiple root causes, including radiation from the cosmic rays and packaging materials, frequency/voltage scaling, and temperature fluctuation. There are two types of induced faults by nature: soft errors and hard errors. The former are transient (e.g., memory bit-flips and logic circuit errors) while the latter are usually permanent (e.g., node crashes from dysfunctional hardware and system abort from power outage). Soft errors are generally hard to detect (e.g., silent data corruption) since applications are typically not interrupted by such errors, while hard errors do not silently occur, causing outputs inevitably partially or completely lost. Note that here we use the terms *failure*, *fault*, and *error* interchangeably. In this work, we aim to theoretically and empirically study if undervolting with a fixed frequency (thus fixed throughput) is able to reduce the overall energy consumption. We study the interplay between the power reduction through undervolting and application performance loss due to the required fault detection and recovery at the raised failure rates from undervolting. Moreover, we consider the overall failure rates from both soft and hard errors, as the failure rates of either type can increase due to undervolting. Table 5.1 lists the key parameters used in the formulation and text henceforth.

Assume that the failures of combinational logic circuits follow a Poisson distribution, and the average failure rate is determined by the operating frequency and supply voltage [119] [154]. We employ an existing exponential model of average failure rate λ in

Table 5.1: Notation in the Formulation and Text.

f	Operating frequency of a core
d, β	Architecture-dependent constant
A	Percentage of active gates in a CMOS processor
C'	Total capacitive load in a CMOS processor
I_{sub}	Subthreshold leakage current of CPU
I'_{sub}	Subthreshold leakage current of non-CPU components
V_{dd}	Supply voltage of a core
V_{safe_min}	The lowest safe voltage for one core of a pre-production processor
V_{th}	Threshold voltage of a core
λ	Average failure rate
λ_0	Average failure rate at the maximum frequency and the maximum voltage
ϕ	Percentage of elapsed time in a segment of compute when an interrupt occurs and the process restarts
τ	Checkpoint interval
τ_{opt}	The optimal checkpoint interval that minimizes the total checkpoint and restart overhead
C	Checkpoint overhead
R	Restart overhead
N	Number of checkpoint interval in an HPC run
n	Number of failures in an HPC run
T_s	Solve/execution time of an HPC run
T_{slack}	Idle time from a core in an HPC run

terms of operating frequency f [154] without normalizing supply voltage V_{dd}, where λ_0 is the average failure rate at the maximum frequency f_{max} and voltage V_{max} ($f_{min} \leq f \leq f_{max}$ and $V_{min} \leq V_{dd} \leq V_{max}$):

$$\lambda(f, V_{dd}) = \lambda(f) = \lambda_0 \; e^{\frac{d(f_{max}-f)}{f_{max}-f_{min}}} \tag{5.1}$$

Exponent d is an architecture-dependent constant, reflecting the sensitivity of failure rate variation with frequency scaling. Previous work [151] modeled the relationship between operating frequency and supply voltage as follows:

$$f = \varphi(V_{dd}, V_{th}) = \beta \; \frac{(V_{dd} - V_{th})^2}{V_{dd}} \tag{5.2}$$

β is a hardware-related constant, and V_{th} is the threshold voltage. Substituting Equation (5.2) into Equation (5.1) yields:

$$\lambda(f, V_{dd}) = \lambda(V_{dd}) = \lambda_0 \; e^{\frac{d(f_{max}-\beta(V_{dd}-2V_{th}+\frac{V_{th}^2}{V_{dd}}))}{f_{max}-f_{min}}} \tag{5.3}$$

Equation (6.1) indicates that the average failure rate is a function of supply voltage V_{dd} only, provided that V_{th} and other parameters are fixed. This condition holds true when undervolting is studied in this work. Denote $\sigma(V_{dd}) = \frac{d(f_{max}-\beta(V_{dd}-2V_{th}+\frac{V_{th}^2}{V_{dd}}))}{f_{max}-f_{min}}$. We calculate the first derivative of $\lambda(f, V_{dd})$ (Equation (6.1)) with respect to V_{dd} and identify values of V_{dd} that make the first derivative zero as follows:

$$\frac{\partial \lambda}{\partial V_{dd}} = \lambda_0 \; e^{\sigma(V_{dd})} \; \frac{-d\beta(1 - (V_{th}/V_{dd})^2)}{f_{max} - f_{min}} = 0 \tag{5.4}$$

$$\Rightarrow V_{dd} = \pm V_{th} \tag{5.5}$$

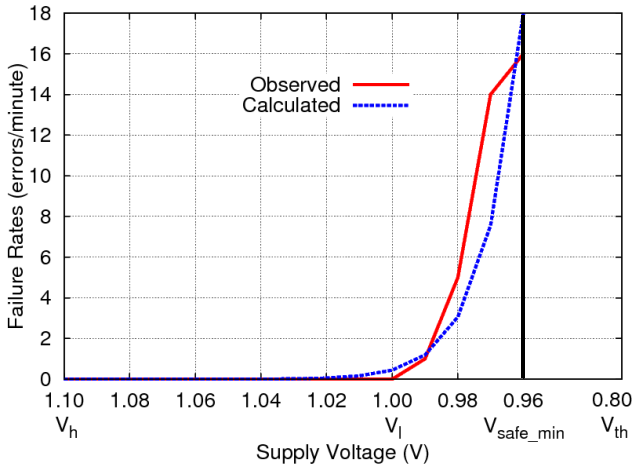


Figure 5.2: Observed and Calculated Failure Rates λ as a Function of Supply Voltage V_{dd} for a Pre-production Intel Itanium II 9560 8-Core Processor (Note that the observed failures are ECC memory correctable errors for one core. V_h: the maximum voltage paired with the maximum frequency; V_l: the minimum voltage paired with the minimum frequency).

Therefore, for $-V_{th} < V_{dd} < V_{th}$, $\lambda(f, V_{dd})$ monotonically strictly increases as V_{dd} increases; for $V_{dd} \leq -V_{th}$ and $V_{dd} \geq V_{th}$, $\lambda(f, V_{dd})$ monotonically strictly decreases as V_{dd} increases. Given that empirically $V_{dd} \geq V_{th}$, we conclude that $\lambda(f, V_{dd})$ is a monotonically strictly decreasing function for all valid V_{dd} values. $\lambda(f, V_{dd})$ maximizes at $V_{dd} = V_{th}$.

Example. Figure 5.2 shows the comparison between the experimentally observed and theoretically calculated failure rates with regard to supply voltage for a pre-production processor, where the observed data is from [39] and the calculated data is via Equation (6.1). As shown in the figure, the calculated values are very close to the observed ones, which demonstrates that Equation (6.1) can be used to model failure rate. Based on the voltage parameters from the vendor [13], we denote several significant voltage levels in

Figure 5.2, where V_h and V_l refer to the the maximum and minimum supply voltages for a production processor. They also pair with the maximum and minimum operating frequencies respectively. In many cases, undervolting is disabled on production processors by vendors based on the assumption that there are no adequate fault tolerance support at the software stack, and it often shuts down a system when its supply voltage is scaled below V_l. For some pre-production processors [39], supply voltage may be further scaled to V_{safe_min}, which refers to the theoretical lowest safe supply voltage under which the system can operate without crashing. But even for these pre-production processors, when V_{dd} is actually reduced below V_{safe_min}, they no longer operate reliably [39].

Note that although there are no observed faults in Figure 5.2 for the voltage range from 1.10 V to 1.00 V, the calculated failure rates are not zero, ranging from 10^{-6} to 10^{-1}. This difference suggests that failures with a small probability do not often occur in real runs. For the voltage range from 0.99 V to 0.96 V, the calculated failure rates match the observation with acceptable statistical errors. The calculated failure rates for voltage levels that are lower than V_{safe_min} are not presented here due to the lack of observational data for comparison. Unless some major circuit-level redesign on the hardware for enabling NTV, we commonly consider the voltage range between V_{safe_min} and V_{th} inoperable in HPC even with sophisticated software-level fault tolerant techniques. Thus, modeling this voltage range is out of the scope of this work.

5.1.2 Performance Modeling under Resilience Techniques

Resilience techniques like Checkpoint/Restart (C/R) and Algorithm-Based Fault Tolerance (ABFT) have been widely employed in HPC environments for fault tolerance.

State-of-the-art C/R and ABFT techniques are lightweight [110] [146], scalable [109] [146], and sound [58]. C/R is a general resilience technique that is often used to handle hard errors (it can also recover soft errors if errors can be successfully detected). ABFT is more cost-efficient than C/R to detect and correct soft errors. But it is not as general as C/R because it leverages algorithmic knowledge of target programs and thus only works for specific applications. Next we briefly illustrate how they function, and present the formulated performance models of both techniques.

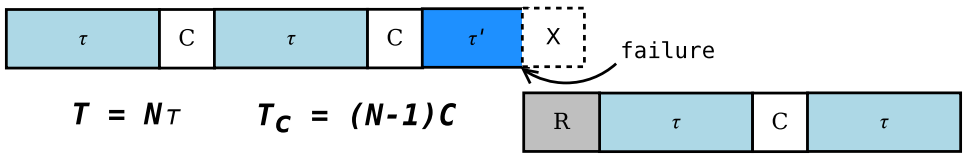


Figure 5.3: Checkpoint/Restart Execution Model for a Single Process.

A checkpoint is a snapshot of a system state, i.e., a copy of the contents of application process address space, including all values in the stack, heap, global variables, and registers. Classic C/R techniques save checkpoints to disks [59], memory [110], and both disks and memory via multi-level checkpointing [109]. Figure 5.3 shows how a typical C/R scheme recovers a single-process failure, where we denote checkpoint overhead as C, restart overhead as R, and compute time between checkpoints as τ respectively. An interrupting failure can arise at any given point of an execution. C/R can capture the failure and restart the application from the nearest saved checkpoint with the interrupted compute period re-executed.

Next we present the issue of determining the optimal checkpoint interval, i.e., τ_{opt} that can minimize the total checkpoint and restart overhead, given a failure rate of λ. This is significant since in the scenario of undervolting, failure rates may vary dramatically as shown in Figure 5.2, which could affect τ_{opt} considerably. Without considering the impacts of undervolting, several efforts on estimating τ_{opt} have been proposed. Young [149] approximated $\tau_{opt} = \sqrt{\frac{2C}{\lambda}}$ as a first-order derivation. Taking restart overhead R into account, Daly [53] refined Young's model into $\tau_{opt} = \sqrt{2C(\frac{1}{\lambda} + R)}$ for $\tau + C \ll \frac{1}{\lambda}$. Using a higher order model, the case that checkpoint overhead C becomes large compared to MTTF (Mean Time To Failure), $\frac{1}{\lambda}$ was further discussed for a perturbation solution in [53]:

$$\tau_{opt} = \begin{cases} \sqrt{\frac{2C}{\lambda}} - C & \text{for } C < \frac{1}{2\lambda} \\ \\ \frac{1}{\lambda} & \text{for } C \geq \frac{1}{2\lambda} \end{cases} \tag{5.6}$$

Note that R has no contributions in Equation (5.6) . Since Equation (5.6) includes the failure rate λ discussed in Equation (6.1), it is suitable to be used to calculate τ_{opt} in the scenario of undervolting.

Consider the basic time cost function of C/R modeled in [53]: $T_{cr} = T_s + T_c + T_w + T_r$, where T_s refers to the solve time of running an application with no interrupts from failures. The total checkpoint overhead is denoted as T_c. T_w is the amount of time spent on an interrupted compute period before a failure occurs, and T_r is defined as the total time on restarting from failures. Figure 5.3 shows the case of a single failure, and how the time fractions are accumulated. Next we generalize the time model to accommodate the case of multiple failures in a run as follows:

$$T_{cr} = N\tau + (N-1)C + \phi(\tau + C)n + Rn \tag{5.7}$$

where N is the number of compute periods and n is the number of failures within a run ($N-1$ is because there is no need for one more checkpoint if the last compute period is completed). ϕ is the percentage of elapsed time in a segment of compute when an interrupt occurs and the process restarts. And we adopt a common assumption that interrupts never occur during a restart. As a constant, $N\tau$ can be denoted as T_s and Equation (5.7) is reformed as:

$$T_{cr} = T_s + (\frac{T_s}{\tau} - 1)C + \phi(\tau + C)n + Rn \tag{5.8}$$

We adopt Equation (5.8) as the C/R time model henceforth. In the absence of failures, this model is simplified with the last two terms omitted. Next we introduce how ABFT works and present its performance model formally.

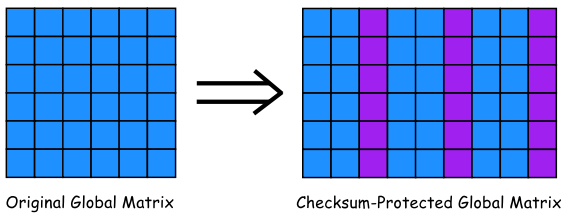


Figure 5.4: Algorithm-Based Fault Tolerance Model for Matrix Operations.

Figure 5.4 shows how ABFT protects the matrix operations from soft errors using the row-checksum mechanism. The checksum blocks are periodically maintained in between the original matrix elements by adding an extra column to the process grid. The redundant checksum blocks contain sums of local matrix elements in the same row, which can be

used for recovery if one original matrix element within the protection of a checksum block is corrupted. In C/R, checkpoints are periodically saved. Likewise, checksum blocks are periodically updated, generally together with the matrix operations. The difference is that the interval of updating the checksum in ABFT is fixed, i.e., not affected by the variation of failure rates, while in C/R, the optimal checkpoint interval that minimizes C/R overhead highly depends on failure rate λ.

Our previous work [146] has modeled ABFT overhead for the dense matrix factorizations, including Cholesky, LU, and QR factorizations. Here we present the overall performance model for ABFT-enabled dense matrix factorizations, taking Cholesky factorization for example due to algorithmic similarity.

$$T_{abft} = \frac{\mu C_f \mathbb{N}^3}{\mathbb{P}} t_f + \frac{\mu C_v \mathbb{N}^2}{\sqrt{\mathbb{P}}} t_v + \frac{C_m \mathbb{N}}{nb} t_m + T_d + T_l + T_c \tag{5.9}$$

where $\mathbb{N}$ represents the dimensions of the global matrix, $\mathbb{P}$ is the total number of processes, and nb is the block size for data distribution. $\mu = 1 + \frac{4}{nb}$ is the factor of introducing checksum rows/columns to the global matrix (the actual factorized global matrix size is $\mu\mathbb{N}$). Cholesky-specific constants $C_f = \frac{1}{3}$, $C_v = 2 + \frac{1}{2}\log\mathbb{P}$, and $C_m = 4 + \log\mathbb{P}$. t_f is the time required per FLoating-point OPeration (FLOP), t_v is the time required per data item communicated, and t_m is the time required per message prepared for transmission. Error detection overhead is denoted as $T_d = \frac{\mathbb{N}^2}{\mathbb{P}} t_f$, and error localization overhead is denoted as $T_l = \frac{nb\mathbb{P}}{\mathbb{N}^3} t_f$. Since one error correction operation only requires one FLOP, the error correction overhead can be described as $T_c = nt_f$. Similarly as in the C/R performance model, the error-related overhead is only valid in the presence of failures. Otherwise the

last three terms are omitted and the sum of the first three terms represents the performance of running Cholesky with ABFT.

Due to the conceptual similarity and space limitation, we only model the performance of these two resilience techniques in this work.

5.1.3 Power and Energy Modeling under Resilience Techniques and Undervolting

In this subsection, we present general power and energy models under a combination of undervolting and resilience techniques. We use C/R as an example for model construction, which can also be applied to other resilience techniques used in this work. Since undervolting will affect the processor power most directly and the processor power has the most potential to be conserved in an HPC system, we assume all the power/energy savings come from processors. Therefore, we focus on modeling processor-level energy. Using the energy models, we can explore in theory what factors will likely affect the overall energy savings through undervolting.

First, we adopt the nodal power model in [129] [122] as follows:

$$\begin{aligned} P &= P_{dynamic}^{CPU} + P_{leakage}^{CPU} + P_{leakage}^{other} \\ &= AC'fV_{dd}^2 + I_{sub}V_{dd} + I'_{sub}V'_{dd} \end{aligned} \tag{5.10}$$

A and C' are the percentage of active gates and the total capacitive load in a CMOS-based processor; I_{sub} and I'_{sub} are subthreshold leakage current of CPU and non-CPU components, and V_{dd} and V'_{dd} are their supply voltages. We denote $I'_{sub}V'_{dd}$ as a

constant P_c since this term is independent of undervolting. Previous work [106] indicates that the power draw of a node during C/R is very close to its idle mode. Therefore, by leveraging this power characteristic for C/R phases and different levels of voltages in Figure 5.2, we propose the following power formulae for a given node:

$$\begin{cases} P_h &= AC'f_hV_h^2 + I_{sub}V_h + P_c \\ P_m &= AC'f_hV_{safe\_min}^2 + I_{sub}V_{safe\_min} + P_c \\ P_l &= AC'f_lV_{safe\_min}^2 + I_{sub}V_{safe\_min} + P_c \end{cases} \tag{5.11}$$

In this Equation, both P_h and P_m use the highest frequency, while P_l scales frequency to the minimum; both P_m and P_l exploit V_{safe_min} to save energy. Here are the scenarios where P_h, P_m, and P_l are mapped into: For the baseline case, we employ P_h to all execution phases of an HPC run. For the energy-optimized case, we apply undervolting to different phases based on their characteristics using P_m and P_l, in order to achieve the optimal energy savings through leveraging resilience techniques. Specifically, without harming the overall performance, we apply P_l to the frequency-insensitive phases including *non-computation* (e.g., *communication*, *memory* and *disk accesses*) and *C/R* phases, while P_m is applied in all the *computation* phases. Using Equations (5.8) and (5.11), we can model the energy costs of a baseline run (E_{base}), a run with undervolting but in the absence of failures ($E_{uv}^{\overline{err}}$), and a run with undervolting in the presence of failures (E_{uv}) as:

$$\begin{cases} E_{base} = P_h T_s \\ E_{uv}^{\overline{err}} = P_m T_s + P_l(\frac{T_s}{\tau} - 1)C \\ E_{uv} = P_m(T_s + \phi\tau n) + P_l\left(\left(\frac{T_s - \tau}{\tau} + \phi n\right)C + Rn\right) \end{cases} \tag{5.12}$$

For processor architectures equipped with a range of operational frequencies, frequency-directed DVFS techniques [141] [115] have been widely applied in HPC for energy saving purposes. Commonly, they predict and apply appropriate frequencies for different computational phases based on workload characteristics. Meanwhile, for the selected frequency (or two frequencies in the case of split frequencies [115]) f_m ($f_l < f_m < f_h$), a paired voltage V_m ($V_l < V_m < V_h$) will also be applied accordingly. One important question emerges: can we further save energy beyond these DVFS techniques by continuing undervolting V_m? To answer this question, we compare our approach with a state-of-the-art DVFS technique named Adagio [115] as an example to demonstrate the potential energy savings from undervolting beyond DVFS. Basically, Adagio runs aside with HPC applications, identifies computation and communication slack, and then apply appropriately reduced frequencies accordingly to save energy without sacrificing the overall performance. We will use the frequencies predicted by Adagio for each phase but further reduce V_m. Therefore, we have:

$$\begin{cases} P_{Adagio}^{slack} = AC' f_m V_m^2 + I_{sub} V_m + P_c \\ P_{Adagio}^{non-slack} = P_h \\ P_{uv}^{slack} = AC' f_m V_{safe\_min}^2 + I_{sub} V_{safe\_min} + P_c \\ P_{uv}^{non-slack} = P_m \end{cases} \tag{5.13}$$

$$E_{Adagio} = P_{Adagio}^{slack} T_{slack} \oplus P_{Adagio}^{non-slack} T_s \tag{5.14}$$

P_{slack} and $P_{non-slack}$ denote the average power during the slack and non-slack phases respectively; T_{slack} is the slack duration due to task dependencies and generally overlaps with T_s across processes [129] (thus we use $\oplus$ instead of $+$). Assume there exists a percentage η $(0 < \eta < 1)$ of the total slack that overlaps with the computation. Therefore we define $\oplus$ by explicitly summing up different energy costs:

$$P_1 T_{slack} \oplus P_2 T_s =$$

$$P_1 T_{slack}(1-\eta) + P_{hybrid} T_{slack} \eta + P_2 T_s \tag{5.15}$$

where P_1 and P_2 denote the nodal power during the slack and computation respectively, and P_{hybrid} $(P_1 < P_{hybrid} < P_2)$ is the average nodal power when slack overlaps the computation. Using Equations (5.12), (5.13), (5.14), and (5.15), we can model the energy consumption E'_{uv}[1] which integrates $(\uplus)$ the advantages of both DVFS techniques (Adagio) and the undervolting beyond DVFS, in the presence of slack:

$$E'_{uv} = E_{Adagio} \uplus E_{uv} = P_{uv}^{slack} T_{slack} \oplus E_{uv} \tag{5.16}$$

Potential energy savings through appropriate undervolting over a baseline run and an Adagio-enabled run can then be calculated as follows:

[1]Since here we treat all cores uniformly during undervolting, i.e., all cores undervolt simultaneously to the same supply voltage, we do not have number of cores as a parameter in our model. But it can be modeled for more complex scenarios.

$$\begin{aligned}\Delta E_1 &= E_{base} - E_{Adagio} \\ &= (P_h - P_{Adagio}^{slack})T_{slack} \\ &= (AC'(f_h V_h^2 - f_m V_m^2) + I_{sub}(V_h - V_m))T_{slack}\end{aligned} \tag{5.17}$$

$$\begin{aligned}\Delta E_1 &= E_{base} - E'_{uv} \\ &= (P_h - P_m)T_s \oplus (P_h - P_{uv}^{slack})T_{slack} - \\ &\quad \left(P_m \phi \tau n + P_l \left(\left(\frac{T_s - \tau}{\tau} + \phi n\right) C + Rn\right)\right)\end{aligned} \tag{5.18}$$

$$\begin{aligned}\Delta E_2 &= E_{Adagio} - E'_{uv} \\ &= (P_h - P_m)T_s \oplus (P_{Adagio}^{slack} - P_{uv}^{slack})T_{slack} - \\ &\quad \left(P_m \phi \tau n + P_l \left(\left(\frac{T_s - \tau}{\tau} + \phi n\right) C + Rn\right)\right)\end{aligned} \tag{5.19}$$

Discussion. Equation (5.17) quantifies energy savings from the DVFS technique Adagio over a baseline run where no energy efficient techniques are employed. We have the following inferences: (a) All energy savings are from appropriately reducing power during T_{slack}, since during computation, Adagio keeps the peak performance of processors to meet performance requirements, which consumes the same energy costs as the baseline run, (b) energy savings can always be achieved ($\Delta E_1 > 0$), given $f_h V_h^2 - f_m V_m^2 > 0$ and $V_h - V_m > 0$. Note that we assume the DVFS operations from Adagio (including slack calculation, frequency approximation, and DVFS itself) costs negligible overhead, as claimed in [115], and (c) the

amount of energy savings achieved is determined by essentially T_{slack}, which affects both the length of time when P^{slack}_{Adagio} is applied, and the value of P^{slack}_{Adagio} since the values of f_m and V_m are determined by T_{slack} (Adagio assumes all slack can be eliminated). Without affecting the analysis, in Equation (5.17), we only consider computation and slack arising from computation for simplicity.

From Equations (5.18) and (5.19), we can observe that the potential energy savings from undervolting is essentially the difference between two terms: Energy savings gained from undervolting (denoted as E_+) and energy overhead from fault detection and recovery due to the increasing errors caused by reduced voltage (denoted as E_-). In other words, the trade-off between the power savings through undervolting and performance overhead for coping with the higher failure rates determines if and how much energy can be saved overall. In E_+, two factors are significant: V_{safe_min} and T_{slack}. They impact the maximum energy savings and generally depend on chip technology and application characteristics. In E_-, three factors are essential: n, C, and R, where n, the number of failures in a run, is affected by undervolting directly. C and R rely on the resilience techniques being employed. Some state-of-the-art resilience techniques (e.g., ABFT) have relatively low values of C and R [58], which can be beneficial to more energy savings.

Here we showcase how to use the above models to explore the energy saving capability of undervolting. We apply the parameters from our real system setup shown in Table 6.2 into Equation (5.18). In order to investigate the relationship among C, R, λ, and the overall energy savings, we let Equation (5.18) > 0 and solve the inequation with appropriate assumptions.

Using the HPC setup in Table 6.2, we have $f_h = 2.5$, $f_l = 0.8$ (we assume $f_m = 1.8$ for Adagio), $V_h = 1.3$, $V_l = 1.1$, and $V_{safe\_min} = 1.025$. We solve $AC' = 20$, $I_{sub} = 5$, and $P_c = 107$, estimate $\phi = \frac{1}{2}$, and adopt $n = \lambda T_s$, $\tau_{opt} = \frac{1}{\lambda}$ (Equation (5.6)) using the methodology in Section 5.2. For simplicity, we assume slack does not overlap any computation and $T_{slack} = 0.2T_s$. So $\oplus$ in Equation (5.17) > 0 becomes $+$. First we consider the case of $C \geq \frac{1}{2\lambda}$. According to Equation (5.6), let $\tau = \tau_{opt}$ and now we have:

$$33.34375\ T_s + 9.6105\ T_s - 164.65625 \times \frac{1}{2} T_s - 128.935 \left(\left(\lambda T_s - 1 + \frac{1}{2} \lambda T_s \right) C + R \lambda T_s \right) > 0 \tag{5.20}$$

It is clear that the sum of the first three terms is negative and the fourth term is positive. Thus in this case no energy savings can be achieved using undervolting. The conclusion continues to stand even if we let $T_{slack} = T_s$ to increase the second term. It is due to the high failure rate that causes the third term overlarge. Next we consider the other case ($\tau_{opt} = \sqrt{\frac{2C}{\lambda}} - C$) in Equation (5.6). Thus the inequation becomes:

$$33.34375\ T_s + 9.6105\ T_s - 164.65625 \times \frac{1}{2} (\sqrt{2\lambda C} - \lambda C) T_s - 128.935 \left(\left(\frac{\lambda T_s}{\sqrt{2\lambda C} - \lambda C} - 1 + \frac{1}{2} \lambda T_s \right) C + R \lambda T_s \right) > 0 \tag{5.21}$$

$$\Rightarrow T_s > \frac{c_3 C}{c_3 \left(\frac{\lambda C}{\sqrt{2\lambda C} - \lambda C} + \frac{1}{2} \lambda C + R\lambda \right) - c_1 + c_2 (\sqrt{2\lambda C} - \lambda C)} \tag{5.22}$$

where $c_1 = 42.95425$, $c_2 = 82.328125$, and $c_3 = 128.935$

If the above condition (5.22) is satisfied, energy savings can be achieved. The condition of T_s can be also reformed into an inequation of C or R in terms of T_s and λ,

without losing generality. Using Equation (6.1), we can also substitute λ with an expression of V_{dd}. Recall that it needs to meet $C < \frac{1}{2\lambda}$ to yield the case $\tau_{opt} = \sqrt{\frac{2C}{\lambda}} - C$.

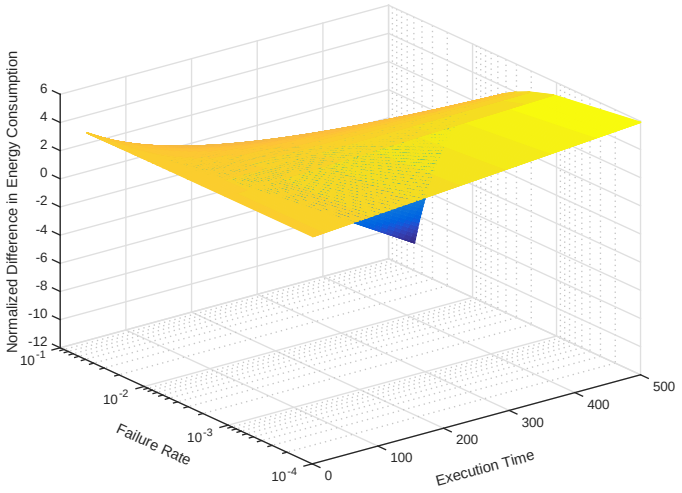


Figure 5.5: Normalized Difference in Energy Consumption w/ and w/o Undervolting and Resilience Techniques.

Although Inequation (5.22) reflects the relationship that needs to be satisfied for energy savings, we can further explore the numerical solution of Equation (5.17) to see the energy saving trend clearly. Figure 5.5 depicts the curve of normalized difference in energy consumption between the original energy costs and the energy costs with undervolting and resilience techniques, using the same assumptions when deriving Inequation (5.22). Moreover, we focus on the effects from λ and T_s, and thus set $C = 10$ and $R = 20$ in seconds for the C/R technique used. From the τ_{opt} case condition $C < \frac{1}{2\lambda}$, we calculate the upper bound of λ as 5×10^{-2}. We use 10^{-4} as the lower bound of λ since it is sufficient to show the trend. We let T_s ranging from 0 to 500 to ensure it covers all execution time of

HPC runs in our experiments. From Figure 5.5, we can see that as T_s increases, more energy savings can be achieved from undervolting. However, when λ increases, energy savings are reduced greatly, for large T_s values in particular. Generally, when λ is small, e.g., within the range $[10^{-4}, 10^{-3}]$, energy savings can be achieved and the variation of T_s barely affects energy efficiency. For large λ and T_s, energy savings cannot be obtained by undervolting due to largely increased fault detection and recovery overhead, as λ increases exponentially (faster than the increase of T_s), such that Inequations (5.21) and (5.22) do not hold.

Similarly, we let Equation (5.19) > 0 and solve the inequation using the above empirical assumptions. In the two cases of τ_{opt}, we obtain two inequations that are similar to Inequations (5.20) and (5.21) individually, except having a smaller second term of T_s. This is straightforward since $P_{Adagio}^{slack} < P_h$, the only difference between Equations (5.18) and (5.19). Therefore, we have similar results and analysis of energy savings from undervolting over DVFS techniques (Adagio).

Model Relaxation. We notice that the relationship among C, R, and λ can be obtained without T_s, if the C/R performance model (see Equation (5.7)) is relaxed. In the scenario of undervolting, τ_{opt} is comparatively small and thus the number of checkpoints $N = \frac{T_s}{\tau_{opt}}$ is large, due to the high failure rates λ. Therefore, we consider $N - 1 \approx N$ in Equation (5.7), and the term -1 in Inequation (5.21) can be ingored, by which T_s in the inequation can be eliminated. Consequently, Inequation (5.21) can be relaxed into:

$$33.34375 + 9.6105 - 164.65625 \times \frac{1}{2}(\sqrt{2\lambda C} - \lambda C) - 128.935\left(\left(\frac{\lambda}{\sqrt{2\lambda C} - \lambda C} + \frac{1}{2}\lambda\right)C + R\lambda\right) > 0 \tag{5.23}$$

$$\Rightarrow R < \frac{c_1}{c_3\lambda} - \frac{c_2}{c_3}\left(\sqrt{\frac{2C}{\lambda}} - C\right) - \left(\frac{1}{\sqrt{2\lambda C} - \lambda C} + \frac{1}{2}\right)C \tag{5.24}$$

In the relaxed performance and energy models, energy savings can be fulfilled as long as the above relationship (5.24) holds. Again it is required that $C < \frac{1}{2\lambda}$ in order to yield $\tau_{opt} = \sqrt{\frac{2C}{\lambda}} - C$, likewise in the non-relaxed models.

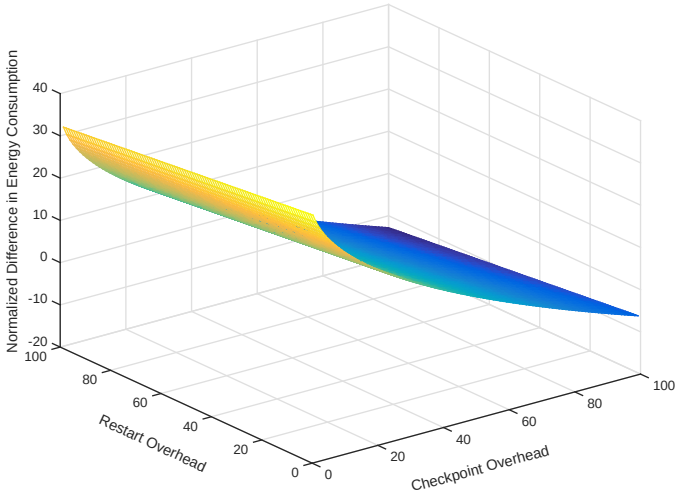


Figure 5.6: Normalized Difference in Energy Consumption w/ and w/o Undervolting and Resilience Techniques (Relaxed).

Figure 5.6 shows the numerical solution of the relaxed model of energy saving difference, i.e., the left hand side of Inequation (5.24), which helps us see the energy saving effects from interested parameters in a straightforward fashion. Since T_s is eliminated after relaxation, we focus on the effects from C and R in this case. We choose [0, 100] in seconds as the range of C and R from the measured data collected during our experiments. We let λ

$= 5 \times 10^{-4}$ such that $C < \frac{1}{2\lambda}$ is satisfied. We can see from the figure a straightforward trend that the variation of C has greater impacts on energy savings compared to that of R: As C becomes larger, energy savings are monotonically decreased until negative values. The growing of R barely improves energy efficiency when C is small, while larger C values make energy saving impacts from R more manifested. Generally, small/large C and R values bring higher energy savings due to small/large overhead on fault detection and recovery, which matches well with Inequations (5.23) and (5.24) that indicate given λ and C, smaller R achieves higher energy savings from undervolting.

Likewise, we can relax the model for Equation (5.19) and the solution is in similar form as Inequation (5.24). The energy saving difference curve is similar to Figure 5.6 as well.

5.2 Experimental Methodology

5.2.1 Experimental Setup and Benchmarks

Table 6.2 lists the hardware configuration of the power-aware cluster used for all experiments. Although the size of the cluster is comparatively small, it is sufficient for the proof of concept of our techniques. For power/energy measurement, PowerPack [71], a framework for profiling and analysis of power/energy consumption of HPC systems, was deployed on a separate meter node to collect power/energy profiling data of the hardware components (e.g. CPUs and Memory) on this cluster. This data was recorded in a log file on the meter node and used for post-processing.

Table 5.2: Hardware Configuration for All Experiments.

Cluster	HPCL
System Size	64 Cores, 8 Compute Nodes
Processor	AMD Opteron 2380 (Quad-core)
CPU Frequency	0.8, 1.3, 1.8, 2.5 GHz
CPU Voltage	1.300, 1.100, 1.025, 0.850 V ($V_h/V_l/V_{safe\_min}/V_{th}$)
Memory	8 GB RAM
Cache	128 KB L1, 512 KB L2, 6 MB L3
Network	1 GB/s Ethernet
OS	CentOS 6.2, 64-bit Linux kernel 2.6.32
Power Meter	PowerPack

Table 5.3: Empirical Resilience Techniques and Applicable Failures.

Resilience Technique	Recovery Model	Failure Type
Disk-Based Checkpoint/Restart (DBCR)	Backward	Hard Errors
Diskless Checkpointing (DC)		
Triple Modular Redundancy (TMR)	Retry	Soft Errors
Algorithm-Based Fault Tolerance (ABFT)	Local/Global	

Table 5.3 shows several mainstream resilience techniques and their applicable failures. Benchmarks used in this chapter are selected from NPB [20] benchmark suite, LULESH [17], AMG [22], and our fault tolerant ScaLAPACK [146]. In the next section, we will show the performance and energy results of running these applications under various resilience techniques and demonstrate if energy savings can actually be obtained using a combination of undervolting and resilience techniques on a conventional HPC system.

5.2.2 Failure Rate Calculation

Recall that the limitation of applying undervolting in HPC is that production machines used in the HPC environment do not allow further voltage reduction beyond the point of V_l, shown in Figure 5.2. To estimate failure rates between V_l and V_{safe_min}, we use

Table 5.4: Northbridge/CPU FID/VID Control Register Bit Format.

Bits	Description
63:32, 24:23, 21:19	Reserved
32:25	Northbridge Voltage ID, Read-Write
22	Northbridge Divisor ID, Read-Write
18:16	P-state ID, Read-Write
15:9	Core Voltage ID, Read-Write
8:6	Core Divisor ID, Read-Write
5:0	Core Frequency ID, Read-Write

Equation (6.1) to calculate the failure rates used in our experiments. As demonstrated in Figure 5.2, our calculated data matches well with the observed data from [39]. Thus the calculated failure rates are appropriate for empirical evaluation, although no real failures can be observed beyond V_l on our platform.

5.2.3 Undervolting Production Processors

Unlike the undervolting approach used in [39] through software/firmware control on a pre-production processor, we conduct undervolting for a production cluster by directly modifying corresponding bits of the northbridge/CPU FID and VID control register [35], where FID and VID refer to frequency and voltage ID numbers respectively. This process needs careful detection of the upper and lower bounds of processors' supply voltage. Otherwise overheat and hardware-damaging issues may arise. The register values consist of 64 bits in total, where different bit fragments manage various system power state variables individually. Table 6.3 summarizes the register bit format [35] for processors on the HPCL cluster: The Core Voltage/Frequency/Divisor ID fragments (CoreVid/CoreFid/CoreDid) are used for undervolting. As a general-purpose software level undervolting approach, the

interested bits of register values are altered using the Model Specific Register (MSR) interface [19]. Next we illustrate how to extract various ID fragments from specific register values and modify voltage/frequency of cores using corresponding formula. For instance, we input the register with a hexadecimal value 0x30002809 via MSR. From the bit format, we can extract the Core Voltage/Frequency/Divisor ID as 20, 9, and 0 respectively. Moreover, from [35], we have the following architecture-dependent formulae to calculate voltage/frequency:

$$\text{frequency} = 100\text{MHz} \times (\text{CoreFid} + 16)/2^{\text{CoreDid}} \tag{5.25}$$

$$\text{voltage} = 1.550\text{V} - 0.0125\text{V} \times \text{CoreVid} \tag{5.26}$$

Given the register value 0x30002809, it is easy to calculate voltage/frequency to be 1.300 V and 2.5 GHz individually using the above equations. Using MSR, undervolting is implemented by assigning the register with desirable voltage values at the voltage bits. The frequency bits are unchanged to ensure fixed frequency during undervolting.

5.2.4 Error Injection and Energy Cost Estimation

As previously discussed in Section 5.1.1, production machines commonly disable the further voltage reduction below V_l. Thus, we need to emulate the errors that may appear for the voltage range between V_l and V_{safe_min}, based on the failure rates calculated by Equation (6.1). Specifically, we inject hard and soft errors respectively at the calculated failure rates to emulate the incurred failures in HPC runs due to undervolting to such voltage levels. For instance, for emulating hard errors occurring at V_{safe_min}, our fault injector

injects errors by arbitrarily killing parallel MPI processes during program execution at OS level. For emulating soft errors, our injector randomly modifies (e.g. bit-flips) the values of matrix elements to erroneous ones at library level for matrix-based HPC applications, likewise as in [146]. Soft error detection is done within the error checking module of matrix benchmarks, as a part of the ABFT technique. Although, due to the hardware constraints, energy costs cannot be experimentally measured when undervolting to V_{safe_min}, we apply the following *emulated scaling* method to estimate the energy costs at V_{safe_min}.

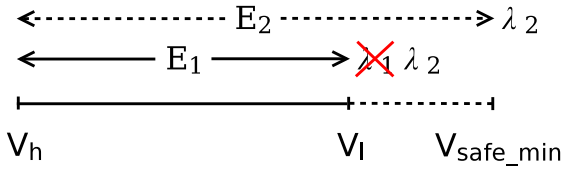


Figure 5.7: Estimating Energy Costs with Undervolting at V_{safe_min} for Production Processors via Emulated Scaling.

Figure 5.7 demonstrates this emulated scaling mechanism, where E_1 and E_2 refer to the energy costs at V_l and V_{safe_min} respectively. λ_1 and λ_2 are the calculated failure rates at the two voltage levels via Equation (6.1). For estimating the energy cost E_2 at V_{safe_min}, we inject the errors at the rate of λ_2 at V_l instead of using the failure rate λ_1. Moreover, in Equation (5.11), replacing V_{safe_min} with V_l, we can solve AC', I_{sub}, and P_c by measuring the system power consumption at V_l by applying P_h, P_m, and P_l to different phases (see Section 5.1.3). Finally, with AC', I_{sub}, and P_c, we can calculate the values of P_h, P_m, and P_l at V_{safe_min} using Equation (5.11), and apply these values into Equation (5.12) to calculate the energy costs at V_{safe_min}.

5.3 Experimental Results

In this section, we present comprehensive evaluation on performance and energy efficiency of several HPC applications with undervolting and various resilience techniques. We experimentally show under what circumstances energy savings using this combinational technique can be achieved. The benchmarks under test include NPB MG (MG), NPB CG (CG), NPB FT (FT), matrix multiplication (MatMul), Cholesky factorization (Chol), LU factorization (LU), QR factorization (QR), LULESH, and AMG. All the experiments are conducted on our 8-node (64 cores) power-aware cluster shown in Table 6.2. All the cases are under the ideal scenario that once an error occurs it will be detected.

Figure 5.8 shows the normalized execution time and energy costs of various benchmarks running under undervolting and four different resilience techniques. Figure 5.9 demonstrates the comparison of the normalized performance and energy efficiency of Adagio, a state-of-the-art DVFS technique for HPC, and our undervolting approach based on Adagio with ABFT. The test scenarios of the two figures are explained as follows: For C/R based resilience techniques (e.g. disk-based and diskless C/R), OneCkpt means checkpoint/restart is only performed once; OptCkpt@$\mathsf{V_x}$ refers to checkpoint/restart is performed with the optimal checkpoint interval at V_x; and OptCkpt@$\mathsf{V_x} + uv$ is to integrate the impacts of undervolting into OptCkpt@$\mathsf{V_x}$ to see time and energy changes. The nature of the Triple Modular Redundancy (TMR) and ABFT determines that the frequency of performing fault-tolerance actions is not affected by failure rates. Therefore, we simply apply undervolting to them during program execution. For the comparison against Adagio, we

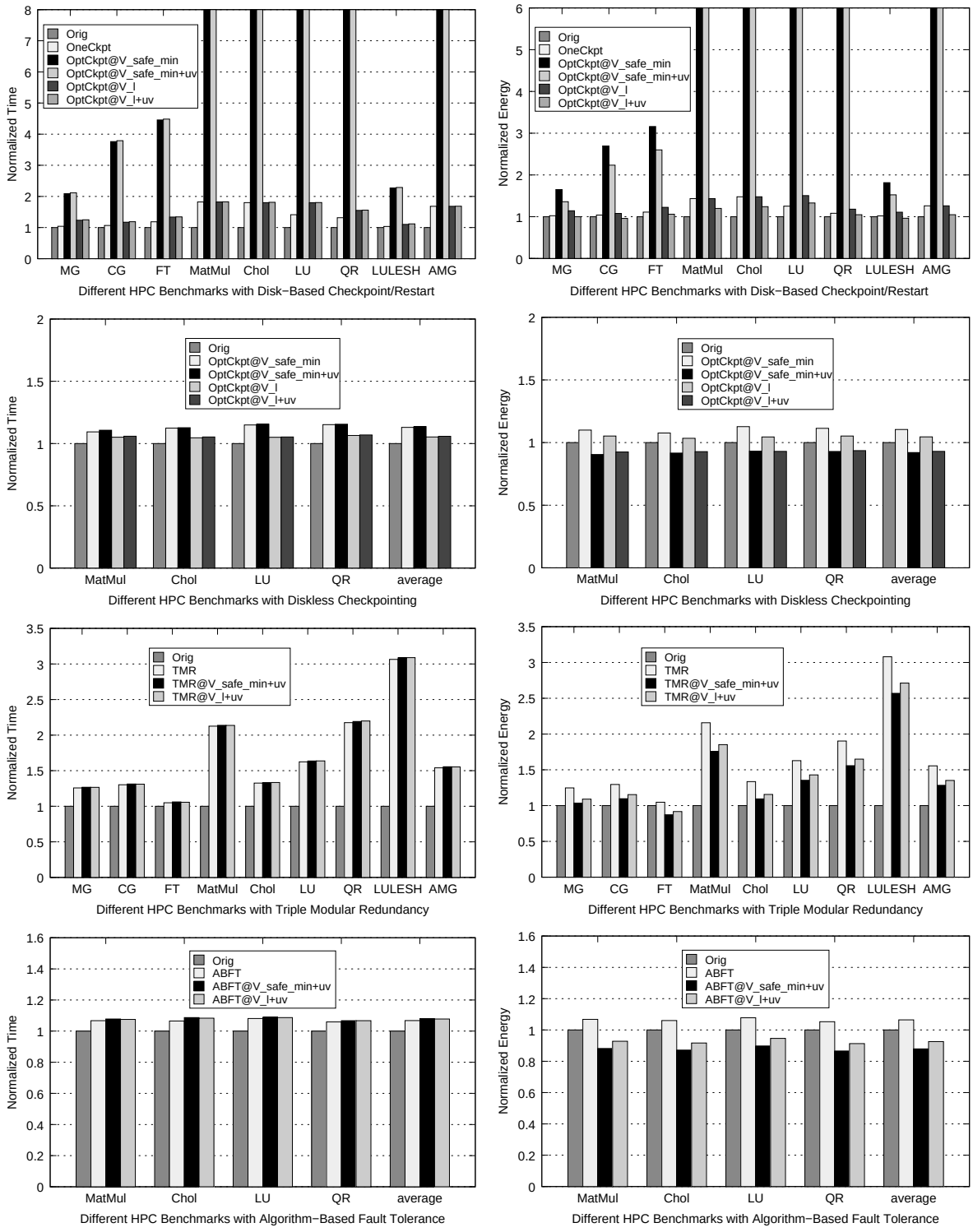


Figure 5.8: Performance and Energy Efficiency of Several HPC Runs with Different Mainstream Resilience Techniques on a Power-aware Cluster.

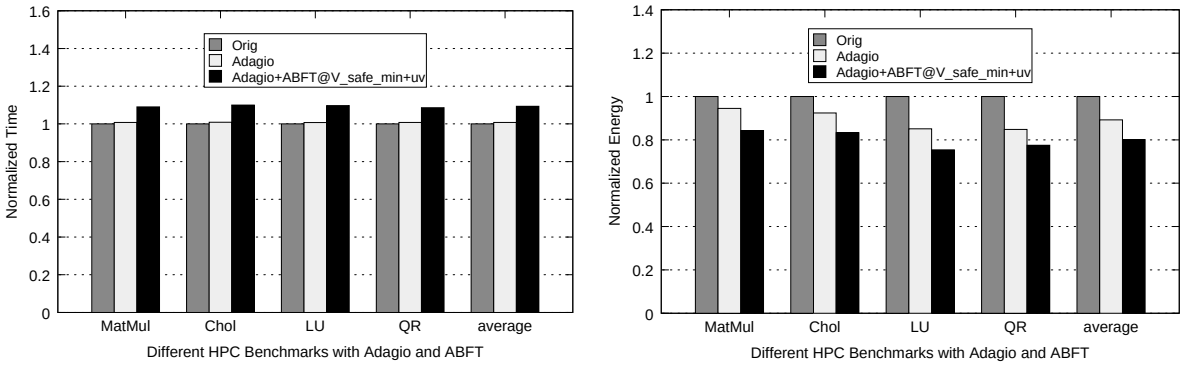


Figure 5.9: Performance and Energy Efficiency of the HPC Runs with an Energy Saving Solution Adagio and a Lightweight Resilience Technique ABFT.

first run applications with Adagio, and then with Adagio and our combinational techniques together.

We use the Berkeley Lab Checkpoint/Restart (BLCR) version 0.8.5 [59] as the implementation for the Disk-Based C/R, and our own implementation of fault tolerant ScaLAPACK [146] for ABFT. We also implemented an advanced version of TMR and Diskless C/R for the selected benchmarks.

In this section, we do not include the case studies that have a mix of hard and soft errors, although it is a more realistic scenario of undervolting. The reason is that the performance and energy impacts of hard and soft error detection and recovery are accumulative, based on a straightforward hypothesis that at any point of execution only one type of failure (either a hard error or a soft error) may occur for a given process. Thus the current single-type-of-error evaluation is sufficient to reflect the trend of the impacts (see Figure 5.8).

5.3.1 Disk-Based Checkpoint/Restart (DBCR)

As discussed earlier, DBCR needs to save checkpoints into local disks, which can cause high I/O overhead. The first two subgraphs in Figure 5.8 show that for some benchmarks (MatMul, Chol, LU, QR, and AMG) single checkpoint overhead is as high as the original execution time without C/R. For matrix benchmarks, DBCR will save the large global matrix with double-precision matrix elements as well as other necessary C/R content, which explains the excessive C/R overhead. Using the scaling technique presented in Section 5.2.4, the failure rate at V_{safe_min} is around 10^{-1} while the one at V_l is of the order of magnitude of 10^{-3}. Based on the relationship between the checkpoint overhead and failure rates shown in Equation (5.6), the optimal numbers of checkpoints at these two voltages differ. Due to the high C/R overhead and a larger number of checkpoints required, undervolting to V_{safe_min} using DBCR cannot save energy at all. With less checkpoints required, undervolting to V_l still does not save energy for DBCR.

5.3.2 Diskless Checkpointing (DC)

As a backward recovery technique, compared to DBCR, DC saves checkpoints in memory at the cost of memory overhead, which is much more lightweight in terms of C/R overhead due to low I/O requirement. From the third and fourth subgraphs in Figure 5.8, we can observe that the C/R overhead significantly drops by an average of 44.8% for undervolting to V_l. Consequently, energy savings are obtained by an average of 8.0% and 7.0% from undervolting to V_{safe_min} and V_l respectively. Note that the energy savings from undervolting to the two voltages are similar, since the extra power saving from undervolting

to a lower voltage level is offset by the higher C/R overhead (e.g. more checkpoints are required). This also indicates that the overhead from a resilience technique greatly impacts the potential energy saving from undervolting.

5.3.3 Triple Modular Redundancy (TMR)

As another effective retry-based resilience technique, Triple Modular Redundancy (TMR) is able to detect and correct error once in three runs, assuming there exists only one error arising within the three runs. Instead of using the naïve version of TMR that runs an application three times, we implemented a cost-efficient TMR that performs triple computation and then saves the results for the recovery purposes, while reducing the communication to only once to lessen overhead. This requires the assumption that we have a reliable network link for communication. Subgraphs in Figure 5.8 shows that we can successfully reduce TMR overhead except for LULESH, for which we had to run three times to ensure resilience. Although similar to DBCR, bearing the high overhead from the resilience technique, undervolting the system to both voltage levels by leveraging TMR generally cannot save energy. This once again demonstrates that the resilience techniques with lower overhead will benefit energy savings from undervolting.

5.3.4 Algorithm-Based Fault Tolerance (ABFT)

We also evaluate the performance counterpart of TMR that handles soft errors, based on local/global recovery using arithmetic checksums for matrix elements, i.e., ABFT. Well known for its low overhead compared to other fault tolerant techniques, ABFT is thus an ideal candidate to pair with undervolting for energy savings for some applications where

ABFT can be applied. As shown in the last two subgraphs of Figure 5.8, without undervolting, ABFT only adds less than 6.8% overhead on average to the original runs. Similarly as in the cases of the other three resilience techniques, undervolting only incurs negligible overhead (around 1%) in ABFT. Therefore, the combined usage of ABFT and undervolting only causes minor overhead (on average 8.0% for undervolting to V_{safe_min} and 7.8% for undervolting to V_l). Another advantage of using ABFT is that ABFT is essentially based on the checksum algorithms. Checksum blocks periodically update with matrix operations and do not require to update more frequently when the failure rates increase, which means the overhead of ABFT is constant regardless of the failure rates. Consequently, the energy savings using ABFT and undervolting can be up to 12.1% compared to the original runs in our experiments. One disadvantage for using ABFT is that it is only applicable to certain applications such as matrix operations, which is not general enough to cover the entire spectrum of HPC applications.

5.3.5 Energy Savings over Adagio

As discussed in Section 5.1.3, we aim to see if further reducing voltages for the selected frequencies of various phases by Adagio will save more energy overall. Figure 5.9 confirms that we are able to further save energy on top of Adagio through undervolting by effectively leveraging lightweight resilience techniques. In this experiment, we adopt the most lightweight resilience technique evaluated above, i.e., ABFT, to maximize the potential energy savings from undervolting. Undervolting was conducted on top of Adagio without modifying the runtime frequency-selecting decisions issued by Adagio, so the energy savings from Adagio are retained. Here we only present the data of undervolting to V_{safe_min}

because we already know from Section 5.4 that undervolting to V_{safe_min} gains the most energy savings for the case of ABFT. On average, combining undervolting and Adagio can further save 9.1% more energy than just Adagio, with less than 8.5% extra performance loss. Note that the majority of the performance loss is from ABFT itself for guaranteeing resilience.

5.4 Summary

Future large-scale HPC systems require both high energy efficiency and resilience to achieve ExaFLOPS computational power and beyond. While undervolting processors with a given frequency could decrease the power consumption of an HPC system, it often increases failure rates of the system as well, hence, increases application's execution time. Therefore, it is uncertain that applying undervolting to processors is a viable energy saving technique for production HPC systems. In this chapter, we investigate the interplay between energy efficiency and resilience at scale. We build analytical models to study the impacts of undervolting on both application's execution time and energy costs. By leveraging software-level cost-efficient resilience techniques, we found that undervolting can be used as a very effective energy saving technique for the HPC field.

Chapter 6

Scalable Energy Efficiency with Resilience for High Performance Computing Systems: A Quantitative Methodology

As the exascale supercomputers are expected to embark around 2020 [63], High Performance Computing (HPC) systems nowadays expand rapidly in size and duration in use, which brings demanding requirements of energy efficiency and resilience at scale, along with the ever-growing performance boost. These requirements are becoming prevalent and challenging, considering two crucial facts that: (a) The costs of powering an HPC system grow greatly with its expanding scale, and (b) the failure rates of an HPC system are dramatically increased due to a larger amount of interconnected computing nodes.

Therefore, it is desirable to consider both dimensions of energy efficiency and resilience, when building scalable, cost-efficient, and robust large-scale HPC systems. Specifically, for a given HPC system, our ultimate goal is to achieve the optimal performance-power-failure ratio while exploiting parallelism.

Nevertheless, for the concerns of energy efficiency and resilience in scalable HPC systems, alleviating one dimension does not necessarily improve the other. Energy efficiency and resilience are essentially mutually-constrained during the efforts of finding the balanced HPC configuration for the integrated optimal performance-power-failure ratio. Despite the straightforward fact that both of energy efficiency and resilience are correlated with execution time of HPC runs, altering some HPC parameters that are closely related to both dimensions, such as supply voltage of hardware components and number of cores used, can be beneficial to one dimension but harmful to the other.

For instance, energy savings can be achieved via Dynamic Frequency and Voltage Scaling (DVFS) techniques [141] [115] [132] [129], for CMOS-based processing components including CPU, GPU, and memory. In general practice, DVFS is often frequency-oriented towards idle time of the components, which means the voltage will be changed if the paired frequency is altered but will be kept the same otherwise. Nowadays state-of-the-art processors with cutting-edge nano-technology are allowed to be supplied with a significantly low voltage, close to the transistor's threshold voltage, e.g., Intel's Near-Threshold Voltage (NTV) design [90]. Further energy savings can be achieved through a fixed-frequency scheme with further reduced voltage, named *undervolting* [143] [29] [39] [131], at the cost of increased failures of the components. However, it is not clear that which variation from

undervolting is more dominant for high energy efficiency: power savings from further voltage reduction or performance loss from the overhead on error detection and recovery. It is thus desirable to investigate the potential of achieving high energy efficiency in HPC by undervolting, with hardware/software-level resilience techniques applied meanwhile to guarantee the correct execution of HPC runs.

There exist only a few efforts investigating this issue in the context of HPC with well-grounded modeling and experimental validation [39] [131]. However, the proposed approach in [39] worked specifically for a customized pre-production multi-core processor with ECC (Error-Correcting Code) memory, and thus their solution considered a single type of potential failures, i.e., ECC errors only. On the other hand, the authors in [131] did not theoretically consider the effects of scalability of HPC systems and discuss the interplay among mutually-constrained HPC parameters at scale, nor empirically evaluated the trade-offs among the HPC parameters towards scalable energy efficiency and resilience. Therefore, in this work, we propose to quantitatively model the entangled effects of energy efficiency and resilience in the scalable HPC environment and investigate the trade-offs among typical HPC parameters for the optimal energy efficiency with resilience. In summary, the contributions of this work include:

- We quantitatively model the entangled effects of energy efficiency and resilience at scale, by extending the Amdahl's Law and the Karp-Flatt Metric;
- We showcase the interplay among typical HPC parameters with internal causal effects, and demonstrate how the trade-offs impact the energy efficiency at scale;
- We provide experimental results using ten HPC benchmarks on two power-aware

clusters, showing that our models are accurate and effective to find the balanced HPC configuration for the optimal scalable energy efficiency under resilience constraints.

The remainder of this chapter is organized as follows. Section 6.1 introduces background. We present our modeling of scalable energy efficiency with resilience, and trade-offs among HPC parameters in Section 6.2. Implementation details and experimental results are provided in Section 6.3. Section 6.4 concludes.

6.1 Background: Energy Savings, Undervolting, and Failures

Numerous efforts have been made to address the demanding requirements of energy efficiency in HPC nowadays. In general, processor-based energy saving techniques can be categorized into two types: *frequency-directed* and *voltage-directed*. Next we present the details of each as the background knowledge for later modeling and discussion.

6.1.1 Frequency-Directed DVFS Techniques

Generally for an HPC run, *slack* refers to a time period when one hardware component waits for another due to imbalanced throughput and utilization [129]. There exist many slack opportunities during an HPC run. For instance, if network components are busy, other components (e.g., CPU, GPU, memory, and disk) are often alternatingly idle when message-passing communication (specifically, message copy among buffers of different compute nodes) is performed. Moreover, if the application is memory and disk intensive,

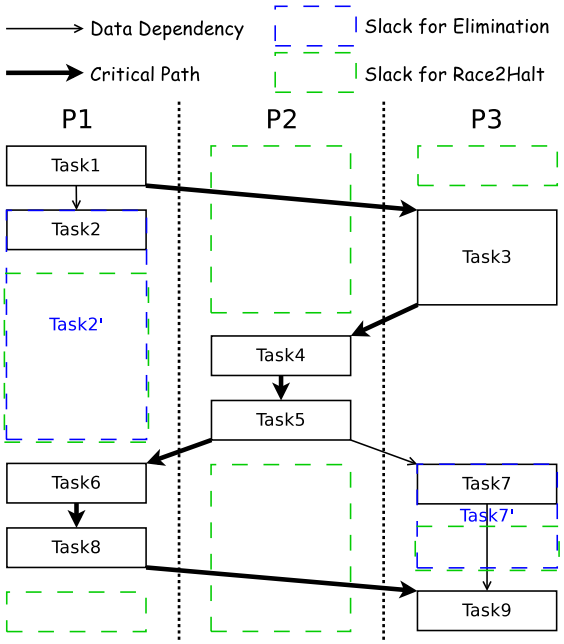


Figure 6.1: DAG Notation of Two DVFS Solutions for a 3-Process HPC Run.

CPU usually waits for the data from memory and disk, due to the performance bottleneck at memory and disk accesses. An ideal method to save energy is to reduce the power of non-busy components when such slack occurs, while keep the peak performance of busy components when otherwise. By exploiting DVFS, existing solutions decrease processor power during communication [116] [132], halt/idle processors when no workloads are available [31] [129], or work based on *Critical Path* (CP) analysis to reclaim slack arising among computation [115] [129]. Energy can be saved with negligible performance loss using the above solutions.

For energy saving purposes, without degrading performance, slack arising in HPC runs can either be eliminated by decreasing processor frequency to extend computation time of program execution fragments (e.g., tasks) with slack appropriately, or be utilized

to make non-working hardware components stay in the halt/idle state, as shown in Figure 6.1 [129], which depicts the Directed Acyclic Graph (DAG) representation of applying two classic DVFS approaches on a task-parallel HPC run. By respecting the CP, slack is not over-reclaimed or over-utilized for halting, and thus the DVFS solutions incur negligible performance loss from DVFS itself in practice. In this work, we adopt state-of-the-art DVFS approaches that outperform other DVFS solutions in different scenarios by completely eliminating potential slack. Similarity among these DVFS solutions is that they are all *frequency-directed*: Processor voltage is only lowered together with processor frequency reduction, in the presence of slack, and is fixed otherwise. In other words, a voltage is always paired with a chosen frequency, and no further voltage reduction is conducted for the given frequency.

6.1.2 Fixed-Frequency Undervolting Technique

Although effective, frequency-directed DVFS approaches may fail to fully exploit energy saving opportunities. Hardware components such as processors nowadays are allowed to be supplied with a voltage that is lower than the one paired with a given frequency, which is referred to as *undervolting* for more power savings beyond DVFS. This *voltage-directed* technique is independent of frequency scaling (i.e., during undervolting, the frequency is fixed after it is chosen), but requires hardware support for empirical deployment. Unlike traditional simulation-based undervolting approaches [143] [29], Bacha *et al.* [39] first implemented an empirical undervolting system on Intel Itanium II processors via software/firmware control, which was intended to reduce voltage margins and thus save power, with ECC memory correcting arising ECC errors. This work maximized potential

power savings since it used pre-production processors that allows the maximum extent of undervolting: They were able to reduce voltage until the levels lower than the lowest voltage corresponding to the lowest frequency supported. In general, production processors are locked for reliability purposes by the OS, and will typically shut down when voltage is lowered below the one paired with the lowest frequency. For *generality* purposes, Tan *et al.* [131] proposed an *emulated scaling* undervolting scheme that works for general production processors, which was deployed on a power-aware HPC cluster as the first attempt of its kind to demonstrate more energy savings compared to state-of-the-art DVFS solutions. They implemented undervolting using the Model Specific Register interface, which does not require the support of pre-production machines and makes no modification to the hardware.

The further power savings from undervolting are however achieved at the cost of higher failure rates λ [154] [39]. As shown in Equation (6.1) below, the average failure rates are quantified in terms of supply voltage only with other parameters known [131] (λ_0: the average failure rate at f_{max} (and V_{max}), d and β: hardware-dependent constants, f_{max}/f_{min}: the highest/lowest operating frequency, V_{th}: threshold voltage). Therefore while undervolting is beneficial to saving power, hardware/software-based fault tolerant techniques are also required to guarantee correct program execution. Bacha *et al.* [39] employed ECC memory to correct memory bit failures (single-bit flips). Tan *et al.* [131] adopted lightweight resilience techniques such as diskless checkpointing and algorithm-based fault tolerance to correct both hard and soft errors. The difference lies in: Pre-production machines are required in the work of Bacha *et al.*, where real errors can be observed at the lowest safe voltage V_{safe_min}, $V_{th} < V_{safe\_min} < V_l$, while the solution proposed by Tan *et al.*

needs production machines only, and thus errors from undervolting cannot be empirically observed, due to close-to-zero failure rates at V_l per Equation (6.1), i.e., the lowest voltage can be scaled to by undervolting for production machines without crashing. The emulated scaling scheme [131] was able to estimate the energy costs at V_{safe_min}, based on real measured power/energy data at V_l and power/energy models under resilience techniques and undervolting.

$$\lambda(f, V_{dd}) = \lambda(V_{dd}) = \lambda_0 \, e^{\frac{d(f_{max} - \beta(V_{dd} - 2V_{th} + \frac{V_{th}^2}{V_{dd}}))}{f_{max} - f_{min}}} \tag{6.1}$$

Since this work is intended for general production HPC systems, we leverage the software-based undervolting approach [131] to save more energy beyond DVFS solutions at the cost of raised failure rates. Thus as in [131], no errors were experimentally observed, but were likewise successfully emulated (see section 6.3.2). In this work, the Checkpoint/Restart technique is employed to recover from errors.

6.1.3 Checkpoint/Restart Failure Model

Computing systems in general suffer from various sources of failures, ranging from computation errors on logic circuits, to memory bit-flips due to frequency and voltage fluctuation [39] [146] [131]. Without loss of generality, in this work we discuss how to detect and recover from a failure in an HPC run using a general-purpose widely used resilience technique Checkpoint/Restart (C/R) [53], and build and evaluate our performance and power models based on C/R. Our methodology of theoretical modeling and experimental evaluation also applies to other resilience techniques such as Algorithm-Based Fault Toler-

ance (ABFT), with minor changes in the proposed models accordingly. Note that we use the terms *errors*, *faults*, and *failures* interchangeably henceforth in the later text.

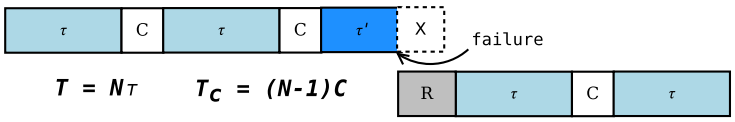


Figure 6.2: Fault Tolerance using the Checkpoint/Restart Technique.

Figure 6.2 demonstrates the scenario of fault tolerance using C/R. A program with the execution time T can be *checkpointed* by making a snapshot of a system state at the end of fragments of an evenly divided program run so that $T = N\tau$, where τ is the length of fragments of the divided program run, and $N - 1$ is the number of checkpoints added into the run. Specifically, a system state is a copy of current application process address space, including the contents of values of heap, stack, global variables, program text and data, and registers. The total checkpoint overhead is thus modeled as $T_C = (N - 1)C$, where C is the time required for making one checkpoint. At any points of time within a program run fragment, a failure can arise and interrupt the program run. We denote the time that a failure occurs as τ'. For continuing the run without re-executing the whole program, we reinstate the last saved checkpoint and restart from the saved information in the checkpoint. The time overhead on restarting the run is denoted as R. The total restart overhead depends on the number of failures during the run.

6.2 Modeling Scalable Energy Efficiency with Resilience

6.2.1 Problem Description

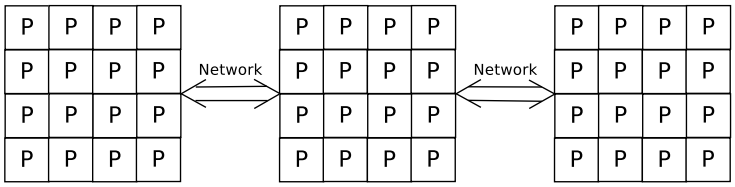


Figure 6.3: Investigated Architecture – Symmetric Multicore Processors Interconnected by Networks.

We aim to achieve the optimal energy efficiency with resilience in a scalable HPC environment as in Figure 6.3: an HPC system with a number of compute nodes, each of which consists of multiple symmetric cores, interconnected by networks. Note that we assume if there exist multiple multicore processors in a node, the cores across processors are also symmetric. Although at the initial stage, applicable architectures are homogeneous HPC systems without accelerators, with minor changes, the methodology proposed and the empirical studies conducted in this work also apply to other architectures, such as emerging heterogeneous GPU/coprocessor-accelerated HPC systems.

6.2.2 Amdahl's Law and Karp-Flatt Metric

Amdahl's Law [36] and Karp-Flatt Metric [88] are two classic metrics that quantify the performance of parallel programs. Amdahl's Law stresses performance impacts from the parallelized code within a parallel program, without considering communication. Karp-Flatt speedup formula takes communication (including data transmission and synchronization in

HPC runs) into account, likewise as the consideration of this work. Specifically, Amdahl's Law is written as:

$$\texttt{Speedup}_{\texttt{a}} = \frac{T_s + T_p}{T_s + \frac{T_p}{P}} = \frac{(1-\alpha)T + \alpha T}{(1-\alpha)T + \frac{\alpha T}{P}} = \frac{1}{1-\alpha+\frac{\alpha}{P}} \tag{6.2}$$

where $T = T_s + T_p$ is the total execution time of the program, consisting of the runtime of the sequential code T_s and the runtime of the parallelized code T_p individually, α is the percentage of code that can be parallelized within the program ($0 \leq \alpha \leq 1$), P is the total number of cores used in the HPC system where the program runs, and N is the problem size. Considering $\kappa(N,P)$ as the communication time, determined by N and P jointly, Karp-Flatt speedup formula can be written in the following form:

$$\begin{aligned}\texttt{Speedup}_{\texttt{kf}} &= \frac{T_s + T_p}{T_s + \frac{T_p}{P} + T_{comm}} = \frac{(1-\alpha)T + \alpha T}{(1-\alpha)T + \frac{\alpha T}{P} + \kappa(N,P)} \\ &= \frac{1}{1-\alpha+\frac{\alpha}{P}+\frac{\kappa(N,P)}{T}}\end{aligned} \tag{6.3}$$

6.2.3 Extended Amdahl's Law for Power Efficiency

The original Amdahl's Law considers performance of HPC systems only. Woo *et al.* [144] incorporated power and energy efficiency into the Amdahl's Law, without considering communication as in the Karp-Flatt speedup formula. In this work, we take into account both the power/energy efficiency and the communication during HPC runs by extending the classic Amdahl's Law for scalable HPC systems.

A General Extension

We first formulate the total power consumption of all hardware components in an HPC system (specifically, including multicore processors across nodes with P cores in total

and other components) during an original error-free run, without considering energy saving DVFS and undervolting solutions applied:

$$\begin{aligned}\texttt{Power} &= \frac{Energy}{Time} \\ &= \frac{(Q+(P-1)\mu Q)(1-\alpha)T + PQ\frac{\alpha T}{P} + P\mu Q\kappa(N,P) + \mathbb{C}((1-\alpha)T + \frac{\alpha T}{P} + \kappa(N,P))}{(1-\alpha)T + \frac{\alpha T}{P} + \kappa(N,P)} \\ &= Q \times \frac{(1+\mu(P-1))(1-\alpha) + \alpha + \mu P\frac{\kappa(N,P)}{T} + \frac{\mathbb{C}((1-\alpha)+\frac{\alpha}{P}+\frac{\kappa(N,P)}{T})}{Q}}{(1-\alpha) + \frac{\alpha}{P} + \frac{\kappa(N,P)}{T}} \end{aligned} \tag{6.4}$$

where Q is the power consumption of a single core at the peak performance, and μ is the fraction of the power the core consumes in the idle state with regard to that at the peak performance ($0 < \mu < 1$). We assume that the core power consumption during communication is roughly the same as that in its idle state [71]. We calculate the total energy consumption by accumulating energy costs at different phases of an HPC run: $(Q + (P-1)\mu Q)(1-\alpha)T$ is the energy costs of all P cores when the sequential code is executed (one core runs at full speed while the others are idle). $PQ\frac{\alpha T}{P}$ is the P-core energy costs when the program runs in parallel. $P\mu Q\kappa(N,P)$ refers to the energy costs the P cores produce during communication. $\mathbb{C}$ is the power costs of non-CPU components in the HPC system (we assume non-CPU components consume constant power during HPC runs, regardless of DVFS and undervolting for CPU only). As an extension to the Amdahl's Law that quantified speedup only of HPC runs, Equation (6.4) gives the total system power costs of HPC runs in general cases.

Without loss of generality, in Equation (6.4), we normalize $Q = 1$ to simplify the later discussion. Moreover, we set $\mathbb{C} = 0$ in the following modeling to focus on CPU power/energy efficiency due to two primary reasons: (a) CPU is the most power/energy con-

sumer in a homogeneous HPC system [71] [134], i.e., saving energy for CPU has the most significant impacts on improving the system energy efficiency, and (b) we investigate the DVFS and undervolting solutions that directly affect CPU power/energy costs but not impact other non-CPU components. More hardware components such as GPU/memory/networks can also be incorporated if power/energy efficient techniques on GPU/memory/networks are considered. Due to space limitation, we elaborate our idea in this work taking CPU for example, and leave studying power/energy efficiency of other components in the HPC system as future work.

Communication of HPC Runs

As stated, we denote the communication time in an HPC run as $\kappa(N, P)$. Empirically, $\kappa(N, P)$ highly depends on the communication algorithm employed. Regardless of the basic point-to-point communication scheme, there exist a large body of studies on highly-tuned communication algorithms [45] [120] [40] [94]. Here we briefly discuss two classic broadcast algorithms: binomial tree and pipeline broadcast. Their performance comparison was summarized in [52] [126], where the time complexity of binomial tree broadcast was modeled as $T_B = \frac{S_{msg}}{BD} \times logP$, and the pipeline broadcast time complexity was modeled as $T_P = \frac{S_{msg}}{BD} \times (1 + \frac{P-1}{\eta(N)})$ (S_{msg} is the message size in one broadcast, BD refers to network bandwidth in the communication, and $\eta(N)$ is the number of message chucks in the message transmission). We can see that performance of binomial tree broadcast depends on P only while performance of pipeline broadcast is determined by both P and N. In practice, the communication scheme and algorithm vary from different HPC applications, which determine what time complexity of $\kappa(N, P)$ is and ultimately affect the power/energy costs of HPC runs.

Power Efficiency with DVFS and Undervolting

Given the fact that the total power consumption of a core is composed of leakage/static power costs and dynamic power costs, the Amdahl's Law can be intuitively rewritten to model the potential core power savings for HPC runs, under the circumstances of DVFS and undervolting respectively:

$$\texttt{PE}_{\texttt{dvfs}} = \frac{P_s + P_d}{\frac{(1-\beta)(P_s+P_d)}{n_1} + \frac{\beta(P_s+P_d)}{n_2}} = \frac{1}{\frac{1-\beta}{n_1} + \frac{\beta}{n_2}} \tag{6.5}$$

$$\texttt{PE}_{\texttt{uv}} = \frac{P_s + P_d}{\frac{(1-\beta)(P_s+P_d)}{n_1} + \frac{\beta(P_s+P_d)}{n_3}} = \frac{1}{\frac{1-\beta}{n_1} + \frac{\beta}{n_3}} \tag{6.6}$$

where P_s and P_d refer to the leakage and dynamic power costs of all used cores individually, and β is the percentage of dynamic power costs within the total power costs. n_1, n_2, and n_3 are the leakage power reduction factor of DVFS/undervolting, the dynamic power reduction factor of DVFS, and the dynamic power reduction factor of undervolting, individually, denoting the ratios of leakage/dynamic power reduction respectively. Equation (6.5) and Equation (6.6) model the power efficiency when DVFS and undervolting are applied separately. For simplicity of the discussion, we assume that the DVFS technique used is able to eliminate all slack in the HPC run, and the undervolting technique adopted is able to scale to the voltage paired with the lowest frequency.

Example. We can use the above two formulae to calculate the theoretical upper bound of power savings using DVFS and undervolting respectively. Assume we have the following parameters already known: $\beta = 0.6$ (dynamic power amounts to 60% of the total core

power, which is empirically reasonable). For the processors used, we assume the maximum frequency $f_h = 2.4$ GHz and the minimum frequency $f_l = 0.8$ GHz. Consider the extreme case that reducing frequency from 2.4 GHz to 0.8 GHz can eliminate all possible slack for all tasks. Since $P_s = I_{sub}V$ [135] (i.e., $\Delta P_s \propto \Delta V$), $P_d = AC'fV^2$ [108], and empirically $\Delta P_d \propto \Delta f^{2.5}$ [60], where A and C' are the percentage of active gates and the total capacitive load in a CMOS-based processor respectively, and I_{sub} refers to subthreshold leakage current, we can derive that $\Delta V \propto \Delta f^{0.75}$, and thus $n_1 = (\frac{f_h}{f_l})^{0.75}_{dvfs} = (\frac{2.4}{0.8})^{0.75} \approx 2.28$ and $n_2 = (\frac{f_h}{f_l})^{2.5}_{dvfs} = (\frac{2.4}{0.8})^{2.5} \approx 15.59$ in the assumed case. By substituting n_1 and n_2 into Equation (6.5), we have the range of power savings from DVFS: $\texttt{PE}_{\texttt{dvfs}} \leq \frac{1}{\frac{0.4}{2.28}+\frac{0.6}{15.59}} \approx 4.67$. Moreover, consider in the case of undervolting, we have the same $n_1 = (\frac{f_h}{f_l})^{0.75}_{uv} = (\frac{2.4}{0.8})^{0.75} \approx 2.28$ and $n_3 = (\frac{f_h}{f_l})^{1.5}_{uv} = (\frac{2.4}{0.8})^{1.5} \approx 5.26$. Substituting all known parameters we have the range of power savings from undervolting: $\texttt{PE}_{\texttt{uv}} \leq \frac{1}{\frac{0.4}{2.28}+\frac{0.6}{5.26}} \approx 3.45$.

6.2.4 Extended Karp-Flatt Metric for Speedup with Resilience

From the Karp-Flatt speedup formula, i.e., Equation (6.3), we can further derive the extended Karp-Flatt speedup formula when failures occur and resilience techniques are employed in HPC runs. Although we take the C/R technique for example in the later discussion, our modeling also applies to other resilience techniques by making changes on the modeling of error detection and recovery. Next we formulate the Karp-Flatt Metric for the scenario with failures and C/R, by adopting Daly's simplified C/R performance model $T_{cr} = \frac{1}{\lambda}e^{R\lambda}(e^{\lambda(\tau+C)} - 1)\frac{T}{\tau}$ [53]. We substitute the original solve time of the parallelized code with the solve time with failures and C/R T_{cr}:

$$\texttt{Speedup}^{\texttt{cr}}_{\texttt{kf}} = \frac{T_{orig}}{T_{cr}} = \frac{(1-\alpha)T + \alpha T}{\frac{1}{\lambda}e^{R\lambda}(e^{\lambda(\tau+C)}-1)\frac{(1-\alpha)T+\frac{\alpha T}{P}+\kappa(N,P)}{\tau}}$$
$$= \frac{1}{\frac{1}{\lambda}e^{R\lambda}(e^{\lambda(\tau+C)}-1)\frac{1-\alpha+\frac{\alpha}{P}+\frac{\kappa(N,P)}{T}}{\tau}} \tag{6.7}$$

6.2.5 Quantifying Integrated Energy Efficiency

Based on the two extended models, further taking HPC communication, energy saving DVFS and undervolting solutions, and failures and resilience techniques into consideration, we define the integrated energy efficiency of an HPC system as follows, in terms of the speedup in different HPC scenarios achieved per unit energy per unit time:

$$\frac{\mathsf{Perf}}{\mathsf{Watt}} = \frac{\texttt{Speedup}}{\texttt{Power}} \tag{6.8}$$

We next consider the following four typical HPC scenarios individually, where different integrated energy efficiency metrics can be produced according to Equation (6.8).

$$\frac{\mathsf{Perf}}{\mathsf{Watt}} = \frac{1}{1-\alpha+\frac{\alpha}{P}+\frac{\kappa(N,P)}{T}} \times \frac{1-\alpha+\frac{\alpha}{P}+\frac{\kappa(N,P)}{T}}{(1+\mu(P-1))(1-\alpha)+\alpha+\mu P\frac{\kappa(N,P)}{T}}$$
$$= \frac{1}{(1+\mu(P-1))(1-\alpha)+\alpha+\mu P\frac{\kappa(N,P)}{T}} \tag{6.9}$$

communication-time-to-total-time-ratio. The later discussion depends on a practical assumption that our models work well for HPC applications with the following characteristics: The ratio of communication time to the total execution time $\frac{\kappa(N,P)}{T}$ (this term appears in Equations (6.9-6.15)) does not vary much as problem size N and number of cores P change. There exist a large body of such applications [85] [147], including a majority of benchmarks evaluated in Section 6.3. Nevertheless, for applications with a great variation

of $\frac{\kappa(N,P)}{T}$ (e.g., the IS benchmark from NPB [20]), the accuracy of our proposed models may be affected but the trend of energy efficiency manifested by our models may still retain.

We first consider the baseline case that there are no failures during an HPC run, without DVFS and undervolting techniques. Based on the definition, we calculate the energy efficiency in this case, as shown in Equation (6.9), where we assume that the HPC run uses P cores in total that solves a size-N problem. A number of parameters are influential to the energy efficiency, including several fixed application-specific and architecture-specific invariants: the percentage of parallelized code α, and the ratio μ, power consumed by one idle core normalized to that by one fully-loaded running core. Equation (6.9) models the baseline for building energy efficiency models of the following HPC scenarios, with energy saving solutions, failures, and resilience techniques.

Scenario 1: HPC Runs with Faults and C/R (No DVFS + No Undervolting)

If faults occur at a rate of λ and the C/R technique is employed in the HPC run, resilience-related factors including failure rates λ, checkpoint intervals τ, checkpoint overhead C, and restart overhead R are involved in the speedup formula, which affect the overall energy efficiency as well. Using the performance model T_{cr} for the scenario with failures given in Equation (6.7), we first model the power costs P_{cr} in this scenario as Equation (6.10), based on the assumption that the power draw of a node during checkpointing and restarting is very close to that during its idle state [106]. In Equation (6.10), $\mathbb{N}$ is the expected number of failures, approximated in [53] as $\mathbb{N} = \lambda T(1 + \frac{C}{\tau})$. If the solve time T is comparatively long, we can assume $\mathbb{N} \gg 1$, and thus further simplify the power formula. The ultimate energy efficiency is given in Equation (6.11). In contrast to the

$$P_{cr} = \frac{E_{cr}}{T_{cr}} = \frac{(1+\mu(P-1))(1-\alpha)+\alpha+\mu P\frac{\kappa(N,P)}{T}+\mu P\frac{(\mathbb{N}-1)C+\mathbb{N}R}{T}}{\frac{1}{\lambda}e^{R\lambda}(e^{\lambda(\tau+C)}-1)\frac{1-\alpha+\frac{\alpha}{P}+\frac{\kappa(N,P)}{T}}{\tau}} \tag{6.10}$$

$$\begin{aligned}\frac{\mathsf{Perf}}{\mathsf{Watt}} &= \frac{1}{\frac{1}{\lambda}e^{R\lambda}(e^{\lambda(\tau+C)}-1)\frac{1-\alpha+\frac{\alpha}{P}+\frac{\kappa(N,P)}{T}}{\tau}} \times \frac{\frac{1}{\lambda}e^{R\lambda}(e^{\lambda(\tau+C)}-1)\frac{1-\alpha+\frac{\alpha}{P}+\frac{\kappa(N,P)}{T}}{\tau}}{(1+\mu(P-1))(1-\alpha)+\alpha+\mu P\frac{\kappa(N,P)}{T}+\mu P\lambda(1+\frac{C}{\tau})(C+R)} \\ &= \frac{1}{(1+\mu(P-1))(1-\alpha)+\alpha+\mu P\frac{\kappa(N,P)}{T}+\mu P\lambda(1+\frac{C}{\tau})(C+R)}\end{aligned} \tag{6.11}$$

energy efficiency in the baseline case, there appears a new term $\mu P\lambda(1+\frac{C}{\tau})(C+R)$ in the new energy model. Intuitively, since the speedup is degraded due to the checkpoint and restart overhead, and the overall energy costs are raised due to the extra energy costs during checkpointing and restarting, the energy efficiency in this scenario is expected to go down, compared to the baseline case.

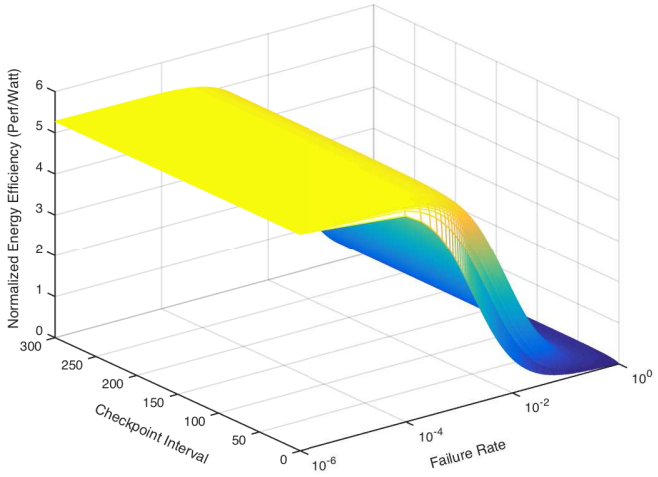


Figure 6.4: Energy Efficiency of HPC Runs with Faults and Resilience Techniques (Checkpoint/Restart).

Figure 6.4 depicts the energy efficiency curve when the C/R technique is used in the presence of failures. Here we adopt some premise to facilitate the analysis: Per empirical measurement (see Section 6.3.2.5), we let $\mu = 0.6$, and without loss of generality, we assume the communication in the HPC run takes 50% of the total execution time, i.e., $\frac{\kappa(N,P)}{T} = 0.5$ (from the evaluation in Section 6.3, we found that values of $\frac{\kappa(N,P)}{T}$ did not alter the trend of energy efficiency). Moreover, we set $P = 50$ and $\alpha = 0.9$ to showcase an HPC environment. Regarding the C/R technique used, we assume checkpoint overhead $C = 10$ and restart overhead $R = 20$ in seconds. From Figure 6.4, we can see that when failure rates λ are comparatively small, i.e., in the range of $[10^{-6}, 10^{-4}]$, the energy efficiency is dominated by the impacts from the sequential and parallelized code and the communication per Equation (6.11). Variation of checkpoint intervals τ barely affects the energy efficiency in this case. However, the energy efficiency experiences a dramatic drop when the failure rates λ lie in around $[10^{-4}, 10^{-2}]$, and the drop becomes flattened when λ is further increased to a value close to 1. The impacts of τ are manifested when τ is small enough so that $\frac{C}{\tau}$ is larger than 1. We next discuss the scenario where energy saving techniques are used to improve the energy efficiency.

Scenario 2: HPC Runs with Faults and C/R, using DVFS to Save Energy

Likewise, we model the energy efficiency in the scenario with failures and C/R, and energy saving DVFS solutions in the presence of slack. From Equation (6.12), we can see that during the slack, using DVFS can indeed improve the energy efficiency due to the reduced power costs by a factor of $\frac{1}{\frac{1-\beta}{n_1}+\frac{\beta}{n_2}}$ (the only difference between Equation (6.11)), if the DVFS techniques incur negligible performance loss as described in Figure 6.1. n_1 and

$$\frac{\text{Perf}}{\text{Watt}} = \frac{1}{\frac{1}{\lambda}e^{R\lambda}(e^{\lambda(\tau+C)}-1)\frac{1-\alpha+\frac{\alpha}{P}+\frac{\kappa(N,P)}{T}}{\tau}} \times \frac{\frac{1}{\frac{1-\beta}{n_1}+\frac{\beta}{n_2}} \times \left(\frac{1}{\lambda}e^{R\lambda}(e^{\lambda(\tau+C)}-1)\frac{1-\alpha+\frac{\alpha}{P}+\frac{\kappa(N,P)}{T}}{\tau}\right)}{(1+\mu(P-1))(1-\alpha)+\alpha+\mu P\frac{\kappa(N,P)}{T}+\mu P\lambda(1+\frac{C}{\tau})(C+R)}$$

$$= \frac{\frac{1}{\frac{1-\beta}{n_1}+\frac{\beta}{n_2}}}{(1+\mu(P-1))(1-\alpha)+\alpha+\mu P\frac{\kappa(N,P)}{T}+\mu P\lambda(1+\frac{C}{\tau})(C+R)} \tag{6.12}$$

n_2 are determined by capability of the DVFS techniques and the amount of slack in the HPC run, and β depends on specific processor architectures. Note that during the non-slack time, power consumption with DVFS is the same as the original run.

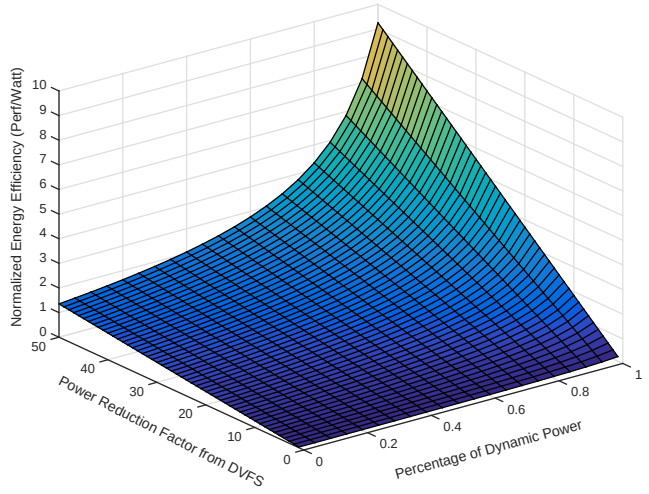


Figure 6.5: Energy Efficiency of HPC Runs with Faults, Checkpoint/Restart, and DVFS Techniques.

As Equation (6.12) is derived based on Equation (6.11), we fix $\lambda = 10^{-5}$ and $\tau = 150$ to discuss the impacts from β and n_1/n_2 on the energy efficiency in the scenario with faults and DVFS. Given the known relationship that $n_1 = \Delta f^{0.75}$ and $n_2 = \Delta f^{2.5}$ (i.e.,

$n_2 = \Delta f^{1.75} n_1$), we rewrite the term using a joint parameter n as $\frac{1}{\frac{1-\beta}{n}+\frac{\beta}{\Delta f^{1.75} n}}$. As shown in Figure 6.5, we can see that a larger β or n value results in higher energy efficiency in general, which is consistent with Equation (6.12). We can also observe that for β values close to 1, the variation of energy efficiency is more manifested, since in which case the energy efficiency is dominated by the larger n_2 ($\Delta f^{1.75} > 1$), according to $\frac{1}{\frac{1-\beta}{n}+\frac{\beta}{\Delta f^{1.75} n}}$. Empirically, typical β values range from 0.65 to 0.8 for different processor technologies nowadays [63] [105] [117].

Scenario 3: HPC Runs with Faults and C/R, using Undervolting (Increased Failure Rates)

If the undervolting technique is leveraged to further save energy during non-slack time, compared to Scenario 2 where DVFS solutions apply to slack only, power can be saved by a factor of $\frac{1}{\frac{1-\beta}{n_1}+\frac{\beta}{n_3}}$ by undervolting, as modeled in Equation (6.13). However, lower supply voltage by undervolting given a chosen operating frequency by DVFS causes higher failure rates ($\lambda' > \lambda$) and shorter checkpoint intervals ($\tau' < \tau$) for tolerating raised failures, and thus inevitably results in longer execution time. Consequently, energy efficiency in this scenario can be either improved or degraded compared to Scenario 2, due to the coexisting power savings and performance loss. Generally, there exists a balanced undervolting scale that maximizes the energy efficiency in this scenario. We present details of this discussion in Section 6.2.6. Next we look into energy saving effects from leakage and dynamic power reduction factors n_1 and n_3.

Figure 6.6 shows the energy efficiency trend as n_1 and n_3 change, in the scenario with failures, C/R and undervolting techniques, with given values $\beta = 0.7$, $\lambda' = 10^{-1}$,

$$\frac{\text{Perf}}{\text{Watt}} = \frac{1}{\frac{1}{\lambda'}e^{R\lambda'}(e^{\lambda'(\tau'+C)}-1)\frac{1-\alpha+\frac{\alpha}{P}+\frac{\kappa(N,P)}{T}}{\tau'}} \times \frac{\frac{1}{\frac{1-\beta}{n_1}+\frac{\beta}{n_3}} \times \left(\frac{1}{\lambda'}e^{R\lambda'}(e^{\lambda'(\tau'+C)}-1)\frac{1-\alpha+\frac{\alpha}{P}+\frac{\kappa(N,P)}{T}}{\tau'}\right)}{(1+\mu(P-1))(1-\alpha)+\alpha+\mu P\frac{\kappa(N,P)}{T}+\mu P\lambda'(1+\frac{C}{\tau'})(C+R)}$$

$$= \frac{\frac{1}{\frac{1-\beta}{n_1}+\frac{\beta}{n_3}}}{(1+\mu(P-1))(1-\alpha)+\alpha+\mu P\frac{\kappa(N,P)}{T}+\mu P\lambda'(1+\frac{C}{\tau'})(C+R)} \tag{6.13}$$

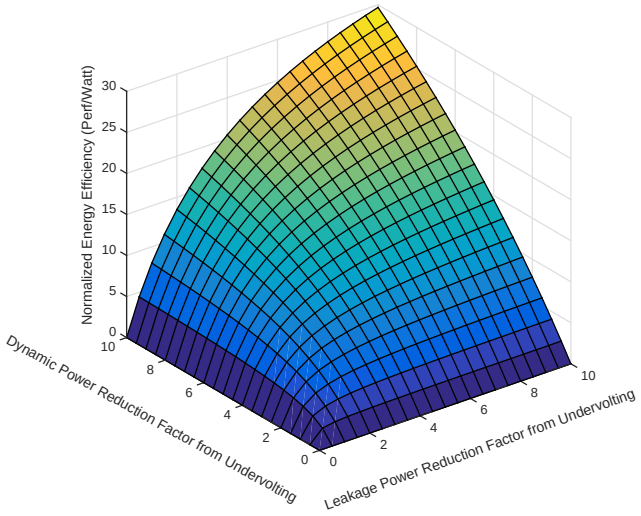


Figure 6.6: Energy Efficiency of HPC Runs with Faults, Checkpoint/Restart, and Undervolting (Setup I).

and $\tau' = 15$ (both of failure rates and the optimal checkpoint interval are increased due to undervolting). Due to the fact that undervolting increases both n_1 and n_3 but $n_3 > n_1$, compared to the leakage power reduction factor n_1, the dynamic power reduction factor n_3 has greater impacts on the variation of energy efficiency – for a given n_1, larger n_3 values increase the energy efficiency more than larger n_1 values given a fixed n_3. From Figure 6.6, we can see that the energy saving effects from n_3 is almost linear for a given n_1, while for a fixed n_3 value, the impacts from n_1 are smaller: The energy efficiency curves of n_1 become flattened as n_1 increases, especially for small n_3 values. Regarding energy saving

techniques, the energy efficiency trend is similar as in Scenario 2: Larger power reduction values n_1, n_2, and n_3 always improve energy efficiency monotonically. These observations can also be drawn from the causal term $\frac{1}{\frac{1-\beta}{n_1}+\frac{\beta}{n_3}}$ in Equation (6.13).

6.2.6 Energy Saving Effects of Typical HPC Parameters

With the energy efficiency models under different circumstances, we can further investigate the impacts of typical HPC parameters on the optimal energy efficiency individually. Since the parameters inherently affect each other, for highlighting the effects of individual factors, we fix other parameters likewise as the previous discussion.

Optimal Checkpoint Interval

Given a failure rate λ, for a certain C/R technique with the checkpoint and restart overhead C and R respectively, there exists an optimal checkpoint interval τ_{opt} that minimizes the total checkpoint and restart overhead. τ_{opt} is beneficial to improving performance, and thus also contributes to energy efficiency. Daly proposed a refined optimal value $\tau_{opt} = \sqrt{2C(\frac{1}{\lambda}+R)}$ for the condition $\tau + C \ll \frac{1}{\lambda}$ [53]. Under the circumstance of undervolting, failure rates vary exponentially according to Equation (6.1). The previously proposed τ_{opt} does not apply to the case that Mean Time To Failure (MTTF, the reciprocal of failure rates) becomes comparable to the checkpoint overhead C. Daly also discussed a perturbation solution in [53] that can be employed to handle this case of large MTTF due to undervolting:

$$\tau_{opt} = \begin{cases} \sqrt{\frac{2C}{\lambda}} - C & \text{for } C < \frac{1}{2\lambda} \\ \frac{1}{\lambda} & \text{for } C \geq \frac{1}{2\lambda} \end{cases} \tag{6.14}$$

Depending on the relationship between C and $\frac{1}{2\lambda}$, we use Equation (6.14) to calculate the most cost-efficient checkpoint interval in the scenario of undervolting. As already shown in Figure 6.4, larger checkpoint intervals τ barely enhance the energy efficiency while smaller ones do. Specifically, the difference between energy saving effects at nominal voltage and those at reduced voltage by undervolting is as follows: For nominal voltage with close-to-zero failure rates, e.g., at failure rate range [10^{-6}, 10^{-4}], the variation of τ barely affects the energy efficiency. For failure rates larger than 10^{-4}, the variation around small τ (compared to C) does affect the energy efficiency. The smaller τ incurs lower energy efficiency overall, since the resulting larger $\frac{C}{\tau}$ in Equation (6.11) makes the negative energy saving effects from C/R more manifested. For failure rates close to 1, the energy efficiency becomes insensitive to the variation of τ again, since in this case, within the C/R term $\mu P\lambda(1 + \frac{C}{\tau})(C + R)$ in Equation (6.11), λ is the dominant factor instead of τ. Generally for large values of λ, there exists an optimal value of τ for energy efficiency – any values greater than it barely affect the energy efficiency.

Optimal Supply Voltage

As stated earlier, values of supply voltage V_{dd} affect the performance and energy efficiency of HPC runs in two aspects: (a) V_{dd} values determine failure rates λ per Equation (6.1) and thus the optimal checkpoint interval τ_{opt} per Equation (6.14), and (b) V_{dd} values determine the leakage and dynamic power costs according to the relationship $P_s \propto V$ and

$P_d \propto fV^2$. Nevertheless, the two aspects by nature conflict with each other in achieving high energy efficiency: Decreasing V_{dd} causes an exponential increase of λ, and thus a decrease of τ_{opt}, which results in higher overhead on error detection and recovery using C/R. Therefore, for energy saving purposes, larger values of V_{dd} should be adopted to minimize C/R overhead. On the other hand, larger V_{dd} brings higher leakage and dynamic power costs that degrade the energy efficiency overall, without affecting the solve time (operating frequency f is already selected per DVFS techniques and not modified by undervolting). Targeting the optimal energy efficiency for different HPC scenarios illustrated in Section 6.2.5, we tend to choose an optimal V_{dd} value that balances the two conflicting aspects above.

$$\frac{\mathsf{Perf}}{\mathsf{Watt}} = \frac{n'}{(1+\mu(P-1))(1-\alpha)+\alpha+\mu P\frac{\kappa(N,P)}{T}+\mu P\lambda'(1+\lambda' C)(C+R)} \tag{6.15}$$

Recall that Figure 6.4 clearly shows that the variation of λ greatly impacts the energy efficiency, as a result of undervolting. However, it does not reflect the interplay between the two conflicting factors (a) and (b). We thus theoretically quantify this interplay to see if there exists an optimal supply voltage for the given HPC configuration. Since undervolting reduces both n_1 and n_3, for simplicity of the discussion of the optimal V_{dd}, we relax the energy efficiency formula in Scenario 4 by letting $n_1 = n_3 = n'$, where n' is the *unified power reduction factor* from undervolting. In addition, we assume the optimal checkpoint interval τ_{opt} is always adopted (due to greatly raised λ by undervolting, we adopt the branch $\tau_{opt} = \frac{1}{\lambda}$ for $C \geq \frac{1}{2\lambda}$ in Equation (6.14)). The simplified energy efficiency

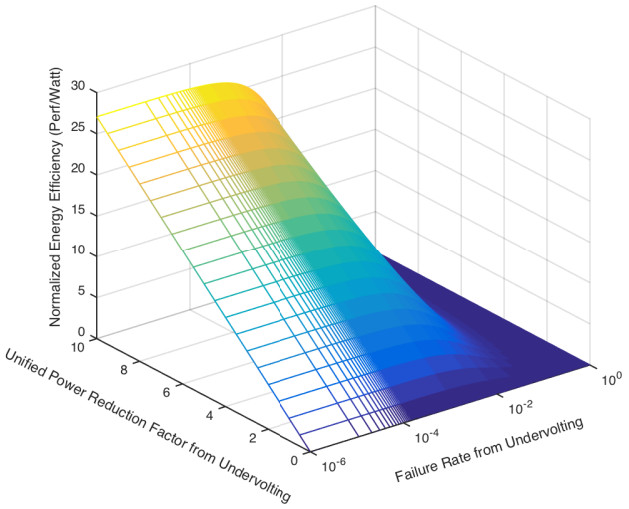


Figure 6.7: Energy Efficiency of HPC Runs with Faults, Checkpoint/Restart, and Undervolting (Setup II).

is modeled in Equation (6.15), which stresses the interplay between the conflicting power savings and raised failure rates from undervolting.

We thus plot the curve of the simplified formula using the same assumption that fixes parameters other than n' and λ', as shown in Figure 6.7. It is clear to see that the larger n' is and the smaller λ' is, the higher the energy efficiency reaches. From the previous discussion, we know that achieving higher power savings from undervolting incurs higher failure rates, and vice versa. Therefore, selecting a voltage value for undervolting that balances the two factors will fulfill the optimal energy efficiency. We next empirically explore the optimal voltage for the highest energy efficiency.

6.3 Evaluation

In this section, we present details of empirical evaluation for our speedup and energy efficiency models in the above different HPC scenarios on two HPC clusters. The goals of the evaluation are to experimentally demonstrate that: (a) the proposed models are well-grounded and accurate to predict the speedup, power and energy efficiency for scalable HPC runs, with prior knowledge on several HPC parameters, and (b) the proposed models are able to capture the interplay among typical HPC parameters in discussed HPC scenarios that ultimately affects the overall energy efficiency.

6.3.1 Experimental Setup

We conducted all the measurement-based experiments on a wide spectrum of HPC applications as our benchmarks, summarized in Table 6.1. The benchmarks were selected from NPB benchmark suite [20], LULESH [17], AMG [22] and our fault tolerant ScaLAPACK [146], including a self-implemented matrix multiplication program [132], For assessing our goals, experiments were performed on two different-scale power-aware clusters: HPCL and ARC. Table 6.2 lists the hardware configuration of the two clusters. Since undervolting requires the permission of root users, we managed to perform undervolting-related experiments on HPCL only while all other experiments were conducted on both clusters. We measured the total power consumption for all compute nodes involved in the experiments, including both dynamic and leakage power costs, collected by Watts up? PRO [28]. Energy consumption of HPC runs was measured using PowerPack [71], a comprehensive software and hardware framework for energy profiling and analysis of HPC systems and applications,

Table 6.1: Benchmark details. From left to right: benchmark name, benchmark suite, benchmark description and test case used, problem domain, lines of code in the benchmark, parallelization system employed, and parallelized code percentage relative to the total.

Benchmark	Suite	Description and Test Case	Domain	LOC	Parallelized in	Percentage of Parallelized Code
MG	NPB	Solve a discrete Poisson equation using multigrid method (Class B).	discrete mathematics	2568	OpenMP/MPI	73.0%
CG	NPB	Estimate eigenvalue of a sparse matrix with conjugate gradient method (Class B).	numerical linear algebra	1864	OpenMP/MPI	93.3%
FT	NPB	Solve a partial differential equation using fast Fourier transform (Class B).	numerical linear algebra	2034	OpenMP/MPI	58.7%
EP	NPB	Generate Gaussian random variates using Marsaglia polar method (Class B).	probability theory and statistics	359	OpenMP/MPI	94.7%
MatMul	Self-coded	Matrix multiplication on two 10k×10k global matrices, saving into a third one.	numerical linear algebra	1532	OpenMP/MPI /Pthreads	99.2%
Chol	FT-ScaLAPACK	Cholesky factorization on a 10k×10k global matrix to solve a linear system.	numerical linear algebra	2182	MPI	92.7%
LU	FT-ScaLAPACK	LU factorization on a 10k×10k global matrix to solve a linear system.	numerical linear algebra	2892	MPI	61.6%
QR	FT-ScaLAPACK	QR factorization on a 10k×10k global matrix to solve a linear system.	numerical linear algebra	3371	MPI	76.5%
LULESH	DARPA UHPC	Approximate hydrodynamics equations using 512 volumetric elements on a mesh.	hydrodynamics	6014	OpenMP/MPI	14.6%
AMG	CORAL	An algebraic multigrid solver for linear systems on a 4×4×6 unstructured grid.	numerical linear algebra	3098	OpenMP/MPI	65.1%

which enables individual power and energy measurement on all hardware components such as CPU, memory, disk, motherboard, etc. of an HPC system. Before presenting experimental results of running the benchmarks for evaluating each proposed goal individually, we detail the implementation of our approach.

Table 6.2: Hardware Configuration for All Experiments.

Cluster	HPCL	ARC
System Size (# of Nodes)	8	108
Processor	2×Quad-core AMD Opteron 2380	2×8-core AMD Opteron 6128
CPU Freq.	0.8, 1.3, 1.8, 2.5 GHz	0.8, 1.0, 1.2, 1.5, 2.0 GHz
CPU Voltage (Undervolting)	1.300, 1.100, 1.025, 0.850 V ($V_h/V_l/V_{safe\_min}/V_{th}$)	N/A
Memory	8 GB RAM	32 GB RAM
Cache	128 KB L1, 512 KB L2, 6 MB L3	128 KB L1, 512 KB L2, 12 MB L3
Network	1 GB/s Ethernet	40 GB/s InfiniBand
OS	CentOS 6.2, 64-bit Linux kernel 2.6.32	CentOS 5.7, 64-bit Linux kernel 2.6.32
Power Meter	PowerPack	Watts up? PRO

6.3.2 Implementation Details

Frequency-Directed DVFS

For demonstrating the effectiveness of our approach, we need to employ start-of-the-art energy efficient DVFS and undervolting techniques to evaluate the impacts from energy savings on the integrated energy efficiency of HPC systems (due to the similarity of energy saving trend between DVFS and undervolting, as shown in Equations (6.12) and (6.13), we only present results on undervolting). For different benchmarks, we used the most efficient DVFS approaches we developed in previous work: an adaptively aggressive energy efficient DVFS technique for NPB benchmarks [127], an energy efficient high performance matrix multiplication [132], and energy efficient distributed dense matrix factorizations [129]. CPU DVFS was implemented via the CPUFreq infrastructure [5] that directly reads and writes CPU operating frequency system configuration files.

Table 6.3: Northbridge/CPU FID/VID Control Register Bit Format.

Bits	Description
63:32, 24:23, 21:19	Reserved
32:25	Northbridge Voltage ID, Read-Write
22	Northbridge Divisor ID, Read-Write
18:16	P-state ID, Read-Write
15:9	Core Voltage ID, Read-Write
8:6	Core Divisor ID, Read-Write
5:0	Core Frequency ID, Read-Write

Fixed-Frequency Undervolting

For saving energy beyond the DVFS solutions, we leveraged an energy saving undervolting approach for HPC systems [131], where undervolting is conducted for a production cluster by modifying corresponding bits of the northbridge/CPU frequency and voltage ID control register. The register values consist of 64 bits in total, where different bit fragments manage various system power state variables individually. Table 6.3 summarizes the register bit format [35] for processors on the HPCL cluster: The Core Voltage/Frequency/Divisor ID fragments (CoreVid/CoreFid/CoreDid) are used for undervolting. As a general-purpose software level undervolting approach, the interested bits of register values are altered using the Model Specific Register (MSR) interface [19]. Next we illustrate how to extract various ID fragments from specific register values and modify voltage/frequency of cores using corresponding formula. For instance, we input the register with a hexadecimal value 0x30002809 via MSR. From the bit format, we can extract the Core Voltage/Frequency/Divisor ID as 20, 9, and 0 respectively. Moreover, from [35], we have the following architecture-dependent formulae to calculate voltage/frequency:

$$\text{frequency} = 100\text{MHz} \times (\text{CoreFid} + 16)/2^{\text{CoreDid}} \tag{6.16}$$

$$\text{voltage} = 1.550\text{V} - 0.0125\text{V} \times \text{CoreVid} \tag{6.17}$$

Given the register value 0x30002809, it is easy to calculate voltage/frequency to be 1.300 V and 2.5 GHz individually using the above equations. Using MSR, undervolting is implemented by assigning the register with desirable voltage values at the voltage bits. The frequency bits are unchanged to ensure fixed frequency during undervolting.

Failure Emulation by Injecting Hard and Soft Errors

As stated, due to hardware constraints of production processors, voltage cannot be scaled to the levels where observable errors occur. Alternatively, we emulate the real error cases as follows: Using the failure rates at error-triggering voltage levels calculated by Equation (6.1) (demonstrated [131] to be highly accurate compared to real failure rates [39]), we inject hard and soft errors respectively at the calculated failure rates to emulate the incurred failures in HPC runs due to undervolting to such voltage levels. Specifically, hard error injection is performed by manually killing an arbitrary MPI processes during program execution at OS level (for general HPC applications). Soft error injection is however conducted at library level (for matrix-based HPC applications): We randomly select some matrix elements using a random number generator and modify values of the matrix elements to erroneous ones (soft error detection is done within the error checking module of matrix benchmarks). Without loss of generality, we adopt the general-purpose widely used resilience technique Checkpoint/Restart (C/R) [59] to detect (hard errors only) and recover

from the introduced failures (note that C/R is not the optimal resilience technique to protect from soft errors, and here we employ C/R to simplify the evaluation of the proposed models).

Power/Energy Measurement and Estimation

Limited extent of undervolting for production machines also presents us from measuring power/energy directly for the case that real errors are observed from undervolting. Nevertheless, the emulated scaling undervolting scheme adopted from [131] allows us to utilize measured power costs at V_l (the lowest undervolted voltage for production machines) to estimate the power costs at V_{safe_min} (the lowest undervolted voltage for pre-production machines) based on the following power models, which enables our approach to work for general production machines. The three power models represent the baseline power costs at the highest frequency and voltage, the power costs at the highest frequency and the lowest voltage, and the power costs at the lowest frequency and voltage individually. Since our approach cannot undervolt to V_{safe_min}, P_m and P_l are empirically not measurable. We manage to obtain P_m and P_l as follows: Substituting V_{safe_min} in P_m and P_l with V_l, we measure the power costs P'_m and P'_l at V_l and P_h to solve constants AC', I_{sub}, and P_c using the three formula. With AC', I_{sub}, and P_c known, we can calculate P_m and P_l using V_{safe_min} in the formula of P_m and P_l. Given the power costs at V_{safe_min}, we can further calculate the energy costs when undervolting to V_{safe_min}.

$$\begin{cases} P_h &= AC' f_h V_h^2 + I_{sub} V_h + P_c \\ P_m &= AC' f_h V_{safe\_min}^2 + I_{sub} V_{safe\_min} + P_c \\ P_l &= AC' f_l V_{safe\_min}^2 + I_{sub} V_{safe\_min} + P_c \end{cases} \tag{6.18}$$

Input Selection and Model Parameter Derivation

For extrapolating the power costs and resilience-aware speedup of HPC runs using the proposed power/speedup models, we need to derive the values of parameters in Equations (6.4) and (6.7). We also need to find out if the model parameters vary across different tests (i.e., with different problem sizes), since the parameter variation can greatly impact the accuracy of our models.

Table 6.4: Architecture-Dependent Power Constants in Our Models on HPCL/ARC Clusters.

Cluster	Single Core Peak Power Q	Non-CPU Power $\mathbb{C}$	CPU Idle Power Fraction μ
HPCL	11.49 Watts	107.02 Watts	0.63
ARC	N/A	N/A	0.75*

We first determine some application/architecture-dependent constants in our models. Power constants Q, $\mathbb{C}$, and μ were straightforward to obtain from empirical power measurement. Table 6.4 lists measured values of the constants for the two clusters, where only HPCL was equipped with PowerPack that is able to isolate CPU power consumption from other components. We obtain the power costs of a single core at its peak performance Q by dividing the total CPU power costs with the CPU counts in a node. Note that for ARC

we report the system idle power fraction instead of CPU. This number is slightly larger than the CPU idle power fraction for HPCL, which is reasonable since the power costs of other components barely vary between busy and idle modes [71]. Similarly, for a given C/R technique and an HPC application, we can empirically profile the checkpoint overhead C and the restart overhead R.

Table 6.5: Calculated Failure Rates at Different Supply Voltage on the HPCL Cluster (Unit: Voltage (V) and Failure Rate (errors/minute)).

Supply Voltage	Calculated Failure Rate	Error Injection Needed?
1.300	3.649×10^{-6}	No
1.250	4.713×10^{-5}	No
1.200	5.437×10^{-4}	No
1.150	1.700×10^{-2}	No
1.100	0.397	Yes
1.050	2.717	Yes

For each supply voltage V_{dd} level, we can easily solve the failure rate λ under the given voltage using Equation (6.1). Table 6.5 shows the calculated failure rates at different voltage levels. Note that we refer to failure rates reported in [39] when estimating the failure rates in our experiments, as their work was able to undervolt to the error-triggering voltage levels. Since the non-test HPC runs in our experiments finish in 71 - 266 seconds, at all voltage levels except for 1.100 V and 1.050 V, error injection is not needed according to the calculated failure rates.

Recall that when developing power and performance models with the consideration of communication, we rely on an assumption that the ratio between communication time $\kappa(N, P)$ and the total execution time T barely varies with the variation of problem size N

Table 6.6: Communication Time to Total Time Ratio for All Benchmarks with Different Number of Cores and Problem Sizes on the ARC Cluster (Unit: $\kappa(N,P)$ (second) and T (second)).

$\kappa(N,P), T, \frac{\kappa(N,P)}{T}$	Run 1	Run 2	Run 3
MG	P=256, N=Class A 0.03, 0.05, **55.6%**	P=256, N=Class B 0.12, 0.22, **52.7%**	P=256, N=Class C 1.36, 2.32, **58.6%**
CG	P=64, N=Class A 0.06, 0.12, **52.2%**	P=64, N=Class B 2.33, 5.69, **41.0%**	P=64, N=Class C 5.64, 14.78, **38.2%**
FT	P=64, N=Class A 0.11, 0.33, **32.8%**	P=64, N=Class B 1.47, 4.20, **35.1%**	P=64, N=Class C 4.60, 16.57, **27.8%**
EP	P=16, N=Class A 0.30, 2.33, **13.0%**	P=64, N=Class B 0.36, 2.70, **13.4%**	P=256, N=Class C 0.37, 2.70, **13.5%**
MatMul	P=64, N=10k×10k 1.34, 6.12, **22.0%**	P=16, N=20k×20k 33.87, 160.37, **21.1%**	P=256, N=30k×30k 11.35, 41.44, **27.4%**
Chol	P=64, N=5k×5k 0.16, 0.983, **16.3%**	P=64, N=15k×15k 1.18, 5.79, **20.4%**	P=64, N=25k×25k 4.39, 23.25, **18.9%**
LU	P=64, N=10k×10k 0.77, 5.68, **13.5%**	P=64, N=20k×20k 7.40, 45.94, **16.1%**	P=64, N=30k×30k 21.33, 182.34, **11.7%**
QR	P=64, N=10k×10k 0.85, 7.59, **11.2%**	P=64, N=20k×20k 8.77, 60.90, **14.4%**	P=64, N=30k×30k 40.25, 236.76, **17.0%**
LULESH	P=8, N=64 1.66, 320.09, **0.52%**	P=64, N=512 4.12, 328.96, **1.25%**	P=216, N=1728 12.39, 336.90, **3.68%**
AMG	P=96, N=90×90×90 5.14, 107.33, **4.79%**	P=49152, N=150×150×150 N/A	P=960k, N=360×360×1080 N/A

and number of cores P. For each benchmark, $\frac{\kappa(N,P)}{T}$ was collected from multiple test runs (we assume that for each test run and experimental run, N and P are fixed during the execution). We selected different test runs by altering both N and P to ensure cross input validation. For NPB benchmarks, we used input size Class A, B, and C and changed number of cores used from 16, 64, to 256 (for LULESH, due to the application characteristics that P must be a cube of an integer, P ranges from 8, 64, 216). For matrix benchmarks, we chose global matrix sizes 10000×10000, 20000×20000, and 30000×30000. Table 6.6 presents the values of the communication time versus the total execution time for all benchmarks with

different N and P, except for AMG where increasing N and P exceeds available resources on our experimental platform. We can observe that for matrix benchmarks, $\frac{\kappa(N,P)}{T}$ has minor variation as the global matrix size N increases; for NPB benchmarks MG, CG, FT, and EP, there also exists such variability on $\frac{\kappa(N,P)}{T}$ for CG and FT, although it is stable for MG and EP. We calculate the percentage of parallelized code α (see the last column in Table 6.1) in accordance with Equations (6.2) and (6.3), where the empirical speedup, P, $\kappa(N,P)$, and T values were obtained in advance from dynamic profiling of such test runs as well.

6.3.3 Validation of Modeling Accuracy

Using the above fine-grained tuned inputs and parameters, we can thus predict power costs and performance of HPC runs with our extended Amdahl's Law and Karp-Flatt Metric built in Section 6.2. Moreover, for validating the accuracy of our models, we next make head-to-head comparison between the real measured data on our experimental platform and the predicted data from the theoretical modeling, and calculate the modeling accuracy in terms of average error rate. Note that the following presented power and energy data are the total values for the whole HPC system evaluated.

Extended Amdahl's Law for Power Efficiency

Figure 6.8 shows the measured and predicted system power costs on HPCL for the ten benchmarks to evaluate the accuracy of our extended Amdahl's Law for power efficiency, i.e., Equation (6.4). Given all parameters in Equation (6.4) using the above measurement

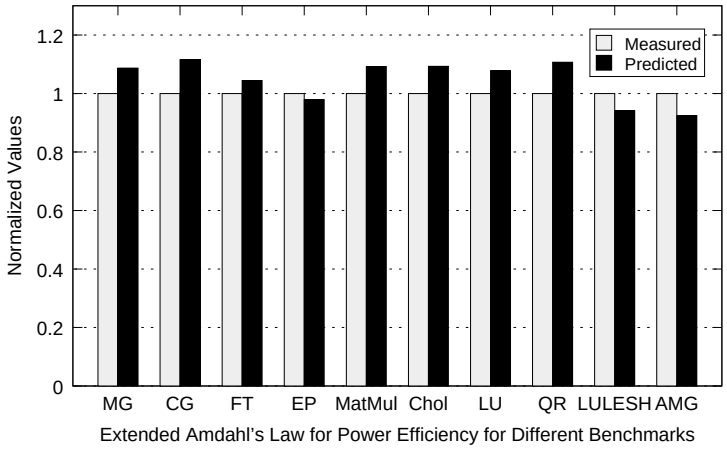


Figure 6.8: Measured and Predicted System Power Consumption for HPC Runs on the HPCL Cluster.

and derivation methods, we can easily extrapolate power costs for various HPC runs. As shown in Figure 6.8, the predicted data matches well with the measured data with an average error rate of 7.7% for three runs of each benchmark with different N and P. Generally, the errors from the power extrapolation for all benchmarks result from the variability of the term $\frac{\kappa(N,P)}{T}$ (see Table 6.6): For applications with stable $\frac{\kappa(N,P)}{T}$, the average error rate of the extrapolation is small (e.g., EP and LULESH), while for application having $\frac{\kappa(N,P)}{T}$ with minor variation, the extrapolation errors are more manifested (e.g., CG and QR).

Extended Karp-Flatt Metric for Speedup with Resilience

We also evaluated the accuracy of our extended Karp-Flatt speedup formula for the HPC runs with failures, equipped with the C/R technique. Figure 6.9 (we averaged the results from the two clusters) compares the measured and predicted speedup for the benchmarks running in such a scenario, based on Equation (6.7). From Figure 6.9, we

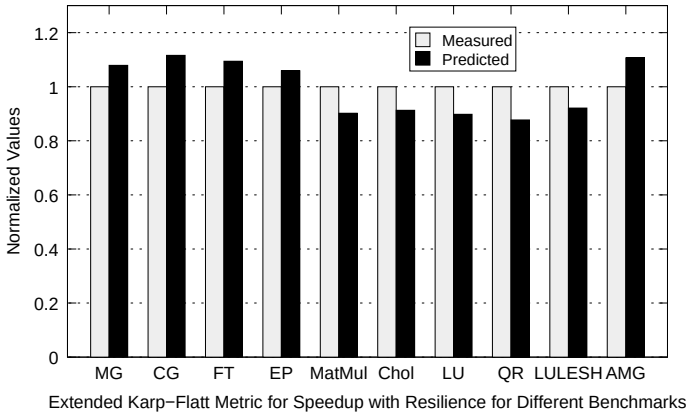


Figure 6.9: Measured and Predicted System Speedup with Resilience for HPC Runs on the HPCL/ARC Clusters.

can see that compared to our extended power model, our extended speedup model for HPC runs with failures and resilience techniques undergoes a higher average error rate (9.4%) against the measurement. In addition to the empirical minor variation of assumed fixed $\frac{\kappa(N,P)}{T}$, another reason for the errors between the measured and predicted speedup comes from the failure distribution formula we adopt. As mentioned in [53], the simplified solve time of applications with C/R in the presence of failures is an approximation to the solution, which may incur errors in the extrapolation. We notice that for matrix benchmarks (MatMul/Chol/LU/QR) and AMG, the predicted results have the most margins with the measured data, which indicates that our speedup model may be less accurate for applications with comparatively large-size checkpoints (i.e., C is large). Refining our model for this type of HPC applications may achieve higher accuracy.

6.3.4 Effects on Energy Efficiency from Typical HPC Parameters

Next we evaluate several critical HPC parameters discussed in Section 6.2.6 that have potentially significant impacts on the integrated energy efficiency in various HPC scenarios. We aim to empirically determine if there exist the optimal values of these parameters for the highest energy efficiency with resilience to validate our models.

Impacts from Checkpoint Intervals

According to the discussion in Section 6.2.6.1, we know that there exists an optimal checkpoint interval τ_{opt} for achieving the highest performance, defined by Daly's two sets of equations individually. One applies for the nominal voltage case and the other works for the undervolting case. For a given failure rate λ and a certain C/R technique (checkpoint overhead C and restart overhead R are known), τ_{opt} is calculated via $\tau_{opt} = \sqrt{2C(\frac{1}{\lambda} + R)}$ for the nominal voltage, and is calculated via Equation (6.14) for the reduced voltage by undervolting. Figure 6.10 shows the normalized system energy efficiency for a given λ (voltage is fixed) and different τ for MG and LULESH. We select to present the data of the two benchmarks because they have similar solve time according to our tests. All other benchmarks follow a similar pattern as the two. We injected errors in the failure rate at V', $V_{safe\_min} < V' < V_l$ (see Table 6.2 for core voltage specification), calculated to be 1.057 errors/minute per Equation (6.1). We also highlight in the figure the calculated optimal checkpoint interval τ_{opt}. From this figure, we can see that the optimal energy efficient τ slightly differs from the theoretical τ_{opt}. Note that in this case the optimal energy efficient τ is also the optimal τ for the highest performance, since the power savings do not change

by using a fixed voltage for the same λ and thus only performance affects energy efficiency.

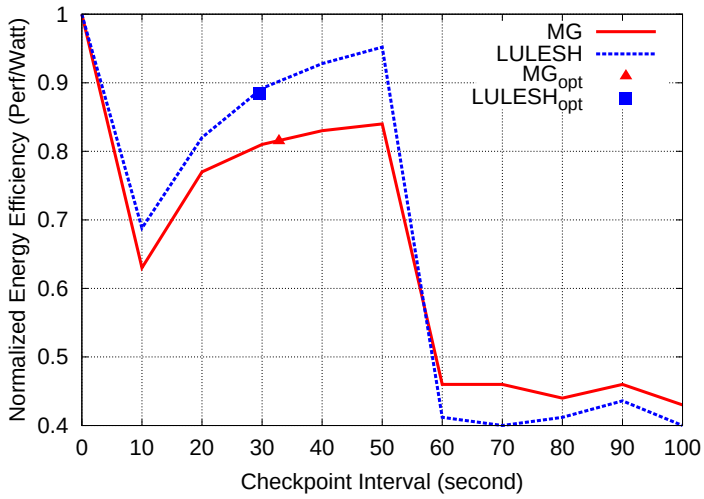


Figure 6.10: Energy Efficiency (MG and LULESH) for Different Checkpoint Intervals on the HPCL Cluster.

Moreover, we can see from Figure 6.10 that there exists the following fluctuation pattern: The energy efficiency drops greatly if checkpointing and restarting was applied (37.4% and 30.9% degradation for MG and LULESH respectively). Note that Figure 6.4 does not show a reference point where the normalized energy efficiency is 1 as in Figure 6.10. Since we injected a fixed number of errors, the restart overhead was also fixed. The energy efficiency increases if less checkpoints were used (i.e., longer checkpoint intervals), which matches well with Figure 6.4. We injected an error at the time around the 55^{th} second (the execution time of MG and LULESH runs is around 100 seconds), the energy efficiency quickly decreases for checkpoint intervals larger than this time period (not shown in Figure 6.4), since the re-execution of the program is necessary due to no checkpoints available (the first checkpoint has not made yet), which greatly increases the total execution time. Lastly,

the energy efficiency barely changes for larger checkpoint intervals due to the same amount of checkpoints (1 in this case).

Impacts from Supply Voltage

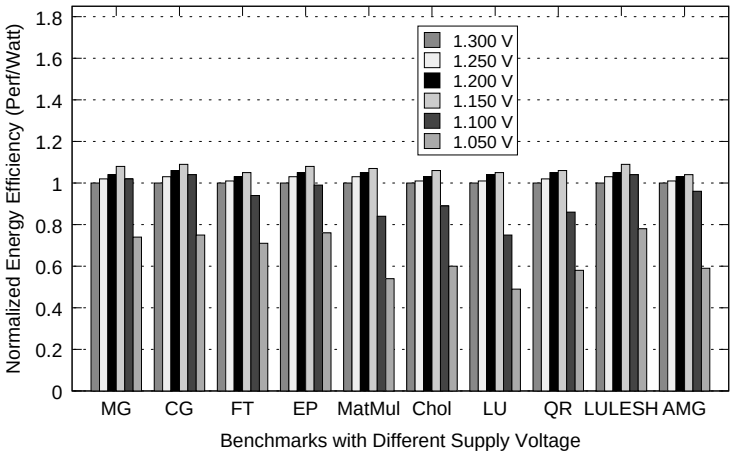


Figure 6.11: Energy Efficiency for HPC Runs with Different Supply Voltage on the HPCL Cluster.

As previously discussed in Section 6.2.6.2, for the scenario of undervolting in the presence of failures, theoretically there exists an optimal supply voltage value that balances the positive energy saving effects from the power savings and the negative energy saving effects from the C/R overhead. The highest energy efficiency is fulfilled at the optimal voltage. Figure 6.11 shows the experimental results on the normalized system energy efficiency under different supply voltage values for all benchmarks. The baseline case is the energy efficiency at the nominal voltage, 1.300 V in our experiments. We observe several findings from the figure: (a) For all benchmarks, the energy efficiency is improved by 1% - 9% when no errors were injected and dramatically degrades at the lowest voltage, i.e.,

1.050 V. This is because the failure rates are exponentially raised as the voltage decreases per Equation (6.1), and at the lowest voltage failure rates are increased to values much larger than 1 error/minute (Table 6.5 lists all used failure rates at different voltage), (b) the optimal voltage for all benchmarks to achieve the highest energy efficiency is 1.150 V, although for some benchmarks (MG, CG, and LULESH), energy can be saved at 1.100 V as well. This is because 1.150 V is in our experiments the lowest voltage that incurs negligible increase in the number of failures during the HPC runs (no error injection is needed per the calculated failure rates and length of runs), and (c) for benchmarks with higher checkpoint and restart overhead C and R, such as MatMul, Chol, LU, QR, and AMG, no energy savings can be achieved from undervolting to V_l or lower compared to the baseline; for benchmarks with lightweight C and R, the energy efficiency at different voltage does not vary much (the highest improvement overall from undervolting is 9%), except for the lowest voltage 1.050 V that incurs significantly more failures. This indicates there exists comparable competition between the positive effects from power savings via voltage reduction, and the negative effects from the extra time costs on detecting the recovering from the failures. The experimental results are very constructive to find the optimal voltage for the highest energy efficiency in the HPC environment. In general, selecting the smallest voltage that does not incur significantly more observable failures during an HPC run should fulfill the optimal energy efficiency.

6.4 Summary

Future HPC systems are required to be encompassing over performance, energy efficiency, and resilience, three crucial dimensions of great concerns by the HPC community nowadays. Enhancing one dimension of the three concerns does not necessarily improve the others, since the variation of some joint HPC parameters can be beneficial to one dimension, while be harmful to the others. There exists a lack of efforts that investigate the entangled effects among the three concerns, between energy efficiency and resilience in particular. In this chapter, we quantify the interplay between energy efficiency and resilience for scalable HPC systems both theoretically and empirically. We propose comprehensive analytical models of the integrated energy efficiency with resilience, by incorporating power and resilience into Amdahl's Law and Karp-Flatt Metric, two classic HPC performance metrics. We also discuss the energy saving effects from typical HPC parameters that have inherent causal relationship with each other. Experimental results on two power-aware HPC clusters indicate that our models for scalable energy efficiency with resilience are accurate to capture the conflicting effects from typical HPC parameters. Our showcases on a wide spectrum of HPC benchmarks also demonstrate that it is feasible using our models to find the balanced HPC configuration for the highest integrated energy efficiency with resilience. We plan to discuss more types of hardware components and resilience techniques, and evaluate on larger scale HPC systems to further validate the capability of our models in the future.

Chapter 7

Related Work

There exist a large body of efforts in the scope of this dissertation. We detail them in the folloing categories: *energy efficient DVFS scheduling strategies* and *interplay between energy efficiency and resilience in HPC*. Next we summarize them individually for comparison purposes.

7.1 Energy Efficient DVFS Scheduling Strategies

Numerous energy saving DVFS scheduling strategies exist without considering possible non-negligible overhead on employ DVFS, including some high performance communication schemes. A considerable amount of other types of energy efficient DVFS scheduling algorithms have been proposed, but only a few of them were designed for high performance scientific computing. We group them into different typical categories below for further discussion.

DVFS Scheduling for Compute Intensive Applications: Numerous work has been done to save energy for compute intensive applications by exploiting CPU slack or idle time from imbalanced CPU-bound applications. Ge *et al.* [70] proposed a runtime system and an integrated performance model for achieving energy efficiency and constraining performance loss through DVFS and performance modeling and prediction. Rountree *et al.* [115] presented another runtime system by improving and extending previous classic scheduling algorithms and achieved significant energy savings with extremely limited performance loss. Kappiah *et al.* [86] proposed a scheduled iteration method that computes the total slack per processor per timestep, then scheduling CPU frequency for the upcoming timestep.

DVFS Scheduling for Data Intensive Applications: There also exists a large amount of work for energy efficient communication via different DVFS scheduling algorithms. A relatively small amount of research has been conducted for reducing energy costs of memory/disk access intensive applications. Kappiah *et al.* [86] devised a system that exploits slack arising at synchronization points of MPI programs by reducing inter-node energy gear via DVFS. Li *et al.* [95] proposed to characterize energy saving opportunities in executions of hybrid MPI/OpenMP applications without performance loss. Predictive models and novel algorithms were presented via statistical analysis of power and time requirements under different configurations. Ge *et al.* [69] observed that memory stalls in the memory-bound sequential application swim from the SPEC CPU2000 benchmark suite produced considerable slack for energy savings via DVFS with alomost no impact on performance. Our work focuses on improving energy efficiency for parallel exeuctions of memory/disk-bound applications on distributed-memory computing systems.

Aggressive and Speculative Mapping and Scheduling: Liu *et al.* [102] leveraged the fact that at runtime some applications typically have shorter execution time than their worst-case execution time, and applied DVFS to dynamically and aggressively reduce voltage and frequency on a heterogeneous system consisting of CPUs and GPUs. Luo *et al.* [104] proposed to improve energy efficiency for thread-level speculation in a same-ISA heterogeneous multicore system with an overhead throttling mechanism and a competent resource allocation scheme. Our work differs from them in that prior knowledge of the worst-case execution time of the application is not a prerequisite, and the target of our work is data intensive applications running on a distributed-memory architecture.

DVFS Scheduling for CPU-bound Operations: Alonso *et al.* [32] leveraged DVFS to enable an energy efficient execution of LU factorization, where idle threads were set into blocked and CPU frequency of the corresponding core was lowered down to save energy with minor performance loss (up to 2.5%). Energy savings achieved were not much (up to 5%) since the approach was only applied to a shared-memory system where the slack can only result from idle CPU usage locally. Kimura *et al.* [93] employed DVFS into two distributed-memory parallel applications to allow tasks with slack to execute at an appropriate CPU frequency that does not increase the overall execution time. Energy savings up to 25% were reported with minor performance loss (as low as 1%). In their work, reducing DVFS overhead was not studied.

DVFS Scheduling for MPI Programs: Kappiah *et al.* [69] presented a dynamic system that reduces CPU frequency on nodes with less computation and more slack to use. With little performance loss, their approach was able to save energy for power-scalable clusters,

where the computation load was imbalanced. Springer *et al.* [124] presented a combination of performance modeling, performance prediction, and program execution to find a near-optimal schedule of number of nodes and CPU frequency to satisfy energy costs and execution time requirements. Li *et al.* [96] proposed a strategy to improve energy efficiency for hybrid parallel applications where both shared and distributed-memory programming models (OpenMP and MPI) were employed. The key difference between our approach and these solutions is that we take the overhead on applying DVFS into account and minimize its costs to save energy.

Improving MPI Communication Performance: Chan *et al.* [45] redesigned MPI communication algorithms to achieve that one node can communicate with multiple nodes simultaneously with lower costs rather than one-to-one at a time. Faraj *et al.* [65] presented a customized system that generates efficient MPI collective communication routines via automatically-generated topology specific routines and performance tuning to achieve high performance consistently. Karwande *et al.* [89] presented an MPI prototype supporting compiled communication to improve performance of MPI communication routines, which allowed the user to manage network resources to aggressively optimize communication. Hunold *et al.* [83] proposed a mechanism that automatically selected a suitable set of blocking factors and block sizes for pdgemm() to maximize performance. Our approach differs from these techniques, since it improves MPI communication performance via highly-tuned pipeline broadcast that maximizes the slack utilization, without modifying MPI communication routines and the pdgemm() routine interface.

OS-level. There exist a large body of OS level energy efficient approaches for high performance scientific applications. Lim *et al.* [101] developed a runtime system that dynamically and transparently reduces CPU power for communication phases to minimize energy-delay product. Ge *et al.* [70] proposed a runtime system and an integrated performance model for achieving energy efficiency and constraining performance loss through performance modeling and prediction. Rountree *et al.* [116] developed a *SC* approach that employs a linear programming solver collecting communication trace and power characteristics for generating an energy saving scheduling. Subsequent work [115] presented another runtime system by improving and extending previous classic scheduling algorithms and achieved significant energy savings with extremely limited performance loss.

Application-level. Kappiah *et al.* [86] introduced a scheduled iteration method that computes the total slack per processor per timestep, then scheduling CPU frequency for the upcoming timestep. Liu *et al.* [103] presented a technique that tracks the idle durations for one processor to wait for others to reach the same program point, and utilizes this information to reduce the idle time via DVFS without performance loss. Tan *et al.* [127] proposed an adaptively aggressive scheduling strategy for data intensive applications with moderated performance trade-off using speculation. Subsequent work [132] proposed an adaptive memory-aware strategy for distributed matrix multiplication that trades grouped computation/communication with memory costs for less overhead on employing DVFS. Liu *et al.* [102] proposed a power-aware static mapping technique to assign applications for a CPU/GPU heterogeneous system that reduces power and energy costs via DVFS on both CPU and GPU, with timing requirements satisfied.

Simulation-based. There exist some efforts on improving energy efficiency for numerical linear algebra operations like Cholesky/LU/QR factorization, but most of them either are based on simulation or only work for a single multicore machine. Few studies have been conducted on power/energy efficient matrix factorizations running on distributed-memory architectures. Slack reclamation methods such as Slack Reduction and Race-to-Idle algorithms [30] [33] have been proposed to save energy for dense linear algebra operations on shared-memory multicore processors. Instead of running benchmarks on real machines, a power-aware simulator, in charge of runtime scheduling to achieve task level parallelism, was employed to evaluate the proposed power-control policies for linear algebra operations. DVFS techniques used in their approaches were also simulated. Subsequent work [32] leveraged DVFS to optimize task-parallel execution of a collection of dense linear algebra tasks on shared-memory multicore architectures, where experiments were performed at thread level. Since no communication was involved, these approaches did not achieve significant energy savings due to no utilization of slack from communication latency.

7.2 Interplay between Energy Efficiency and Resilience in High Performance Computing

To the best of our knowledge, our energy saving undervolting work is the first of its kind that models and discusses the interplay between energy efficiency and resilience at scale. There exist few efforts investigating the joint relationship among performance, energy efficiency, and resilience for HPC systems. Rafiev *et al.* [112] studied the interplay among time, energy costs, and reliability for a single-core and a multicore system respectively,

while they focused on concurrency and did not quantitatively elaborate the impacts of frequency/voltage on performance and reliability. Yetim *et al.* [148] presented an energy optimization framework using mixed-integer linear programming while meeting performance and reliability constraints. Targeting the application domain of multimedia, this work exploited the workload characteristics that limited error tolerance can be traded off for energy reduction. Most of the related efforts have been conducted in the following areas:

Real-Time/Embedded Processors and Systems-on-Chip: Extensive research has been performed to save energy and preserve system reliability for real-time embedded processors and systems-on-chip. Zhu *et al.* [154] discussed the effects of energy management via frequency and voltage scaling on system failure rates. This work is later extended to reliability-aware energy saving scheduling that allocates slack for multiple real-time tasks [153], and a generalized Standby-Sparing technique for multiprocessor real-time systems, considering both transient and permanent faults [75]. These studies made some assumptions suitable for real-time embedded systems, but not applicable to large-scale HPC systems with complex hardware and various types of faults. Pop *et al.* [111] explored heterogeneity in distributed embedded systems and developed a logic programming solution to identify a reliable scheduling scheme that saves energy. This work ignored runtime process communication, which is an important factor of performance and energy efficiency for HPC systems and applications. The Razor work [62] implemented a prototype 64-bit Alpha processor design that combines circuit and architectural techniques for low-cost speed path error detection/correction from operating at a lower supply voltage. With minor error recovery overhead, substantial energy savings can be achieved while guaranteeing correct operations

of the processor. Our work differs from Razor since we consider hard/soft errors in HPC runs due to undervolting. Razor power/energy costs were simulated at circuit level while our results were obtained from real measurements in an HPC environment at cluster level. Similar power-saving and resilient-against-error hardware techniques have been proposed such as Intel's Near-Threshold Voltage (NTV) design [90] on a full x86 microprocessor.

Memory Systems: As ECC memory prevails, numerous studies have explored energy efficient architectures and error-detection techniques in memory systems. Wilkerson *et al.* [143] proposed to trade off cache capacity for reliability to enable low-voltage operations on L1 and L2 caches to reduce power. Their subsequent work [29] investigated an energy saving cache architecture using variable-strength ECC to minimize latency and area overhead. Unlike our work that is evaluated with real HPC systems and physical energy measurements, they used average instructions per cycle and voltage to estimate energy costs. Bacha *et al.* [39] employed a firmware-based voltage scaling technique to save energy for a pre-production multicore architecture, under increasing fault rates that can be tolerated by ECC memory. Although our work similarly scales down voltage with a fixed frequency to save power, it is different in two aspects: Ours targets *general faults* on common *HPC* production machines at scale, while theirs specifically handles *ECC errors* on a *pre-production* architecture. To balance performance, power, and resilience, Li *et al.* [100] proposed an ECC memory system that can adapt memory access granularity and several ECC schemes to support applications with different memory behaviors. Liu *et al.* [56] developed an application-level technique for smartphones to reduce the refresh power in DRAM at the cost of a modest increase in non-critical data corruption, and empirically showed that such errors have few impacts on

the final outputs. None of the above approaches analytically model the entangling effects of energy efficiency and resilience at scale like ours.

Saving Energy for Resilience Techniques: There exist a number of studies that investigate energy saving opportunities for resilience techniques. They either exploit metric-based mechanism to evaluate energy efficiency for failure-prone large-scale HPC systems, or capture energy saving opportunities during the use of resilience techniques during HPC runs. Grant *et al.* [74] conducted a comparison study between energy saving techniques that takes into account reliability of HPC sytems at scale. They proposed an energy-reliability metric that imposes a quantifiable penalty to energy savings techniques for increased runtime providing reliability, and accounts for the probability of failure increase due to runtime overhead from energy saving techniques. Diouri *et al.* [57] proposed an energy estimation framework for selecting the most energy efficient fault tolerance protocol without running HPC applications for different execution setting. The accuracy of the proposed estimation framework was evaluated with real HPC runs and energy monitoring. Mills *et al.* [107] argued that resilience techniques such as checkpoint/restart incur considerable performance and energy overhead on data movement. They attempted to reduce CPU power during the I/O intensive checkpointing during HPC runs and 10% total energy savings were reported with little performance loss. Aupy *et al.* [38] developed an energy-aware checkpointing method for divisible workloads, where the total execution time is bounded/enforced by soft or hard deadlines and the total energy costs are minimized. Rajachandrasekar *et al.* [113] discussed that the naïve use of power capping during checkpointing phases can incur considerable performance degradation, and proposed a novel power-aware checkpointing

framework named Power-Check that efficiently utilizes I/O and CPU by data funneling and selective core power capping. Experimental results demonstrated significant energy savings up to 48% and also performance gain up to 14% during checkpointing.

Nevertheless, there exist a large body of studies that quantify only energy costs and performance, or resilience and performance, at single-node level, at scale, or based on simulation.

Energy Efficiency Quantification. Woo and Lee [144] built an analytical model that extends the Amdahl's Law for energy efficiency in scalable many-core processor design. They considered three many-core design styles of processors only without communication, while we focus on networked symmetric multicore processors, and communication time and power costs for processors across nodes are involved in our models. By augmenting the Amdahl's Law, Cassidy and Andreou [43] derived a general objective function linking performance gain with energy-delay costs in microarchitecture and applied it to design the optimal chip multiprocessor (CMP) architecture, while our work aims to achieve the optimal performance/energy efficiency in terms of FLOPS per watt. Song *et al.* [122] developed an energy model to evaluate and predict energy-performance trade-offs for various HPC applications at large scale. Ge and Cameron [68] devised a power-aware speedup metric that quantify the interacting effects between parallelism and frequency, and used it to predict performance and power-aware speedup for scientific applications. Their models were accurate and scalable, but did not incorporate the effects of energy saving DVFS and undervolting techniques as in our work. Moreover, they did not evaluate the interplay among typical HPC parameters, where we conduct an extensive theoretical/empirical study.

Resilience Quantification. Zheng and Lan [152] modeled the impacts of failures and the effects of resilience techniques in HPC, and used their models to predict scalability for HPC runs with possibility of failures. Wang *et al.* [140] proposed a unified speedup metric that incorporates checkpointing overhead into classic speedup metrics in the presence of failures. Yu *et al.* [150] created a novel resilience metric named data vulnerability factor to holistically integrate application and hardware knowledge into resilience analysis. Again, all of these approaches however did not consider energy saving DVFS and undervolting techniques, both of which can significantly affect energy efficiency and resilience, with negligible performance loss. Moreover, analytical models in [140] are not as fine-grained as ours, since relationship among important HPC parameters was simplified such that it is not clear how they interact and by what extent they affect speedup (and energy efficiency).

Simulation-Based Quantification. Li and Martínez [97] investigated power-performance correlation of parallel applications running on a CMP, aiming to optimize power costs under a performance budget and vice versa. They conducted simulation-based experiments to evaluate the proposed power-performance optimization for one single parallel application. Suleman *et al.* [125] proposed a technique that accelerates the execution of critical sections, a code fragment of shared data accessed by only one thread at a given time, which differs from the sequential code in the Amdahl's Law. Experimental results using lock-based multithreaded workloads on three different CMP architectures indicated significant performance and scalability improvement. Bois *et al.* [41] developed a framework of generating synthetic workloads to evaluate energy efficiency for multicore power-aware systems. The proposed framework effectively showed the energy and performance trade-off for gen-

erated workloads. All these simulation approaches are at single-node level, which may need considerate adaptation to work for large-scale HPC systems.

Chapter 8

Conclusions

My PhD research focuses on building fast [132] [127] [126], energy efficient [131] [129] [134] [128] [132] [127] [130], and fault tolerant [131] [130] HPC applications, and HPC software debugging [133] for large-scale emerging HPC architectures. Below, I first summarize the research efforts presented in this dissertation, and then briefly discuss my research plan in the near future.

8.1 Conclusive Remarks

8.1.1 Consolidating Energy Efficient High Performance Scientific Computing in Large-scale HPC Systems

Scientific applications running nonstop on large-scale HPC systems need to efficiently use energy for execution. Comprising millions of components, todays HPC systems already consume megawatts of power; to meet an insatiate demand for performance from mission-critical applications, future systems will consist of even more components and con-

sume more power. Efficient use of energy by scientific applications not only reduces energy costs but also allows greater performance under a given power budget and improves system reliability.

Linear algebra has been widely used in almost all science and engineering fields, and has been considered as the core component of high performance scientific computing nowadays. While many linear algebra libraries have been developed to optimize their performance, no linear algebra libraries optimize their energy efficiency at the library design time. Demanding requirements of energy efficiency in HPC in this era are becoming prevalent due to the growing costs for powering supercomputers. Emerging heterogeneous systems combining CPU with accelerators, such as GPU and coprocessors, are superior in performance and energy efficiency. Yet, software-based hardware techniques can further gain energy savings per application characteristics at runtime.

Widely used linear algebra libraries are often designed to maximize performance without considering energy efficiency. While existing OS level energy saving approaches can often be directly applied to linear algebra libraries to save energy without modifying the library source, these approaches usually can not optimize the energy efficiency according to the algorithmic characteristics of the linear algebra operations. In our work, we show that substantial energy savings can be achieved in task-parallel applications running on distributed-memory architectures, taking distributed linear algebra libraries for example, by partially giving up the generality of OS level approaches to leverage the algorithmic characteristics of the applications. Although working at library level, the proposed algorithmic energy saving approach does not have the generality of OS level approaches and

requires source modification of target libraries, the substantial energy savings is worthwhile for widely used linear algebra libraries like ScaLAPACK and DPLASMA because, once it is done, it will benefit a large number of users and a wide range of applications that employ routines from such libraries.

8.1.2 Balancing Energy Saving and Resilience Tradeoffs in HPC Systems

Lowering operating frequency and/or supply voltage of hardware components, i.e., DVFS is one important approach to reduce power and energy consumption of a computing system for two primary reasons: First, CMOS-based components (e.g., CPU, GPU, and memory) are the dominant power consumers in the system. Second, power costs of these components are proportional to the product of operating frequency and supply voltage squared. In general, supply voltage has a positive correlation with (not strictly proportional/linear to) the operating frequency for DVFS-capable components, i.e., scaling up/down frequency results in voltage raise/drop accordingly.

Nevertheless, existing DVFS techniques are essentially frequency-directed and fail to fully exploit the potential power reduction and energy savings. With DVFS, voltage is only lowered to comply with the frequency reduction in the presence of slack. For a given frequency, cutting-edge hardware components can be supplied with a voltage that is lower than the one paired with the given frequency. The enabling technique, undervolting is independent of frequency scaling, i.e., lowering only supply voltage of a chip without reducing its operating frequency. Undervolting is advantageous in the sense that: (a) It can keep the component frequency unchanged such that the computation throughput is well

maintained, and (b) it can be uniformly applied to both slack and non-slack phases of HPC runs for power reduction.

The challenge of employing undervolting as a general power saving technique in HPC lies in efficiently addressing the increasing failure rates caused by it. Both hard/soft errors may occur if components undergo undervolting. Several studies have investigated architectural solutions to support reliable undervolting with simulation. The study by Bacha *et al.* presented an empirical undervolting system on Intel Itanium II processors that resolves the arising ECC memory faults yet improves the overall energy savings. While this work aims to maximize the power reduction and energy savings, it relies on pre-production processors that allow such thorough exploration on the undervolting schemes, and also requires additional hardware support for the ECC memory. We thereby investigate the interplay between energy efficiency and resilience for large-scale parallel systems, and demonstrate theoretically and empirically that significant energy savings can be obtained using a combination of undervolting and mainstream software-level resilience techniques on todays HPC systems, without requiring hardware redesign. We aim to explore if the future exascale systems are going towards the direction of low-voltage embedded architectures in order to guarantee energy efficiency, or they can rely on advanced software-level techniques to achieve high system resilience and efficiency.

8.2 Future Directions

Going forward, I hope to leverage my area of expertise to explore a broad spectrum of other areas. Below, I discuss two potential research directions that I plan to take in the

near future: (a) algorithm-based energy efficiency, and (b) integrated high performance, energy efficient, and fault tolerant hardware and software.

8.2.1 Algorithm-Based Energy Efficiency

The pressing demands of improving energy efficiency for high performance scientific computing nowadays have motivated a large body of software-controlled hardware solutions that strategically switch hardware components to a low-power state, when the peak performance of the components is not necessary. Although OS level solutions can effectively save energy in a black-box fashion, for applications with random/variable execution patterns, slack prediction can be error-prone and thus the optimal energy efficiency can be blundered away. I thereby propose to utilize algorithmic characteristics to predict slack accurately and thus maximize potential energy savings.

DVFS approaches have been widely adopted to improve energy efficiency for task-parallel applications. With high generality, OS level solutions are considered effective. We observe that for applications such as distributed dense matrix factorizations, the optimal energy efficiency cannot be achieved by OS level solutions due to inaccurate slack prediction. Giving up partial generality, I propose to utilize algorithmic characteristics for obtaining slack accurately and thus saving more energy, with negligible performance loss.

8.2.2 Integrated High Performance, Energy Efficient, and Fault Tolerant Hardware and Software

Future large-scale HPC systems require both high energy efficiency and resilience to achieve ExaFLOPS computational power and beyond. Performance, energy efficiency,

and resilience are three crucial challenges for HPC systems to reach exascale. While the three issues have been extensively studied individually, little has been done to understand the interplay between performance, energy efficiency, and resilience for HPC systems.

I propose to investigate HPC hardware and software that is capable of balancing performance, energy efficiency, and resilience at scale. Specifically, detailed work can be: (a) building analytical and quantitative models to study the impact of undervolting and DVFS solutions on HPC applications execution time, energy costs, and fault tolerance for state-of-the-art emerging HPC architectures, (b) developing reliable and cost-efficient HPC software that leverages software-level cost-efficient resilience techniques, and (c) devising novel parallel programming abstractions and failure detection and recovery models to lessen programming efforts for energy efficiency and resilience at extreme scale in the HPC environment.

Bibliography

[1] *ASC Sequoia Benchmark Codes*. https://asc.llnl.gov/sequoia/benchmarks/.

[2] *Automatically Tuned Linear Algebra Software (ATLAS)*. http://math-atlas.sourceforge.net/.

[3] *BIOS and Kernel Developers Guide (BKDG) For AMD Family 10h Processors*. http://developer.amd.com/wordpress/media/2012/10/31116.pdf.

[4] *BLAS (Basic Linear Algebra Subprograms)*. http://www.netlib.org/blas/.

[5] *CPUFreq - CPU Frequency Scaling*. https://wiki.archlinux.org/index.php/CPU_frequency_scaling.

[6] *CPUSpeed*. http://carlthompson.net/Software/CPUSpeed/.

[7] *CULA: NVIDIA GPU Accelerated Linear Algebra*. http://www.culatools.com.

[8] *DPLASMA: Distributed Parallel Linear Algebra Software for Multicore Architectures*. http://icl.cs.utk.edu/dplasma/.

[9] *Exascale Computing Initiative Update 2012, US Department of Energy*. http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/aug12/2012-ECI-ASCAC-v4.pdf.

[10] *Green500 Supercomputer Lists*. http://www.green500.org/.

[11] *HPL - High-Performance Linpack Benchmark*. http://www.netlib.org/benchmark/hpl/.

[12] *Intel Hyper-Threading Technology*. http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[13] *Intel® Itanium® Processor 9560 Specifications*. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/itanium-9500-brief.pdf.

[14] *K computer*. http://www.aics.riken.jp/en/k-computer/about/.

[15] *LAPACK-Linear Algebra PACKage*. http://www.netlib.org/lapack/.

[16] *Linux File Copy Command*. http://www.linux.org/article/view/useful-commands-the-cp-command.

[17] *Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)*. https://codesign.llnl.gov/lulesh.php.

[18] *MAGMA (Matrix Algebra on GPU and Multicore Architectures)*. http://icl.cs.utk.edu/magma/.

[19] *Model-Specific Register (MSR) Tools Project*. https://01.org/msr-tools.

[20] *NAS Parallel Benchmarks (NPB)*. http://www.nas.nasa.gov/publications/npb.html.

[21] *NVIDIA System Management Interface (nvidia-smi)*. https://developer.nvidia.com/nvidia-system-management-interface/.

[22] *A Parallel Algebraic Multigrid (AMG) Solver for Linear Systems*. https://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/amg/.

[23] *Parallel MPI BZIP2 (MPIBZIP2)*. http://compression.ca/mpibzip2/.

[24] *PIC: Pacific Northwest National Laboratory Institutional Computing*. https://cvs.pnl.gov/PIC/wiki/PicCompute.

[25] *Renewable Energy and Energy Efficiency for Tribal Community and Project Development, US Department of Energy*. http://apps1.eere.energy.gov/tribalenergy/pdfs/energy04_terms.pdf.

[26] *ScaLAPACK - Scalable Linear Algebra PACKage*. http://www.netlib.org/scalapack/.

[27] *TOP500 Supercomputer Lists*. http://www.top500.org/.

[28] *Watts up? Meters*. https://www.wattsupmeters.com/.

[29] A. R. Alameldeen, I. Wagner, Z. Chishti, W. Wu, C. Wilkerson, and S.-L. Lu. Energy-efficient cache design using variable-strength error-correcting codes. In *Proc. International Symposium on Computer Architecture (ISCA)*, pages 461–472, 2011.

[30] P. Alonso, M. F. Dolz, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. DVFS-control techniques for dense linear algebra operations on multi-core processors. *Computer Science - Research and Development*, 27(4):289–298, November 2012.

[31] P. Alonso, M. F. Dolz, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Reducing energy consumption of dense linear algebra operations on hybrid CPU-GPU platforms. In *Proc. International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 56–62, 2012.

[32] P. Alonso, M. F. Dolz, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Saving energy in the LU factorization with partial pivoting on multi-core processors. In *Proc. International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 353–358, 2012.

[33] P. Alonso, M. F. Dolz, R. Mayo, and E. S. Quintana-Ortí. Improving power efficiency of dense linear algebra algorithms on multi-core processors via slack control. In *Proc. International Conference on High Performance Computing and Simulation (HPCS)*, pages 463–470, 2011.

[34] P. Alonso, M. F. Dolz, R. Mayo, and E. S. Quintana-Ortí. Modeling power and energy of the task-parallel Cholesky factorization on multicore processors. *Computer Science - Research and Development, Special Issue*, August 2012.

[35] AMD. *BIOS and Kernel Developers Guide (BKDG) For AMD Family 10h Processors*. http://developer.amd.com/wordpress/media/2012/10/31116.pdf, 2012.

[36] G. M Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. In *Proc. AFIPS Spring Joint Computer Conference*, pages 483–485, 1967.

[37] H. Anzt, V. Heuveline, J. Aliaga, M. Castillo, J. C. Fernández, R. Mayo, and E. S. Quintana-Ortí. Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms. In *Proc. International Green Computing Conference (IGCC)*, pages 1–6, 2011.

[38] G. Aupy, A. Benoit, R. Melhem, P. Renaud-Goud, and Y. Robert. Energy-aware checkpointing of divisible tasks with soft or hard deadlines. In *Proc. International Green Computing Conference (IGCC)*, pages 1–8, 2013.

[39] A. Bacha and R. Teodorescu. Dynamic reduction of voltage margins by leveraging on-chip ECC in Itanium II processors. In *Proc. International Symposium on Computer Architecture (ISCA)*, pages 297–307, 2013.

[40] G. Ballard, J. Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. Communication-optimal parallel algorithm for strassen's matrix multiplication. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 193–204, 2012.

[41] K. D. Bois, T. Schaeps, S. Polfliet, F. Ryckbosch, and L. Eeckhout. SWEEP: Evaluating computer system energy efficiency using synthetic workloads. In *Proc. International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC)*, pages 159–166, 2011.

[42] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemariner, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Distributed dense numerical linear algebra algorithms on massively parallel architectures: DPLASMA. Technical Report UT-CS-10-660, September 2010.

[43] A. S. Cassidy and A. G. Andreou. Beyond Amdahl's Law: An objective function that links multiprocessor performance gains to delay and energy. *IEEE Transactions on Computers*, 61(8):1110–1126, August 2012.

[44] M. Castillo, J. C. Fernández, R. Mayo, E. S. Quintana-Ortí, and V. Roca. Analysis of strategies to save energy for message-passing dense linear algebra kernels. In *Proc. International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 346–352, 2012.

[45] E. Chan, R. van de Geijn, W. Gropp, and R. Thakur. Collective communication on architectures that support simultaneous communication over multiple links. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 2–11, 2006.

[46] G. Chen, K. Malkowski, M. Kandemir, and P. Raghavan. Reducing power with performance constraints for parallel sparse applications. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2005.

[47] L. Chen, Z. Chen, P. Wu, R. Ge, and Z. Zong. Energy efficient parallel matrix-matrix multiplication for DVFS-enabled clusters. In *Proc. International Workshop on Power-Aware Systems and Architectures, with ICPP*, pages 239–245, 2012.

[48] Z. Chen. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 167–176, 2013.

[49] J. Choi. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. In *Proc. High Performance Computing on the Information Superhighway (HPC-Asia)*, pages 224–229, 1997.

[50] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. The design and implementation of the ScaLAPACK LU, QR and Cholesky factorization routines. *Scientific Programming*, 5(3):173–184, August 1996.

[51] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 18–28, 2004.

[52] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–12, 1993.

[53] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, February 2006.

[54] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proc. International Conference on Autonomic Computing (ICAC)*, pages 31–40, 2011.

[55] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High performance linpack benchmark: A fault tolerant implementation without checkpointing. In *Proc. International Conference on Supercomputing (ICS)*, pages 162–171, 2011.

[56] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. Memscale: Active low-power modes for main memory. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 225–238, 2011.

[57] M. Diouri, O. Glück, L. Lefèvre, and F. Cappello. ECOFIT: A framework to estimate energy consumption of fault tolerance protocols for HPC applications. In *Proc. International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 522–529, 2013.

[58] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 225–234, 2012.

[59] J. Duell. The design and implementation of Berkeley lab's Linux Checkpoint/Restart. Technical report, Lawrence Berkeley National Laboratory, 2003.

[60] R. Efraim, R. Ginosar, C. Weiser, and A. Mendelson. Energy aware race to halt: A down to EARtH approach for platform energy management. *IEEE Computer Architecture Letters*, 13(1):25–28, January 2014.

[61] M. Elgebaly and M. Sachdev. Efficient adaptive voltage scaling system through on-chip critical path emulation. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED)*, pages 375–380, 2004.

[62] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proc. International Symposium on Microarchitecture (MICRO)*, pages 7–18, 2003.

[63] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proc. International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[64] M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Understanding the future of energy-performance trade-off via dvfs in HPC environments. *Journal of Parallel and Distributed Computing*, 72(4):579–590, April 2012.

[65] A. Faraj and X. Yuan. Automatic generation and tuning of MPI collective communication routines. In *Proc. International Conference on Supercomputing (ICS)*, pages 393–402, 2005.

[66] E. Feller, C. Rohr, D. Margery, and C. Morin. Energy management in IaaS clouds: A holistic approach. In *Proc. International Conference on Cloud Computing (CLOUD)*, pages 204–212, 2012.

[67] V. W. Freeh and D. K. Lowenthal. Using multiple energy gears in MPI programs on a power-scalable cluster. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 164–173, 2005.

[68] R. Ge and K. W. Cameron. Power-aware speedup. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, 2007.

[69] R. Ge, X. Feng, and K. W. Cameron. Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 34, 2005.

[70] R. Ge, X. Feng, W.-C. Feng, and K. W. Cameron. CPU MISER: A performance-directed, run-time system for power-aware clusters. In *Proc. International Conference on Parallel Processing (ICPP)*, page 18, 2007.

[71] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. W. Cameron. PowerPack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):658–671, May 2010.

[72] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, and F. H. Streitz. Extending stability beyond CPU millennium: A micron-scale atomistic simulation of Kelvin-Helmholtz instability. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 58, 2007.

[73] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(3):12–21, August 1993.

[74] R. E. Grant, S. L. Olivier, J. H. Laros, R. Brightwell, and A. K. Porterfield. Metrics for evaluating energy saving techniques for resilient HPC systems. In *Proc. International Workshop on High-Performance, Power-Aware Computing (HPPAC), with IPDPS*, pages 790–797, 2014.

[75] Y. Guo, D. Zhu, and H. Aydin. Generalized standby-sparing techniques for energy-efficient fault tolerance in multiprocessor real-time systems. In *Proc. International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 62–71, 2013.

[76] W. Harrod. A journey to exascale computing. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC) Companion*, pages 1702–1730, 2012.

[77] T. Hoefler, C. Siebert, and W. Rehm. A practically constant-time MPI broadcast algorithm for large-scale InfiniBand clusters with multicast. In *Proc. International Workshop on Communication Architectures for Clusters, with IPDPS*, pages 232–239, 2007.

[78] J. D. Hogg. A dag-based parallel cholesky factorization for multicore systems. Technical Report RAL-TR-2008-029, October 2008.

[79] Y. Hotta, M. Sato, H. Kimura, S. Matsuoka, T. Boku, and D. Takahashi. Profile-based optimization of power performance by using dynamic voltage scaling on a PC cluster. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[80] C.-H. Hsu and W.-C. Feng. A power-aware run-time system for high-performance computing. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 1, 2005.

[81] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proc. ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 38–48, 2003.

[82] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED)*, pages 275–278, 2001.

[83] S. Hunold and T. Rauber. Automatic tuning of PDGEMM towards optimal performance. In *Proc. International European Conference on Parallel Processing (Euro-Par)*, pages 837–846, 2005.

[84] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED)*, pages 197–202, 1998.

[85] H. Jin and R. F. Van der Wijngaart. Performance characteristics of the multi-zone NAS parallel benchmarks. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[86] N. Kappiah, V. W. Freeh, and D. K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 33, 2005.

[87] C. Karlsson, T. Davies, C. Ding, H. Liu, and Z. Chen. Optimizing process-to-core mappings for two dimensional broadcast/reduce on multicore architectures. In *Proc. International Conference on Parallel Processing (ICPP)*, pages 404–413, 2011.

[88] A. H. Karp and H. P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, May 1990.

[89] A. Karwande, X. Yuan, and D. K. Lowenthal. CC–MPI: a compiled communication capable MPI prototype for ethernet switched clusters. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 95–106, 2003.

[90] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar. Near-threshold voltage (NTV) design – opportunities and challenges. In *Proc. Design Automation Conference (DAC)*, pages 1153–1158, 2012.

[91] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. C. Hoe. Multi-bit error tolerant caches using two-dimensional error coding. In *Proc. International Symposium on Microarchitecture (MICRO)*, pages 197–209, 2007.

[92] K. H. Kim, R. Buyya, and J. Kim. Power aware scheduling of bag-of-tasks applications with deadline constraints on DVS-enabled clusters. In *Proc. International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 541–548, 2007.

[93] H. Kimura, M. Sato, Y. Hotta, T. Boku, and D. Takahashi. Empirical study on reducing energy of parallel programs using slack reclamation by DVFS in a power-scalable high performance cluster. In *Proc. International Conference on Cluster Computing (CLUSTER)*, pages 1–10, 2006.

[94] I. A. Lee, C. E. Leiserson, T. B. Schardl, J. Sukha, and Z. Zhang. On-the-fly pipeline parallelism. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 140–151, 2013.

[95] D. Li, B. R. de Supinski, M. Dolz, D. S. Nikolopoulos, and K. W. Cameron. Strategies for energy efficient resource management of hybrid programming models. *IEEE Transactions on Parallel and Distributed Systems*, 24(1):144–157, January 2013.

[96] D. Li, B. R. de Supinski, M. Schulz, K. W. Cameron, and D. S. Nikolopoulos. Hybrid MPI/OpenMP power-aware computing. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, 2010.

[97] J. Li and J. F. Martínez. Power-performance considerations of parallel computing on chip multiprocessors. *ACM Transactions on Architecture and Code Optimization*, 2(4):397–422, December 2005.

[98] J. Li, J. F. Martinez, and M. C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, page 14, 2004.

[99] J. Li, K. Shuang, S. Su, Q. Huang, P. Xu, X. Cheng, and J. Wang. Reducing operational costs through consolidation with resource prediction in the cloud. In *Proc. International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 793–798, 2012.

[100] S. Li, D. H. Yoon, K. Chen, J. Zhao, J. H. Ahn, J. B. Brockman, Y. Xie, and N. P. Jouppi. MAGE: Adaptive granularity and ECC for resilient and power efficient memory systems. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 33, 2012.

[101] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 107, 2006.

[102] C. Liu, J. Li, W. Huang, J. Rubio, E. Speight, and X. Lin. Power-efficient time-sensitive mapping in heterogeneous systems. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 23–32, 2012.

[103] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin. Exploiting barriers to optimize power consumption of CMPs. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[104] Y. Luo, V. Packirisamy, W.-C. Hsu, and A. Zhai. Energy efficient speculative threads: Dynamic thread allocation in same-ISA heterogeneous multicore systems. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 453–464, 2010.

[105] H. Mair, A. Wang, G. Gammie, D. Scott, P. Royannez, S. Gururajarao, M. Chau, R. Lagerquist, L. Ho, M. Basude, N. Culp, A. Sadate, D. Wilson, F. Dahan, J. Song, B. Carlson, and U. Ko. A 65-nm mobile multimedia applications processor with an adaptive power management scheme to compensate for variations. In *Proc. VLSI Symposium*, pages 224–225, 2007.

[106] E. Meneses, O. Sarood, and L. V. Kalé. Assessing energy efficiency of fault eolerance protocols for HPC systems. In *Proc. International Symposium on Computer Architecture and High Performance Computing (SBACPAD)*, pages 35–42, 2012.

[107] B. Mills, R. E. Grant, K. B. Ferreira, and R. Riesen. Evaluating energy savings for checkpoint/restart. In *Proc. International Workshop on Energy Efficient Supercomputing (E2SC), with SC*, page 6, 2013.

[108] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *Proc. International Conference on Supercomputing (ICS)*, pages 35–44, 2002.

[109] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2010.

[110] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.

[111] P. Pop, K. H. Poulsen, V. Izosimov, and P. Eles. Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems. In *Proc. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 233–238, 2007.

[112] A. Rafiev, A. Iliasov, A. Romanovsky, A. Mokhov, F. Xia, and A. Yakovlev. Studying the interplay of concurrency, performance, energy and reliability with ArchOn – an architecture-open resource-driven cross-layer modelling framework. In *Proc. International Conference on Application of Concurrency to System Design (ACSD)*, pages 122–131, 2014.

[113] R. Rajachandrasekar, A. Venkatesh, K. Hamidouche, and D. K. Panda. Power-Check: An energy-efficient checkpointing framework for HPC clusters. In *Proc. International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 261–270, 2015.

[114] N. B. Rizvandi, J. Taheri, and A. Y. Zomaya. Some observations on optimal frequency selection in DVFS-based energy consumption minimization. *Journal of Parallel Distributed Computing*, 71(8):1154–1164, August 2011.

[115] B. Rountree, D. K. Lowenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: Making DVS practical for complex HPC applications. In *Proc. International Conference on Supercomputing (ICS)*, pages 460–469, 2009.

[116] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz. Bounding energy consumption in large-scale MPI programs. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–9, 2007.

[117] S. Rusu, S. Tam, H. Muljono, J. Stinson, D. Ayers, J. Chang, R. Varada, M. Ratta, S. Kottapalli, and S. Vora. A 45 nm 8-core enterprise xeon® processor. *IEEE Journal of Solid-State Circuits*, 45(1):7–14, January 2010.

[118] V. Sarkar, Ed. Exascale software study: Software challenges in extreme scale systems. Technical report, US DARPA IPTO, Air Force Research Labs, September 2009.

[119] S. M. Shatz and J.-P. Wang. Models and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems. *IEEE Transsactions on Reliability*, 38(1):16–27, April 1989.

[120] E. Solomonik, A. Bhatele, and J. Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 77, 2011.

[121] S. W. Son, K. Malkowski, G. Chen, M. Kandemir, and P. Raghavan. Reducing energy consumption of parallel sparse matrix applications through integrated link/CPU voltage scaling. *Journal of Supercomputing*, 41(3):179–213, September 2007.

[122] S. Song, C.-Y. Su, R. Ge, A. Vishnu, and K. W. Cameron. Iso-Energy-Efficiency: An approach to power-constrained parallel computation. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 128–139, 2011.

[123] S. Song, C.-Y. Su, B. Rountree, and K. W. Cameron. A simplified and accurate model of power-performance efficiency on emergent GPU architectures. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 673–686, 2013.

[124] R. Springer, D. K. Lowenthal, B. Rountree, and V. W. Freeh. Minimizing execution time in MPI programs on an energy-constrained, power-scalable cluster. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 230–238, 2006.

[125] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 253–264, 2009.

[126] © 2013 IEEE. Reprinted, with permission, from L. Tan, L. Chen, Z. Chen, Z. Zong, R. Ge, and D. Li. Improving performance and energy efficiency of matrix multiplication via pipeline broadcast. In *Proc. International Conference on Cluster Computing (CLUSTER)*, pages 1–5, 2013.

[127] © 2013 IEEE. Reprinted, with permission, from L. Tan, Z. Chen, Z. Zong, R. Ge, and D. Li. A2E: Adaptively aggressive energy efficient DVFS scheduling for data intensive applications. In *Proc. International Performance Computing and Communications Conference (IPCCC)*, pages 1–10, 2013.

[128] © 2014 IEEE. Reprinted, with permission, from L. Tan and Z. Chen. TX: Algorithmic energy saving for distributed dense matrix factorizations. In *Proc. International Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), with SC*, pages 23–30, 2014.

[129] © 2015 ACM. Reprinted, with permission, from L. Tan and Z. Chen. Slow down or halt: Saving the optimal energy for scalable HPC systems. In *Proc. International Conference on Performance Engineering (ICPE)*, pages 241–244, 2015.

[130] © 2015 ACM. Reprinted, with permission, from L. Tan, Z. Chen, and S. L. Song. Scalable energy efficiency with resilience for high performance computing systems: A quantitative methodology. *ACM Transactions on Architecture and Code Optimization*, 12(4):35, October 2015.

[131] © 2015 IEEE. Reprinted, with permission, from L. Tan, S. L. Song, P. Wu, Z. Chen, R. Ge, and D. J. Kerbyson. Investigating the interplay between energy efficiency and resilience in high performance computing. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 786–796, 2015.

[132] L. Tan, L. Chen, Z. Chen, Z. Zong, R. Ge, and D. Li. HP-DAEMON: High performance distributed adaptive energy-efficient matrix-multiplication. In *Proc. International Conference on Computational Science (ICCS)*, pages 599–613, 2014.

[133] L. Tan, M. Feng, and R. Gupta. Lightweight fault detection in parallelized programs. In *Proc. International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, 2013.

[134] L. Tan, S. R. Kothapalli, L. Chen, O. Hussaini, R. Bissiri, and Z. Chen. A survey of power and energy efficient techniques for high performance numerical linear algebra operations. *Parallel Computing*, 40(10):559–573, December 2014.

[135] Y. Taur, X. Liang, W. Wang, and H. Lu. A continuous, analytic drain-current model for DG MOSFETs. *IEEE Electron Device Letters*, 25(2):107–109, February 2004.

[136] A. H. T. Tse, D. B. Thomas, K. H. Tsoi, and W. Luk. Dynamic scheduling Monte-Carlo framework for multi-accelerator heterogeneous clusters. In *Proc. International Conference on Field Programmable Technology (FPT)*, pages 233–240, 2010.

[137] A. Varma, B. Ganesh, M. Sen, S. R. Choudhury, L. Srinivasan, and J. Bruce. A control-theoretic approach to dynamic voltage scheduling. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 255–266, 2003.

[138] D. M. Wadsworth and Z. Chen. Performance of MPI broadcast algorithms. In *Proc. IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing, with IPDPS*, pages 1–7, 2008.

[139] X. Wang, X. Fu, X. Liu, and Z. Gu. Power-aware CPU utilization control for distributed real-time systems. In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 233–242, 2009.

[140] Z. Wang. Reliability speedup: An effective metric for parallel application with checkpointing. In *Proc. International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT)*, pages 247–254, 2009.

[141] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, page 2, 1994.

[142] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-L. Lu. Reducing cache power with low-cost, multi-bit error-correcting codes. In *Proc. International Symposium on Computer Architecture (ISCA)*, pages 83–93, 2010.

[143] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu. Trading off cache capacity for reliability to enable low voltage operation. In *Proc. International Symposium on Computer Architecture (ISCA)*, pages 203–214, 2008.

[144] D. H. Woo and H.-H. S. Lee. Extending Amdahl's Law for energy-efficient computing in the many-core era. *Computer*, 41(12):24–31, December 2008.

[145] A. S. Wu, H. Yu, S. Jin, K.-C. Lin, and G. Schiavone. An incremental genetic algorithm approach to multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):824–834, September 2004.

[146] P. Wu and Z. Chen. FT-ScaLAPACK: Correcting soft errors on-line for ScaLAPACK Cholesky, QR, and LU factorization routines. In *Proc. International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, pages 49–60, 2014.

[147] X. Wu, V. Deshpande, and F. Mueller. ScalaBenchGen: Auto-generation of communication benchmarks traces. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1250–1260, 2012.

[148] Y. Yetim, S. Malik, and M. Martonosi. EPROF: An energy/performance/reliability optimization framework for streaming applications. In *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 769–774, 2012.

[149] J. W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, September 1974.

[150] L. Yu, D. Li, S. Mittal, and J. S. Vetter. Quantitatively modeling application resilience with the data vulnerability factor. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 695–706, 2014.

[151] Y. Zhang, K. Chakrabarty, and V. Swaminathan. Energy-aware fault tolerance in fixed-priority real-time embedded systems. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 209–213, 2003.

[152] Z. Zheng and Z. Lan. Reliability-aware scalability models for high performance computing. In *Proc. International Conference on Cluster Computing (CLUSTER)*, pages 1–9, 2009.

[153] D. Zhu and H. Aydin. Energy management for real-time embedded systems with reliability requirements. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 528–534, 2006.

[154] D. Zhu, R. Melhem, and D. Mossé. The effects of energy management on reliability in real-time embedded systems. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 35–40, 2004.

[155] Q. Zhu, J. Zhu, and G. Agrawal. Power-aware consolidation of scientific workflows in virtualized environments. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2010.