OPTIMIZATIONS FOR ENERGY EFFICIENCY WITHIN DISTRIBUTED MEMORY PROGRAMMING MODELS

A Dissertation Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Siddhartha Jana

December 2016

OPTIMIZATIONS FOR ENERGY EFFICIENCY WITHIN DISTRIBUTED MEMORY PROGRAMMING MODELS

Siddhartha Jana

APPROVED:

Dr. Edgar Gabriel,
Committee Chair, Associate Professor,
Computer Science, University of Houston

Dr. Barbara Chapman,
Research Advisor, Committee Co-chair, Professor,
AMS / IACS, Stony Brook University

Dr. Jaspal Subhlok,
Department Chair, Professor,
Computer Science, University of Houston

Dr. Weidong Shi,
Assistant Professor,
Computer Science, University of Houston

Dr. Oscar Hernandez,
R&D Research Staff,
CSMD, Oak Ridge National Laboratory, UT-Battelle

Dean, College of Natural Sciences and Mathematics

"... But I have Joules to keep,

And I have FLOPS to run before I sleep ..."

– Yours Truly,

A gross corruption of an excerpt from Robert Frost's

"Stopping by Woods on a Snowy Evening", 1992

Acknowledgements

Firstly, I would like to thank my research advisor Dr. Barbara Chapman, and my mentors - Tony Curtis, Deepak Eachempati, and Dounia Khaldi, for taking me under their wings, and providing me with ample guidance within my research group - HPCTools. I couldn't be more thankful of the opportunities I was presented to mingle with the HPC (High Performance Computing) community and present my progress in the form of talks, research posters, and webinars. The direct outcome of this visibility has led to multiple internship opportunities and successful publications with some of the top contributors to the community - Intel, Cray Inc., Techinische Universität Dresden, and Oak Ridge Associated Universities.

This work has been an outcome of multiple successful joint research ventures and I am very thankful for this. A major fraction of financial support has been provided by the Computer Science department at the University of Houston. Additional funding sources that I am grateful for include the US DOD (United States Department of Defense), LANL (Los Alamos National Laboratory), ORNL (Oak Ridge National Laboratory), and Total Oil&Gas.

One of the fundamental challenges of working in the field of High Performance Computing is the ability to provide empirical evidence of ones hypotheses on large scale distributed systems. I am obligated to the multiple organizations that have directly or indirectly, provided me access to state-of-the-art computational resources. These include - the OLCF (Oak Ridge Leadership Computing Facility), the HPC

center at ZIH (Zentrum für Informationsdienste und Hochleistungsrechnen), and the NSF (National Science Foundation).

I am obligated to my parents - Soumitra Kumar Jana and Kalpana Neogy Jana, for their love and support. Their never-ending patience during my term in the graduate school, has never ceased to amaze me.

OPTIMIZATIONS FOR ENERGY EFFICIENCY WITHIN DISTRIBUTED MEMORY PROGRAMMING MODELS

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Siddhartha Jana

December 2016

Abstract

With the breakdown of Dennard Scaling and Moore's law, power consumption appears to be a primary challenge on the pathway to exascale computing. Extreme Scale Research reports indicate the energy consumption during movement of data off-chip is orders of magnitude higher than within a chip. The direct outcome of this has been a rising concern about the energy and power consumption of large-scale applications that rely on various communication libraries and parallelism constructs for distributed computing. While innovative designs of hardware set the upper bounds for power consumption, there is a need for the software to adapt itself to achieve maximum efficiency at minimal joules.

This work presents detailed analyses of multiple factors within the software stack, that affect the energy consumption of large scale *distributed memory* HPC applications and programming environments. As part of this empirical analyses, we isolate multiple constraints imposed by the communication, memory, and the execution model that affect energy profiles of such applications. With regards to the communication model, empirical analyses in this thesis reveals significant impact due to constraints like the size of the data payload being transferred, the number of data fragments, the overhead of memory management, the use of additional OS threads, as well as the hardware design of the underlying processor. Additional software design characteristics that have been shown to have a significant impact on communication-intensive kernels include – the design of remote data-access patterns (greater than 40% energy savings), the transport layer protocols (25X improvement in bytes/joules) as well as the choice of the interconnect (760X improvement in bytes/joules).

This dissertation also revisits a two-decade-old programming paradigm - Active

Messages, and presents empirical evidence that suggests that integrating it within current SPMD execution models leads to significant performance and energy efficiency.

It is hoped that the work presented in this literature paves the way for taking software design into consideration while designing current and future large-scale energy-efficient systems operating within a power budget.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

This past decade has seen the end of two major computing laws. The first of these, the *Dennard Scaling* suggested an exponential rise in performance per watt of microprocessors with silicon transistors. Its breakdown was brought about by an increase in power leakage within circuits that led to the inability of continuing to boosting CPU performance by increasing the operating frequency. This led to a design switch by hardware vendors with the architecture; hardware parallelism was introduced. Almost a decade has passed since then and as of this writing, the second law has undergone a similar fate. The *Moore's law*, predicted an biannual exponential rise in growth of transistors that can be packed into a silicon chip. Two factors that have contributed to the slowdown of this law - the limits imposed by the laws of Physics and the high power-density of the silicon transistors.

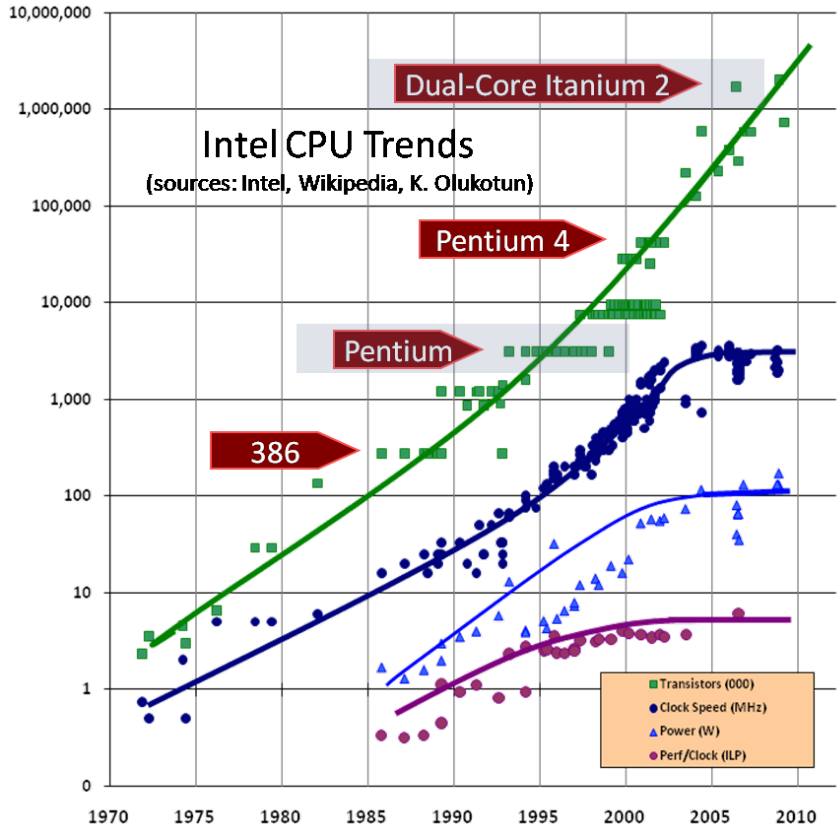


Figure 1.1: The history of Intel chip introductions by clock speed and number of transistors (1970-2010). [Source: Blog article "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software", URL: http://www.gotw.ca/publications/concurrency-ddj.htm]

It is in this backdrop of architectural transition that the field of High Performance Computing experienced a transition from the Terascale era (2^{13} Flop/s) to the Petascale era (2^{15} Flop/s). From this point, the HPC community has set a bold time-frame of arriving at the Exascale era (2^{18} Flops/s) by 2025. A challenge boldly accepted by multiple national governments including the White House[145] The challenge? Achieving this with a tight power budget of 20-25MW!

Given the urgent need for innovative (yet easy to code!) designs for the hardware

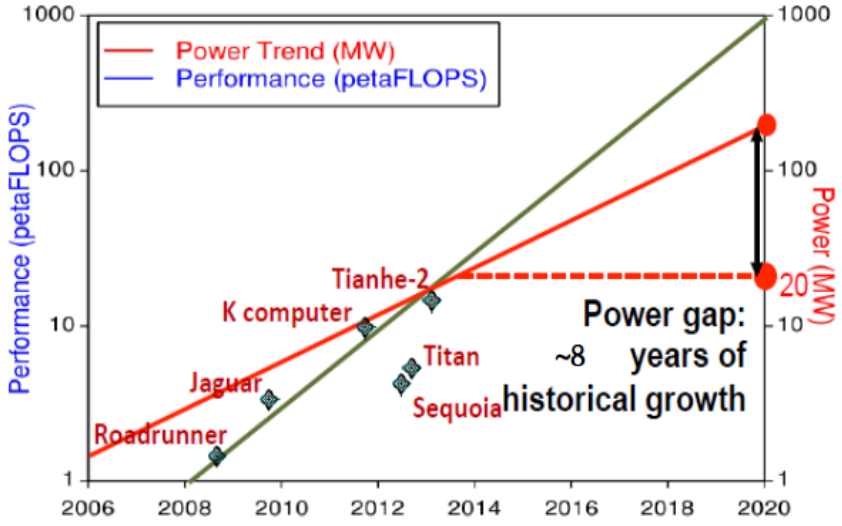


Figure 1.2: Top 500 trends - Performance and Energy efficiency [Source: Energy-aware workshop, SuperComputing Conference, November 2015]

and software stack, the NSF-SRC has put forth a solicitation calling for "breakthroughs" in the field of energy efficient computing.

Some of the excerpts from the program synopsis[120] reads,

> "There is a consensus across the many industries touched by our ubiquitous computer infrastructure that future performance improvements across the board are now severely limited by the amount of energy it takes to manipulate, store, and critically, transport data ..."

and,

> "Truly disruptive breakthroughs are now required, and not just from any one segment of the technology stack. Rather, due to the complexity of the challenges, revolutionary new approaches are needed at each level in the hierarchy. Furthermore, simultaneous co-optimization across all levels is essential for the creation of new, sustainable computing platforms ..."

1.2 Motivation

Recent studies on the challenges facing the Exascale era, express a need for understanding the various factors that affect the energy profiles of applications scaling multiple nodes. It is well established that hardware innovations set the upper bound on the energy efficiency achievable. However, it is the design of the software stack that dictates the degree to which an application can reach this bound, within a system.

This work highlights the notion that the energy cost factors can be mapped to multiple layers across the hardware and software stack. Adopting distributed memory programming models allows applications to run across multiple compute nodes. In order to study the impact of porting such applications to a typical SPMD programming environment, it becomes crucial to explore the energy costs associated with managing the consistency of the distributed memory.

Data movement may be either local within a single processing node, or, it may be external among multiple nodes. The external data transfer takes the form of communication among one or more compute or storage servers. The use of data transfers between processes running in a distributed environment is tightly dependent on the programming model used to design an HPC application. For example, in case of PGAS models like OpenSHMEM, the data movement corresponds to explicit interfaces provided by its communication model. While the transfer routines play an integral role in implementing the algorithm of a distributed application, the synchronizing constructs help ensure memory consistency between different phases of an application. The amount of energy consumed during such data movement poses

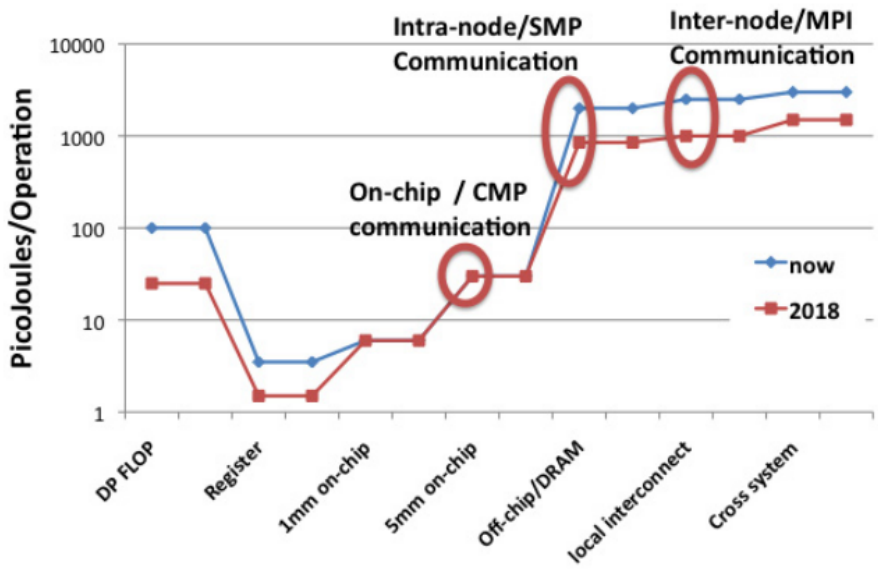


Figure 1.3: Data transfer trends (Source: "Exascale Computing Technology Challenges", John Shalf, Sudip Dosanjh, and John Morrison[139])

a serious threat to the usability of distributed memory models on future systems.

1.3 Research Statement

The research goals are as follows:

- Identify different design factors within the software stack that have the potential of affecting the energy consumption of a distributed memory application
- Study the impact of DVFS on the communication and synchronization model
- Identify changes within standard bulk synchronous execution model that stand to benefit the energy efficiency of HPC applications
- Design of a prototype framework that implements the said changes

- Design of applications and microbenchmarks and perform empirical analyses to evaluate the extent of impact of various code optimizations.

1.4 Scope of the Study

As discussed before, the overarching goal of this thesis is to explore different factors within the software stack that have the potential of affecting the energy consumption of HPC applications that employ distributed memory programming models. The different components of this stack are summarized in Figure 1.4. The components highlighted are discussed in detail in this text. As shown, three main components define a distributed memory programming model:

- The Communication Model: This describes the behavior of explicitly initiated data transfers across distributed memory. This dictates the design of different communication patterns at the application layer. At a lower level, this also affects the design of the middleware that enables the actual data transfer across the interconnect.

- The Memory Model: This describes the consistency model followed in order to maintain a coherent view of the distributed memory among multiple processes. This gives the programmer a set of synchronizing constructs that ensure the correctness of the outcome of parallel execution of instruction paths.

- The Execution Model: This describes the mapping of the actual algorithm of an application to the underlying machine. At a lower level, factors like the actual types of instructions used within the computation kernel come into play.

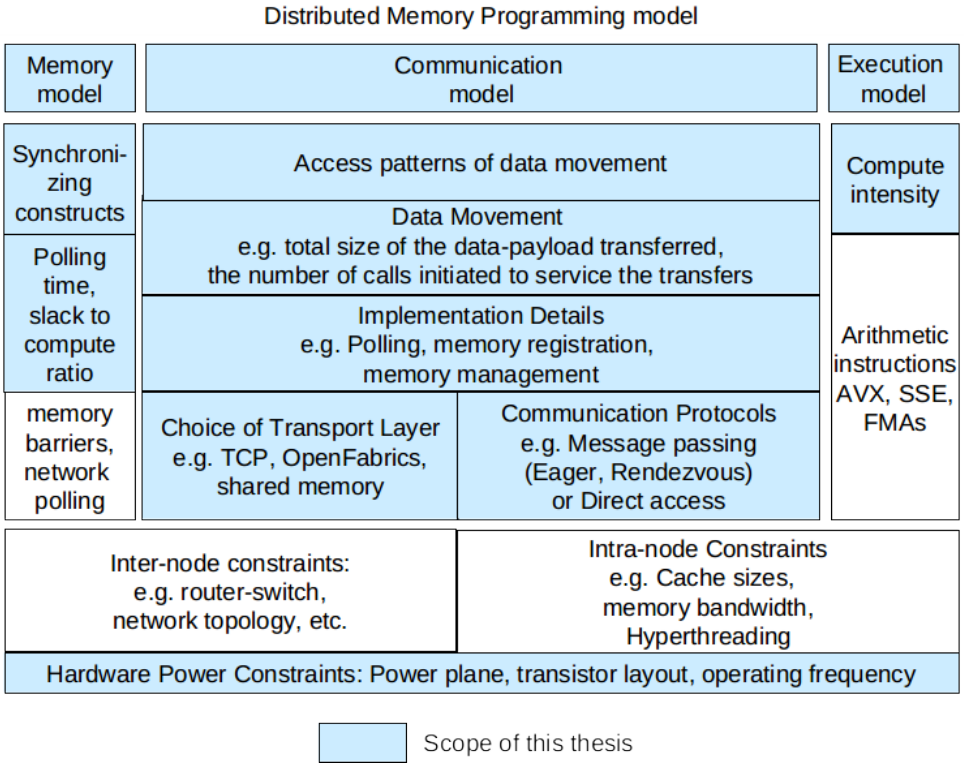


Figure 1.4: The software stack for a distributed memory programming model. The shaded region highlights the scope of this thesis.

1.5 Thesis Contribution

Three parts: computation, communication, memory

This work explores the impact of explicitly initiated communication and synchronization operations on the energy consumption within OpenSHMEM applications.

- Case Study:
  - Analysis of real-world applications to identify shortcomings in distributed memory programming models that lead to poor CPU utilization and in turn, poor energy efficiency.

- Translation of a CORAL petascale application benchmark - LSMS (Locally Self-consistent Multiple Scattering) from MPI the de facto HPC programming model to OpenSHMEM, a popular PGAS programming model.
- Exploring energy saving opportunities give the communication and memory model used by CORAL application benchmarks.
- Description of a detailed empirical study that highlights the various design decisions within distributed memory programming models that effect energy consumption.

- Communication Model: With regards to the communication model, this work strives to explore the different factors that effect the energy costs of a communication intensive application. This is complemented with empirical evidence of the impact on energy savings due to proposed optimizations.
  - Description of a number of factors characterizing individual data transfers that have the potential of impacting the energy signatures of PGAS applications
  - Empirical evidence motivating the transformation of data-access patterns in order to achieve energy efficiency of communication-intensive application kernels. This is presented in terms of the reduction in CPU energy consumption, DRAM energy consumption, communication latency, and the Energy Delay Product (or EDP).
  - Empirical evidence of the feasibility of adopting techniques for fine tuning not only the performance but also the energy efficiency of applications.

- Memory Model: With regards to the memory model, this work aims at exploring opportunities to reduce the energy costs associated with non-uniform work loads.
  - Description of the impact of synchronization constructs on the energy consumption of processes in case of non-uniform work loads.
  - A design and implementation of framework for dynamic management of energy consumption of unbalanced work loads.
- Execution Model: With regards to the execution model, this work aims at exploring opportunities that map computational units to the underlying distributed memory.
  - Identification of shortcomings in current de facto programming models that lead to poor CPU utilization.
  - Revisiting the potential of Active Messages as a solution to increase CPU utilization, and indirectly, energy efficiency.
  - Design of a prototype that integrates the execution model of Active Messages within the OpenSHMEM reference implementation.
  - Design of multiple versions of graph-based applications (like Traveling Salesman Problem, Minimum Spanning Tree) using MPI, standard OpenSHMEM and the proposed Active Message execution models.
  - Empirical analysis highlighting better CPU utilization using Active Messages as compared to standard OpenSHMEM and MPI.

1.6 Chapter-wise Layout

This section describes the layout of this literature. There have been voluminous amount of research efforts to manage energy efficiency across the hardware and software stack. An overview of this is presented in Chapter 3.

We group them based on the software stack layer they correspond to. The layout of all the empirical study is depicted in Figure 2.2. One of the common misconceptions in the community is that the best approach of achieving energy efficiency is to execute applications as fast as possible. Chapter 4 presents empirical evidence against this commonly-held computing approach called, "Race-to-Halt".

The following chapters present a detailed study about different design parameters across the hardware and software stack. An overview of these factors are presented in Chapter 5. These factors are mapped to either the communication or the memory model imposed by the programming models. A profiling-based study of a petascale application is discussed that lists various opportunities of mapping the observations made in past chapters in order to achieve energy savings among its phases.

With respect to the application-initiated remote data transfers, Chapter 6 discusses the costs associated with message sizes and the count of message fragments. In Chapter 7, we present empirical results indicating that the design of the transport layer and the choice of the interconnect solutions, both have a significant impact on the number of joules consumed by a system for every byte transferred. In Chapter 8, we extend the study to highlight the notion that the energy profiles of the application are significantly altered based on the communication patterns used while designing

a distributed algorithm. Chapter 9 describes how energy costs rises in proportion to the number of CPU cycles invested by a process polling at a synchronizing costs. Another factor like the number of processes participating in the synchronizing point is also discussed. Chapter 10 introduces the State-of-the-Art approach of using DVFS or Dynamic Voltage Frequency Scaling approaches to achieve energy and power savings. The following chapters highlight some of the challenges with merging DVFS techniques with current programming models and architectures. Issues with using DVFS for eliminating slack is discussed in Chapter 11. Unintended consequences of performing inter-process communication on a hardware subjected to DVFS is discussed in Chapter 12.

One of the fundamental takeaway from this work is that there is a need for exploring alternative programming models that are better amenable to runtime approaches of achieving energy savings. As a solution, this work brings back the concept of Active Messages into modern programming models like OpenSHEM. A prototype implementation along with empirical results highlight the usability of this execution model. All of these are summarized in Chapter 13.

A summary, takeaway messages, concluding remarks, and proposed future work are discussed in Chapters 14 and 15.

Details like the experimental setup, microbenchmark designs, and energy measurement approaches are discussed as auxiliary notes in the Appendices towards the end of the book. The final segment of the book is devoted to the list of bibliography that was referenced throughout this work.

Chapter 2

Guide to Terminology and Plots

This section describes a few essential terms used throughout the text.

2.1 Power Versus Energy of a Data Transfer

In this section, we hope to establish the difference between optimizing for energy versus power, with respect to data transfers. It must be noted that one doesnt always have to sacrifice the lowest possible latency to achieve energy efficiency. Consider the plots shown in Figure 2.1.

The plots (a) and (b) depict the average power consumed by CPU cores (Y-axis) and the corresponding latency (X-axis) incurred while transferring a 32KB payload across the network (MPI Send-Recv over InfiniBand). If this payload is divided into 64 fragments, the energy consumption by the CPU cores is about 6 mJ and the transfer takes about 370 μs to complete. The average power consumption during this transfer is about 16.21 Watts (Figure 2.1a). If instead, we chose to split this

(a) Using 64 fragments to transfer data. Energy=6mJ

(b) Using 2 fragments to transfer data. Energy=0.33mJ

Figure 2.1: Power Versus Latency. Use of a 32KB data payload transferred using MPI_Send() over InfiniBand

payload into only 2 fragments (16KB each), the energy consumption drops to 0.33 mJ (by 94.5%) and latency to 20 μs (by 94.6%). However, this comes at the cost of a rise in power consumption to 16.565W, i.e. an increment by 1.8% (Figure 2.1b). Thus, despite the higher power consumption, choosing the latter option enables the CPU cores to service the transfer using lesser energy.

2.2 Interpreting Colored Plots

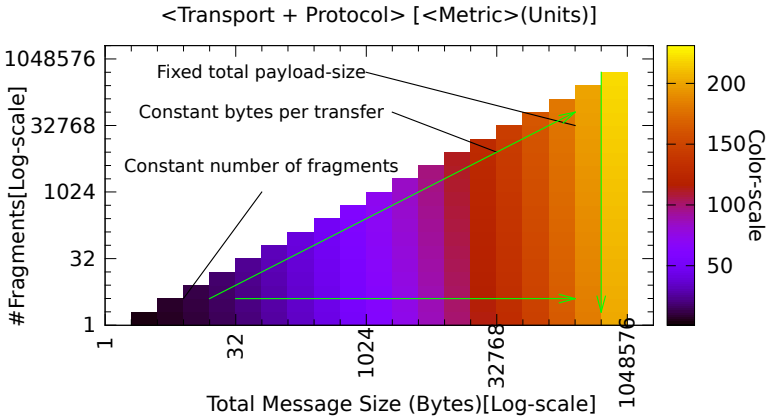


Figure 2.2: Layout of the plots in this work

We briefly discuss the method of interpreting the plots presented in the following

sections. Each plot illustrates empirical results in terms of an energy metric. It corresponds to a specific transport layer and a communication protocol.

The coordinate axes (log-scale) correspond to two controllable factors that define a communication phase in an application - the total size of data transferred during that phase (X-axis) and the number of explicit MPI-calls (Y-axis) used to transfer that payload. Throughout this text, we refer to the latter as the count of *fragments*[1]. The shade of a point in this coordinate space indicates the value of monitored metric that is represented by the color-scale to the right of each plot.

[1]It must be noted that each fragment may further be divided into smaller chunks by the underlying layers based on the middleware design, NIC hardware constraints, etc.

Chapter 3

Related Work

There has been a great deal of research directed towards measuring and managing the energy and power consumption of applications. Proposals like Thrifty[150] have been put forth to direct large-scale research towards redesigning the complete computing stack. The goal of such efforts is directed towards building power-aware Exascale platforms. The interested reader is directed to the survey report by Benedict[24] that provides a detailed taxonomy of power and energy measurement techniques. These describe the current state of the art in terms of energy saving solutions that either hardware-based, software-based, or a combination of both.

3.1 Hardware-controlled Power Management

3.1.1 Processing Units

Some of the model-based techniques provided by chip manufacturers to dynamically monitor and manage the power or energy consumption include: Intel's RAPL[79],

AMD's APM module[10], NVIDIA's NVML[121].

3.1.2 Interconnect Solutions

Hoefler[67] mentions discussions by the IEEE standard on energy efficient Ethernet specifications including - dynamic link-speed reduction, receiver modification, network routing, and deep sleep states. However, initial research indicates latencies and network jitter with these techniques.

3.1.3 Dynamic Voltage Scaling Techniques

Dynamically varying the operational voltage and frequency of a processor is commonly used as a technique for reducing the power and energy consumption trends of applications[47, 71, 114]. The notion that energy efficiency can always be achieved by sacrificing performance has been been challenged by Miyoshi et al.[114].

The authors study power/energy trends of applications by establishing the difference between energy consumed when a system is idle and when it is active. Although the former depends on architecture and a processor's operating frequency, the latter is additionally dependent on the actual load of the task. The authors put forth a metric - the Critical Power Slope. This metric, a function of power at idle mode, the minimum operational frequency and the power at active load at that frequency, is a deterministic factor to whether it is energy efficient to execute a load at a lower frequency or not. Similar results can also be drawn from the roofline model (above). Intel's RAPL(as described) also provides interfaces to enforce power capping by allowing the user to provide the hardware with power consumption limits[79] to limit energy consumption at the cost of performance.

3.1.4 CPU Gating

Orthogonal to DVFS, Leverich et al.[102] make a case for per-core power gating or PCPG. The technique aims at selectively switching off cores within a die to reduce power consumption (or power leakage) even when cores are idle. Empirical data indicate savings over a wide range 3% to 64%

3.2 Software-controlled power management

3.2.1 Compiler-driven

Static analysis to aid DVFS-based techniques Since the impact of DVFS schemes on the efficiency of an application is heavily dependent on the design of the application, many efforts are directed towards exploiting static analysis tools like compilers, to determine the feasibility[3, 71]. One of the domains where this approach is helpful is in case of applications designed for real-time or time-sensitive tasks. While working with such applications, the user has the advantage of knowing the worst case execution time of such applications. This is because of the strict time deadlines that the applications are expected to adhere to. A compiler can thus exploit any slack due to the difference between the time taken to execute the compute load and the time take to meet the deadline.

Using code-optimization techniques One of the ways to control energy costs is to reduce the count of execution cycles by the application. Such optimizations typically target loops and arrays. Rahman et al. show that code transformations like loop parallelization using OpenMP, loop blocking, loop unroll-and-jam, array

copying+strength reduction, scalar replacement+strength reduction, loop unrolling all have the potential of driving energy savings. To significantly reduce power consumption,a scientific application may benefit from using fewer number of threads and then fine-tuning the cache-blocking and loop unrolling factors to ensure that both the CPUs and the memory hierarchy are being used in an efficient manner.

Issues: However, the time and energy efficiency for a given optimizations are often not in correlation with each other. At such times, the user is burdened with the responsibility of assigning priorities to similar static-guided frameworks in order to compensate for the conflict. Dependence on performance counters like L1-cache misses makes the model-predictions dependent on the architecture. In other words, the compiler-flags supported by the model might not be applicable for the same application running on different systems. Also, work on roofline model for energy suggests that the power consumption depends on the compute intensity of the application (ratio of the count of arithmetic to memory access instructions). Bellosa[22] arrives at similar conclusions while evaluating the power consumption of CPUs with variable frequency/voltage. His findings indicate that the optimal configuration of these parameters is only possible only if the memory reference characteristics are taken into consideration.

Exploiting architectural design of microprocessors There have been past research efforts analyzing the impact of instruction scheduling on the energy consumption of applications. If a compiler is aware of the energy cost of switching activity among instruction operands within a processor, a scheduling algorithm may

be designed to order the instruction to reduce the energy consumption during instruction execution. This leads to significant power savings[142, 100]. Similarly, this may be extended to register allocation algorithms[15, 33, 54].

3.2.2 Operating Systems

The operating system analyzes the active and idle times of a device (e.g., CPU, hard disk or display) and makes assumptions about the future use of the device. The disadvantage is that an application may behave differently during different phases and its performance and power usage are not predicted correctly. Advanced Configuration and Power Interface (ACPI) in operating systems like Windows2000 use such power management schemes[112]. In [22], the authors introduce an OS-based power management scheme called Joule-watchers that throttles low priority thread to maintain the average power consumption below a threshold. This is a solution that is dependent on the OS being aware of the pre-determined priority of threads. Moreover, the authors found that excessive throttling might lead to more energy consumption due to an increase in cache misses. Such an execution-environment does not map directly to an HPC system where the applications are usually executed on dedicated nodes with threads of equal priority with OS-oblivious energy signatures.

3.2.3 DVFS Based Efforts

Past efforts towards understanding and managing the power consumption trends of applications have been significant. One of the static-based approaches for managing power consumption by processes is for the compiler to evaluate a program and determine sections within the code where the energy consumption profile changes.

This knowledge in the form of *power management hints* can then be conveyed to the runtime to adjust the voltage/frequency scaling of applications[3]. Korthikanti and Agha[95] study the power consumption behavior of shared memory architectures while handling applications with different problem sizes. Li et al.[104] use DCT and DVFS techniques to study the opportunities of reducing power consumption of hybrid MPI-OpenMP applications.

There has been a great deal of research in managing the energy consumption of applications. Most of these efforts target energy-based optimizations for applications running in a shared memory environment. The maximum impact on the energy savings in such platforms are governed by the avoidance of penalty due to cache misses and memory-intensive operations. For example, Rahman et al.[127] propose reducing power consumption in scientific applications by decreasing the number of active threads and fine-tuning cache blocking and loop unrolling factors to achieve efficient execution. Research efforts show that power bottlenecks are common in case of "disagreements" between the application activity and the system power consumption and quite often the source of inefficiency can be tracked down to the use of power-hungry busy-waits[5, 6, 30].

Barreda et al.[18] discuss work on a Framework for a posteriori detection of power-sinks in the form of discrepancies between the application activity and the CPU C-states. Choi et al.[40] explore opportunities of using DVFS in case of memory intensive phases of applications. Their approach relies on prediction of this intensity by dynamically measuring the ratio of off-chip versus on-chip accesses.

The work closest to our focus are those by Kandemir et al.[88], Vishnu et al.[153],

and Venkatesh et al.[151]. Kandemir et al.[88] discuss static-based techniques like traditional data flow analysis and polyhedral algebra to detect redundant communications and unwanted synchronizations in HPF-like languages. Vishnu et al.[153] exploit voltage frequency scaling and interrupt-based methods to achieve energy savings during remote memory operations. They implement this technique in ARMCI[118]. The energy savings discussed in this work only target individual data transfer operations. Venkatesh et al.[151] discuss techniques of energy measurement of MPI-based data transfers using Intel's RAPL scheme. Energy readings of point-to-point and collective operations are discussed. However, these efforts do not take into account the impact of multiple factors across the hardware and software stack. As we discuss in this work, the cost of an independent data transfer construct is dependent on its semantics and the data-access pattern it participates in. The following sections discuss a number of similar factors and present analysis of empirical results that are significantly impacted by them.

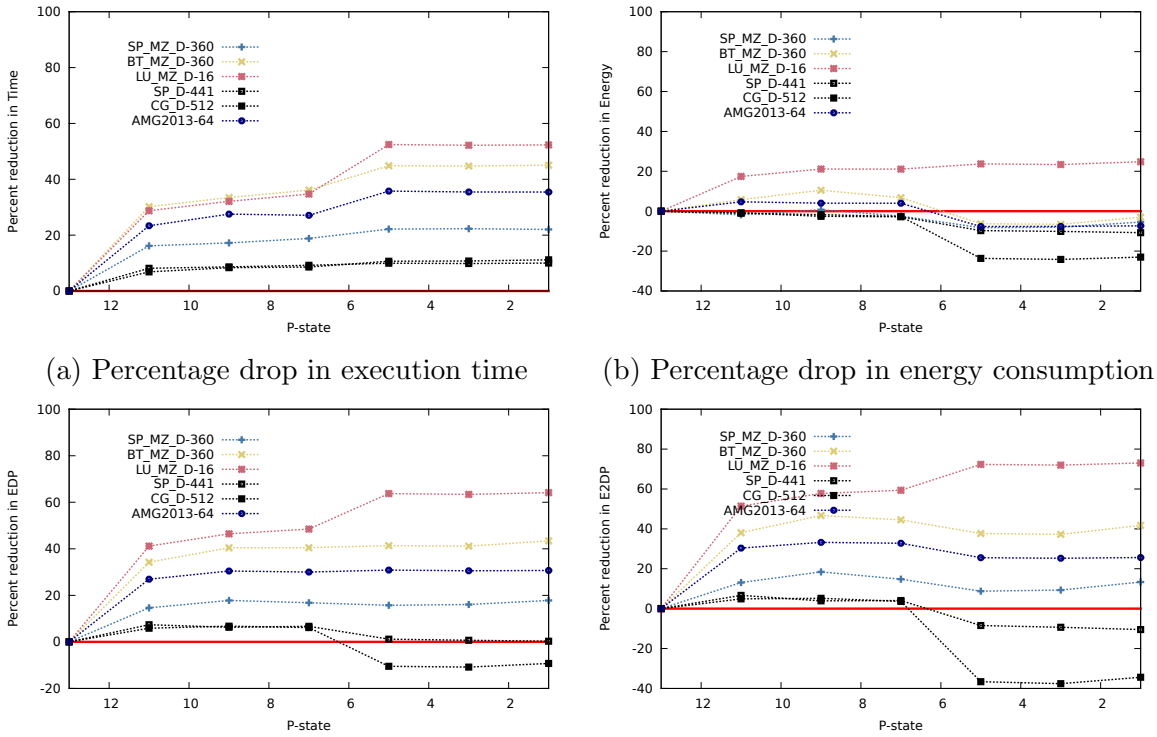
Chapter 4

Debunking the 'Race-to-Halt' Approach

The faster you run your code, more energy efficient it gets

4.1 Frequency scaling at job level

To test whether this claim holds true, a number of benchmarks from the NAS Parallel Test suite were selected and executed across multiple compute nodes all operating at a same core frequency. The CPU frequency chosen was the lowest operating point supported by the model. The execution time and total energy consumed by all the nodes (sockets and memory combined) were monitored. The same experiment was repeated with multiple higher frequencies including the highest operating point (also called, the non-turbo base frequency) supported by the model. Figure 4.1 depicts these results. As shown, the y-axes of the plots correspond to a percentage drop in



(a) Percentage drop in execution time

(b) Percentage drop in energy consumption

(c) Percentage drop in EDP

(d) Percentage drop in E2DP

Figure 4.1: Percentage drop in efficiency metric with respect to lowest operating frequency

each of the 4 efficiency metrics - execution time, total energy consumption, EDP, and E^2DP, with respect to the lowest operating frequency.

From the plots, we observe that almost all applications reach their peak performance by 2.2GHz. The plots corresponding to time savings reaches a plateau beyond this point. Application benchmarks like like LU-MZ, BT-MZ, AMG-2013, and SP-MZ attain greater than 20% reduction in execution time on increasing the frequency from 1.2GHz to 2.6GHz (more than 2X). The time savings for other benchmarks like 3D-FFT, CG, and SP is not greater 10% and are relatively less affected. In

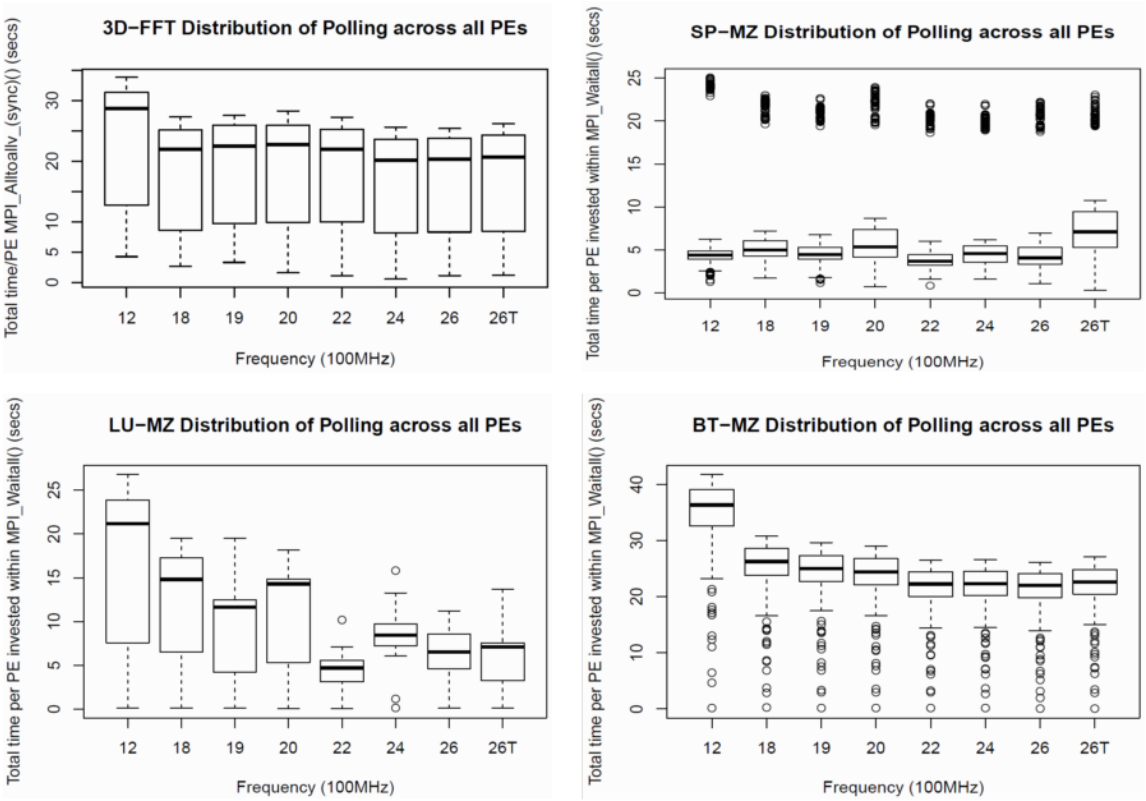


Figure 4.2: A box plot (or whisker diagram) depicting the distribution of total polling time across all processes operating at varying CPU frequencies. While in statistic analysis, the small circles represent outliers in a data set, the presence of these points in the data above suggest high variation in CPU time spent within polling-based constructs.

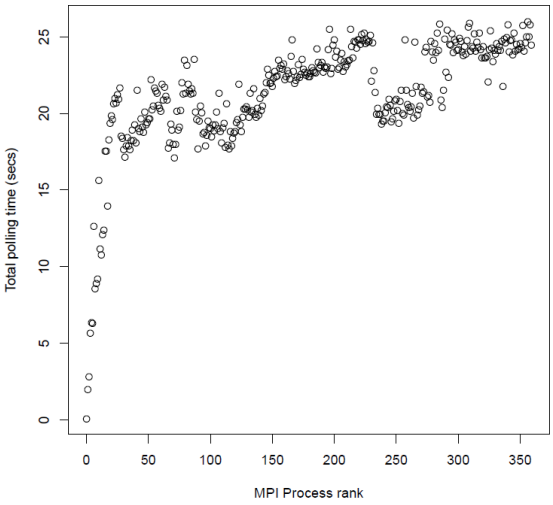
terms of energy savings, we observe that most applications have a negative trend indicating an *increase* in the energy consumption. The net effect on energy efficiency, captured by the metric EDP and E^2DP, clearly indicates that applications that do not gain much in performance due to CPU scaling have the potential of facing a significant drop in energy efficiency. These results show that the *race-to-halt* strategy is not always the best approach for attaining maximum energy efficiency in HPC Applications.

CPU cycles spent polling for network events are obvious energy sinks and contribute to a drop in the utilization of system in terms of time and energy budget. Based on the observations from Figure 4.1, it is evident that beyond a certain operating point, there is no significant improvement in time and energy utilization. In order to understand whether this lack of improvement in efficiency can be attributed to the time spent within slack periods, the time spent within slack periods for a subset of these benchmarks have been plotted in Figure 4.2. Due to the irregular behavior of these applications, it was observed that there is a high variation in the time spent by all the processes. Therefore, the metric has been represented as box plots; one for each operating frequency at which an application was executed.

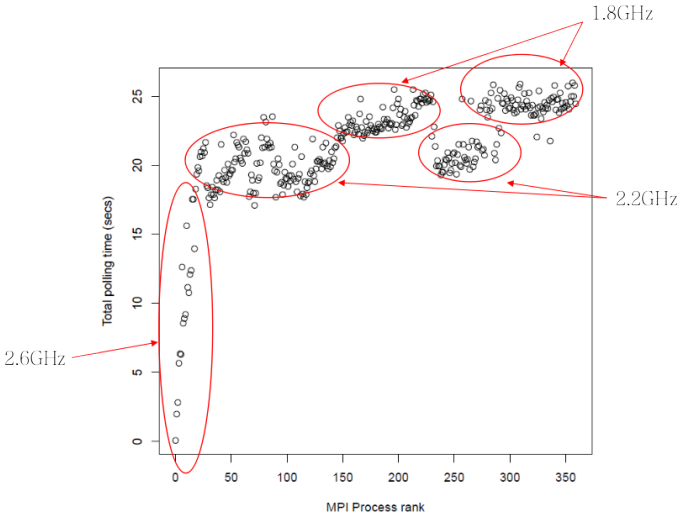
In Figure 4.2, we observe that an increase in the operating frequency does not always lead to a drop in the execution time. For example in case of BT-MZ, an increase in the frequency beyond 2.2GHz has very negligible effect on the value of the median slack time. This behavior is in alignment with those depicted in Figure 4.1.

4.2 Frequency scaling at process level

Commonly used job schedulers like SLURM and ALPS provide the capability of changing the frequency while launching a job step. However, the current design allows the assignment of a single user-specified P-state to all the processes participating in that job-step. To the best of our knowledge, there has been no empirical evidence justifying a need for operating subset of processes at different frequencies than the rest participating in the same job-step.



(a) Varying polling-time within MPI_Waitall



(b) Assignment of CPU frequencies to clusters of processes

Figure 4.3: Distribution of time spent polling within MPI_Waitall by every MPI rank during the parallel execution of the BT-MZ benchmark with 360 processes

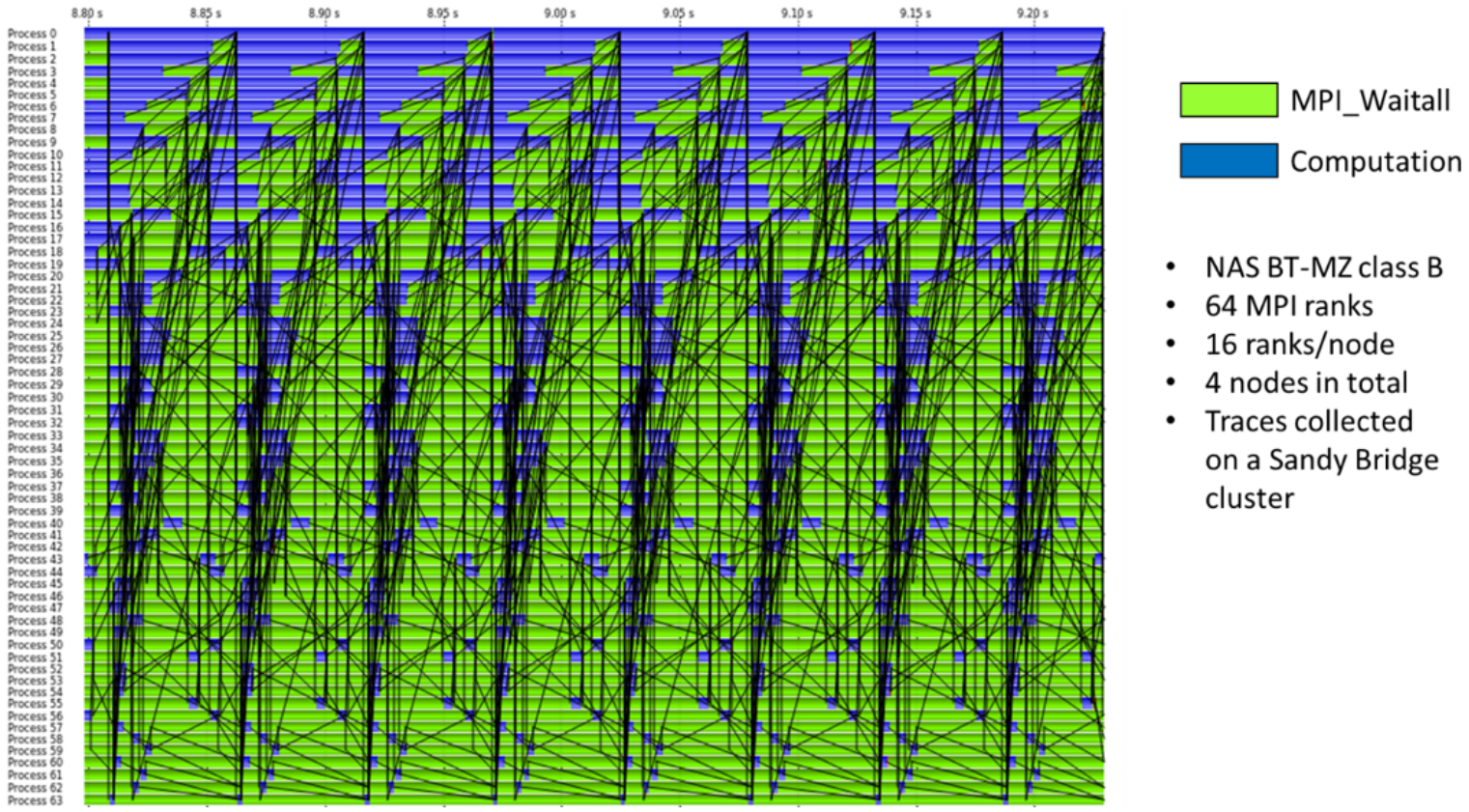


Figure 4.4: Snapshot of Vampir Trace of the profile trace of the MPI version of BT-MZ benchmark

This indicates a missed opportunity for applications with irregular load distribution across processes. As a case study, we inspect the Block Tridiagonal Multizone Benchmark (BT-MZ). Figure 4.4 shows a snapshot of a Vampir trace generated by profiling a parallel run of the benchmark, across 64 processes. The blue boxes correspond to compute-intensive phases and the green ones are map to network polling as part of the function MPI_Waitall. We observe that there is a non-uniform distribution of the polling time across the processes. This variation in the polling time is plotted in Figure 4.3a using data collected by the parallel execution of the same benchmark with 360 processes.

The research question at this point is whether there is a way to reduce the energy consumption of the application by reducing the time spent within the "polling-phase". Another question is whether it is possible to achieve energy savings without significant drop in execution time. To answer these, a visual inspection of the plot in Figure 4.3a shows that the 360 data points can be grouped into multiple clusters based on the relative polling time. One naive approach of tackling this question is to alter the operating frequency of all the processes with the expectation that varying the number of cycles spent within the polling and computation phases may affect the energy savings. This gives us three possible frequency setting modes:

- *Standard Setting:* Operate all MPI ranks at the highest frequency (also called the base frequency). This is the default setting in most HPC machines. For this workload, this should enable a reduction in execution time of processes with high computational workload.

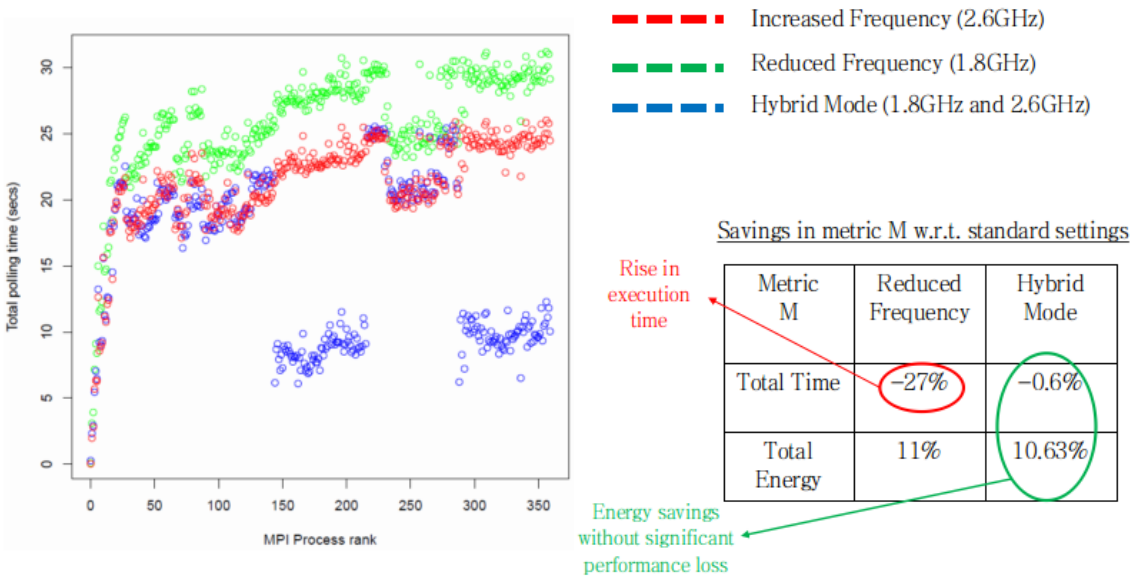


Figure 4.5: Polling time for all processes with different frequency operating modes

- *Reduced Frequency:* Operate all MPI ranks at a low frequency. This enables a reduction in the number of cycles spent polling by a give process.

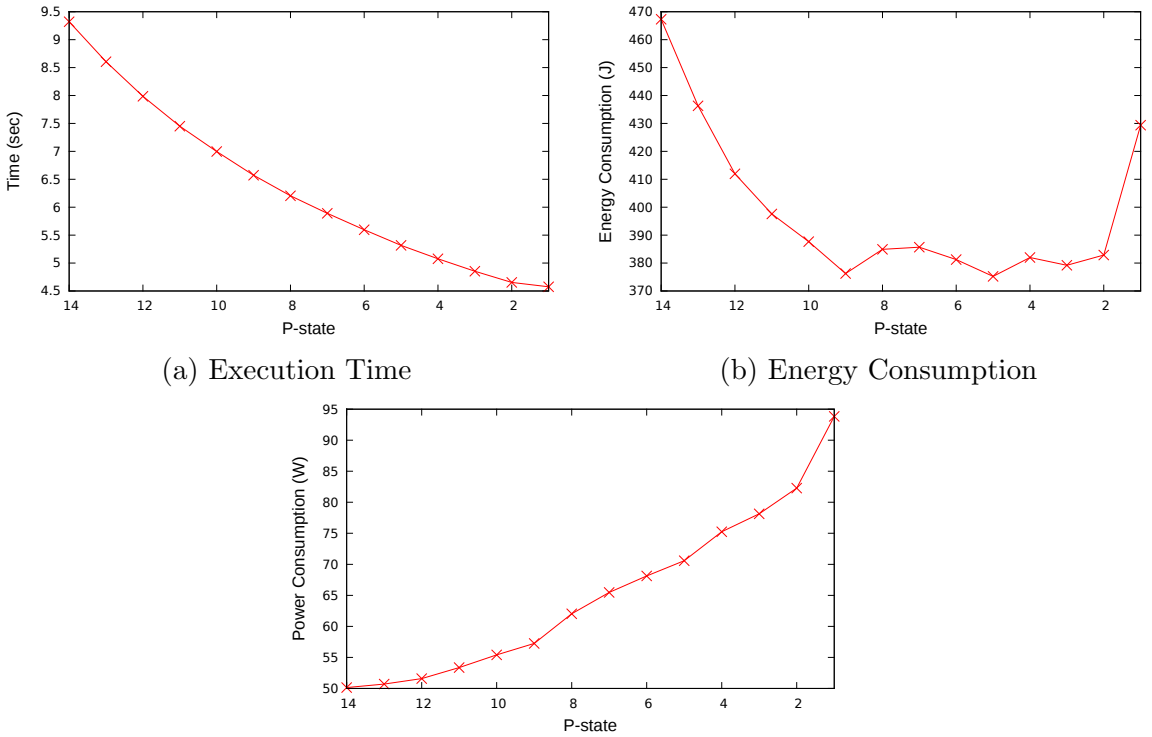
- *Hybrid Mode:* Operate specific MPI ranks with different frequencies. This enables a reduction of the operating frequencies of those MPI ranks that spend more time polling and likewise increase the frequency for those that spend less time in comparison. The rationale behind this hybrid approach is that processes with higher polling time correspond to code paths that have a lesser computation load and hence wait longer at synchronizing points. Similarly, processes with lesser polling time have higher compute workload. This can be observed from Figure 4.4. The assignment of CPU frequencies to different cluster of MPI ranks is illustrated in Figure 4.3b.

Figure 4.5 depicts the time spent within the polling phase for all the MPI processes, when executed using each of the operating modes listed above. We see that using the Hybrid approach (blue dots) lead to significant drop in polling-time for many processes. The table in the figure lists the reduction in execution time and energy consumption for the "Reduce Frequency" and the "Hybrid" modes, with respect to the "Standard Setting". We see that using that using the Hybrid mode, i.e., selectively choosing different operating frequencies for different processes helps achieve high energy savings (up to 10.6% in this case) with negligible losses in execution time (about 0.6% in this case). On the other hand, setting on the processes to the a single fixed frequency might lead to significant degradation of execution time (up to 27% with the "Reduced Frequency" mode).

4.3 Frequency scaling at phase level

For a give application phase, energy efficient execution is not always guaranteed by running the application at the highest possible CPU frequency. This section provides empirical evidence that highlights this claim. As discussed before, in order to reduce the energy consumption, the most common approach is to reduce the number of CPU cycles that are underutilized by an application. This can typically be mapped to a pipeline stall cycles. This is true for memory intensive applications.

Figures 4.6 and 4.7 show the impact of scaling the frequency of CPU cores servicing compute intensive and a memory intensive kernels respectively. The X-axis plots the P-state corresponding to the operating frequency and the Y-axis plots the execution time, energy consumption, and average power consumption.



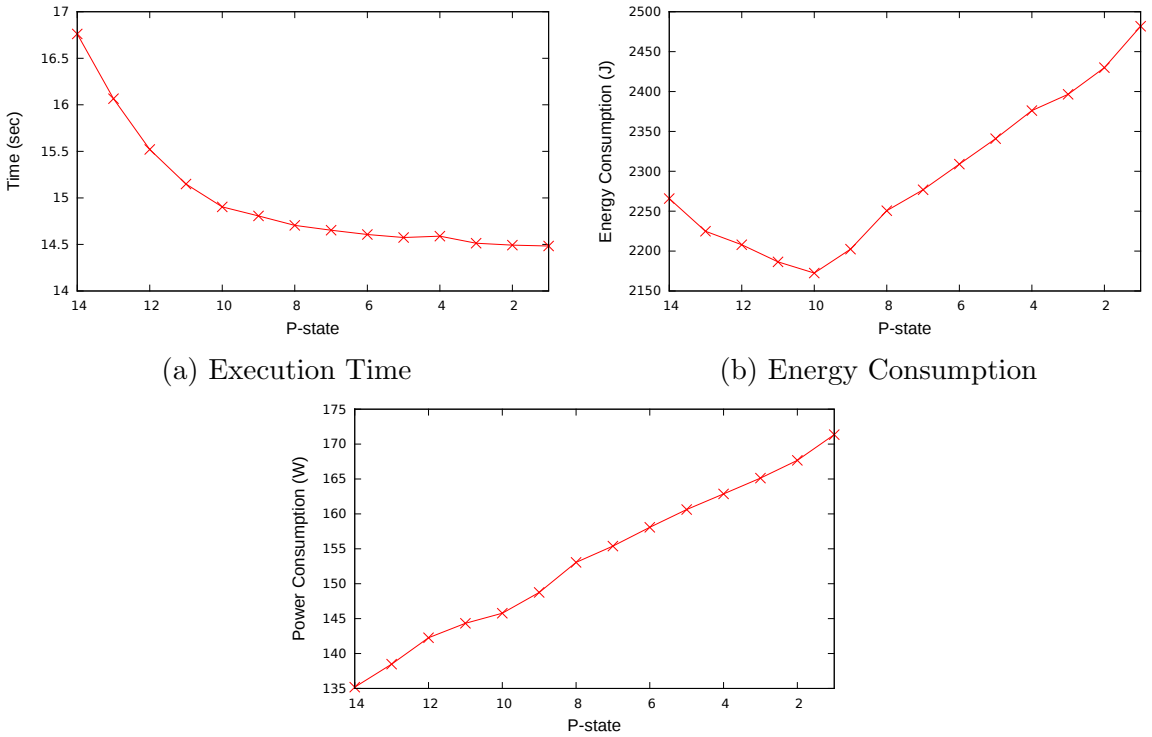
(a) Execution Time

(b) Energy Consumption

(c) Power Consumption

Figure 4.6: Impact of frequency scaling on the behavior of a compute intensive kernel

From Figure 4.6a, we observe that increasing the frequency (decreasing the P-state) of the kernel leads to a drop in the execution time. For a computation-intensive kernel, this is expected because of the increase in the instruction per count. It can also be observed that the energy consumption of the kernel decreases with a drop in the P-state (rise in frequency). However, it must be noted than there isn't a significant difference in energy consumption when the P-state drops below P-9. In fact, there is a rise in the energy consumption when the CPU cores switch from P-2 to P-1 (Figure 4.6b). Another observation is that the drop in the energy consumption is not proportional to the drop in the execution time. This is evident from the power-curve Figure 4.6c.



(a) Execution Time

(b) Energy Consumption

(c) Power Consumption

Figure 4.7: Impact of frequency scaling on the behavior of STREAM TRIAD, a memory intensive kernel

From the above results, we learn that executing a given phase of an application at the highest frequency does not always lead to the lowest energy consumption. In fact, for memory intensive kernels, higher energy efficiency may be achieved by executing a kernel at a lower CPU frequency.

4.4 Chapter Summary

This chapter investigates the feasibility of using "race-to-halt" approaches for executing large scale distributed applications. Empirical results were discussed which analyzes these applications at three different levels of granularity - job level, process

level, and application phase level. For each level, it was shown that executing an application at the highest supported CPU frequency ("race-to-halt" approach) does not always lead to an energy-efficient solution.

Chapter 5

Energy Costs Associated with Distributed-Memory Programming

Shared memory models are characterized by implicit data transfers that are bounded by the distance between the CPU and the different levels of the memory hierarchy. Such data transfers include intra-node cache and memory accesses that consume very low energy, typically of the order of a few hundred pico Joules[38]. In contrast, inter-process communication patterns in PGAS models are initiated by the programmer and bounded by a number of factors internal and external to a single compute node.

OpenSHMEM decouples the communication and synchronization operations[60]. A process may progress in its execution of code segments while being oblivious to communication operations initiated by other processes. In other words, processes are permitted to have an inconsistent view of the globally shared memory during a phase of an application.

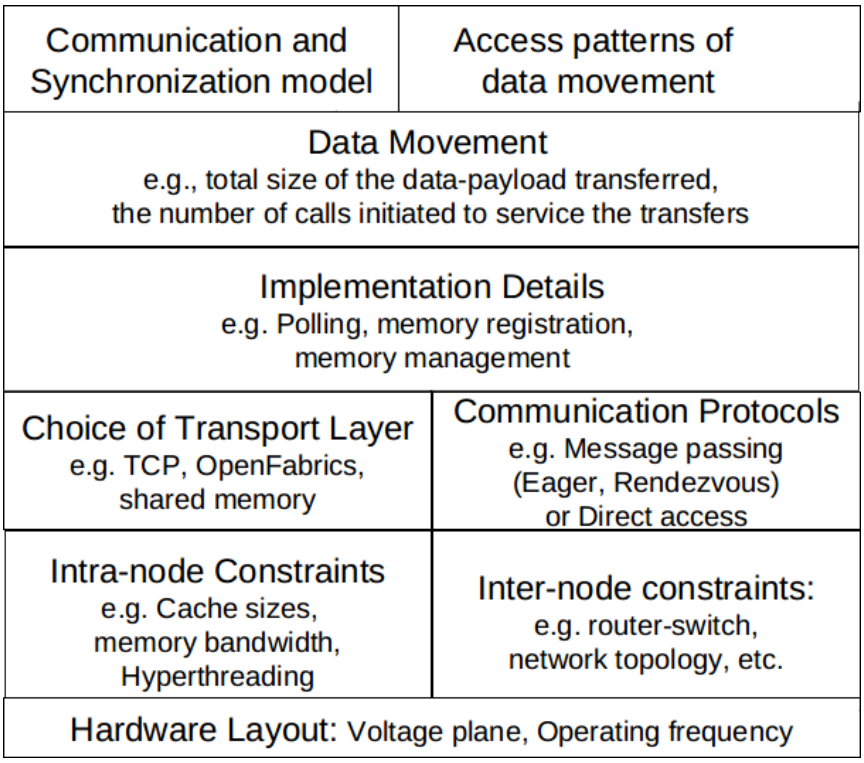


Figure 5.1: Factors impacting the energy and power consumption across the hardware and software stack

In this chapter, we highlight the notion that synchronization and data movement constructs within a distributed programming model have a potential of affecting the total energy consumption of OpenSHMEM applications. Costs in terms of execution time, CPU energy, and DRAM energy is presented as a motivation towards the need for reducing their impact on energy profiles of applications.

The following sections discuss the different components of communication and synchronization constructs in OpenSHMEM.

5.1 Communication Costs

The OpenSHMEM memory model permits RDMA operations. Our studies indicate that during the progress of such operations, there is a significant impact on the power consumed by the CPU and DRAM due to multiple factors including the design of the data transfer patterns within an application, the design of the communication protocols within a middleware, the architectural constraints laid by the interconnect solutions, and also the levels of memory hierarchy within a compute node.

Figure 5.1 lists many such factors throughout the hardware and software stack.

5.2 Synchronization Costs

PGAS implementations like OpenSHMEM stand out with respect to their memory consistency model.

To ensure sequential consistency and an ordering of remote data transfer operations, OpenSHMEM applications may use synchronizing constructs like *shmem_quiet()*, *shmem_fence()*, and *shmem_barrier_all()*. The impact of such barriers on the performance and scalability of distributed applications is well known[111].

The Issue For applications in which the work distribution among multiple processes is non-uniform, using synchronizing constructs[1] result in a subset of processes waiting for varying intervals of time without making any progress. Applications become bounded by the speed of the slowest process.

[1] In the rest of the text, we use the words 'synchronizing construct' and 'barriers' interchangeably.

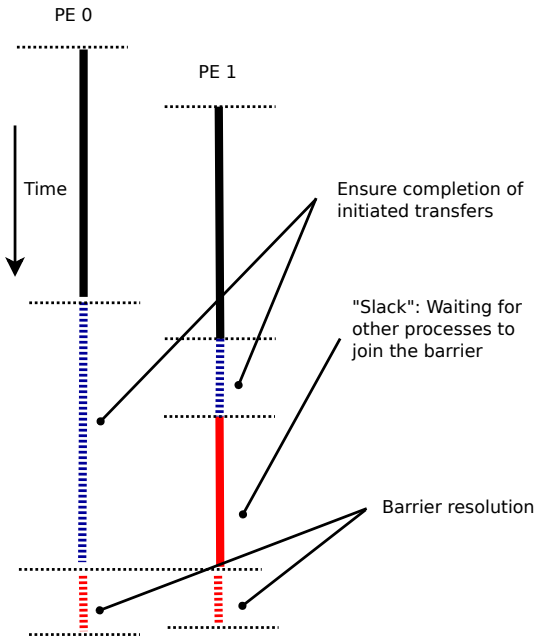


Figure 5.2: Excess slack within fast processes corresponds to energy consumption without any application progress.

The impact on scalability due to such load imbalance has been well explored in studies like[111]. In addition to the impact on performance, the lack of progress also leads to CPU cycles being wasted, which manifests into a rise in the total energy consumption by the applications. The excess CPU cycles invested can be treated as an additional *slack* that a fast process can exploit before meeting the deadline imposed upon by the speed of the slowest process. This notion of a 'slack' is depicted in Figure 5.2.

A Note on Implementation of Barriers In an OpenSHMEM program, the factors contributing to the time spent executing a synchronization construct maybe

attributed to a number of factors governed by the communication and execution model of the programming model. These include:

1. Ensure that all remote read/write operations have completed on the destination process.

2. Ensure that local user buffers that were used for remote read/write operations can be reused.

3. Ensure that all the synchronizing processes have a consistent view of the local shared-memory.

4. Ensure that all the processes reach a synchronization point within their execution paths.

Common implementations of a barrier incorporate the use of shared semaphores which are subjected to repeated atomic polling by each process. The purpose of this polling is to keep track of the state of the semaphore objects. These are typically *globally shared* so that they remain accessible by other processes[2]. The polling is always *atomic* in nature to ensure that only one process can test or set it at any point in time. Furthermore, this polling is typically performed directly over the copy of the semaphore object within the remotely accessible memory, thus avoiding accesses to stale cached versions. This in turn increases the pressure on the memory. Additionally, the polling is *continuous*, to ensure that there is no significant delay

[2]If RDMA is supported by the interconnect, the overhead of the management of semaphores reduces when they are remotely accessible

between the time each process signals entering the barrier and the time this event is detected.

In this work, we focus on reducing the energy cost invested by a process that is enforced by the execution model to wait for all the processes to reach the barrier point before proceeding along its execution path.

5.3 Computation Costs

The energy costs associated with computational kernels correspond to execution of the set of instructions supported by the target processor architecture. The actual value of the energy consumption by an instruction depends on a large number of variables including the arrangement and count of transistors servicing the transfer, the on-chip components used to transfer the instruction, caching effects, pipeline stalls, etc[11, 155, 101].

There have been multiple research efforts that have proposed extensions to existing Instruction Set Architectures (ISA) that enable the software stack to leverage energy saving opportunities on a given platform[12, 65, 161, 13].

5.4 Case Study: a CORAL Benchmark

5.4.1 LSMS

LSMS or Locally Self-consistent Multiple Scattering is a chemistry-based application and is part of the CORAL benchmark-suite. When scaled beyond two processes, the communication pattern follows a two-stage master/worker behavior. This is illustrated in Figure5.3. As shown, all processes, except the root, are sub-divided

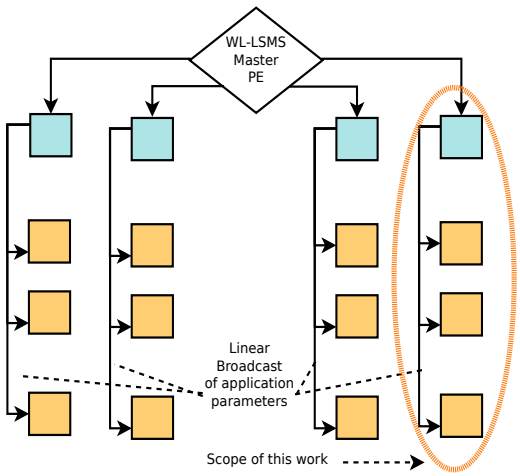


Figure 5.3: Two-stage communication pattern within WL-LSMS

into smaller groups. Each group has its own master which in turn communicates with the root process. Within a group, each master transfers a set of parameters to the other worker processes that belong to the same group.

In order to study the opportunities of energy savings during communication and synchronization, the application was profiled using VampirTrace and the collected traces were analyzed using Vampir.

5.4.2 Communication Phases

Consider Figure 5.4, which depicts the communication-intensive phase of the application between the master and the worker processes.

To communicate the parameters, a programmer may choose to design the data movement pattern following any of the access patterns discussed in Chapter 8. The interaction between the root, master and other worker processes is illustrated as a line diagram in Figure 5.6.

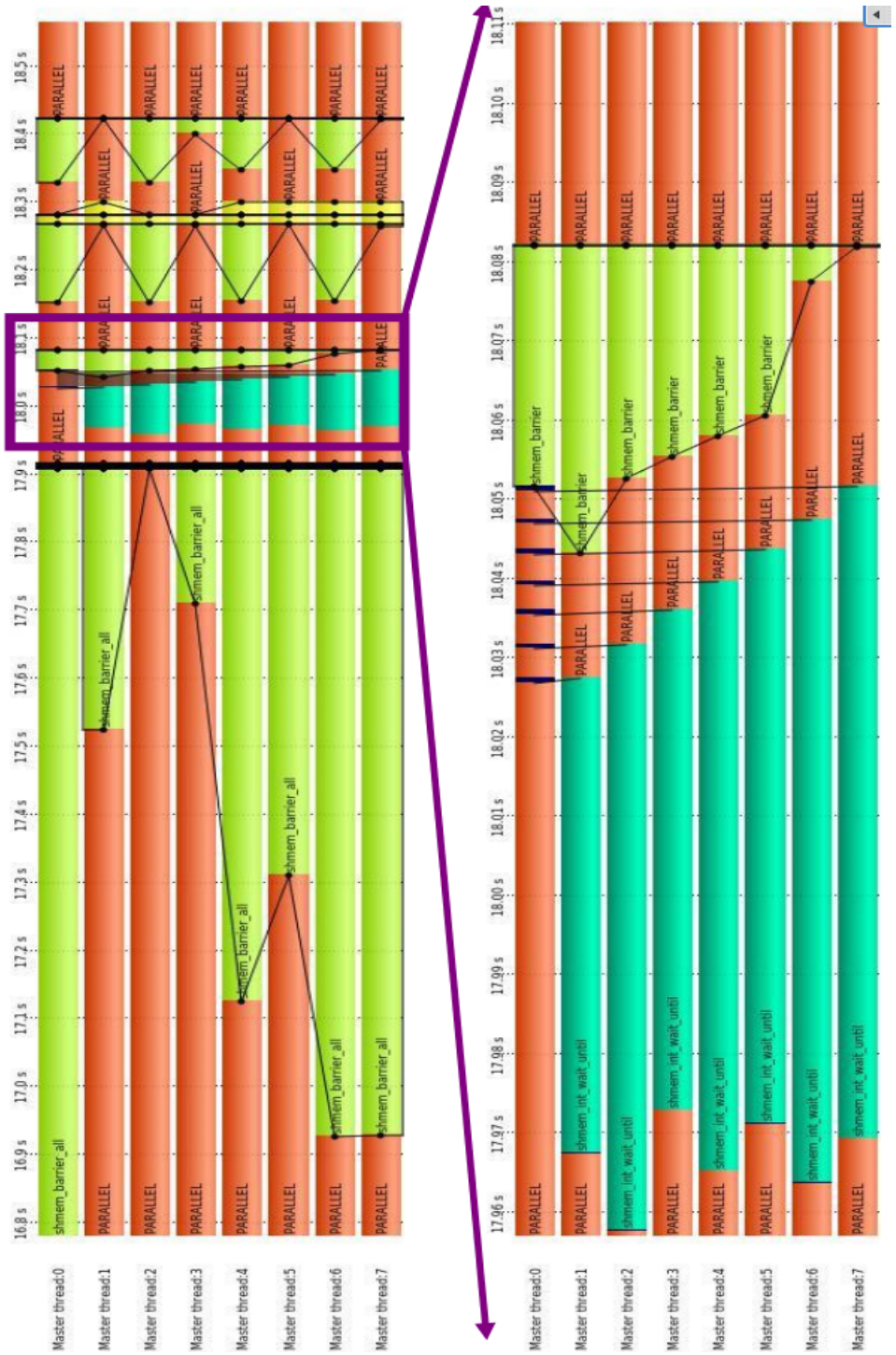


Figure 5.4: Visualization of communication pattern within LSMS, as generated by the Vampir visualizer.

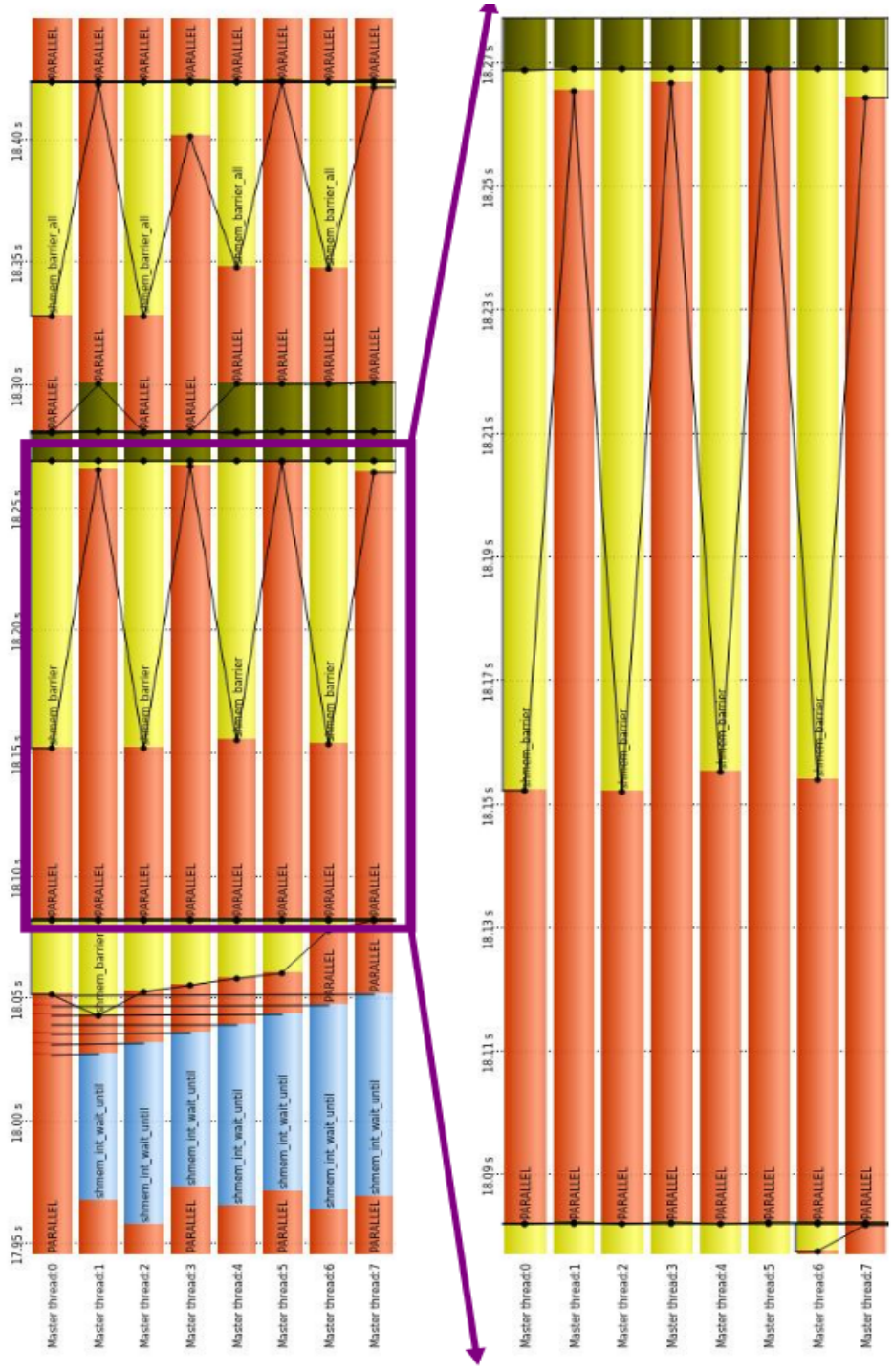


Figure 5.5: Visualization of synchronization behavior within LSMS, as generated by the Vampir visualizer.

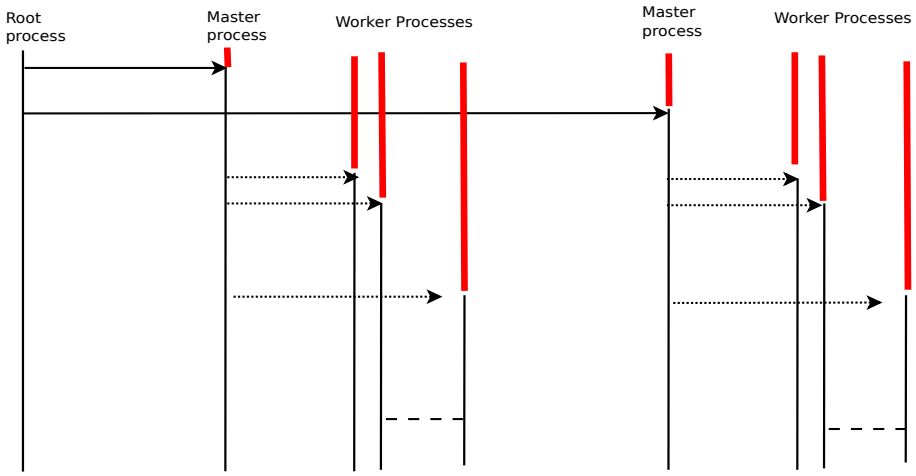


Figure 5.6: Interaction between root, master, and worker processes

As the application is scaled to higher process counts, the number of workers communicating with a given master increases. Also, the number of iterations of the data transfer phase also increases. The energy costs of such communication-intensive phases can be reduced by altering the data movement pattern.

5.4.3 Synchronizing Phases

Consider Figure 5.5 which depicts the load imbalance between the master and the worker processes during a computation phase of the application.

As the application is scaled to higher process counts, the excess time invested waiting for slower processes to reach the synchronization barrier corresponds to excess energy loss. Ensuring that all the processes reach the barrier at the same time provides opportunities to nullify this energy cost.

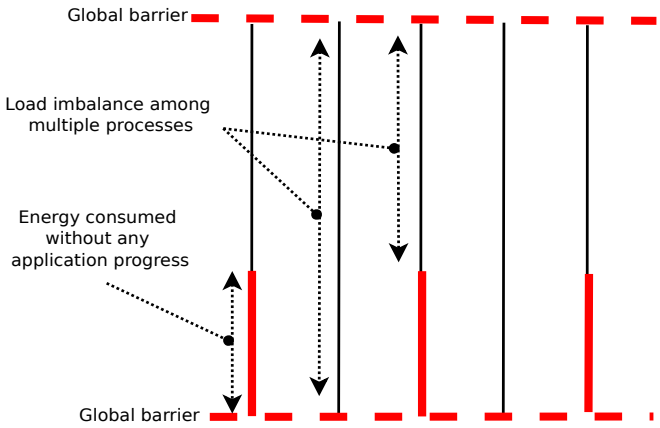


Figure 5.7: Load imbalance among processes lead to extra energy invested waiting for the slower processes to catch up.

5.5 Chapter Summary

This chapter presents an overview of some of the cost factors that affect the energy and power consumption behavior of processes while participating in synchronizing global barriers and remote data transfers.

We observed that the energy and power cost is dependent on the time spent within barriers and the number of processes participating in the barriers.

Additionally, our study indicates that the energy and power cost incurred by a system while servicing remote data transfers are dependent on a number of factors characterizing the underlying the hardware and software stack. These include the sizes of the memory hierarchy, design of the communication protocols, and the capabilities of the interconnect solutions.

The impact of these factors depend on the size of the total data to be transferred within a communication phase of an application. In addition, the number of data transfers initiated to transfer this load also impact the energy and power consumption behavior of OpenSHMEM-like PGAS applications.

Chapter 6

Communication: Fragment Count and Payload Sizes

We identify two application characteristics to analyze the communication patterns in OpenSHMEM programs:

- *Total size of the data to be transferred*

  This factor is governed by the problem size of the application and granularity of parallelism chosen[1].

- *Number of explicit calls (or fragments) used to transfer the data*

  This factor is dependent on the nature of the design of the application by the programmer.

[1]The granularity of parallelism is typically determined by the number of processes participating in the data/task distribution.

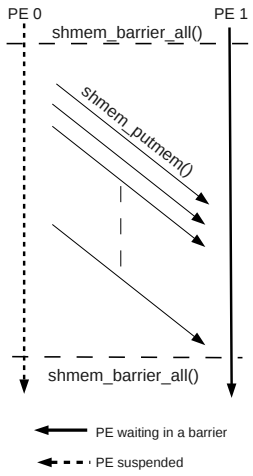


Figure 6.1: Line diagram for microbenchmark used to detect the impact of data sizes and fragments

The study of the effects of inter-node factors on the energy profile of remote data transfers is outside the scope of this work. Nevertheless, in order to account for their impact, we abstract their effects in terms of the net achievable bandwidth. Figure 6.3 illustrates this constraint with respect to the two communication-based parameters discussed above. We observe that for any given data transfer size, maximum bandwidth is achievable with minimum amount of fragmentation.

The impact of the intra-node factors on the energy and power cost incurred in Sections 6.1 and 6.2, respectively.

This section discusses the impact of the use of explicit data transfer routines on the energy cost of OpenSHMEM applications. While using these routines, a programmer may decide to transfer the program data in multiple fragments based on the design of an application. While this practice makes it easier to align the

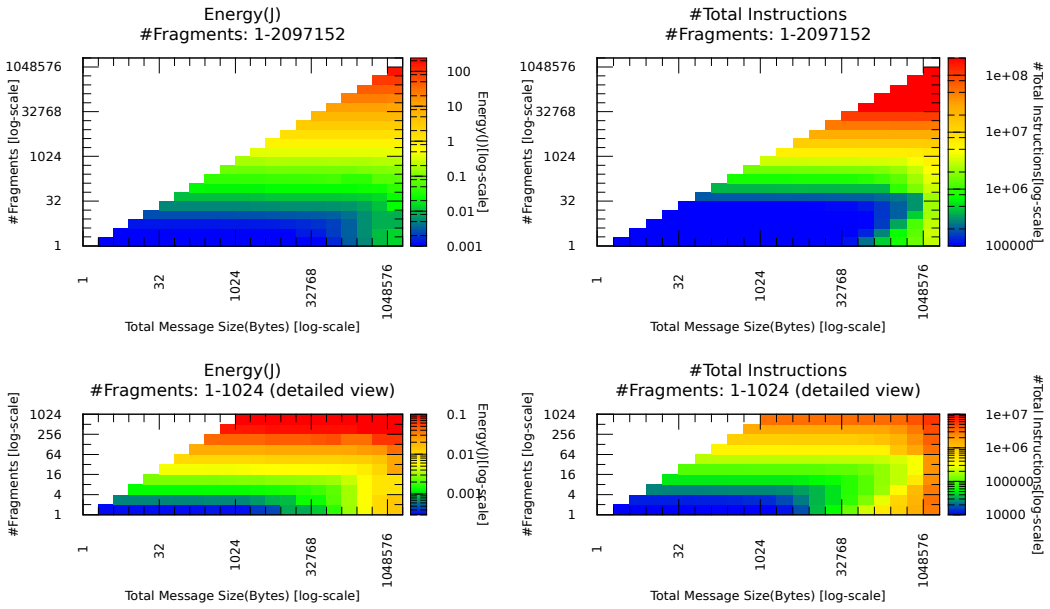


Figure 6.2: Relationship between energy consumption by cores(left) and the total number of instructions executed(right). Top: Results for cases where: $Fragments \in [1, 2097152]$. Bottom: Results for cases where: $Fragments \in [1, 1024]$

semantics of an algorithm to an implementing program, our studies indicate that such practices come at a significant cost.

6.1 Energy-Consumption Observations

Figure 6.2 illustrates the energy consumption by the CPU and the DRAM with respect to the different message sizes of data transferred (in bytes along X-axis) and the number of fragments used to transfer the total data (along Y-axis). The noteworthy observations are:

- Energy consumed holds a correlation to the number of instructions executed. Since an increase in the number of data transfers initiated implies a rise in the

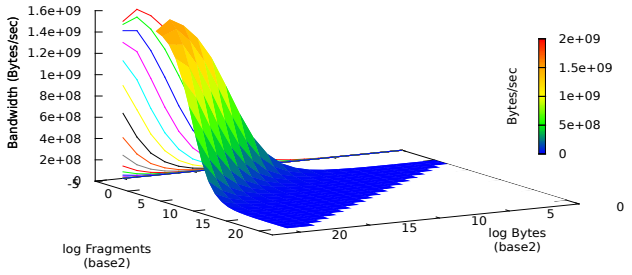


Figure 6.3: The impact on the peak achievable bandwidth with respect to: (i) size of the total data to be transferred; (ii) number of fragments into which the transfer is divided into

number of instructions executed, the energy consumption increases with rise in fragmentation.

- For large bulk transfers with a fixed message size, the energy consumed remains independent of the *initial* rise in fragmentation.

- Using a constant number of fragments, the energy consumed in servicing the transfer of small to medium sized messages (2 to 65536 bytes) is independent of the total size of the data transferred. This behavior can also be observed in terms of the spectrum of the achievable bandwidth shown in Figure6.3. This behavior can be explained by the fact that for such small-sized messages, the cost in managing the data buffers for remote transfers overshadows the actual movement of the data. This cost is independent of the message size and hence leads to a steady bandwidth and energy consumption.

- For large bulk transfers (>65536 bytes), the energy consumed increases with the size of the data to be transferred. This can be attributed to cost incurred

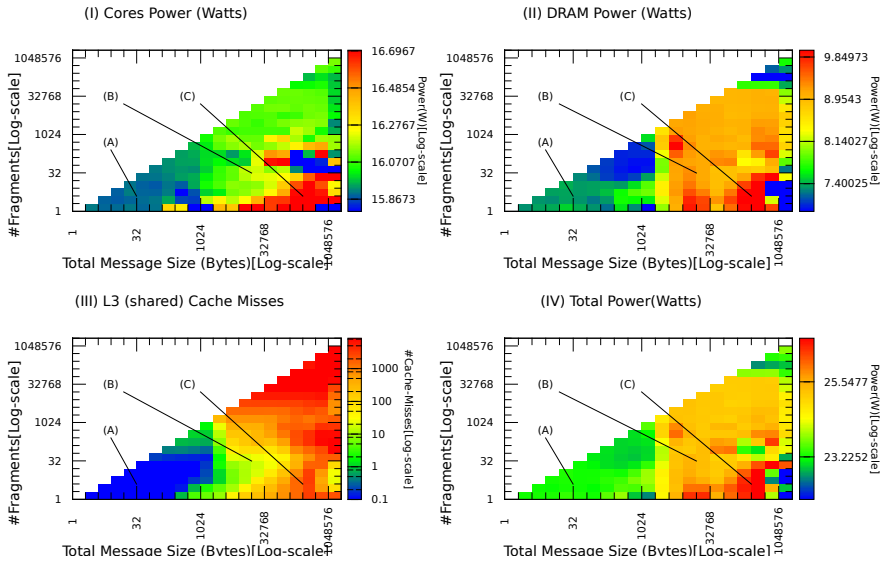


Figure 6.4: (I,II,IV)Power consumed by CPU, DRAM, total system (III) Total L3 cache misses. The various distinct levels of power are represented as:
(A)Small payload sized(up to 2KB) transfers lead to less power consumption by the cores and DRAM;
(B)Medium to large message sizes(4K and beyond) imply accesses of large memory regions and this impacts power consumption;
(C)Large payload sizes with minimum fragmentation leads to higher power consumption by the cores. The underlying NIC is generally responsible for chunking such large transfers, the effect on which is not accounted for by the cores.

in handling data buffers. For large messages, this becomes dependent on the actual size of data that is being transferred.

6.2 Power-Consumption Observations

Figures 6.4 depicts the power consumption by the CPU cores and the DRAM for different message sizes and number of fragments.

- For small data transfer sizes, the power consumed by the CPU (16 Watts) and the DRAM (7 Watts) is low.

- The power consumed by the CPU during transfer of large bulk data payloads (16.2 Watts) is marginally more (1.25%) than that consumed during small data transfers.

- The power consumed by the DRAM during transfer of large bulk data payloads (9 Watts) is significantly more (22%) than that consumed during small data transfers.

- With very low fragmentation, the CPU consumes more power than with fragmented data payload.

- As the message size is increased (along x-axis), the transition of the change in the power consumption behavior by the CPU appears to hold a correlation to the sizes of the intermediate levels of the cache hierarchy. The transition levels correspond to the sizes of the L1 and L2 caches - 32KB and 256KB respectively. Since the caches were flushed after every set of readings, one can speculate that every cache miss in L1 and L2 adds on to the memory pressure on the shared L3 cache thereby resulting in a proportional rise in cache misses. This effect can be observed in Figure 6.4(III), which illustrates the number of L3 cache misses.

- From Figure 6.4, the average power consumption by the system (CPU+DRAM) while servicing large bulk message sizes (28 Watts) is 21.73% higher than that consumed by small message sizes (23 Watts).

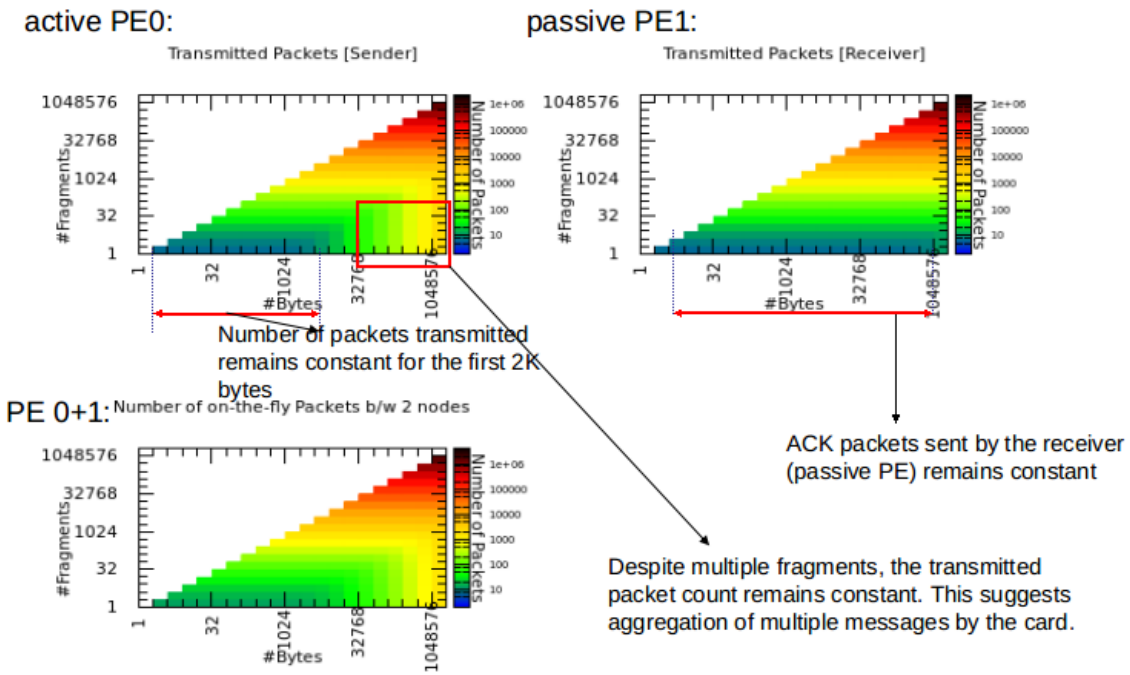


Figure 6.5: The number of raw Infiniband Packets transmitted / received by the NIC during a point-to-point data transfer:(i) Number of packets transmitted by the NIC servicing the sender process; (ii) Number of packets transmitted by the NIC servicing the receiver process; (iii) number of packets on-the-fly transmitted between the two nodes during the lifetime of the transfer

6.3 Network-Card behavior

The energy and power consumption discussed above encompass the behavior of the CPU and the DRAM. As of this writing, to the best of our knowledge, there exists a lack of infrastructure that is capable of capturing the energy/power consumption by the physical layer in general, especially the network card. To circumvent this lack of technology, we study the behavior of the NIC used in our experiments, we studied the number of raw packets transmitted and received by its endpoints. The assumption behind this approach is that the number of packets transmitted holds a direct correlation to the energy consumed by the card. The results are plotted in Figure 6.5. It can be observed that indeed for each combination of payload size and fragment count, the number of raw packets transmitted and received is proportional to the execution time of the data transfer.

6.4 Chapter Summary

Parallel applications that rely on data distribution as a means of achieving parallelism, are typically characterized with inter-process communication of large amount of data. The overhead in the movement of data in distributed systems may overshadow any achievable performance gain due to the reduction of the problem size.

OpenSHMEM-like PGAS implementations are characterized by explicit data transfers as part of the application design. The performance of such transfers in a distributed environment require the participation of not only the CPU and the memory hierarchy but also the interconnect solutions. As a result, the contribution of such transfers to the overall power and energy consumption of large scale OpenSHMEM

applications should not be ignored while evaluating the power characteristics of distributed programming models.

From the view of an OpenSHMEM programmer, the *controllable* factors that affect the bandwidth of a communication kernel include the total size of data to be transferred across the network and the number of fragments (or explicit data transfer calls) that are used to perform this transfer. We evaluated the power and energy consumption by the cores and the DRAM to study the effects of performing transfers of different payload sizes and different number of fragments. We observed that energy consumption increases for transfer of large message sizes. Additionally, for a fixed-data payload, an increase in fragmentation leads to an increase in the energy consumed. This motivates the need for aggregation of small-sized messages when the data transfer size covered by the problem size is large.

For more details about the topics discussed in this Chapter, the interested reader is directed to the literature documented by Jana et al. under [84].

Chapter 7

Communication: Network-Stack Design

This work is an extension of our previous experience of studying the impact of one-sided communication in PGAS models (OpenSHMEM) [84]. We had learned that managing small-sized data transfers on RDMA-capable networks are more energy efficient than handling large bulk transfers. In this work, we present empirical evidence highlighting the contribution of design factors within the software stack to the power consumption by the underlying system. Our takeaway from this study is that the protocols used to implement such interfaces, play a significant role in impacting its power-cost. In addition, since the design of communication libraries are tuned to specific interconnect solutions, the choice of the transport layer adopted for servicing data transfers plays an equally significant role.

In Section 7.1, we discuss the impact of the above factors on the behavior of two-sided communication interfaces within MPI, the de facto standard for distributed memory model. This is an extension of past work on analyzing the impact of data-transfer characteristics on one-sided communication interfaces[84]. This is followed by a description of our observations of the impact on power consumption by CPU cores and the DRAM while relying on Ethernet (via traditional TCP) and Infiniband (via OFED or OpenFabrics Enterprise Distribution[122]) fabrics (Section 7.2). All of these are discussed with respect to the implementation of two basic message-passing schemes - the Eager and Rendezvous protocols. Finally in Section 7.2, we summarize our findings by discussing the total power efficiency achievable for each of the above configurations. We hope this work motivates the practice of taking power-metrics into consideration while designing middleware solutions for Exascale-era machines.

7.1 Factors affecting Power and Energy profile of remote data transfers

Two-sided data-transfer in distributed-memory models such as MPI, sockets, etc., require the active participation of both the sender and the receiver of the data. The impact on the achievable latency and bandwidth of such transfers depend on the design of the transport layer (and the associated interconnect) and the data transfer protocol. As part of this work, we learned that the impact of these factors on the energy metrics is very important.

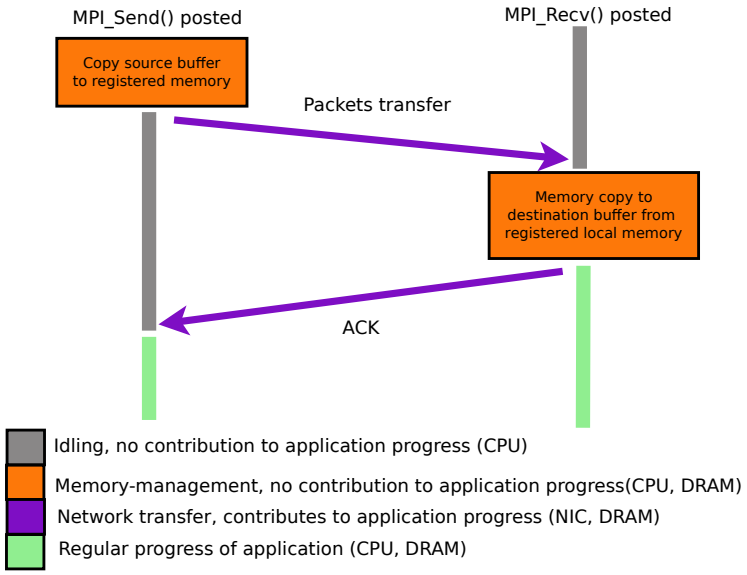


Figure 7.1: Eager Protocol

7.1.1 Choice of transport layer and the associated interconnect

If the target platform relies on an OS-based TCP protocol for servicing data transfers, CPU cores undergo multiple switches between user and supervisor operating modes. In addition, relying on Ethernet-based fabric has the potential of degrading the achievable efficiency both in performance and energy consumption (as discussed later). To avoid this, a communication library may exploit kernel-bypass mechanisms and RDMA-based capabilities of the OFED stack on top of modern interconnects like InfiniBand, etc.

7.1.2 Design of data-transfer protocols

Data transfers within message-passing libraries are based on two well-established paradigms - the eager and rendezvous protocols. The primary phases involved in

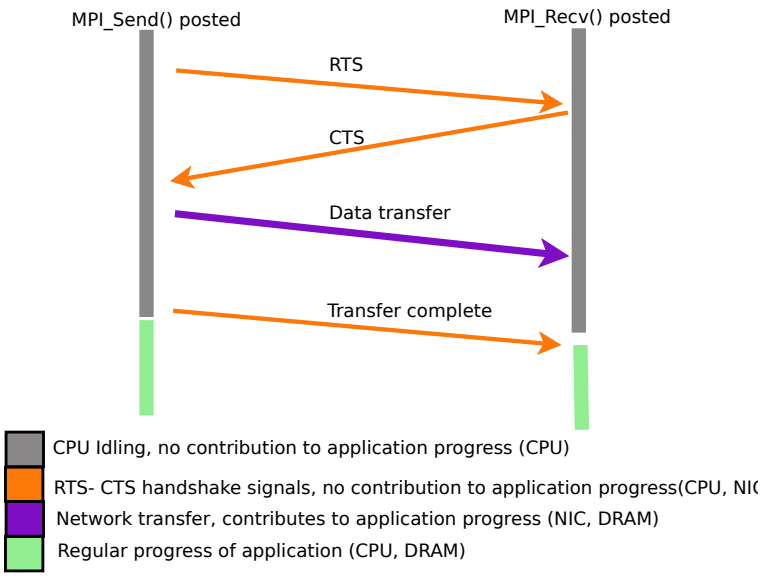


Figure 7.2: Sequence Diagrams for Rendezvous Protocol

these protocols are depicted in the line diagrams in Figures 7.1 and 7.2.

Rendezvous protocols incorporate RTS-CTS[1] handshaking to ensure that the sender waits for an explicit request from the receiver before servicing the actual transfer. Such an exchange ensures that the receiver's buffer is ready for being over-written with the incoming payload. This method has proven to be beneficial for large bulk transfers since the overhead of the handshaking operation gets eclipsed by the gain in the throughput of the end-to-end data movement[19]. For small message sizes, however the additional round trip proves expensive.

Eager protocols help mitigate the above overhead by reducing the time and energy spent by the sender waiting for the receiver to post the destination buffer address.

[1]Request-To-Send / Clear-To-Send two-sided handshake signal

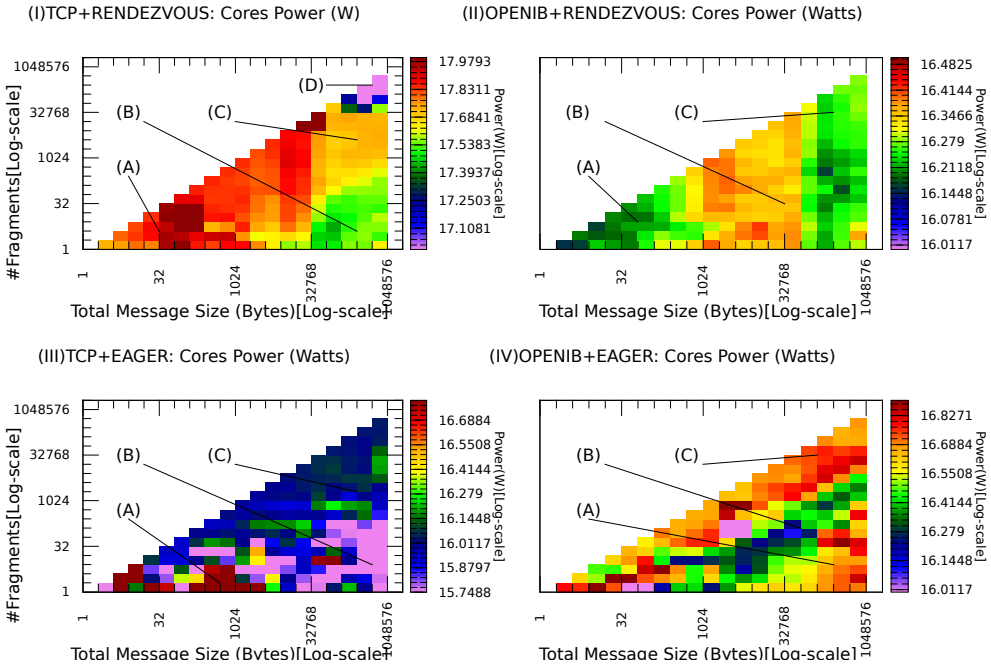
The sender may choose to start transferring its data to a pre-allocated buffer without waiting for the receiver to send a CTS signal. This is easily facilitated by an underlying interconnect solution that supports RDMA-based transfers. Once the receiver calls MPI_Recv(), it can copy-out the data from this pre-allocated buffer. Not surprisingly, the impact of latency of such techniques is bounded by the costs of memory registration and the additional in-memory copies both at the sender's and the receiver's end.

7.2 Empirical Observation and Analysis

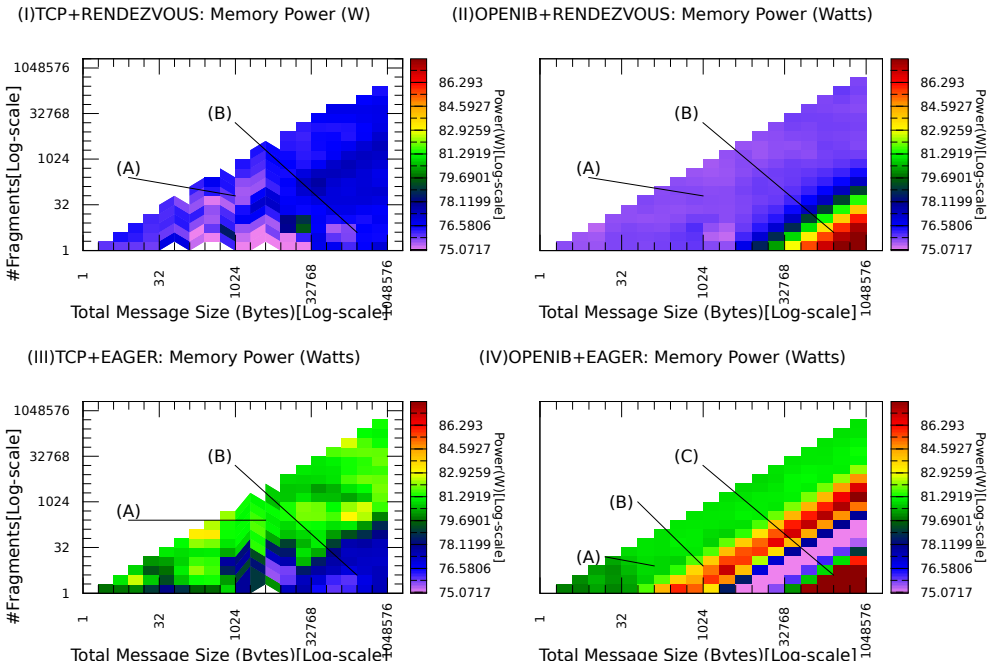
In this section, we present our observations of the impact on the energy and power consumption by the CPU cores and memory due to the factors discussed in the previous section.

7.2.1 Using TCP over Ethernet

Using Rendezvous protocol Consider the power consumption by the CPU cores servicing the sender process (Figure 7.3a(I)). While handling small data-payloads ($< 1KB$) the CPU cores suffer a high power cost (region A). The reason for this may be attributed to the very low latency of the operation, the frequent context switches between the operating modes (see Section 7.1) and the high overhead incurred during the handshake operations. This cost reduces for large bulk transfers ($> 32KB$) due to a rise in the latency of the data transfer and a drop in the rate of active participation by the CPU cores (region B). Dividing such bulk buffers into smaller fragments again leads to a rise in the cost (region C). However, this rise in power-cost in limited due to high latency that arises with heavy fragmentation. Due to this, the inverse relation

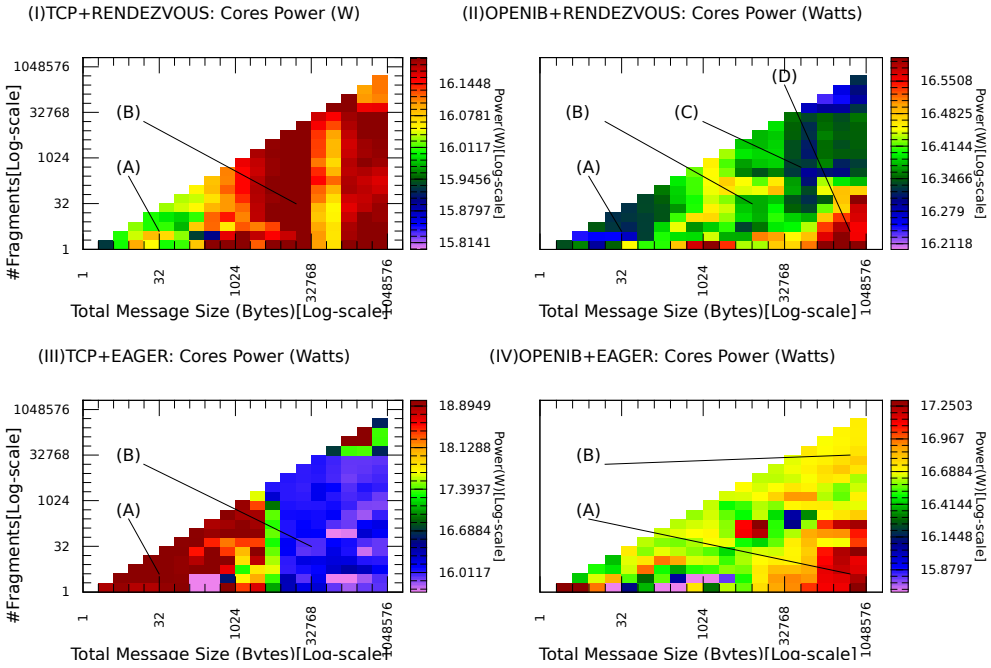


(a) Power consumed by the CPU cores

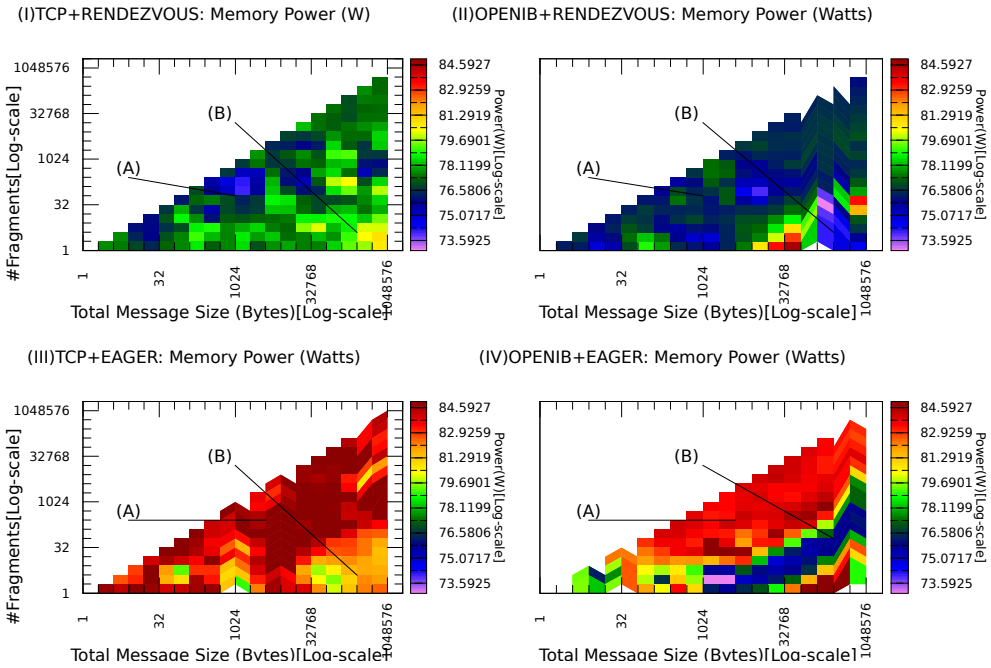


(b) Power consumed by the DRAM

Figure 7.3: Power consumed by the CPU cores and the DRAM while servicing remote data transfers by the sender process



(a) Power consumed by the CPU cores



(b) Power consumed by the DRAM

Figure 7.4: Power consumed by the CPU cores and the DRAM while servicing remote data transfers by the receiver process

between the increase in latency and the drop in the average power consumption can be observed in region D. On the receiver's end (Figure 7.4a(I)), the power consumed primarily depends on the size of the data being transferred (regions A,B). It must also be noted that the passive participation by the receiver (when compared to the sender), leads to a lower range of power-consumption (15.5-16.2W as compared to 17-18W). From the memory unit's perspective (Figure 7.3b(I)) the power cost incurred by the sender process while servicing small transfers (region A) is lesser than that while servicing large transfers (region B).

Using Eager protocol Switching from a rendezvous protocol to an eager protocol definitely reduces the operating power-cost incurred by the CPU cores while servicing large data transfers by the sender and the receiver processes (Figs. 7.3a(III), 7.4a(III)(regions A-B)). The negative impact of fragmentation can be observed in terms of the rise in the power-cost incurred by the memory modules, both by the sender as well as the receiver (Figs. 7.3b(III), 7.4b(III)(regions A-B)). Implementing an eager protocol using a non-RDMA based fabric like Ethernet leads to a significant rise in power-consumption at the receiver's end (Figure 7.4b(III)(region A)). A rise in the number of bytes transferred per fragment leads to a rise in the energy consumed by the memory. However, we see from region B in Figs. 7.3b(III), 7.4b(III) that the power consumption by the memory module drops. This can be attributed to the rise in the latency in completion of the transfer of the entire data-payload.

7.2.2 Using OpenIB/OFED stack over InfiniBand

Using Rendezvous protocol: At first glance, Figs. 7.3a(II), 7.4a(II) (regions A,B,C), depict that the power consumed by the CPU cores is dependent on the total size of the data-payload and not so much on the degree of fragmentation. However, one must take into account that using the rendezvous protocol over the OFED stack leads to a combination of two different types of overhead. The first is the power-penalty of using either memory-pinning or local memcpy operations (as explained in the next subsection). The second is the overhead due to the handshaking operations (as explained in the previous sub-sections).

With regards to the power consumed by the memory at the sender's side, the cost increases monotonically with a rise in the size per fragment. As discussed in the following bullet point below, using the OFED stack is accompanied with the power-penalty of either memory-pinning or local memcpy operations. This cost varies with the number of bytes transferred with each fragment (Figure7.3b(III)).

Using Eager protocol: Parallel diagonally-colored bands in Figs. 7.3a(IV), 7.3b(IV), 7.4a(IV), 7.4b(IV) show that the power consumed by the cores and the memory unit, both depend on the number of bytes transferred within each fragment. As discussed before, either the memory space containing these fragments are dynamically pinned-down (registered) with the NIC or its contents are copied over to some pre-registered buffer. The performance penalty of dynamic registration of small buffers is expensive. Thus, a runtime implementation would typically perform a local copy of the contents into a pre-registered buffer. Our experience shows that the power cost of this memory copy increases with rise in the fragment size (i.e.

Table 7.1: Symbols in Eqn. 7.1

Symbols	Metric
Bw	Achievable bandwidth (bytes/sec)
P_{net}	Net average power consumed (W)
$B_{payload}$	Total number of bytes transmitted
ΔE_s	Energy consumption by sender (J)
ΔE_r	Energy consumption by receiver (J)
$P_{s,cpu}$	Cores power consumption at sender (W)
$P_{r,cpu}$	Cores power consumption at receiver (W)
$P_{s,mem}$	Memory power consumption at sender (W)
$P_{r,mem}$	Memory power consumption at receiver (W)

bytes/fragment). This can be observed in region C. As the size of each fragment increases, an implementation would typically start dynamically registering user buffers with the NIC. Either way, this keeps the CPU cores active. It is during this inflection point that we observe a slight drop in the cores power consumption. Further increase in the size of fragment again leads to a rise in this cost (region A).

Complementary to the CPU power consumption, the power-cost incurred by the memory rises with the size per fragment (region A). It too hits a cool spot (region B) and then rises up monotonically with a rise in the achievable bandwidth on the NIC.

7.3 Energy Efficiency of Data Transfers

To study the net impact of the choice of the communication protocols and the transport layer, we evaluated the power efficiency using a metric tuned towards communication-intensive kernels. The energy efficiency of a compute-intensive application kernel is given by the total number of machine/floating-point operations per second per watt of power consumed (MOPS/Watt or FLOPS/Watt). To evaluate the cost of data transfer operations, we use a similar metric - the net bandwidth

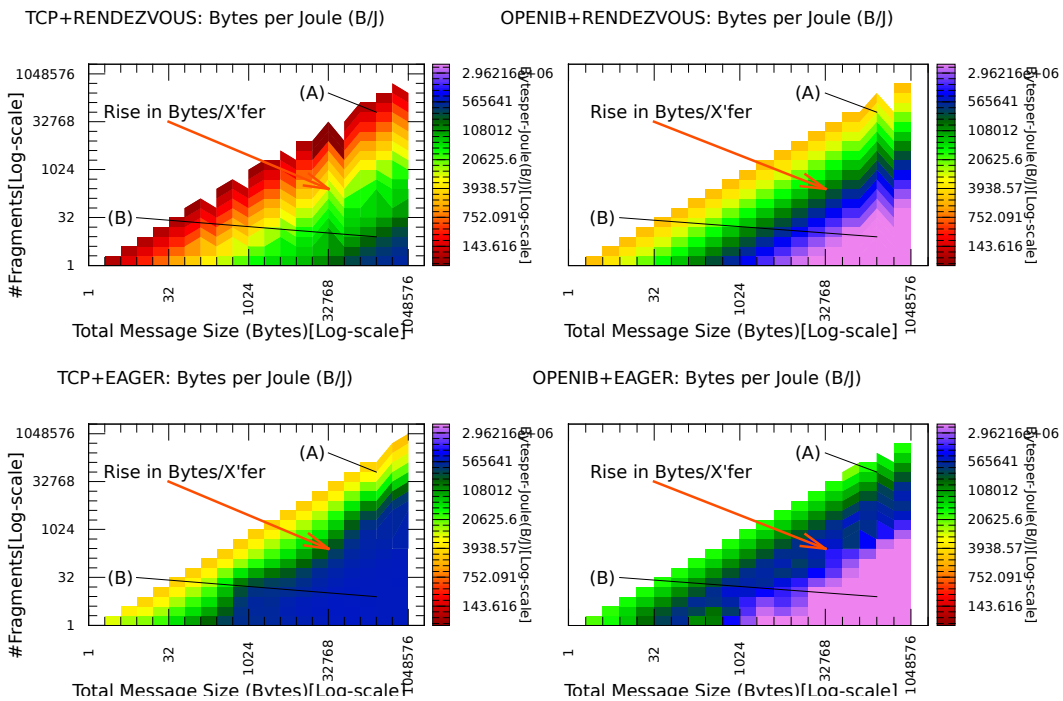


Figure 7.5: A summary of the total bytes transferred per Joule of energy consumed by the sender and the receiver while participating in remote data transfers.

achievable per watt of power consumed by the participating processes; in other words - the number of bytes that can be transferred across the network for each joule of energy consumed by the sender and receiver. For a point-to-point communication model like MPI, this may be represented by the equation below:

$$\frac{Bw}{P_{net}} = \frac{Bw}{(P_{s,cpu}+P_{s,mem}+P_{r,cpu}+P_{r,mem})} = \frac{B_{payload}}{\Delta E_s+\Delta E_r} \quad (\frac{Bytes}{Joule}) \tag{7.1}$$

The symbols used in this equation are listed in Table 7.1. The net impact on this metric is discussed in Fig: 7.5. The primary observations are:

- The net bandwidth achievable using an interconnect directly impacts the maximum value of energy efficiency. Thus the peak bytes transmitted per joule is an order of a magnitude higher when using the OpenIB over InfiniBand as compared to TCP over Ethernet. Moreover, irrespective of the type of transport adopted, energy-efficient communication can be achieved using an eager-based protocol.

- *Impact of number of bytes packed per transfer:*

  - In the figure, the arrow points towards the direction of the increase in the number of bytes transferred per call. For TCP+Rendezvous configuration (Figure 7.5(I)), we see that the peak energy efficiency during a data transfer (about 0.565MB/Joule) may be attained only when the total message size per transfer is higher than 128KB. With the TCP+Eager protocol however, this peak is attained for message sizes beyond 1KB in size.

  - The highest power efficiency among all the configurations is achievable while using an eager-based protocol over the OpenIB stack - A maximum of 3MB of data is transferred for every joule of energy consumed.

7.4 Chapter Summary

Data movement across large-scale systems has the potential of impacting not only the performance of distributed programming models, but also the power-signatures. In this work, we established the notion that the choice of the transport layer and the design of communication protocols play a significant role in terms of the energy

and power consumption. The empirical results discussed in this work highlighted the behavior of this impact on the CPU cores and the memory. It was observed that the power consumption by CPU cores and the memory bandwidth is not only impacted by the latency of the remote transfers, but also the memory bandwidth between the CPU cores and the memory.

While using traditional TCP over Ethernet, energy savings can be obtained by choosing an eager-based protocol over a rendezvous-based one. While using an eager protocol, an efficiency of up to 600bytes/joule may be obtained. Despite these savings, it must be noted that mapping an eager protocol over a non-RDMA based fabric leads to high power consumption by the memory. While using an RDMA-capable network like InfiniBand, the use of eager-based protocol lends itself naturally to the semantics of the transport layer (OpenFabrics OFED, in our case).

Irrespective of the type of transport and protocol, higher efficiency (bytes transferred per joule) can be achieved by aggregating user buffers into contiguous larger fragments before servicing the transfer. In addition, the net bandwidth achievable during a transfer impacts this efficiency. We hope that results of energy efficiency as well as a detailed study of the impact on the various sub-components of the system would motivate the design of "power-aware" middleware for use with HPC applications.

In the future, we plan to extend this study to evaluate the impact on large-scale multi-node systems. It is equally essential to study the contribution of communication kernels to the energy profiles of large scale real-world HPC applications.

For more details about the topics discussed in this Chapter, the interested reader is directed to the literature documented by Jana et al. under [83].

Chapter 8

Communication: Access Patterns

We extend this study to incorporate the impact of design of communication patterns characterized by multiple occurrences of such transfers. In case of PGAS models, the variation of this impact on the energy and the performance can be attributed to the flexibility of decoupling synchronization costs from the actual transfer of the data-payload. We discuss a number of one-sided based communication patterns and perform an empirical analysis of the maximum possible savings that may be obtained while choosing one access pattern over the other. These also motivate the need for static or dynamic transformations among these communication techniques. We evaluate some well known techniques like aggregating contents of source buffers of multiple remote write operations, using non-blocking data transfer semantics, using pinned-down buffers, and managing the size of data payload packed within each transfer. We present empirical results that indicate that the savings (in terms of performance and energy) obtained through such techniques varies significantly and

there is plenty of opportunity for system programmers to tune for energy-efficient implementations of PGAS models.

In Section 8.1, we describe the various characteristics within PGAS communication kernels, that have an impact on the energy and latency cost of applications. In Section 8.2, we identify a set of basic operations that define an RDMA access. We then list a small subset of communication patterns that are may be built upon these operations. This is followed by some examples of transformations from one access pattern to another and which suggest potential energy savings. We complement this discussion with empirical evidence that provides an optimistic estimate of the maximum possible savings that can be practically achievable. This is presented in Section 8.3.

8.1 Design Factors Impacting Communication-Energy Costs

This section describes some application-level design factors that have the potential of impacting the energy signatures of communication-intensive kernels. While these factors are controllable at the user-level, their use directly impacts the behavior of the underlying communication library.

At a higher level, we categorize these on the basis of -(a) properties of the communication kernel, and (b) properties of individual data transfers

8.1.1 Properties of the Communication Kernel

The total size of the payload being transferred In case of communication-intensive kernels, past work indicates that the total size of all the user buffers participating in RDMA operations have a direct impact on the energy consumption[151, 153, 83]. Since this metric impacts the memory footprint of the application, it is essential to incorporate this metric in empirical studies[1].

The number of explicitly initiated data transfers While the payload size associated with data movement is important, the overhead associated with the software stack that services the transfer of the payload is equally significant. Therefore, one of the crucial factors that needs to be considered while evaluating energy and performance costs is the number of explicitly initiated data accesses to service the transfer of a fixed payload. We refer to this metric as the number of "fragments" or "chunks" that a payload is divided into, while issuing a transfer. The impact of this metric affects multiple layers in the software stack:

- At the application level, this metric typically corresponds to the number of discrete user buffers used to design a communication pattern. The exact count of such buffers and their actual size is dependent on the application's problem size and the algorithm design.

- At the data transfer layer, the impact of this metric supplements the impact of completion semantics of RDMA transfers. For example, in case of non-blocking

[1] It must be noted that the significance of the impact of such a metric depends on the actual ratio of the number of local compute-based operations to those servicing remote transfers.

remote write operations, this metric corresponds to the number of outstanding in-progress PUTs. In such cases, the energy and latency costs are impacted not only by the cost of servicing the actual transfers, but also that of managing and polling for the status' of multiple communication handlers.

- At the bytes transfer layer, the bandwidth and the message rate are dictated by the constraints imposed by the underlying interconnect and physical layer. Due to this limit, this metric also corresponds to the actual number of chunks that the middleware divides the user buffer into, before transferring its contents over the network.

8.1.2 Properties of the Individual Data Transfers

The data-transfer completion semantics Most modern interconnects support non-blocking transfers of data between the local and remote memories. The latency due to such remote transfers may therefore be overlapped by the available computation. This ensures efficient use of CPU cycles. Without support for asynchronous transfer by the underlying hardware, these CPU cycles would instead be invested in busy polling in order to track the completion status of the communication. The use of non-blocking transfers however, comes with the price of: (a) having to manage multiple communication handlers within the runtime, and (b) the possibility of having the count of the number of in-progress transfers exceeding the capability of the hardware. As we discuss later, the overhead of software management of this high count of asynchronous calls lead to an increase in the participation of the CPU, thereby raising the potential energy consumption per byte of data transferred.

The contiguity of the data-buffers in memory While handling small-to-medium-sized transfers, an application developer or the PGAS implementation itself may exploit the peak bandwidth of the underlying interconnect by merging multiple non-contiguous source buffers into a single contiguous chunk before sending the contents across the network. This technique is well established among PGAS implementations that support strided, indexed, or vectorized transfers[118]. However, one has to be wary of the latency and the energy cost associated with such mechanisms due to (a) the impact of local *memcpy()*s which are CPU and DRAM intensive, and (b) the maximum achievable bandwidth of the underlying interconnect. The benefits therefore depend on the extent of hardware support and the amount of computation available for overlapping the latency associated with bulk transfers.

The registration status of the source buffers with an RDMA-capable NIC PGAS implementations built on top of OS-bypass mechanisms require the virtual-to-physical address mapping to be pinned-down. This pinned region is registered with the NIC to enable RDMA-based accesses. If the application programmer uses a source buffer that is not pinned to the memory, a PGAS implementation typically performs a local copy of the contents of the buffer to pre-registered memory locations[2]. As shown in further sections, such local memory copies are CPU and DRAM intensive and their cost is proportional to the size of the copied contents.

[2]An alternative to performing local memory copy operations is to dynamically register memory locations. However this is a very expensive operation [107, 156] and is used while handling only bulk-sized buffers. The study of the impact of this metric is out of scope of this work.

8.2 Code Transformations that Impact Energy Consumption

In order to evaluate the impact of the elimination of cost factors discussed in the Section 8.1, we designed a number of microbenchmarks simulating different possible data access patterns using one-sided constructs. We discuss these next. We then present a list of transformations from one pattern to another. These help eliminate the different cost factors discussed before.

The results presented in this work are intended to motivate such transformations using either static or dynamic approaches. It must be noted that in real-world applications, the feasibility of switching such transformations would be constrained by a number of other factors such as data dependencies, algorithm design, the memory model, the communication model, etc. The discussions here and the empirical results in Section 8.3 are therefore aimed at aiding the reader in obtaining an optimistic estimate of the potential energy savings.

8.2.1 Design of Data-access Patterns

In order to design multiple data-access patterns within a communication kernel, we needed to identify a set of design elements, based on which any one-sided communication-intensive pattern may be modeled. These "design elements" correspond to different phases over which a remote transfer may be built upon.

Design Elements RDMA Write operations (or PUTs) in PGAS implementations may be divided into the following basic phases:

$P(x)$**:** This operation corresponds to the initiation of a one-sided WRITE operation, of x bytes, from the user address space of the active, sender process to that of the passive receiver. A call to this function does not guarantee completion of the data transfer. For an RDMA-capable interconnect solution with kernel-bypass capabilities, this operation can be serviced without the intervention of the CPU. This is typically achieved by using a pinned-down memory segment as sender and receiver buffers on either endpoint of the communication. This pinning of memory with the OS corresponds to the registration of the memory location with the NIC, thereby eliminating the need for CPU participation. From the point of view of an OpenSHMEM developer, this corresponds to a call to *shmem_putmem()* where both the sender and the receiver addresses correspond to a portion on the globally accessible "symmetric" memory.

Q**:** This phase typically corresponds to a polling operation which guarantees completion of previously posted PUT (P) operations. In terms of OpenSHMEM terminology, this corresponds to a call to *shmem_quiet()* that returns back the control to the user after ensuring that the data payload of all previously posted PUT operations have been copied over to the destination buffer at the receiver.

M**:** This phase corresponds to the preparation of a user buffer before initiating an RDMA operation. This involves memory management tasks like copying the contents of user buffers to pinned-down memory buffers. In case of OpenSHMEM, the communication model does not mandate that the source

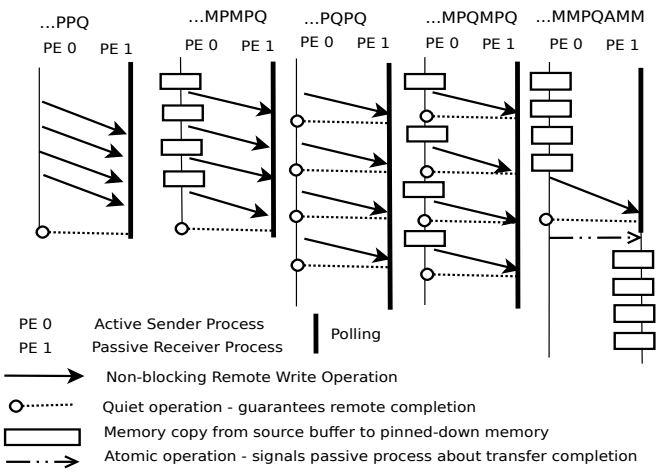


Figure 8.1: Line Diagrams of data-access patterns

buffer to be a symmetric object. As a result, this operation is typically performed by the underlying implementation. We mimic this, within our benchmarks, using simple function calls to *memcpy()*.

A: This phase corresponds to an atomic operation to signal the occurrence of a particular event to the remote passive process. In terms of OpenSHMEM, a call to *shmem_swap* achieves such semantics.

Examples We discuss some examples of data-access patterns that were designed as a combination of the basic elements discussed above[3][4]. Figure 8.1 illustrates these patterns, the impact of which, are discussed later in Section 8.3:

[3]A note on the nomenclature used: A repetition of a substring in each pattern name corresponds to a discrete user buffer. e.g. Each 'MP' in ...$MPMPQ$ corresponds to operations over a different fragment at successive addresses in the heap. The actual count of this repetition i.e. the number of fragments, corresponds to the number of disjoint user buffers over which the access pattern operates.

[4]A note on the design of the microbenchmarks: Obtaining steady energy readings require running the synthetic microbenchmarks for large number of iterations. To avoid a data-access pattern from falling prey to caching effects from past runs, it is essential to clear the contents of the cache before the start of each iteration.

...PPQ **:** Having multiple consecutive PUTs followed by a single *quiet* takes into account the overhead of maintaining multiple handlers for non-blocking PUTs, followed by polling for their remote completion.

...$MPMPQ$ **:** Having every non-blocking PUT be preceded by a memory memcpy operation from the buffer in the user address space to a "symmetric" buffer, takes into account the additional memory management involved while using non-registered source buffers. After all the memory copies and the PUTs, this pattern ends with a single *quiet* operation, thereby accounting for the costs of remote completion of all the transfers.

...$PQPQ$ **:** Having every PUT be immediately followed by a *quiet* takes into account the impact of using multiple blocking WRITE operations.

...$MPQMPQ$ **:** This benchmark represents the worst case among all patterns - a combination of all the cost factors described above. Each PUT operation is preceded by a *memcpy()* and is followed by the *quiet* operation.

...$MMPQAMM$ **:** To study the impact of packing data contents on the sender's side and unpacking them on the receiver's end, a data-access pattern may be modeled by incorporating the costs associated with multiple *memcpy()s* at the sender's side (to copy data from the user-buffer into a pinned-down source buffer), the actual transfer of the buffer contents to the remote process (using a single PUT), checking for remote completion of the transfer by the sender (a single *quiet*), signaling the completion of the transfer to the receiver process (a single *atomic* operation), polling for the completion-signal by the receiver,

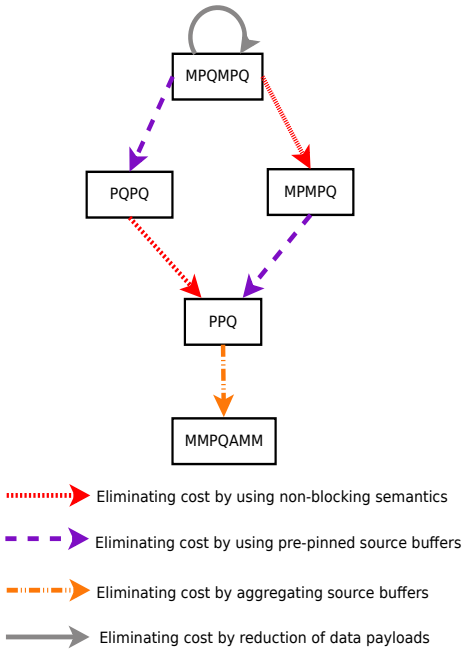


Figure 8.2: Different transformations of remote data-access patterns, that have the potential of energy savings within communication-intensive application kernels.

and, copying back the contents from the communication buffer into the final destination buffer at the receiver's end (using multiple *memcpy()s*). It must be noted that unlike the above patterns where the number of PUTs is equal to the number of user buffers, this pattern contains a single PUT that follows as many memory copy operations as the number of discrete buffers that the data payload spans across.

8.2.2 Transformations of access Patterns

Figure 8.2 illustrates the set of microbenchmarks that were evaluated and the relation between them. The edges connecting the nodes of the graph depict different code transformations, the impact of which, are discussed in Section 8.3. We study the

impact of four different OpenSHMEM code transformations that take into account the factors discussed in Section 8.1. Using the nomenclature above, we describe these transformations below:

Impact of using pinned source buffers: The impact of using unpinned source buffers and additional *memcpy()* operations may be studied by using globally visible or symmetric user-buffers, i.e. $MPQMPQ \rightarrow PQPQ$ and $MPMPQ \rightarrow PPQ$.

Impact of using non-blocking remote transfers: The possible cost-savings associated with converting blocking remote write operations to non-blocking ones may be studied by eliminating unnecessary calls to *quiet* after each PUT, i.e., $MPQMPQ \rightarrow MPMPQ$ and $PQPQ \rightarrow PPQ$.

Cost of aggregating user-buffers: The impact of using multiple memory copy operations to aggregate data into pinned memory instead of explicitly issuing multiple PUTs may be evaluated through the transformation $PPQ \rightarrow MMPQAMM$.

Cost of reducing the data-payload: While all the above transformations are dependent on the characteristics of the data-access pattern within a communication kernel, this transformation deals with the the size of the data-payload as dictated by the input problem size. The impact of the data-payload size can be analyzed by studying the same test-case – $MPQMPQ$ – with different transfer-payload sizes.

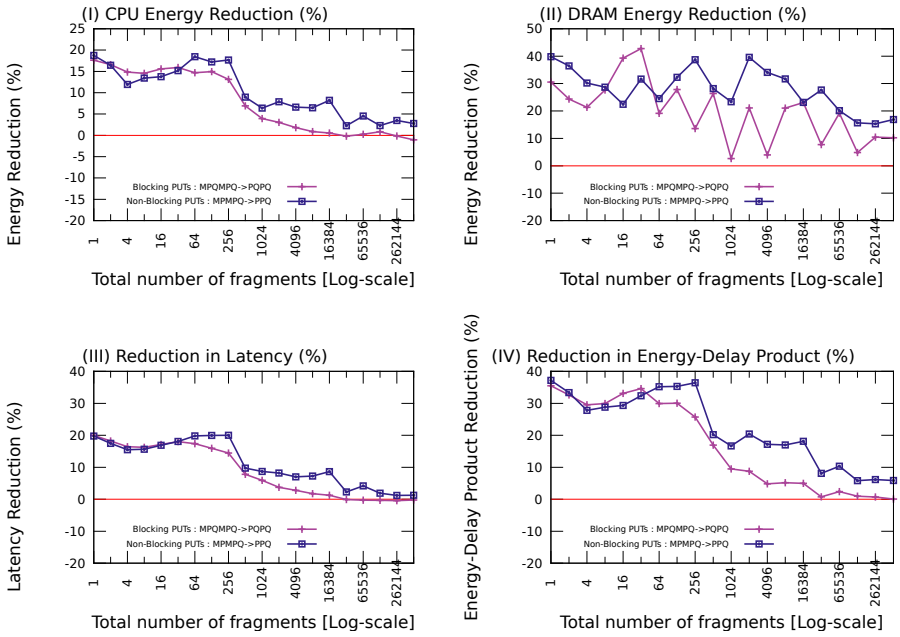


Figure 8.3: Impact of using pinned data buffers : Data-payload size = 0.5MB

It must be noted that many other transformations and their combinations such as $MPQMPQ \rightarrow MMPQAMM$ are also possible. However, since our scope lies on studying each of transformations independently, we do not discuss such cases here which account for multiple cost factors. Their impact may be compounded over the affect of multiple transformations listed above.

8.3 Empirical Results

While one of the primary purposes of this work is to discuss the potential savings achievable by transformation of a certain data-access pattern to another, it is essential to understand the behavior of independent patterns themselves. This can be achieved by analyzing the energy costs, latency, message rate, and bandwidth

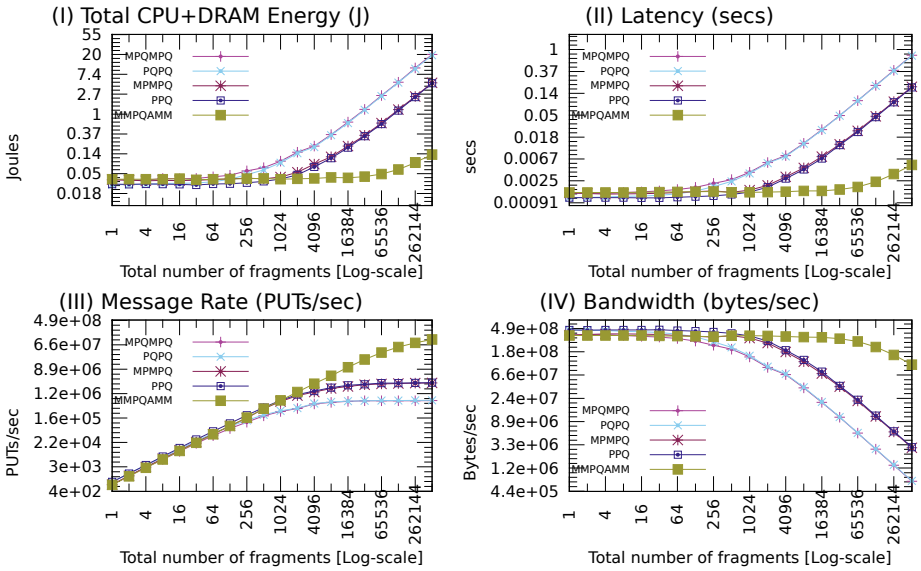


Figure 8.4: Impact of use of various data-access patterns on the CPU+DRAM energy and the achievable latency for a remote PUT operation w.r.t. number of explicitly initiated transfers : Total Data-payload size = 0.5MB

of each of these patterns. Figure 8.4 illustrates these transfer characteristics of the communication patterns while servicing a 0.5MB remote write operation using different number of PUT operations (represented as "#Fragments" on the x-axis). We observe that access-patterns with different blocking semantics have significant change in energy and performance traits beyond 256 PUTs (<2KB/PUT). Communication using blocking semantics ($MPQMPQ$ and $PQPQ$) have the highest energy and latency cost. This is accompanied with a lower bandwidth and message rate. This trend can be attributed to the penalty associated with polling-based operations and additional memory management necessary to ensure remote completion of the transfers. This impact on the energy and performance is reduced due to the use of non-blocking semantics ($MPMPQ$ and PPQ). With the number of PUTs greater

than 256 (<2KB/PUT), aggregation of data buffers ($MMPQAMM$) lead to minimal energy consumption and latency, and a sharp rise in the bandwidth and the message rate[5].

Moreover, for all data-access patterns except $MMPQAMM$, the message-rate becomes limited beyond 64 PUTs (<8KB/PUT) and is accompanied with a steady drop in the bandwidth of the transfer. This corresponds to the software overhead due to multiple explicitly initiated PUTs.

While these raw values provide an overview of the behavior of the data patterns, they do not present a fine-grained insight into the impact on the CPU and the DRAM. Moreover, they do not present a clear indication of the potential savings due to the factors discussed in Section 8.1. To address this, we present a detailed study using the transformations discussed in Section 8.2. These are shown in Figures 8.3, 8.5, and 8.6. These figures illustrate the impact of the transformations on various cost metrics.

In our case, the cost "metrics" studied are:

- Energy consumption by the CPU
- Energy consumption by the DRAM
- Latency of the transfer
- EDP or Energy Delay Product[6]

[5]In case of $MMPQAMM$, all the buffers are serviced by a single PUT operation, irrespective of the number of user buffers. Thus the metric - *message rate* corresponds to *#User-buffers/sec.*

[6]While CMOS circuits have the ability to trade performance for energy savings, it becomes

The "impact" of each cost metric is calculated in terms of the percent *reduction* in one of the above metrics. If a transformation T is applied on a data-access pattern $C_{initial}$ such that: $T(C_{initial}) \rightarrow C_{final}$, then the impact of T in terms of percent reduction in a cost-metric M may be calculated as:

$$I = \frac{M(C_{initial}) - M(C_{final})}{M(C_{initial})} * 100$$

For all of these experiments, the graphs depict the values of various metrics as measured at the compute node servicing the active sender processes responsible for initiating the remote write operations. We restrict our discussion to study the behavior of this process and not the passive receiver process.

It must be noted that the energy consumption of a passive process that's polling at a barrier, waiting for the completion of a transfer, cannot be ignored while performing large scale studies of distributed applications. In fact, our past study[83] indicates that the energy consumption increases proportionally with the time and its scale is very high. However, since the polling activity corresponds to a constant power consumption, it can be safely ignored in the following discussions that focus on the impact due to the remote data-access patterns.

8.3.1 Impact of Using Pinned Buffers

From Figure 8.3, we observe that the impact on (or, the percent of reduction in) the CPU energy consumption and the latency is as high as 20% in case of bulk

challenging to optimize for both simultaneously. The EDP, first proposed by Horowitz[57, 69], takes into account both the energy and the time costs in an implementation-neutral manner. For cases, where energy and performance have equal importance, this metric can be calculated as a product of the energy consumed and the time taken. For more complicated cases, where performance is given a higher priority, the weight of the "delay" factor is increased by squaring or cubing it[99].

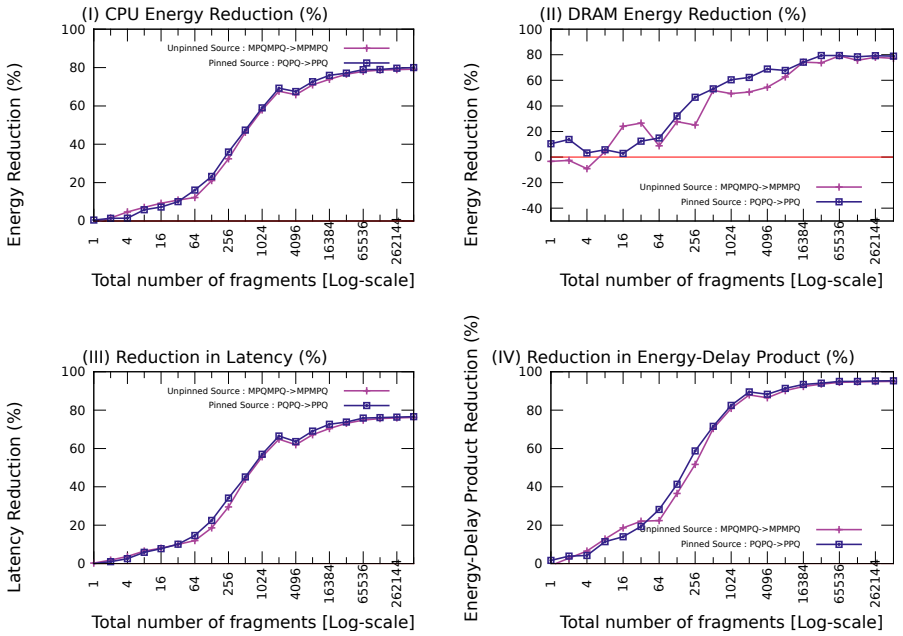


Figure 8.5: Impact of transforming multiple blocking operations to non-blocking

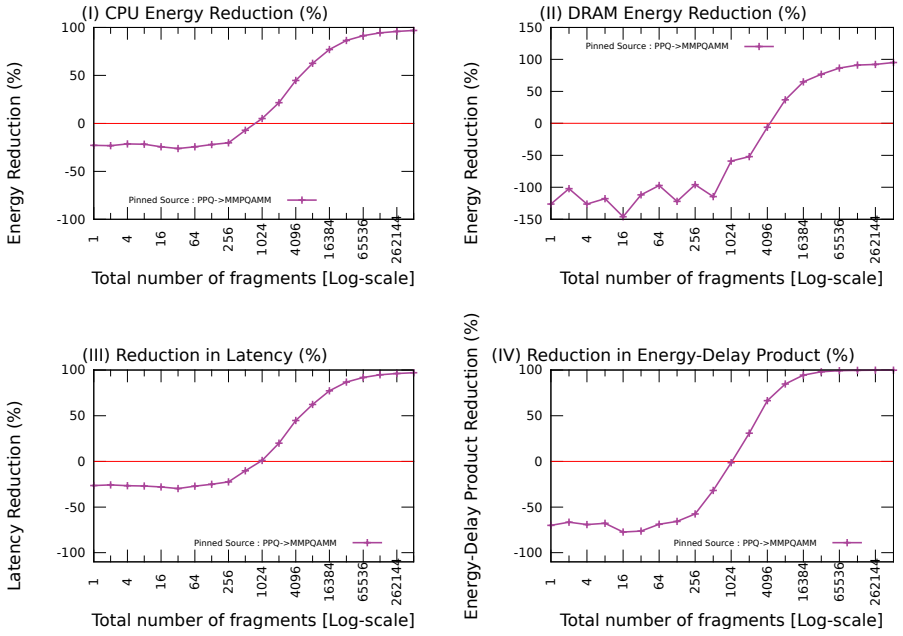


Figure 8.6: Impact of aggregation of multiple data buffer

transfers. This is not surprising, as this type of transformation results in elimination of unwanted memory copy operations of bulk buffers, which directly benefit the energy cost and the latency. This elimination of bulk memory copy operations leads to DRAM energy savings as high as 40%. The impact of this transformation however drops to less than 5% in terms of CPU energy and almost zero in case of latency. This downward trend is observable especially when the number of PUTs increases beyond 512 (i.e., buffer size $< 1KB$ per PUT). This condition corresponds to the case where the CPU energy cost of small memory copy is negligible and the message rate reaches its threshold (see Figure 8.4).

Influence of other cost factors: We see that there exists a variation in the impact based on the blocking semantics of the PUT operations. The plots in the figure depict these two possible outcomes as "Blocking PUTs: $MPQMPQ \rightarrow PQPQ$" and "Non-Blocking PUTs: $MPMPQ \rightarrow PPQ$". The primary observation with regards to the DRAM energy consumption is that there is an overall lesser impact of this transformation on blocking PUTs when compared with non-blocking PUTs. An interesting observation is the oscillating trend in DRAM energy savings in case of using non-blocking PUTs. This was surprising because the total size of the data payload being handled across all the data points remains constant (0.5MB) and the primary source of DRAM energy savings in this transformation is the elimination of local memory copy operations. We are currently investigating the reason for this trend.

8.3.2 Impact of Using Non-Blocking Remote Transfers

From Figure 8.5, we observe that the impact of replacing blocking transfers by non-blocking versions is significant in terms of reduction in CPU energy, latency, and the energy-delay product. As shown, the positive impact on the energy and the latency rises with an increase in the number of discrete PUTs and hits a limit (80%) when this count rises beyond 256. This can be attributed to the fact that the benefits of launching multiple non-blocking transfers is overshadowed by the cost of ensuring completion of these transfers (during the *quiet* operation). The benefits on the energy-delay product is significant. The 80% reduction in CPU energy and latency corresponds to an improvement in energy delay product by almost 95%.

Influence of other cost factors: We see that there is very little difference between the cases corresponding to using pre-registered and non-registered data buffers. The plots in the figure depict these two possible outcomes as "Unpinned Source: $MPQMPQ \rightarrow MPMPQ$" and "Pinned Source: $PQPQ \rightarrow PPQ$".

8.3.3 Impact of Aggregation of Buffers

Unlike other data-access patterns discussed in the text, an access pattern similar to $MMPQAMM$ corresponds to an active participation by both the sender and the receiver. Moreover the RDMA-based data transfer is limited to a single PUT operation. The cost associated with handling multiple user buffers is dependent solely on the cost of local memory copy operations.

Figure 8.6 depicts the impact of converting a pattern like PPQ to $MMPQAMM$.

The observations are described below.

For bulk transfers (#Fragments≤1024): For this case, a pattern like PPQ is characterized by few bulk PUTs directly from the source buffer to that target. The transformed pattern $MMPQAMM$ is characterized by bulk local memory copy operations first, by the sender, and then, by the receiver. The latter pattern significantly raises the CPU and DRAM energy consumption. Moreover, this phase corresponds to the peak bandwidth achievable using PPQ. Thus, we see a negative impact on the energy metrics (about -25% on CPU and -125% on DRAM) and the latency (-25%). This negative impact amortizes any potential energy savings achievable through the use of a single bulk blocking PUT.

For small transfers (#Fragments>1024): We observe that the overall CPU and DRAM energy savings achieved using this transformation increases with the count of discrete source data buffers (fragmentation). The high energy savings in PPQ may be attributes to not only the obvious elimination of the software overhead (associated with servicing multiple PUTs and in-progress transfers) but also its limiting message-rate and dropping bandwidth (Figure 8.4).

To summarize, we learn that with no sufficient overlap for transfers of bulk-sized buffers, initiating a non-blocking RDMA operation does not yield much benefit. Additionally, we observed that despite data-access patterns like aggregation having a positive impact on energy and performance based metrics for large number of small-sized user buffers, its adoption for bulk-sized messages becomes inefficient. Similarly,

enforcing the use of pinned-down memory for small-sized user buffers is not beneficial.

8.4 Chapter Summary

In this work, we established the notion that the design of data-access patterns play a critical role in impacting the energy profiles of communication-intensive PGAS applications. We investigated a number of factors that affect the energy cost of a process initiating a remote data transfer. These include – the contiguity of the data buffers in the memory, the total size of the payload being transferred, the registration status of the source buffers, the completion semantics of the data transfer operations, etc. For a fixed size of data-payload that is transferred to a remote process, the extent of impact of these factors depends on the number of explicitly initiated data transfers.

We investigated the impact of different pattern transformation techniques on the energy and performance characteristics of communication intensive kernels, using: registered memory buffers (up to 40% EDP savings), non-blocking operations (up to 97% EDP savings), and aggregation of source buffers (-70% to +98% EDP savings). Some of the lessons learned include -(a) Energy savings achieved by using pinned-down source buffers reduces with a rise in the number of explicitly initiated PUT operations, (b) Energy savings due to the use of non-blocking semantics is higher for smaller sized transfers; such savings hit a limit due to additional overhead of management of multiple outstanding remote transfers, (c) Aggregating bulk-sized buffers into contiguous memory locations has a negative impact on the energy savings, the latency and the energy-delay product. Using multiple smaller transfers tend to benefit significantly in terms of such savings.

For more details about the topics discussed in this Chapter, the interested reader is directed to the literature documented by Jana et al. under [85].

Chapter 9

Synchronization: Scale and Time

This chapter presents an empirical analysis highlighting the effect that unbalanced workloads have on the energy consumption by processes using synchronization constructs. We study this impact on the energy costs in terms of two factors - the cost incurred by processes waiting for different time periods within a barrier, and the cost incurred by the entire system with a rise in the number of processes participating in a barrier.

9.1 Synchronizing Time

The line chart and the code snippet of the microbenchmark used to verify this is presented in the first row of Table 9.1.

Figure9.1 illustrates that a linear growth in the time spent by a process within a barrier leads to a linear rise in total energy consumed by the system (cores and the

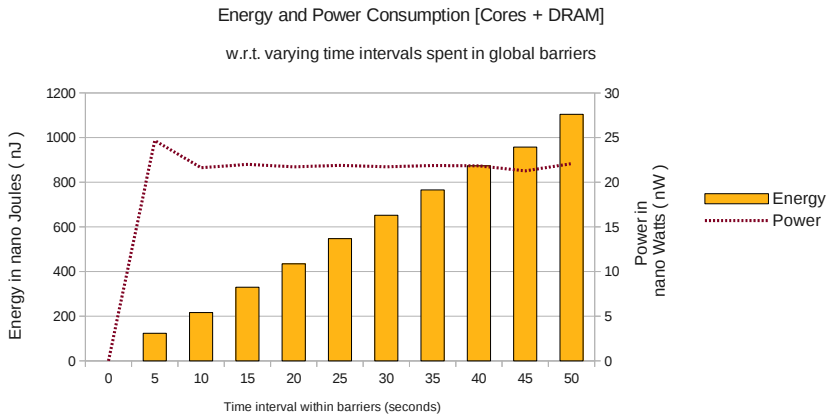


Figure 9.1: Impact of wait period within a barrier

DRAM)[1]. In addition, we also observe that the power consumption or the rate of change in energy, is independent of the time spent by a process waiting at a barriers.

9.2 Scale of Synchronization

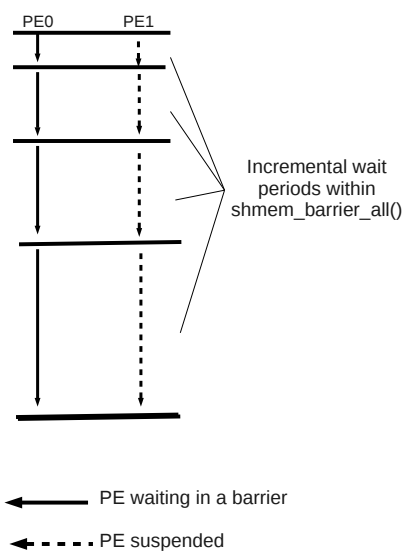
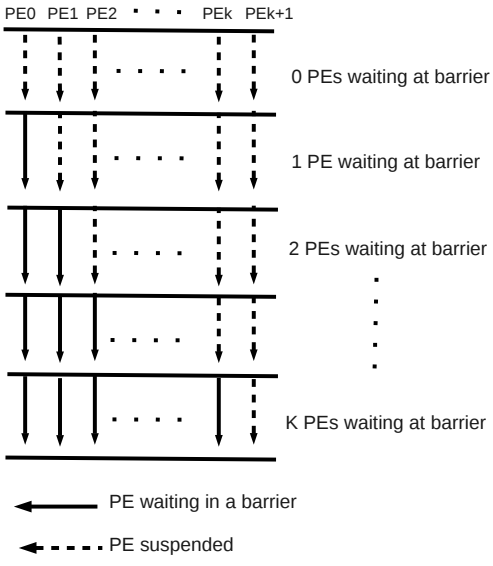
The line chart and the code snippet of the microbenchmark used to verify this is presented in the second row of Table 9.1

The results depicted in Figure 9.2 verify the claim that an increase in the number of processes waiting at a barrier leads to a linear rise in the energy consumed over the entire system which, in turn implies a linear rise in the average power consumption.

It is only when a semaphore signals the end of the barrier, that the CPU executes a code fragment that prepares the process to exit the barrier region. In accordance with this design, Figure 9.3 depicts the change in the energy and power consumption pattern with respect to the types of instructions executed by the CPU. The waste in

[1]For our experiments, the linear relationship between the energy (E) consumed and the time (T) spent within a barrier was: E = (33.1446*T)-1.88467. Unsurprisingly, the model had a high Coefficient of determination (r^2=0.999027)

Table 9.1: Line charts for studying the impact of barrier on energy and power costs

Varying time spent in barrier	Varying number of processes
	

CPU cycles can be observed by the linear rise in the difference between the number of conditional-branch instructions that are 'taken' and 'not taken'.

Also, the high correlation between the *total* number of instructions executed and the total number of *conditional-branch* instructions hint at the execution of the same set of instructions, irrespective of the time spent in the barrier. This *homogeneity* in the instruction types result in a constant power consumption by the system (Figure 9.1).

It is in case of the latter, that a process can be characterized as consuming energy

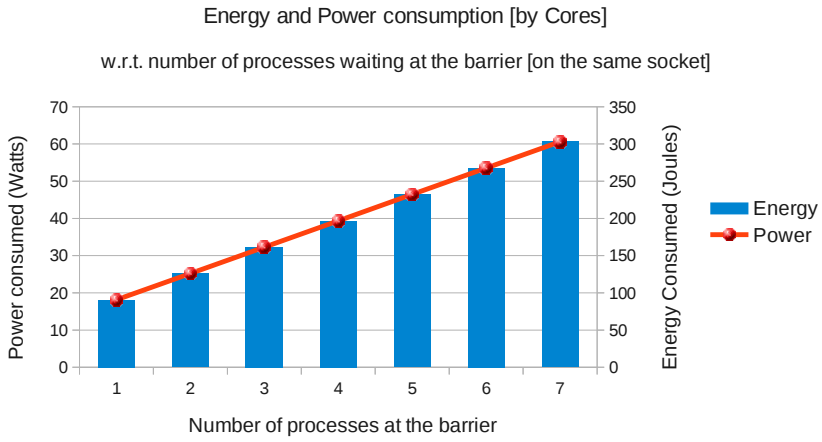


Figure 9.2: Impact of number of processes participating in a barrier

(wasting CPU cycles) without making any progress in the application execution. We focus on applications with unbalanced load, wherein the difference between the time to reach a barrier point may be attributed to processes following different control flow paths during their execution. It must be noted that the time taken by the processes to reach a barrier-point may be different despite following the same execution path. However, this observation may be attributed to a number of non-application specific factors like irrelevant system noise, the topological mapping of the processes on a large scale system, etc.

9.3 Chapter Summary

This chapter introduces the basic notion that the use of synchronization constructs is one of the most common factors that lead to a reduction of CPU utilization and waste of energy consumed by the system.

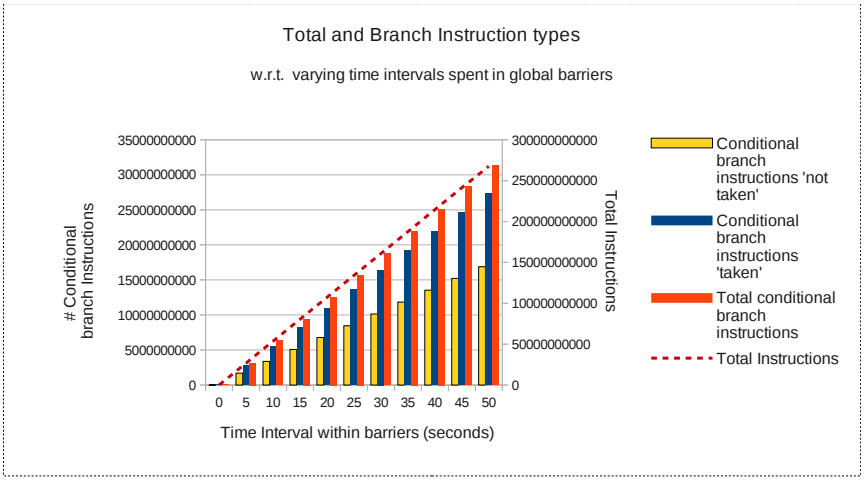


Figure 9.3: Comparing the types of instructions executed by the CPU while waiting at a barrier. The count includes (i) Total number of instructions (ii) Number of conditional branch instructions (iii) Number of conditional branch instructions that are 'taken' (iv) The number of conditional branch instructions that are 'not taken'

Chapter 10

State-of-the-Art: Using DVFS

From the past chapters, we learned that communication and synchronization constructs contribute to the energy consumption of applications that run on distributed systems. We also learned that the extent of impact of such parallelism constructs depend on the data-access pattern and distribution of work load among processes. In the following chapters, we introduce energy saving opportunities in case of applications with unbalanced workloads. More specifically, we target OpenSHMEM applications that heavily rely on extensive use of synchronization constructs in order to maintain a consistent view of the global memory.

The implementation of synchronization constructs, as dictated by the execution model, results in processes having to wait for significant time periods without making any apparent progress within the application. While this is essential to establish a consistent view of the distributed global memory, this leads to waste of computational resources, thereby contributing to a rise in the energy consumption by an application.

This *wait period* can be treated as excess *slack* within a process.

In this chapter, we present an overview of the past attempts at reducing the energy costs associated with *slacks* and the challenges that arise while using such approaches.

10.1 State of the Art

Reduction of Slack: There exists a vast breadth of the work tackling energy concerns with slacks in processes. The notion of 'slack' across all approaches have been the same - short time-bursts within the application where the processor wastes multiple cycles without making any progress into the application which, in turn leads to energy hot-spots. Based on the design of the application, the duration for the occurrence of these 'phases' may range from anywhere from a few nanoseconds to as high as seconds.

Examples of such slack include the delay in application progress due to pipeline stalls and cache misses. Such slack typically appears around loops and branches. There have been many research efforts in alleviating such slack and significant energy savings have been achieved on simpler architectures.

Past efforts using DVFS techniques have focused on the application phase that is responsible for managing the operating frequency of the underlying platform. Such a phase is typically called a **Power Management Point** or **PMP**. On reaching a PMP, the software stack triggers a change in the operating state of the underlying hardware during the course of its energy-aware execution. This trigger may be initiated directly by the user process or indirectly via operating system interfaces.

In past research efforts, a naive technique for controlling the frequency has been

to statically insert a PMP at the start of every basic block. The challenge in altering the operating frequency of code regions is to avoid penalizing the total execution time of that regions, and by extension, the entire application. The code region that is a candidate for frequency scaling is commonly referred to as a **Power Management Region** or **PMR**.

The rationale behind this approach is to enable programmers and tools to control frequency settings at a finer granularity. This is especially important when trying to tune the hardware to map the application semantics to the hardware. This is not easy to analyze solely by the hardware. This method comprise of two main phases:

- Determine the frequency setting for a particular code-region. This phase includes accumulating all the information required to represent the dynamic behavior of the application. We refer to this as **Power Hint Points** or **PHPs**.
- Configure the hardware to make the actual transition to the new operating frequency. This is based on the information conveyed by the PHP. We refer to this as **Power Control Points** or **PCPs**.

10.2 DVFS efforts for serial applications:

To reduce energy consumption due to slack arising from CPUs stalling for completion of memory transfers, some have attempted to scale down CPU frequencies in order to reduce the number of cycles invested in waiting for completion of memory operations[74]. This translates to energy savings. However, the opportunities of

such savings on modern day processors with architectural features like a superscalar design, multi-issue pipelines, VLIW support, etc. is minimal.

In case of real-time or time-sensitive applications, slacks occur for cases where processes complete their tasks before the known deadline. This results in applications stalling without making any execution progress. Aboughazaleh et al.[3] target such types of applications and describe techniques that rely on an OS thread to periodically monitor the progress of the application and dynamically change the CPU frequency settings.

An alternative approach is to empirically model the execution time of an application in terms of CPU cycles and place a PMP every so many cycles and have an additional phase responsible for collecting application status with low overhead[2]. This method involves altering the voltage-frequency characteristics of the target processor while executing different code regions within an application. As discussed before, incorporating such DVFS techniques leads to significant overheads due to the calculation and the setting of new speeds. Aboughazaleh et al.[3] incorporate these into their model to determine the optimal speed of each code region. This model, however, assumed the knowledge of the worst execution time of each code region. While this can be determined for periodic real-time applications, its application to irregular applications remains a challenge. Nevertheless, where applicable, this approach ensures constant application progress with a significant reduction in energy consumption.

Alternatively, statistical approaches like Moose et al.[115] may be used to evaluate the feasibility of investing time in altering the voltage-frequency characteristics.

Aboughazaleh et al.[2] have provided a theoretical solution for calculating this overhead and inserting the power-management hints in a manner that minimizes its effect. The drawback, however, is that their approach assumes that the execution time of the application, when running at the maximum frequency, is known apriori.

10.3 Extending DVFS to Parallel Applications

One such effort that closely maps to this approach is that by Kappiah et al.[90]. Their work focuses on reducing the impact of slacks generated by MPI communication and barrier interfaces. However, their scope of power management is the main outer loop. Their approach relies on summing the slack time associated with every MPI function within an iteration and then calculating the ratio of this sum to the total iteration time. This slack is compared against a certain threshold to determine whether the frequency needs to be decreased for future iterations or not.

The drawbacks of this approach is as follows:

- Since the calculation of the slack and the adjustment of the operating frequency is performed once per iteration, the scope of exploiting slack over finer code regions is lost.
- The calculation of the 'gross' slack in essence spreads the effect of the slack period over the entire iteration. Thus, for cases where the slack is concentrated over only a specific region, distributing the slack across the entire iteration leads to a more inaccurate analysis.
- Moreover, after every iteration, this approach compares the slacks across all

the processes. It doesn't take into account cases of irregular workloads where only a group of processes cooperate with each other to perform a certain set of tasks.

Besides these shortcomings, past efforts remain oblivious to the semantics of various parallelism constructs. We address these shortcomings in the following chapters.

10.4 Types of Scaling

With regards to the location of a PCP, there are two main types of frequency scaling. One of the approaches is to place a PCP at the start of a PMR. In other words, choosing an operating frequency at the start of execution of a code region. Such an approach is called *Proactive Scaling*. This approach is primarily applicable for applications with PMRs that are characterized by a deterministic slack periods. Examples of such regions include body of iterative constructs like for loops. The advantage of this approach is that the impact of the change in the operating frequency affects not only the slacks but also the compute kernel surrounding them.

An alternative approach is to place a PCP at the start of a slack region. In order words, a change in the operating frequency is initiated only just before the CPU enters a slack. Such an approach is called *Reactive Scaling*. The goal behind this approach is typically to reduce the energy consumption of the system during a slack period. The advantage of this approach is that the runtime environment does not require a deterministic occurrence of slacks. This is specially applicable for irregular applications with unpredictable occurrences of slacks. The disadvantage is that any savings achievable through frequency scaling is distributed over a short time-frame.

Chapter 11

Challenges: DVFS for Eliminating Slack

11.1 Opportunities for eliminating slacks

As discussed before, a *slack* corresponds to an application phase that is serviced by system resources without making sufficient progress into the application. The extent to which the energy consumed during such phases is dependent on multiple factors listed below:

- *Compute Ratio (CR):* The ratio of the number of compute instructions executed by PEs reaching the barrier later to that executed by PEs waiting at the barrier. A higher value of this metric implies more load imbalance. A value of 1 implies uniform load distribution.

- *Early Core Count (EC):* The count of the number of PEs polling at the barrier at any given time

- *Compute Intensity (CI):* The ratio of the number of compute operations (arithmetic instructions) to memory operations (data-access instructions like loads and stores)

11.2 Proactive Scaling

11.2.1 Approach and Challenges

Proactive scaling techniques strive to use DVFS techniques over a code region with the goal of reducing the energy invested servicing 'slacks' contained in that region. With this scaling technique, the scope of a PMR [1] extends beyond the duration of slack phases. Figure 11.1 illustrates four cases of execution timelines of a pair of synchronizing PMRs with unbalanced workloads[2].

The first three cases in the figure highlight the opportunities of altering frequency settings to achieve a reduction in the slack within each process. The last case highlights a case where a poor choice of P-state may lead to a negative impact on the performance. We see that PE-0 on operating at a lower frequency might delay the termination of a global barrier. All four cases are described below.

- *Case A: Baseline Mode:* This corresponds to the initial condition where PE-0 enters a synchronizing barrier - slack period, before PE-1. As a result, it spends

[1] for definition of PMR, see Chapter 10.

[2] For simplicity, all the PMRs in the figure are assumed to start at the same timestamp and the slack phases are assumed to be aggregated at the end of the regions. This allows for a one-to-one comparison of the impact on computation and slacks during proactive scaling.

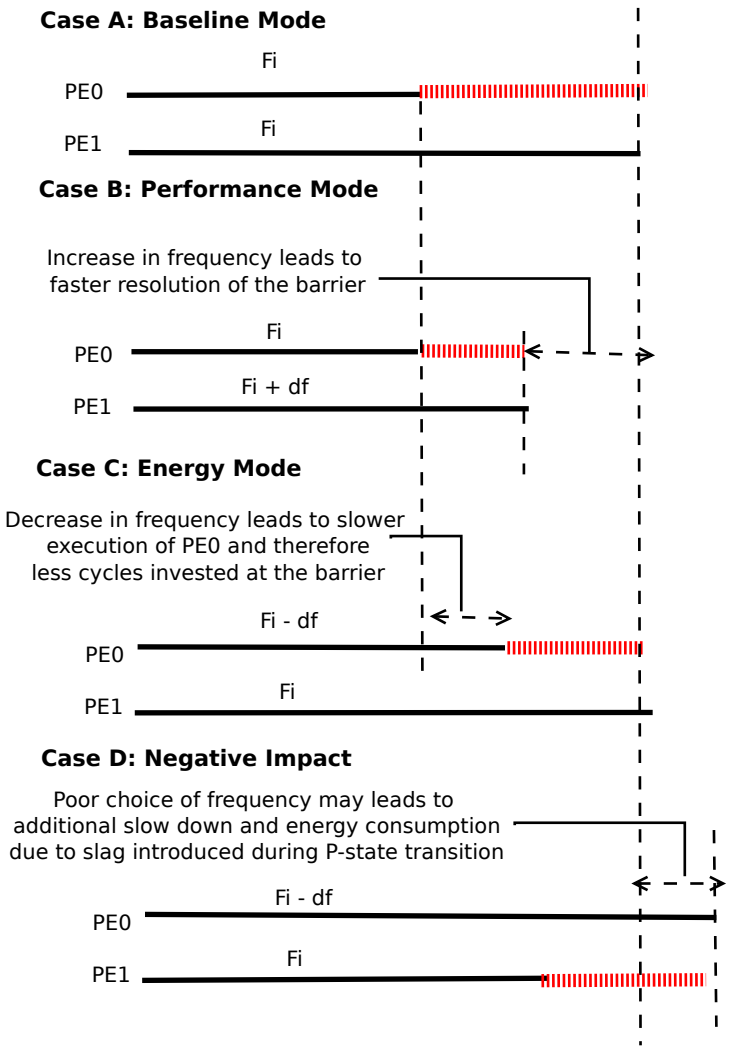


Figure 11.1: Different approaches of using Proactive Scaling for energy savings. The compute regions are represented with horizontal bold lines. The slack regions are represented with dashed red lines. Four possible execution timelines are represented: *(A) Baseline mode:* Both the processes, PE-0 and PE-1, operate at the same operating frequency; *(B) Performance Mode:* The operating frequency of PE-1 is boosted; *(C) Energy Mode:* The operating frequency of PE-0 is reduced in order to reduce the number of cycles wasted polling, which leads to energy savings; *(D) Negative Impact due to energy mode:* Depicts a case corresponding to a short slack period in which case operating in an energy mode adds additional overhead due to P-state transition. This affects performance.

considerable number of CPU cycles polling at a barrier, waiting for PE-1 to signal its entry.

- *Case B: Performance Mode:* This case corresponds to the approach where the speed (CPU frequency) of PE-1 is increased at the start of the PMR region. The hypothesis is that running the process with high compute load (in this case, PE-1) leads to an earlier resolution of the barrier. This in turn results in less wait time for the process with low compute load (in this case, PE-0), thereby reducing the energy invested polling at the barrier[3].

- *Case C: Energy Mode:*, if the frequency of the CPU servicing PE-0 is decreased at the start of the PMR, the number of cycles invested within the slack region decreases. This is also accompanied by a drop in energy consumed.

- *Case D: Negative Impact of operating in Energy Mode:* If the time-period of the slack region is too small or the compute intensity too high, a drop in operating frequency of the process with low compute load (in this case, PE-0) might lead to a later resolution of the barrier. This may introduce additional slack as shown thereby leading to a negative impact on execution time and energy.

11.2.2 Empirical study

Compute Intensity Over the past two decade, there have been multiple research efforts within the field of embedded processors, multi-core systems, and real-time memory-intensive applications that exploit DVFS approaches to achieve energy

[3]The extent of reduction in execution time is dependent on how compute-intensive the application is. If the execution segment is memory-bound, the speed of execution remains bounded by the speed of the memory hierarchy, which is unaffected by the CPU frequency.

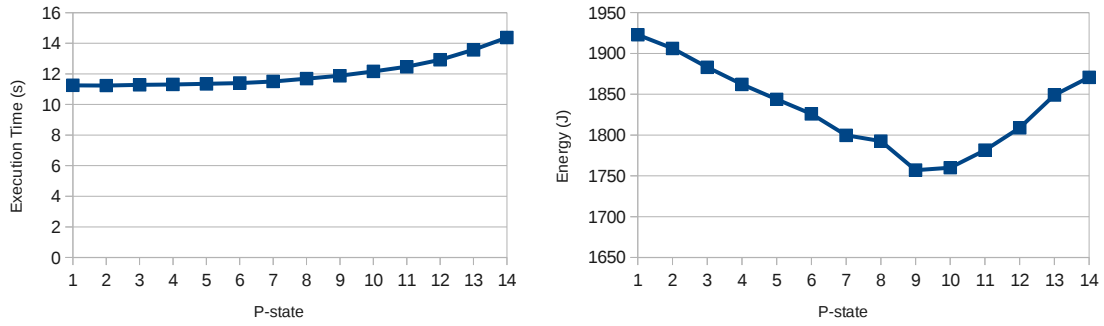


Figure 11.2: DVFS over STREAM COPY kernel. Compute Intensity (CI) = 1/2 = 0.50

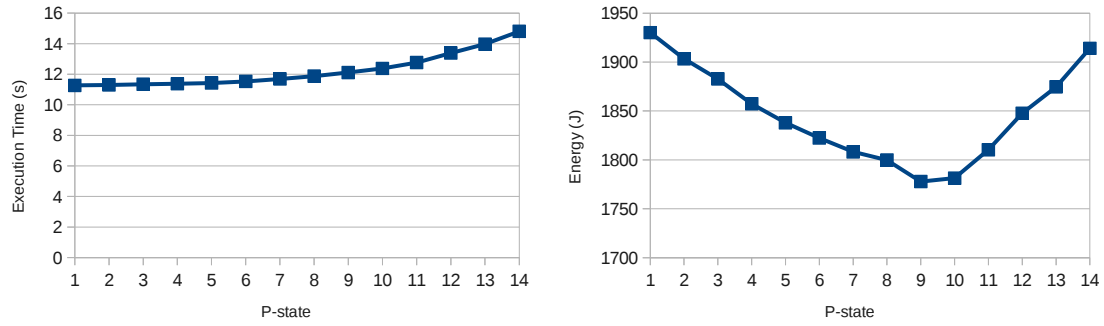


Figure 11.3: DVFS over STREAM SCALE kernel. Compute Intensity (CI) = 2/3 = 0.67

savings[2, 71, 41, 42, 92, 39, 86]. The primary hypothesis is to reduce the energy consumption during pipeline stalls that arise due to the differences in operating speed between the CPU and the memory hierarchy. This section discusses a microbenchmark-based study that evaluates the optimum CPU operating frequency for different memory and compute-intensive kernels.

Figures 11.2 through 11.6 depict the difference in behavior of different kernels operated at multiple CPU operating frequencies (X-axis). This "difference in behavior" is represented in terms of the coordinates of the minima of the curves representing

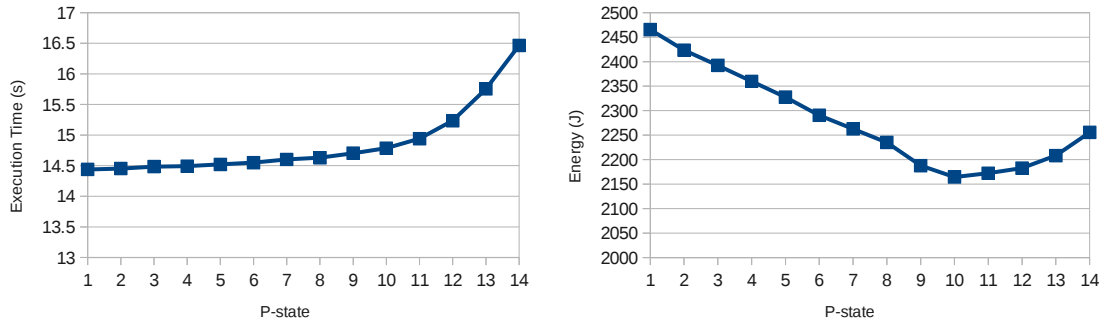


Figure 11.4: DVFS over STREAM ADD kernel. Compute Intensity (CI) = 2/3 = 0.67

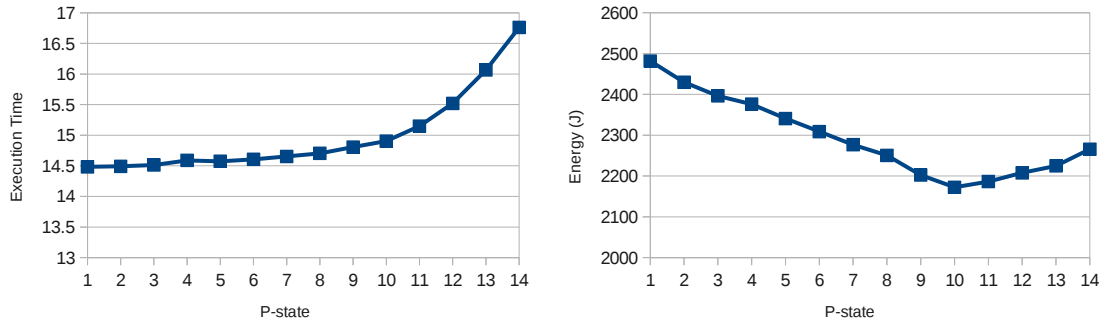


Figure 11.5: DVFS over STREAM TRIAD kernel. Compute intensity (CI) = 3/4 = 0.75

the execution time and CPU energy consumption. Figures 11.2 through 11.5 correspond to different kernels within the STREAM benchmark. Figure 11.6 corresponds to a hand-written compute-intensive kernel characterized with FMA operations using double-precision floating point data objects.

The fact that the CPU frequency corresponding to the minima for any kernel is different from the others, suggest that the optimum energy efficiency on a given processor is not always achieved at the highest operating frequency. Also, the observation that the energy and time curves are not proportional for all the kernels,

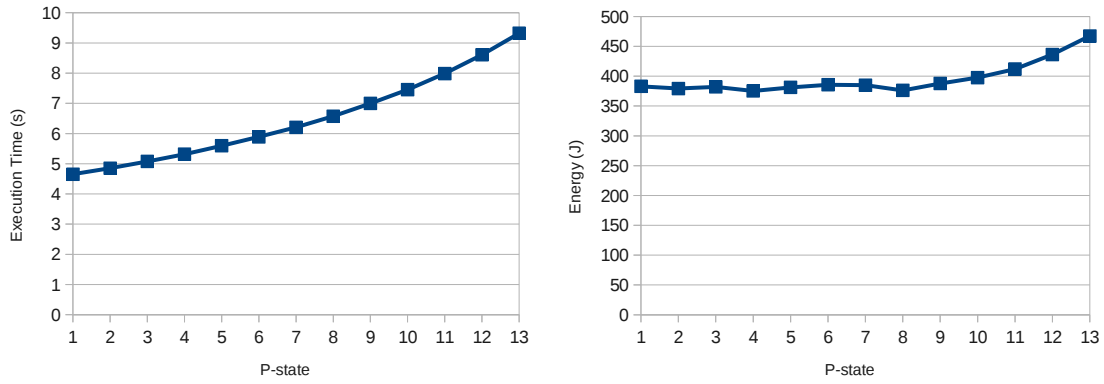


Figure 11.6: DVFS over hand-written Compute-intensive kernel. Compute Intensity (CI) ¿ 6

suggests that the trend in execution time and energy is not always proportional for all the kernels.

Compute Ratio and Early Core Count This section describes a microbenchmark based study that analyzes the potential savings in energy, time, and average power using proactive scaling. The line diagram of the microbenchmark is depicted in Figure 11.7. The vertical blocks corresponds to the progress of PEs all converging at a barrier. The blue-shaded portions corresponds to a computationally intensive region whereas the yellow-colored portions depict the time spent polling at a synchronizing construct like a global barrier. Two different variables are used to plot the energy metrics against - the *compute ratio (CR)* and the *early core count (EC)*.

From Figure 11.8a, we observe that for high compute ratio (large load imbalance, $CR > 1$) while using proactive scaling, operating in performance mode or hybrid mode leads to savings in time (as compared to baseline mode). The extent of savings depends on the actual operating frequencies used to execute the PEs. For $CR = 1$

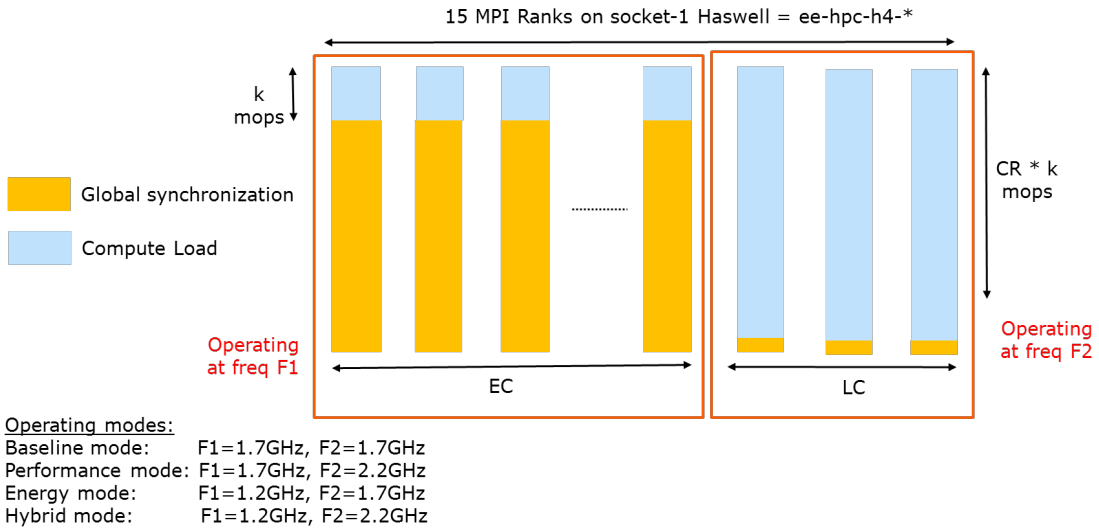
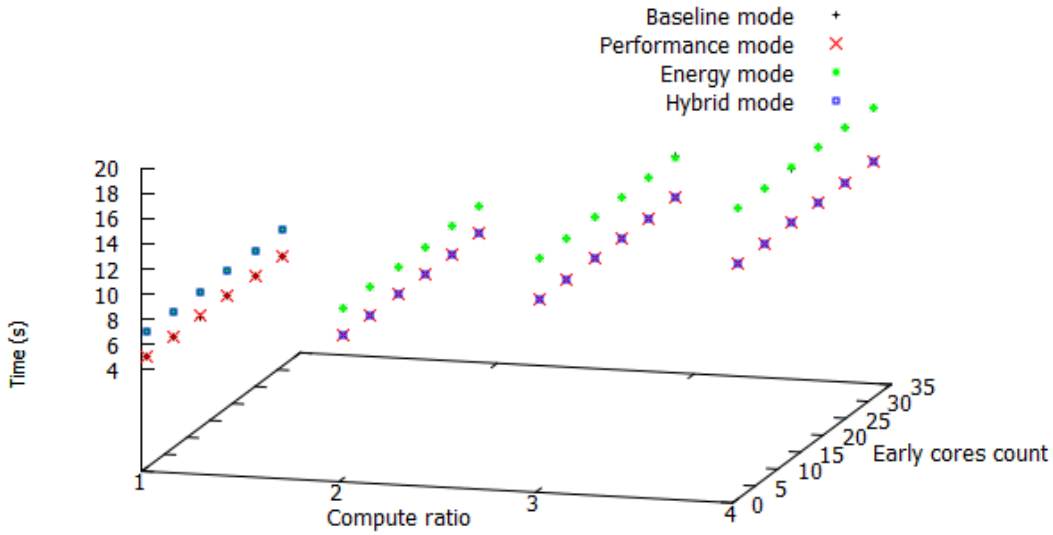


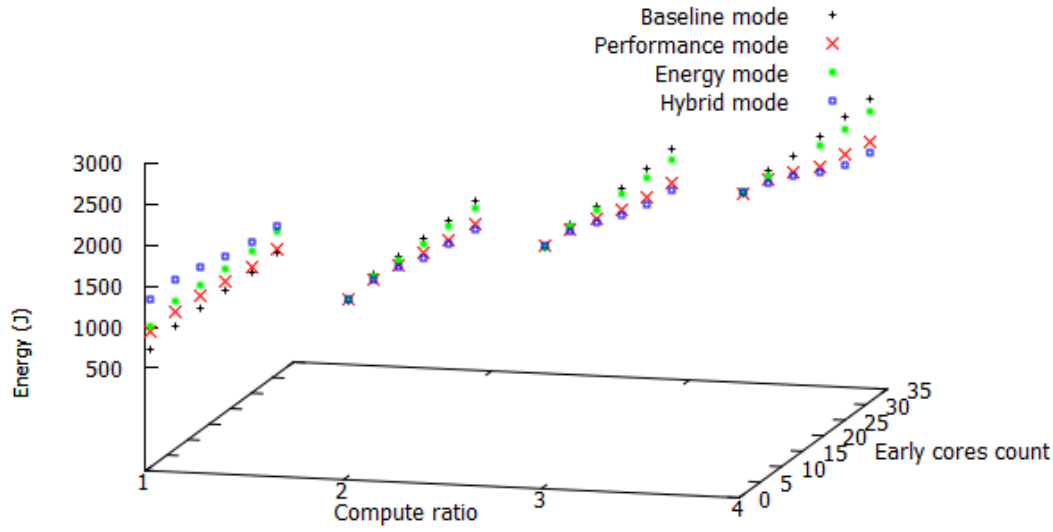
Figure 11.7: Line diagram for microbenchmark to evaluate the potential savings using proactive scaling

(zero load imbalance), we see that running in hybrid or energy mode leads to a drop in performance. This is because, attempting to reduce the frequency of even a single PE introduces an unwanted delay in the critical path, thereby hampering performance. With regards to energy consumption (Figure 11.8b), we observe that operating at performance and hybrid mode is more beneficial than energy mode for unbalanced load distribution ($CR > 1$). This gap increases further for higher compute ratio. Another major observation is that for a fixed CR, the impact of DVFS increases with the number of cores polling at the barrier (EC). This impact is reversed for CR=1.

To summarize, for cases with high load imbalance, the extent of energy savings using proactive scaling by boosting the frequency (performance/hybrid mode) is higher than simply reducing the frequency (energy mode).



(a) Execution Time



(b) Energy Consumption

Figure 11.8: Results of the microbenchmark based study on the impact on execution time and energy consumption due to proactive frequency scaling

11.3 Reactive Scaling

11.3.1 Approach and Challenges

Reactive scaling techniques strive to use DVFS on detecting a 'slack' phase within an application. In other words, for reactive scaling, the PMR region[4] corresponds to the 'slack' region within an application kernel. Figure 11.9 illustrates four cases of execution timelines of a pair of synchronizing PMRs with unbalanced workloads. They highlight the opportunities of altering frequency settings to achieve a reduction in the slack within each process. For simplicity, all the PMRs are assumed to start at the same timestamp. This allows for a one-to-one comparison of the different *slacks* accumulating at the end of each PMR. However, it must be noted that synchronizing PMRs may have different start timestamps.

- *Case A: Baseline Mode:* This corresponds to the initial condition where PE-0 enters a synchronizing barrier - slack period, before PE-1. As a result, it spends considerable number of CPU cycles polling at a barrier, waiting for PE-1 to signal its entry.

- *Case B: Performance Mode:* This case corresponds to the approach where the speed (CPU frequency) of PE-1 is increased at the same moment that PE-0 enters a barrier, thereby leading to a reduction in the time spent by the latter within the slack period [5]. This leads to energy savings.

[4] for definition of PMR, see Chapter 10

[5] The extent of reduction in execution time is dependent on how compute-intensive the application is. If the execution segment is memory-bound, the speed of execution remains bounded by the speed of the memory hierarchy, which is unaffected by the CPU frequency.

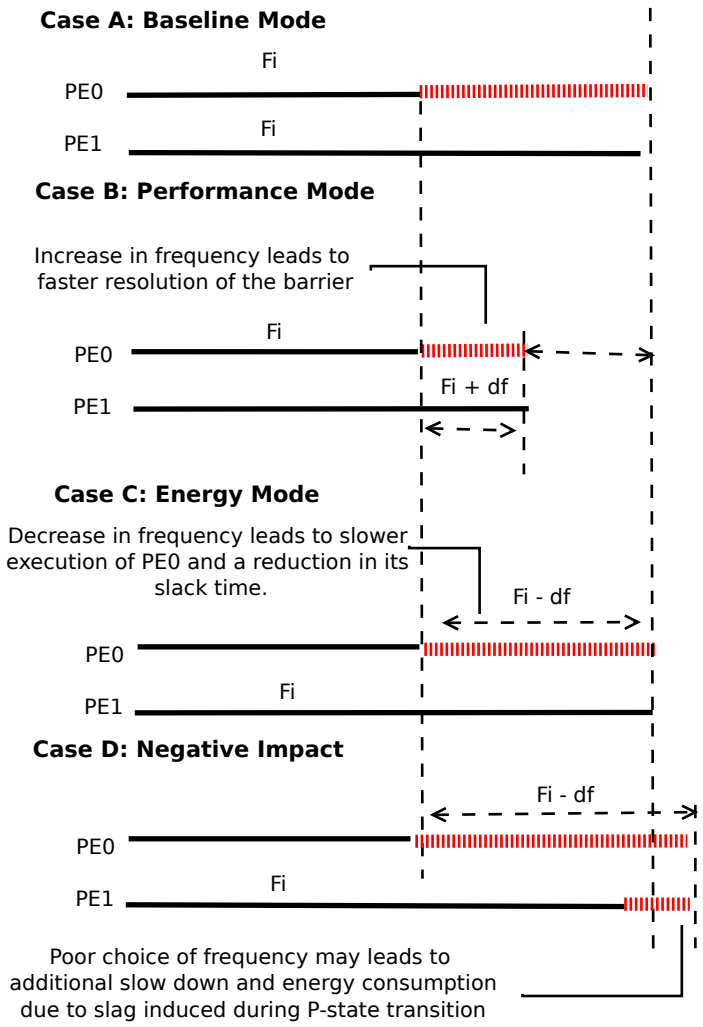


Figure 11.9: Different approaches of using Reactive Scaling for energy savings – *(A) Baseline mode:* Both the processes operate at the same operating frequency; *(B) Performance Mode:* The operating frequency of the process that reaches the barrier later, is boosted at the time when the other process enters a slack region; *(C) Energy Mode:* The operating frequency of the process that enters the slack region first is reduced in order to reduce the number of cycles wasted polling, which leads to energy savings; *(D) Negative Impact due to energy mode:* Depicts a case corresponding to a short slack period in which case operating in an energy mode adds additional overhead due to P-state transition. This affects performance.

- *Case C: Energy Mode:*, if the frequency of the CPU servicing PE-0 is decreased the moment it enters the barrier, the number of cycles invested within the slack region decreases. This is also accompanied by a drop in the power and energy consumed by the PE.

- *Case D: Negative Impact of operating in Energy Mode:* If the time-period of the slack region is too small, the time and energy costs associated with switching between the frequencies overshadows any potential savings achievable. As depicted in the Figure, this may lead to a slight delay in the barrier being resolved, thereby leading to performance degradation.

11.4 Chapter Summary

From the barrier experiments, we conclude that any interval of time spent by a process within a barrier not only impacts the performance but also the energy signature of distributed applications. With rise in the number of processes participating in the barrier, the total power consumed by the system increases as well. This motivates the need for exploitation opportunities of overlapping asynchronous communication operations with computation thereby delaying a process from entering a synchronizing construct.

While the observation is not surprising, it hints at the need to avoid global barriers whenever possible. Instead the use of point-to-point barriers should be practiced, especially when there is a likelihood of load imbalance among processes.

Chapter 12

Challenges: DVFS with Data Movement

As power consumption continues to be a major concern for exascale systems, research efforts have been directed towards using hardware and software codesign principles to achieve energy efficiency. In accordance to this, one common approach is to use frequency scaling capabilities of modern processors to achieve energy savings. This is commonly referred to as Dynamic Voltage Frequency Scaling (DVFS). Reducing the frequency allows the processors to operate at a lower voltage level thereby leading to energy savings.

In a one-sided point-to-point communication model, a single software agent (like an OS process) is responsible for managing the data transfer between itself and a passive agent (process). A common scenario in the design of PGAS kernels is for the passive process to rely on the completion of this data transfer in order to make

further progress into the application. The CPU servicing this process is therefore subjected to polling for a certain semaphore-based event that signals the completion of the transfer. At such a point, theoretically, energy savings may be achieved by scaling down the frequency of a CPU core that services the passive process. This time-frame is commonly referred to as 'slack period' and there have been multiple research efforts directed towards using DVFS techniques to reduce the energy consumption without significant performance impact[117, 52, 104, 106]. This chapter highlights the claim that this lack of "significant performance impact" during data movement in a distributed environment is heavily dependent on the underlying implementation. It presents empirical evidence that the extent of these savings depends on the implementation approach of one-sided communication interfaces. This analysis is presented with respect to the use of one-sided remote write (PUT) operations in OpenSHMEM[34], an SPMD-based PGAS model. This chapter discusses the potential impact on the energy and latency costs incurred by the sender and the receiver process[1] in an environment where the latter is serviced at a reduced CPU frequency. This chapter covers the following:

- Discussion of the challenges of using DVFS in a distributed environment (Section 12.2)
- Description of different cost factors within the software stack that affect the energy consumption and the performance of remote data transfers (Section 12.3)
- Discussion on common approaches of implementing remote PUT operations that have the potential of being affected by DVFS (Section 12.4)

[1] or in OpenSHMEM terminology, *'processing element (PE)'*

- An empirical analysis that presents the impact of using DVFS on the above approaches (Section 12.6)

The empirical analysis incorporates a fine-grained study of the energy consumption by the CPU and the DRAM servicing the sender and receiver processes. These readings were obtained using computational resources described in Section 13.4.2.

The results presented in this work should be useful to system programmers incorporating DVFS techniques in a distributed environment.

Note: In this chapter, the term *'Sender'* refers to an SPMD process that initiates an RDMA operation to access a remote data object, and the term *'Receiver'* refers to the process that owns the remote data object. No additional meaning is implied in terms of the extent of participation while servicing the data movement.

12.1 Related work

There have been multiple research efforts directed towards exploring energy savings using DVFS during blocking data transfer and synchronization operations. Some examples include work by Newsom et al.[117], Gamell et al.[52], Li et al.[104], and Lim et al.[106].

Newsom et al.[117] use locality-aware of PGAS data transfers to determine the feasibility of using DVFS for energy savings. Their analysis takes into account the energy savings achievable using hardware-controlled as well as application (user/-compiler) driven DVFS techniques. They highlight the potential energy savings achievable at the application layer, by discussing the impact of applying frequency

scaling while prefetching remote data objects within stencil-based kernels.

Gamell et al.[52] explore the feasibility of using DVFS during different UPC operations. Their experiments are limited to communication among multiple cores within a single node. They conclude that energy savings using DVFS is achievable during UPC *memget* and *wait* operations.

Li et al.[104] explore opportunities for energy savings using DVFS and DCT within Hybrid MPI+OpenMP applications. In their work, they introduce a power-aware performance prediction model which aid in determining the frequency and concurrency (number of threads) settings for different OpenMP phases in hybrid applications.

Lim et al.[106] use DVFS techniques within the MPI runtime library. Their approach is geared towards controlling the frequency at the granularity of individual MPI calls. For cases where the overhead of frequency scaling is too high, the granularity is increased to control frequency switch across multiple MPI function calls.

All the above efforts focus on using frequency scaling as a means to reduce the energy consumption during slack-periods during data movement in a distributed-memory environment. One of the major questions that remain unanswered is the performance impact on the data movement due to (a) the actual value of the CPU frequency and, (b) the data-access pattern adopted by PGAS communication phases. This chapter addresses these questions by comparing different implementation approaches of point-to-point communication interfaces within PGAS models.

12.2 Constraints imposed by Hardware Design

- *Choosing the correct frequency level:* CPU cores in modern processors are capable of operating at multiple different frequencies. While operating a CPU at a lower frequency leads to power savings, running it at a higher frequency leads to increased throughput. It has been well established that the choice of this operating frequency depends on the design of the application kernel. Recent study by Gotz et al.[58] present empirical evidence that shows that applications with varying computational demands attain energy efficiency at different CPU clock speeds. This observation is in alignment with the Roofline Model of Energy[37], which relates these 'computational demands' to the ratio of the number of compute operations to memory accesses within application kernels. This article focuses on only RDMA transfers. More specifically, it describes how operating a receiver at different frequencies during a point-to-point data transfer, leads to varying performance and energy consumption.

- *Sibling cores with contradicting frequency demands:* While designing an energy efficient software, one must be aware of the impact of frequency scaling of a single CPU core on the performance of other cores. The extent of this impact varies with the architectural design of the target processor. For example, in case of the Sandy Bridge architecture, all the CPU cores lie on the same frequency plane[131]. This means that a single CPU core cannot operate at a different frequency than others[2]. In such an environment where all the cores share the

[2]A hardware logic unit, called the Power Control Unit, is responsible for ensuring that the internal clock of all the cores are maintained at a frequency that meets the core with the highest performance demand.

same clock-speeds, conflicting demands by a single CPU core may affect the performance of the rest of the cores. In case of PGAS applications targeting a multi-core environment, using DVFS has the potential of severe performance degradation.

- *DVFS dependent Memory/Cache bandwidth:* DVFS also affects the local cache and memory bandwidths within a processor. Schöene et al.[134], in their study, present empirical evidence suggesting that this impact varies among different x86_64 processors. For a Sandy Bridge-EP processor, they show that the memory bandwidth can drop by as much as 44% depending on the operating frequencies and the number of cores in use. The L3 cache bandwidth also gets affected and follows an almost linear relationship with the drop in the CPU frequency[3]. In case of PGAS applications, this impact on the bandwidth of the internal memory hierarchy leads to an impact on the performance of data transfers - both remote or local to a process.

[3]This is because in a Sandy Bridge architecture, the interconnecting ring-bus runs on the same frequency as the CPU cores. Also, the cores, the bus, and the last-level shared L3 cache, all lie on the same power plane[131].

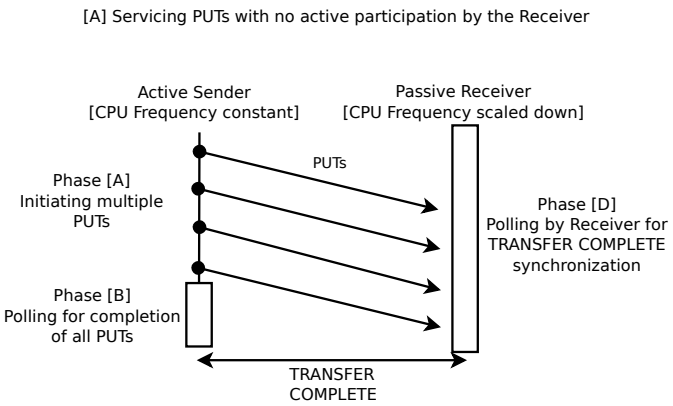


Figure 12.1: Line Diagram for remote write implementation: Servicing PUTs with no participation by the receiver

12.3 Energy cost factors associated with RDMA transfers

In this section, we identify multiple energy and performance factors within the software stack which have the potential of affecting the energy consumption of PGAS implementations of point-to-point interfaces. These cost factors are mapped to various *phases* of implementation approaches illustrated in Figures 12.1, 12.2, and 12.3. This section describes the cost factors with respect to remote write (PUT) operations. However, it must be noted they are also applicable for implementations of remote read (GET) operations.

To complement this discussion, Table 12.1 maps these costs to the CPU and the DRAM servicing the sender and the receiver processes.

- *Initiating Asynchronous PUTs (Phase A):* This phase, executed by the sender, corresponds to the initiation of a one-sided PUT operation of x bytes, from its

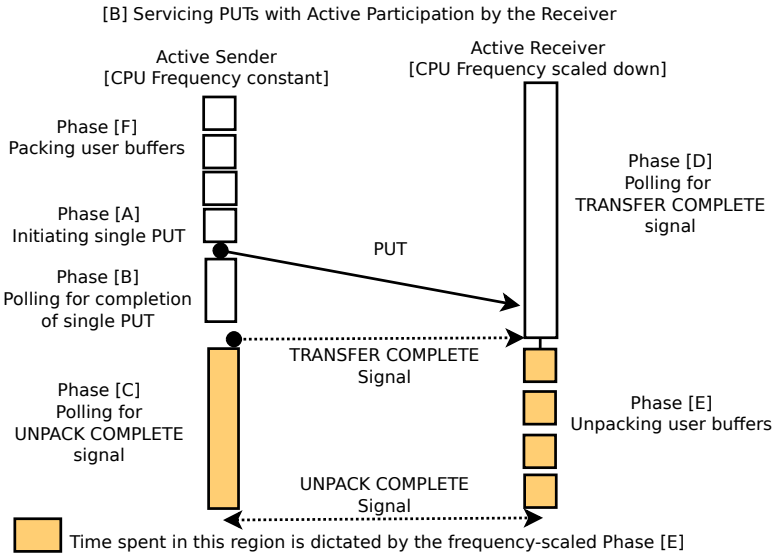


Figure 12.2: Line Diagram for remote write implementation: Servicing PUTs with active participation by the receiver

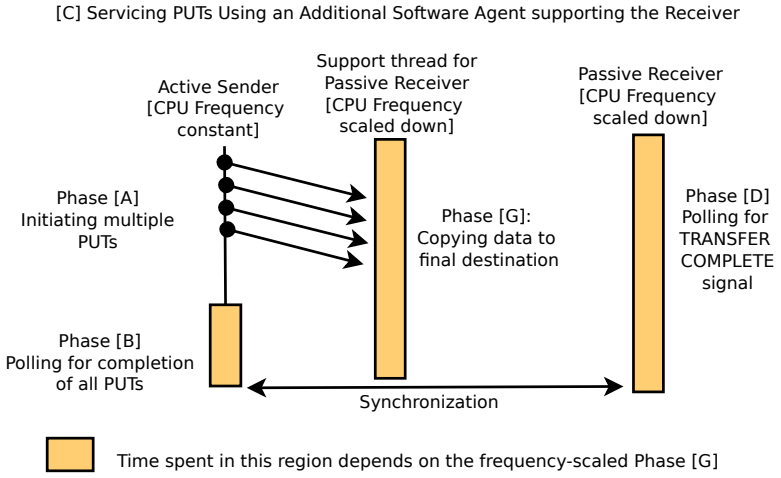


Figure 12.3: Line Diagram for remote write implementation: Servicing PUTs with an additional thread supporting the receiver

Table 12.1: Overview of different factors that contribute to the performance and energy consumption. Each row lists the cost factor, the system components involved as well as the potential impact on the CPU and DRAM energy/performance metrics

(I) Phase	(II) Components affected	Impact of scaling down frequency of Receiver CPU					
		(III) Time spent within this phase	(IV) Activity of components affected	(V) Sender CPU Energy	(VI) Sender DRAM Energy	(VII) Receiver CPU Energy	(VIII) Receiver DRAM Energy
A: Initiating async PUTs	Sender CPU, Sender DRAM	No impact	No	Constant	Constant	–	–
B: Polling for completion	Sender CPU, Sender DRAM (DMA)	No impact	No	Constant	Constant	–	–
C: Polling for UNPACK COMPLETE	Sender CPU	Increased time period	No	Energy rise proportional to the time spent	–	–	–
D: Polling for TRANSFER COMPLETE	Receiver CPU, Receiver DRAM (DMA)	No impact	Yes	–	–	Reduction of energy	Reduction of energy
E: Unpacking user buffers	Receiver CPU, Receiver DRAM	Increased time period	Yes	–	–	Trade off between time spent idling v/s initiating local memory copy operations	Depends on the impact on memory access rate
F: Packing user buffers	Sender CPU, Sender DRAM	No impact	No	Constant	Constant	–	–
G: Polling for incoming packets + Copying data to destination buffers	Receiver CPU, Receiver DRAM	Increased time period	Yes	–	–	Trade off between energy saved while polling v/s initiating local memory copy operations	Depends on the impact on memory access rate + rate of incoming buffers

85].

- *Polling for completion (Phase B)* This phase, executed by the sender, corresponds to a polling operation which, on completion, guarantees completion of all previously initiated PUTs during Phase A. In terms of OpenSHMEM, this corresponds to a *shmem_quiet* operation.

- *Polling for UNPACK COMPLETE (Phase C)* This phase is applicable for implementations that require unpacking of received data packets and distribution of its contents to discrete user buffers. In such cases, this phase is an additional overhead borne by the sender after Phase B. It corresponds to a polling operation by the sender to receive an acknowledgment by the receiver that the unpacking phase is complete. The completion of phase C indicates that the remote memory locations have been updated with the corresponding data contents and are available for future local/remote accesses.

- *Polling for TRANSFER COMPLETE (Phase D)* This phase, executed by the receiver, corresponds to polling for a signal sent by the sender to flag the arrival of data packets at the receiver. This is a crucial factor in case of implementations that rely on the receiver to participate in the data transfer operations.

- *Unpacking user buffers (Phase E)* It must be noted that the completion of Phase D does not guarantee that the incoming data contents have arrived at the final destination buffers. It might be the case that the contents need to be copied from a temporary storage buffer to the final destination. This phase corresponds to this software overhead borne by the receiver while transferring

the contents to its final intended destination address. It must be noted that this Phase E is identical to Phase C; the only major difference being the actual process servicing it.

- *Packing user buffers (Phase F)* This phase, executed by the sender, corresponds to the preparation of a user buffer before initiating an RDMA operation. This involves memory management tasks like copying the contents of user buffers from user address space to pinned-down memory buffers. In case of OpenSHMEM, the communication model does not require the source buffer of a remote PUT operation to be remotely accessible ('symmetric' in SHMEM terminology) itself. In such a case, this operation is typically performed by the underlying implementation.

- *Memory management by a support thread (Phase G)* Unlike phase E where the receiver bears the overhead of managing the contents at the intended destination addresses, an implementation may choose to use a dedicated software agent to handle this operation. Depending on the implementation, this asynchronous agent may be launched during the initialization phase of an application (*shmem_init*, in case of OpenSHMEM) and remain active throughout the lifetime of a process. The use of computational resources to service this agent is an additional cost factor that needs to be accounted for.

12.4 Approaches for implementing RDMA PUTs

This section highlights some common approaches for implementing remote PUT operations in a PGAS library. As described below, these approaches may be divided

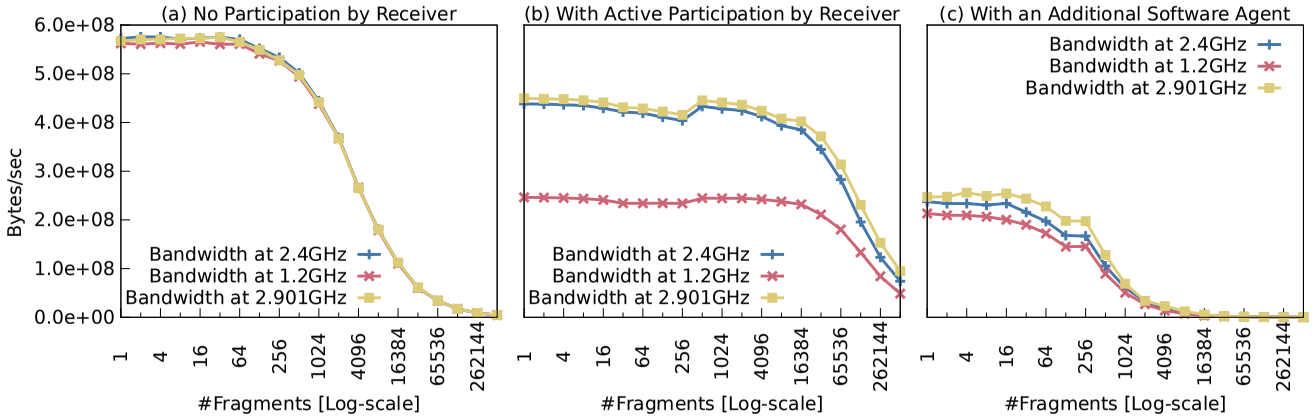


Figure 12.4: Achievable RDMA PUT Bandwidth with the sender process operating at 2.901GHz and the receiver process operating at a Turbo frequency of 2.901GHz, and non-Turbo frequencies of 1.2GHz and 2.4GHz. The 3 subplots correspond to implementations (a) without any active participation by the receiver (Mellanox Scalable SHMEM) (b) with active participation by the receiver (Mellanox Scalable SHMEM), and (c) using an additional software agent (OpenSHMEM reference implementation over GASNet - IBV conduit)

into a number of different phases listed in Section 12.3. The data flow within these approaches are illustrated as line charts in Figures 12.1, 12.2, and 12.3. The corresponding pseudocodes are listed in Appendix-B. The achievable bandwidth for each of these patterns at different CPU (receiver) frequencies is depicted in Figure 12.4. Clearly, changing the frequency of the receiver leads to an impact in the performance of the data transfer. This is discussed later in Section 12.6.

1. ***Servicing PUTs with No Active Participation by the Receiver***
   (Refer Figure 12.1)

   This case corresponds to the ideal scenario with minimal CPU intervention and software overhead during a remote write operation (PUT). Low latency of such transfers is typically achieved using RDMA support provided by modern interconnects like InfiniBand. Such operations do not require the active participation of the remote CPU and bypass the OS on the remote node.

2. ***Servicing PUTs with Active Participation by the Receiver***
   (Refer Figure 12.2)

   This corresponds to cases where additional software overhead is added by the communication library to implement data access patterns that are not directly supported by the underlying hardware. In order to handle the transfer of discrete user buffers across the network, an implementation may choose to aggregate or pack multiple discrete memory fragments into a single contiguous memory chunk. This operation is performed locally before transferring the contents to the remote host. On detecting the arrival of the incoming packets

(which typically involves a handshaking signal), the receiver is responsible for unpacking the contents of the buffer and copying them to their intended destination buffers. One such use case is implementation of strided-data communication interfaces, which are common among PGAS models. These interfaces allow the user to initiate transfer of multiple data objects that are not aligned contiguously in memory.

3. ***Servicing PUTs Using an Additional Software Agent Supporting the Receiver*** *(Refer Figure 12.3)*

   In order to ensure progress of asynchronous PUT operations without interrupting the receiver's CPU, an additional thread may be launched at the receiver's end for polling the network for incoming transfers. Once this thread detects an incoming packet, it aids the completion of the data transfer operation by copying the data contents to the final destination buffers. This leaves the receiver free to perform a different set of operations, thereby leading to communication-computation overlap. A use case for such an approach is ensuring the progress of asynchronous communication on platforms that lack network support for RDMA-based transfers.

12.5 Experimental setup

12.5.1 Method

In order to conduct the study with respect to different approaches of implementing RDMA patterns, we designed synthetic microbenchmarks based on data-access patterns in communication libraries. The pseudo codes for each of these benchmarks are listed in Appendix-B.

The patterns were evaluated using two OpenSHMEM processes (PEs), each launched on a separate but identical compute node and bound to their respective CPU cores. Each PE played the role of either the sender or the receiver, but not both. This isolation ensured a comparative study between the two processes. The patterns depicted in Figures 12.1 and 12.2 were evaluated using the *Mellanox Scalable SHMEM ver-2.2* (over *OpenFabrics* Byte Transport Layer). For the third pattern, the OpenSHMEM reference implementation was used (over GASNet with IBV conduit)[6].

As discussed before, the purpose of the study is to analyze the impact of a receiver operating at a scaled-down CPU frequency. In our experiments, the CPU frequency of the sender was held constant at 2.901GHz (turbo-frequency). To study the impact of frequency scaling of the compute node servicing the receiver, the experiment was repeated multiple times with the node operating initially at 2.901GHz and later at 2.4GHz and 1.2GHz.

[6]The reference implementation spawns an additional thread that uses the GASNet Active Message framework to detect and handle incoming PUT requests targeting destination buffers that are declared (i) as *global* or *static* (in C), or (ii) as *save* or within *common* blocks (in Fortran).

The results of the microbenchmarks are depicted in Figures 12.5,12.6, and 12.7. Every data point in the figure corresponds to a transfer of a fixed data payload of 512KiB. The x-axis indicates the number of fragments used to transfer the fixed payload. In this work, the term 'fragments' corresponds to the number of explicitly initiated OpenSHMEM PUT operations - a user controllable parameter. It must be noted, however, that the data payload may be further split into smaller packets by the underlying software and hardware stack.

The effect of frequency scaling was studied in terms of the impact on 6 different metrics:

1. The energy consumed by the CPU servicing the sender process
2. The energy consumed by the CPU servicing the receiver process
3. The energy consumed by the DRAM servicing the sender process
4. The energy consumed by the DRAM servicing the receiver process
5. The unidirectional point-to-point latency (as measured at the sender's side)
6. The Energy Delay Product (EDP)[7]

The *impact* I may be represented as the percent reduction in the above metrics M due to application of a DVFS technique T that scales down the frequency of the

[7]While CMOS circuits have the ability to trade performance for energy savings, it becomes challenging to optimize for both simultaneously. The EDP, first proposed by Horowitz[57, 69], takes into account both the energy and the time costs in an implementation-neutral manner. For cases, where energy and performance have equal importance, this metric can be calculated as a product of the energy consumed and the time taken. For more complicated cases, where performance is given a higher priority, the weight of the "delay" factor is increased by squaring or cubing it.

receiver from an operating frequency $F_{initial}$ to a reduced frequency, F_{final}. It can be calculated as follows:

$I = \frac{M(F_{initial}) - M(F_{final})}{F(C_{initial})} * 100$

From the expression above, it must be noted that a negative or positive impact value of I suggests a corresponding rise or drop in a metric M due to application of T. A zero value indicates an absence of any impact on M. The impact on each metric is illustrated in Figures 12.5, 12.6, and 12.7.

12.5.2 Test-bed Characteristics

Our test platform comprised of two Sandy Bridge nodes connected via InfiniBand. The characteristics of these nodes are listed in Table 12.2.

12.5.3 Power/Energy Measurement

Each node contains power monitoring support and reports energy/power readings at the CPU, DRAM, and the node level. Each node has instrumented voltage regulators (VRs) that are sampled at a frequency of 1 KHz for both sockets and the four voltage lanes of the DIMMs (Dual In-line Memory Modules) on board. With the help of an FPGA, a digital filter is applied to smooth the samples. Furthermore, a linear correction is applied to the measurement data coming from the VRs in order to ensure an error margin not exceeding 3%. Our study was aimed at performing a fine-grained analysis of the impact on two main components that dictate the energy and power consumption of a system - the CPU and the DRAM[8].

[8]More information about the High Definition Energy Efficiency Monitoring (HDEEM) project is available at `http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/projekte/hdeem`

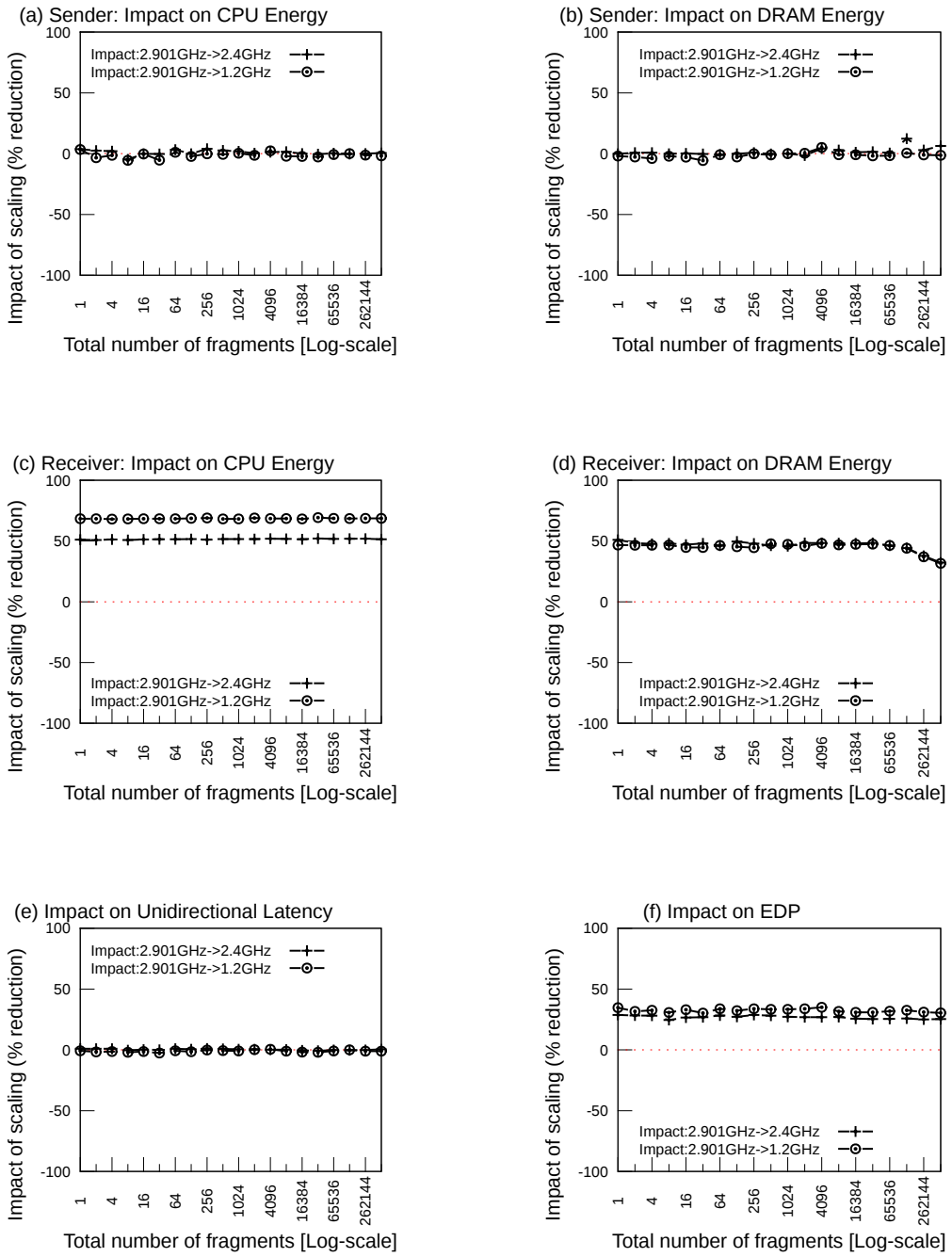


Figure 12.5: Impact of frequency scaling on energy and performance metrics for implementations which *do not require active participation by the receiver* during a one-sided point-to-point remote PUT operation. The line-chart and the pseudo-code of this approach is depicted in Figure 12.1.

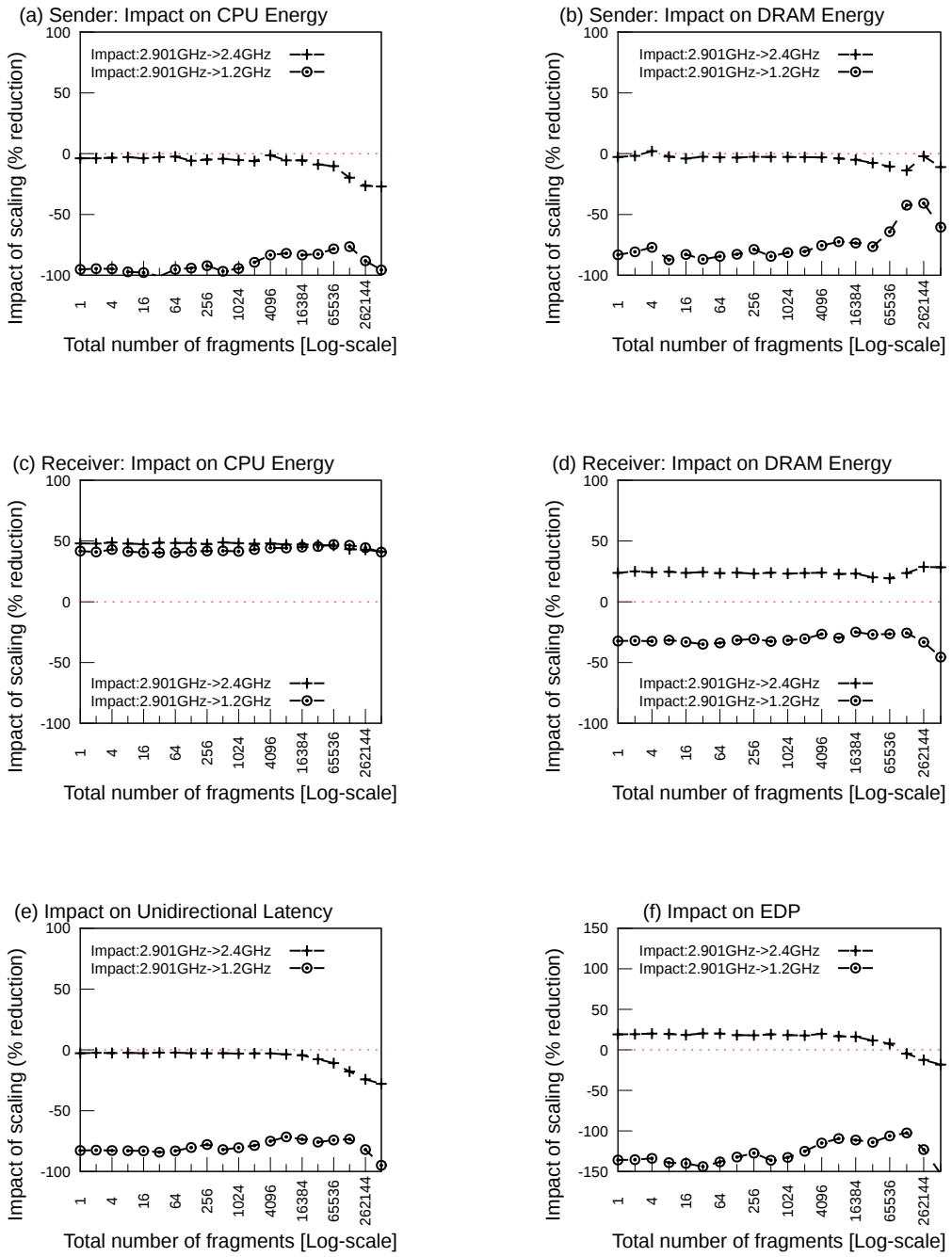


Figure 12.6: Impact of frequency scaling on energy and performance metrics for implementations which *depend on active participation by the receiver* in order to ensure completion of one-sided point-to-point remote PUT operation. The line-chart and the pseudo-code of this approach is depicted in Figure 12.2.

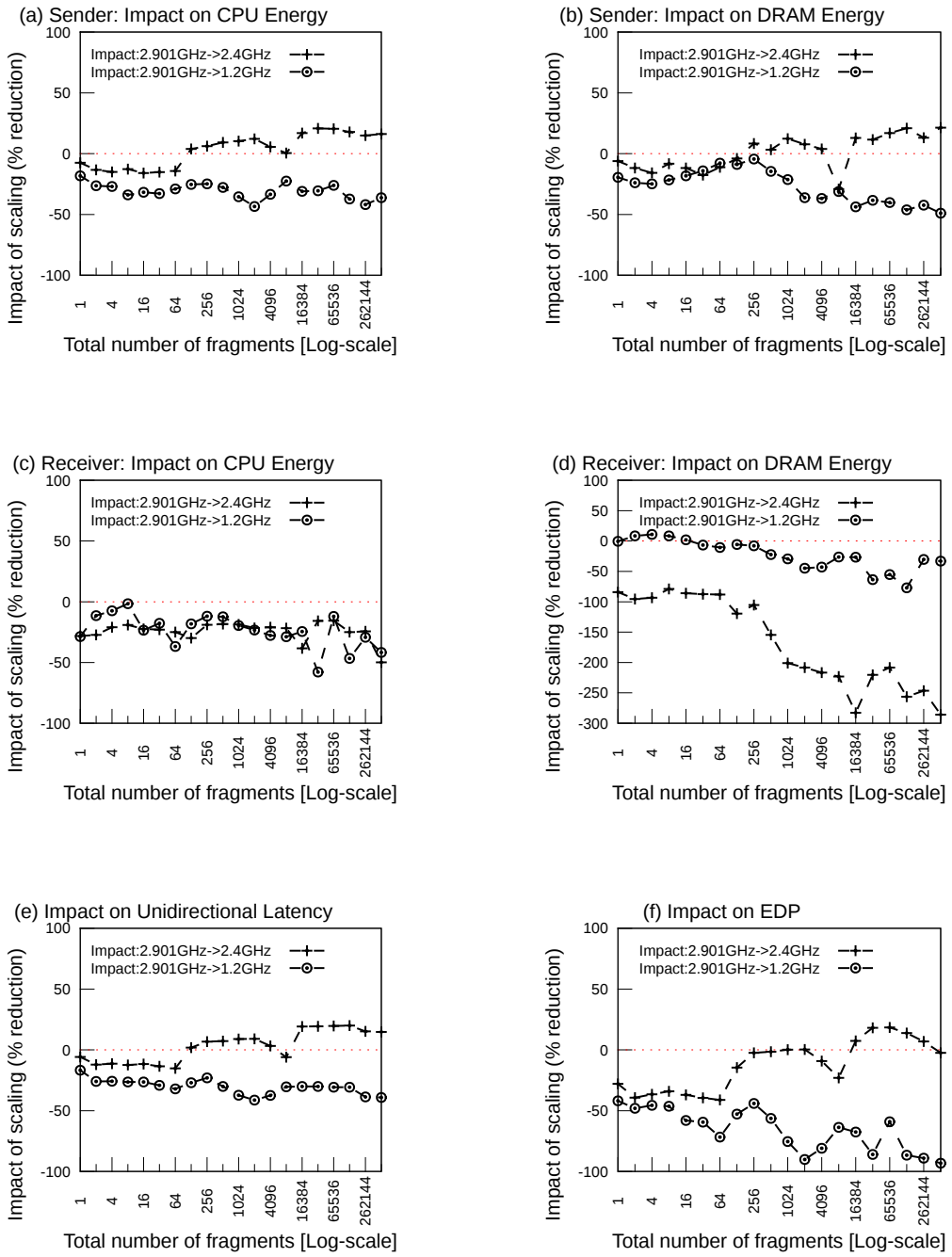


Figure 12.7: Impact of frequency scaling on energy and performance metrics for implementations which *relies on an additional asynchronous software agent* to ensure completion of one-sided point-to-point remote PUT operation. The line-chart and the pseudo-code of this approach is depicted in Figure 12.3.

Table 12.2: Characteristics of the Test Platform

Processor	Intel Xeon CPU E5-2670
Microarchitecture	Intel's Sandy Bridge
L3 cache per die	20MiB
Cores	2x 8
Main Memory	32GiB
Infiniband card	Mellanox MT27500, ConnectX-3
Linux kernel version	2.6.32 x86_64

12.6 Results

This section discusses the empirical results obtained on scaling down the operating frequency of the CPU servicing the receiver process. The impact is discussed for each of the implementation approaches depicted in Figures 12.1, 12.2, and 12.3.

12.6.1 No Participation by the Receiver

- For the sender process, Figures 12.5-a/b suggest that there is no significant impact ($\approx 0\%$) of frequency scaling on the energy consumption by the CPU and DRAM. This is true regardless of the extent of fragmentation (number of discrete buffers) used for transferring the data payload (Phase A) and ensuring its completion (Phase B). This may be attributed to the fact that the latency of the transfer is dictated by the bandwidth of the network interconnect (InfiniBand, in this case) which is orders of magnitude smaller than that of the I/O interconnect between the network adapter and the last-level (L3) cache on the receiver's side. Since scaling down the frequency of the receiver CPU does not affect the bandwidth of the network interconnect, there is no significant impact during the actual RDMA-based transfers (Phases A and B).

- For the receiver CPU, Figure 12.5-c indicates that definite energy savings can be achieved due to scaling down of the operating frequency. Also, these savings are higher with a greater drop in the frequency ($\approx 50\%$ versus 68% when the frequency is scaled down from 2.901GHz to 2.4GHz and 1.2GHz respectively). This holds true regardless of the number of discrete fragments being transferred. This is not surprising because the CPU at the receiver's end does not contribute to the data transfer operation and therefore the savings can be attributed to the reduced rate of polling at the synchronization point (Phase D).

- Figure 12.5-d shows that the energy savings for the receiver DRAM is almost constant ($\approx 50\%$) regardless of the frequency level to which the CPU is scaled down to. It must be noted that due to Intel's direct-I/O technology[78], there is almost no participation by the receiver DRAM during this transfer; the contents of the data transfer is directed to the L3-cache without the need for accessing the DRAM. Nevertheless we see significant savings in its energy consumption when the CPU frequency is scaled down. The fact that the savings is non-zero and independent of the final frequency suggests that the energy consumed by the DRAM is higher when the CPU operates at the turbo-frequency (2.901GHz) and is almost constant at other lower non-turbo frequency levels. Another observation is that for high fragmentation count (number of PUTs $> 32K$), there is a drop in energy savings. This suggests a rise in memory accesses for higher fragmentation. This is because, despite the use of the L3-cache described above, Intel's chipsets limits the use of this cache up to 10%

of its size - which, on our platform is about 2MB. With a rise in fragmentation (and hence, smaller sized PUTs), the relative overhead per network packet increases. This may be the potential cause for the L3-cache limit getting exhausted thereby leading to direct-I/O operations that target the DRAM.

- Figure 12.5-e highlights one of the major observations of this experiment. It affirms the fact that reducing the CPU frequency of the receiver in case of a one-sided transfer with no participation by the receiver CPU leads to no impact on the latency of the data transfer pattern.

- In terms of the net impact on the two-node system, we see that the energy savings at the CPU servicing the receiver dominates the savings in Energy Delay Product (EDP) (Figure 12.5-f).

12.6.2 Active Participation by the Receiver

Figure 12.6 depicts the impact of frequency scaling on a remote PUT operation that involves aggregation of discrete user buffers by the sender and the corresponding unpacking by the receiver. It can be observed that the impact of this pattern is significantly different from that discussed in Figure 12.5.

- From Figure 12.2, we see that the time spent by the sender CPU within Phase C is dependent on the performance of the receiver in Phase E. From Figure 12.6-a, we observe that the energy consumption by the sender CPU is dependent on the frequency to which the receiver CPU is scaled down to. During Phase C, the sender CPU is primarily involved in a polling operation. As a result, the

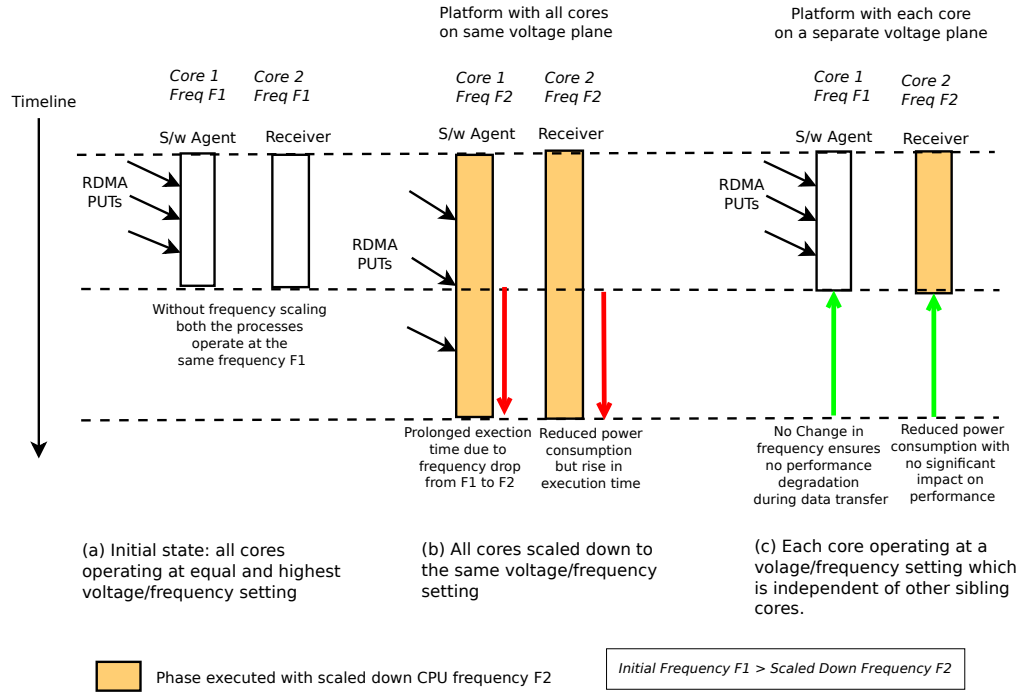


Figure 12.8: Benefit of using DVFS at the granularity of individual cores

energy consumption is directly proportional to the time spent in this phase, which in turn is dependent on the frequency of the receiver. This explains the relatively higher (negative) impact on the receiver CPU energy during Phase C ($\approx$ 0 to $(-20)\%$ versus (-75) to $(-98)\%$ when the frequency is scaled down from 2.901GHz to 2.4GHz and 1.2GHz respectively).

- Figure 12.6-b suggests that there is a negative impact on the sender's DRAM (rise in the energy consumption) when the receiver is operated at 1.2GHz. In this communication pattern, the two phases during which the sender's DRAM participates are phases F and B. From Table 12.1, neither of these phases have the potential of being affected by scaling down the frequency of the receiver. Therefore, we do not completely understand the cause for the rise in energy. We

are currently performing additional experiments to understand this behavior. It must be noted that this does not affect further analyses of this pattern simply because the magnitude of the DRAM energy here, is of the order of tens of milliseconds, which is negligible in comparison to that of the CPU (with energy consumption that is higher by two orders of magnitude).

- Figure 12.6-c shows that there is a definite rise in the energy savings of the receiver's CPU, due to scaling down its frequency. It is important to note that regardless of the frequency down to which the CPU is scaled, the impact remains almost equal. This suggests that as long as the CPU is not operating in turbo frequency (2.901GHz), consistent energy savings ($\approx 50\%$) can be expected.

- Figure 12.6-d shows that the impact of energy consumption by the receiver DRAM is dependent on the CPU frequency scaling. We observe that reducing the frequency from turbo (2.901GHz) to 2.4GHz has a positive impact on memory access rate. This leads to energy savings for the DRAM ($\approx 25\%$). However, dropping the frequency to 1.2GHz drops the memory access rate to a point that leads to an energy inefficient transfer of the same size of data payload ($\approx (-30)\%$).

- The unidirectional latency of this approach appears to follow a similar trend to that of the sender's CPU energy: From Figure 12.6-e, the extent of impact on the latency is dependent on the frequency to which the (≈ 0 to $(-20)\%$ versus (-75) to $(-98)\%$ when the frequency is scaled down from 2.901GHz to

2.4GHz and 1.2GHz respectively).

- In terms of impact on the Energy Delay Product (EDP) due to frequency scaling, Figure 12.6-f shows that the behavior is dictated by the impact on the receiver's DRAM. We see that reducing the receiver CPU frequency from 2.901GHz to 2.4GHz leads to positive savings ($\approx 10\%$). However, reducing the frequency to 1.2GHz leads to negative impact as high as 90%.

12.6.3 Additional Thread Supporting the Receiver

Figure 12.7 depicts the impact of frequency scaling on a remote PUT operation that is completed with the assistance of an additional thread coupled with the receiver. This implementation approach suffers from the design of CPUs with compute cores sharing the same voltage plane. In this study, this architecture characteristic was true for the target SandyBridge processors.

- In order to decrease the frequency of the SandyBridge core servicing the receiver process, all the cores on the same voltage plane have to be scaled down. As seen in Figure 12.7-e, this impacts the unidirectional latency of the transfer process. More specifically, when the frequency of the receiver CPU is scaled down from 2.901GHz to 1.2GHz, the latency increases by up to 50% (negative impact). However, the impact is different when the CPU is scaled to 2.4GHz instead. In fact, it is observed that there is either zero or up to 20% drop (positive impact) in the latency. The variability in the impact may be attributed to the fact that there is a trade-off between the energy costs associated with

two different phases at the receiver's end - (a) The polling operation by the receiver process (Phase D) (b) The memory management by the support thread (Phase G). On reducing the frequency to 2.4GHz, the energy savings during Phase D dictates that of the entire CPU. However, reducing the frequency to 1.2GHz significantly impacts the performance of the support thread which makes Phase G contribute strongly to a rise in the latency (negative impact).

- Similar to the latency, the energy consumption by the sender's CPU also varies based on the operating frequency of the receiver CPU (Figure 12.7-a). Since the frequency of this CPU is not altered, the similarity in the energy and latency characteristics may be attributed to the CPU time invested to synchronize with the receiver (Phase B).

- Figure 12.6-f summarizes the net impact of frequency scaling in terms of the achievable Energy Delay Product (EDP). Up to 64 PUTs, the EDP of the data transfer at 2.4GHz is higher by 50% than at 2.901GHz. Beyond 256 PUTs, the impact is almost negligible. However, while on dropping the frequency to 1.2GHz, there is a significant performance degradation or 50% or higher.

12.7 Using DVFS in a multicore environment

The scope of this chapter is limited to evaluating the performance/energy metrics of RDMA operations between two processes, each bound to a single CPU core on different nodes. For the first two implementation approaches, the study was restricted to studying the impact of only these two cores participating in the data transfer operation. However, in real-world multicore HPC environment, it is almost always

the case that a process running on one core is accompanied by additional software agents (OS processes / threads) running on sibling cores. In such an environment, using DVFS on a single core has a potential of affecting the activity of other cores.

The feasibility of using DVFS in a multicore environment is heavily dependent on its design and architecture of the target processor. Consider the case of Sandy Bridge processors. In this case, all the CPU cores lie on the same voltage/frequency plane[131]. In other words, all the cores operate at the same frequency level. Thus, a naive energy efficient solution of operating all the frequencies at a lower frequency in order to favor a single core might lead to significant performance degradation of software agents operating on other cores. This is illustrated in Figure 12.8(b). This issue may be alleviated in case of processors like the Haswell series where each core can be operated at a voltage/frequency setting that is independent of other cores[64]. This approach has the potential of alleviating the performance impact of implementations which rely on using an additional thread to handle data transfers. This is shown in Figure 12.8(c).

12.8 Lessons learned

The main lessons learned using empirical analysis of each of the above approaches are listed below:

- High energy savings with negligible performance impact may be achieved when the target process of a remote PUT operation does not participate in servicing the data transfer. This is applicable for implementation approaches that rely on RDMA-based capabilities of the underlying interconnect.

- For an implementation where the target process does participate in a data transfer operation, scaling down the frequency of that process not only affects the unidirectional latency but also the energy consumption, which worsens with a drop in the CPU operating frequency.

- For an implementation using an additional software agent for servicing a transfer, the extent of impact depends on: (a) The number of explicit PUT operations used to transfer the data payload, (b) The actual operating frequency of the CPU servicing the receiver, and (c) Architectural design of the target CPU in terms of whether multiple cores on a CPU share the same voltage plane or not.

12.9 Chapter Summary

This work details the impact of CPU frequency scaling on the performance and energy consumption of remote data transfers. The empirical results presented are instructional for system developers of energy efficient solutions for distributed memory programming models, especially PGAS. The focus was to analyze the impact of using DVFS (Dynamic Voltage Frequency Scaling) on the performance and energy metrics of system components servicing one-sided RDMA operations. Multiple cost factors that affect the energy and performance during DVFS-based techniques were identified. These factors are dependent on not only software stack but also various microarchitectural design factors. Their impact was analyzed with respect to three common implementation approaches of PGAS point-to-point communication - (a) Using RDMA capable underlying software and hardware stack to service transfers

without the active participation of a target process, (b) Relying on the receiver process to participate in the data transfer to ensure its completion, and (c) Using an additional software agent (e.g., OS thread) at the receiver's side to assist in completion of the operation.

The main lessons learned using empirical analysis of each of the above approaches are listed below:

- High energy savings with negligible performance impact may be achieved when the target process of a remote PUT operation does not participate in servicing the data transfer. This is applicable for implementation approaches that rely on RDMA-based capabilities of the underlying interconnect.
- For an implementation where the target process does participate in a data transfer operation, scaling down the frequency of that process not only affects the unidirectional latency but also the energy consumption, which worsens with a drop in the CPU operating frequency.
- For an implementation using an additional software agent for servicing a transfer, the extent of impact depends on: (a) The number of explicit PUT operations used to transfer the data payload, (b) The actual operating frequency of the CPU servicing the receiver, and (c) Architectural design of the target CPU in terms of whether multiple cores on a CPU share the same voltage plane or not.

For more details about the topics discussed in this Chapter, the interested reader is directed to the literature documented by Jana et al. under [81].

Chapter 13

Proposed Solution: Reviving Active Messages

Recent reports on challenges of programming models at extreme scale suggest a shift from traditional block synchronous execution models to those that support more asynchronous behavior. The OpenSHMEM programming model enables HPC programmers to exploit underlying network capabilities while designing asynchronous communication patterns. The strength of its communication model is fully realized when these patterns are characterized with small low-latency data transfers. However, for cases with large data payloads coupled with insufficient computation overlap, OpenSHMEM programs suffer from underutilized CPU cycles.

In order to tackle the above challenges, this chapter explores the feasibility of introducing Active Messages in the OpenSHMEM model. Active Messages is a well established programming paradigm that enables a process to trigger execution of computation units on remote processes. Using empirical analyses, we show that this approach of moving computation closer to data provides a mechanism for OpenSHMEM applications to avoid the latency costs associated with bulk data transfers. In addition, this programming pattern helps reduce the need for unwanted synchronization among processes, thereby exploiting more asynchrony within an algorithm. As part of this preliminary work, we propose an API that supports the use of Active Messages within the OpenSHMEM execution model. We present a microbenchmark-based performance evaluation of our prototype implementation. We also compare the execution of a Traveling-Salesman Problem designed with and without Active Messages. Our experiments indicate promising benefits at scale.

13.1 Introduction

In recent years, research surveys that highlight the challenges faced by current programming models at extreme scale, have indicated a shift from the de facto SPMD style message passing models. With regards to the need for asynchrony within programming models, the report on *ASCR Programming Challenges for Exascale Computing*[8] states that, "The increased variation on execution speed of various components, due to error recovery and power management, will require codes that are more tolerant to noise, hence, more asynchronous". In accordance with this, multiple research efforts are being directed towards adopting programming languages and

libraries that support task-based algorithm design.

In this chapter, we explore the feasibility of introducing support for Active Messages to OpenSHMEM[1], a one-sided SPMD-based PGAS programming model. Active messages (AM) provide a means of triggering a user-specified unit of computation at a different process (or Processing Element or PE). The main motivation is to enable asynchronous execution of small compute paths and overlap of communication, with very little synchronization overhead incurred at the source and the target PE. The user-specified function (called a *'handler'*) has access to the user address space at the target PE. Thus, Active Messages (or AM) let PEs inject computation on remote destinations that host memory objects that are either remotely inaccessible due to the memory model or too costly for data movement.

The contribution of this work and the chapter layout is as follows: (i) Description of a point-to-point interaction between a pair of processes using Active Messages (Section 13.2) and comparison of the AM handler with a task. (ii) Proposal of an API that introduces Active Messages within the OpenSHMEM programming model (Section 13.3) (iii) A prototype implementation of AM within

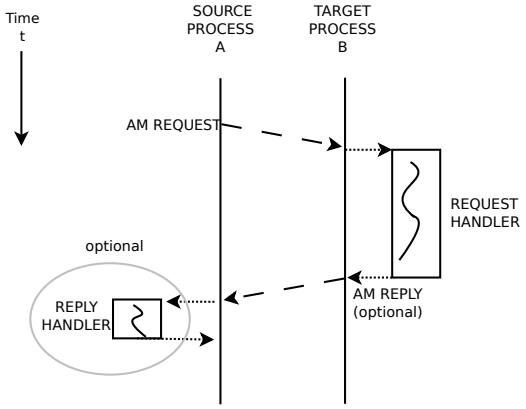


Figure 13.1: Execution flow of an Active Message Request

[1]OpenSHMEM is a trademark of Silicon Graphics International Corp.

the OpenSHMEM reference implementation over GASNet (iv) Empirical study using synthetic microbenchmarks and a miniapp that evaluates the performance of the prototype (Section 13.4) (v) List of different research efforts in the field of task management in a distributed environment (Section 13.5). (vi) A summary of the lessons learned and potential future work (Section 13.6).

13.2 Overview of Active Messages

Figure 13.1 depicts the flow diagram of two processes communicating using Active Messages. The progress of the communication between the source process A and the target process B is described below:

1. Both A and B register the function handlers with the AM library.

2. Source process A sends an AM request to remote process B. This AM request mainly comprises (1) the identity of B, (2) the identity of the handler to be executed at B, and (3) optionally, contents of the data buffer to be passed as input to the handler.

3. On receiving the AM request, process B chooses to asynchronously execute the requested function handler. At the start of the execution, it gains access to any data buffer that was transferred. This function that is executed at process B is called the *'request handler'*.

4. During the execution of the request handler, process B may optionally choose to post a reply AM back to A. Similar to the AM sent by A, this reply AM

contains the identity of the handler to be executed at A along with an optional data payload.

5. At some point in time, on detecting the arrival of the above reply AM, A executes the handler corresponding to the identity listed in the incoming message. The handler which is executed as a response to this AM is called the *'reply handler'*.

13.2.1 Active Message v/s Intra-node Tasking Models

Active Messages can be viewed as a mechanism to launch a unit of task on a user-specified process that may be located on a remote or a local node. This is unlike common intra-node tasking models where one has to rely on a scheduler to assign resources for execution, the AM model allows the programmer to explicitly specifying the destination for the execution. Another difference is that while the computation associated with an intra-node task is expected to return a specific result to a 'parent' unit, computation of AM handlers are usually intended to update local data structures. Another point to note is that while intra-node tasking models allow establishing dependence among multiple tasks, inter-node tasking supported by AM models focus on asynchronous execution of independent handlers.

13.3 Proposed Extensions for Supporting Active Messages

This section describes the proposed interface of AM handlers and the related AM management functions [2] related to: (1) design of an AM handler, (2) registration of

[2]Note: As a norm in the OpenSHMEM community, all the AM related functions in this chapter have been prefixed with 'shmemx_' instead of 'shmem_' to indicate that they are *proposed extensions*

AM handlers, (3) initiating AMs, (4) the completion of AMs, and (5) handler-safe locking. The set of the proposed interfaces for C is shown in Listing13.1.

Design of an AM Handler

The actual body of an AM handler is enclosed within a user-defined function. The purpose of active messages is to enable injection of code paths that contain a small set of computation that remains independent of the progress of other PEs. The design of an AM handler should therefore adhere to the following set of constraints:

- The handler body should not call other function routines from the OpenSHMEM library, that have the potential to trigger an inter-PE communication. This includes point-to-point communication, synchronization constructs, atomic operations, and other AM related functions (excluding those related to mutual exclusion).

- The execution of an AM handler can progress in an OS thread that runs concurrent to the one servicing the critical path of a PE. It becomes the responsibility of the programmer to ensure that no race conditions occur when a data object is made accessible to both an AM handler as well as the execution path of a PE. If one of the accesses is a write operation, handler-safe locks can be used to ensure mutual exclusion.

- If a data object is a target of a write operation during the execution of the handler routine, handler-safe locks should be used to avoid race conditions.

to the standard and *not* part of the current specification.

```
/** Function Handler Signature **/
void user_function_name (void* data_buffer, size_t buffer_size, int
    calling_peid, shmemx_am_token_t token)

/** (De)Registration of Active Message handlers **/
typedef void (*shmemx_am_handler) (void *buf, size_t nbytes, int
    req_pe, shmemx_am_token_t token)

void shmemx_am_attach (int handler_id, shmemx_am_handler handler_ptr)
void shmemx_am_detach (int handler_id)

/** Initiating Active Messages **/
void shmemx_am_request (int dest, int handler_id, void* source_addr,
    size_t nbytes)
void shmemx_am_reply (int handler_id, void* source_addr, size_t
    nbytes, shmemx_am_token_t temp_token)

/** Progress and Completion **/
void shmemx_am_quiet ()
void shmemx_am_poll ()

/** Handler-Safe Locking **/
void shmemx_am_mutex_init (shmemx_am_mutex* t)
void shmemx_am_mutex_destroy (shmemx_am_mutex* t)
void shmemx_am_mutex_lock (shmemx_am_mutex* t)
void shmemx_am_mutex_unlock (shmemx_am_mutex* t)
int shmemx_am_mutex_trylock (shmemx_am_mutex* t)
```

Listing 13.1: Proposed API routines for Active Messages in OpenSHMEM

Registration of AM Handlers This features the following two collective API routines:

- *shmemx_am_attach:* Enables the calling PE to register the function pointed to by the function pointer. The user passes a handler-id that is used to map the handler to the corresponding function. On return from this function, a PE can use the handler_id to launch an AM until its association to the handler function is removed using shmemx_am_detach. It must be noted that the remote PE itself need not register the handler if it does not intend to execute it during its lifetime. Since a function being registered can only be used as an AM handler after it has been registered, some type of synchronization between the two PEs may be necessary to ensure that the function registration is complete on the target PE. Different PEs can register the same function with different handler-ids.

- *shmemx_am_detach:* This removes the mapping between a handler-id and the function mapped to the id. Once detached, it is illegal for any other PE to reuse the same handler id to launch an AM unless it is explicitly remapped using shmemx_am_attach by the current PE.

Initiating Active Messages

- *shmemx_am_request:* This function is used to launch an AM on a remote PE destination. The contents of the user buffer is transferred to the target PE along with the id of the function. On receiving this request, the target PE executes the corresponding handler. On return from this request function, there is no

guarantee of the completion of execution of the handler by the target PE. This asynchrony reduces the overhead at the source PE. To enable the source PE to reuse the data buffer, it is essential that this function copies the contents to a temporary buffer internally before returning to the user address space.

- *shmemx_am_reply:* In a two-sided request-reply communication model, this function is used by the request AM handler to launch a reply AM handler at the source PE that had issued the AM.

The Completion of Active Messages

- *shmemx_am_quiet:* This function enables the calling PE to ensure that the request handlers of all previously posted Active Messages and their corresponding response handlers (if any) have completed their execution.

- *shmemx_am_poll:* This polls the network for any outstanding AM requests. It must be noted that while this function can be used by a programmer to wait for a certain event to occur, it is not necessary for an OpenSHMEM implementation to rely on this function to make progress. An implementation should be free to exploit interrupt driven mechanisms or asynchronous notification capabilities of the underlying operating system or the hardware platform, respectively.

Handler-Safe Locking

Since the critical path of the PE and the AM handler may run concurrently, it becomes necessary to ensure mutually exclusive accesses to shared data structures.

For this, we propose a new data type called shmemx_am_mutex. It becomes the responsibility of the programmer to ensure that an object of this data type be visible to both the AM function handler as well as the main PE thread. An object of this type represents a mutex variable that can be passed to the following functions to avoid overlapping access of shared memory.

- *shmemx_am_mutex_init:* Initializes the mutex variable. Typically, the purpose is to ensure that the initial state of the variable becomes visible to both the critical path of the PE as well as the AM handler.
- *shmemx_am_mutex_destroy:* Ensures that the variable is no longer usable by the critical path of the PE or the calling thread. This provides an opportunity for an implementation to clean up memory associated with the variable.
- *shmemx_am_mutex_lock:* Attempts to acquire the mutex variable exclusively. If unsuccessful, the calling PE remains blocked until it gains access to the mutex.
- *shmemx_am_mutex_unlock:* Releases the ownership of the mutex variable.
- *shmemx_am_mutex_trylock:* Attempts to acquire the mutex variable exclusively. If unsuccessful, it returns 0 to the callee and its execution continues with blocking. If successful, it returns a non-zero number.

13.4 Prototype Evaluation

13.4.1 Implementation Design

The prototype implementation[3] was designed as part of the OpenSHMEM reference implementation[34] which, in turn uses GASNet[27] for inter-process communication. Our prototype is built on top of the existing support of Active Messages that is offered by GASNet. The incorporation of the prototype within the OpenSHMEM reference implementation is illustrated in Figure 13.2. It must be noted that fine-tuned implementations of Active Messages in OpenSHMEM should take advantage of network hardware capabilities (if any) and the exploration of different design approaches is out-of-scope of this chapter.

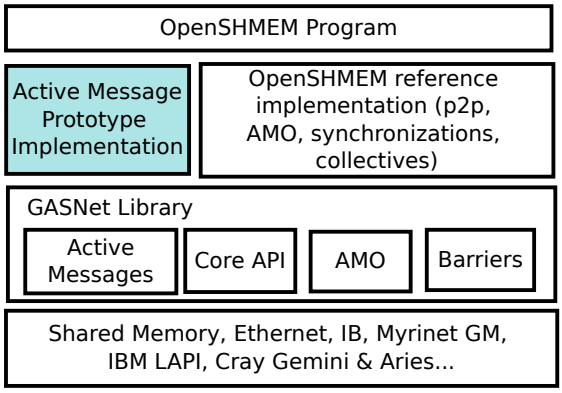


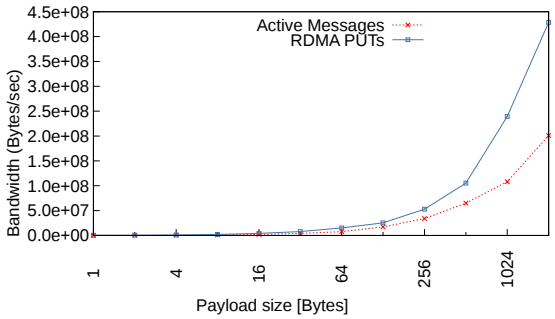
Figure 13.2: Incorporation of the the proposed Active Messages prototype into the OpenSHMEM reference implementation

13.4.2 Experimental Setup

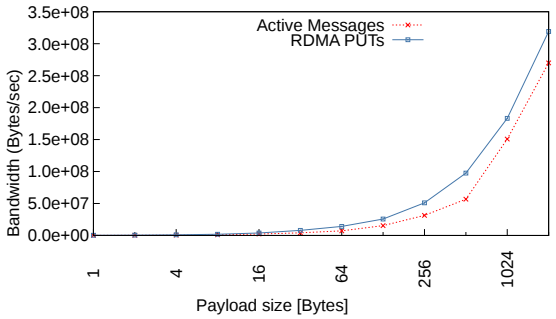
The experimental results presented in the following sections were obtained using a cluster with AMD Opteron processors (model 6174) and Infiniband interconnect (Mellanox MT26418). Each compute node comprises of a total of 48 cores (4 sockets/node, 12 cores/socket) with approximately 5MB shared L3 cache and 16GB main memory. The OS distribution on each compute node is OpenSUSE Linux (ver. 3.11).

Process Layout The results from the bandwidth and message rate tests, microbenchmarks were obtained by binding each process (PE) to a specific core on

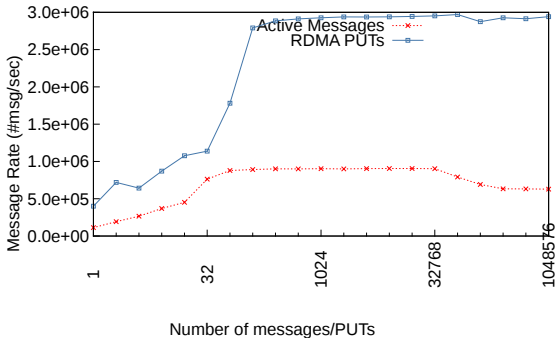
[3]The Active Message prototype implementation is available as a fork of the OpenSHMEM reference implementation and is available as a git repository at `https://github.com/openshmem-org/openshmem-am`



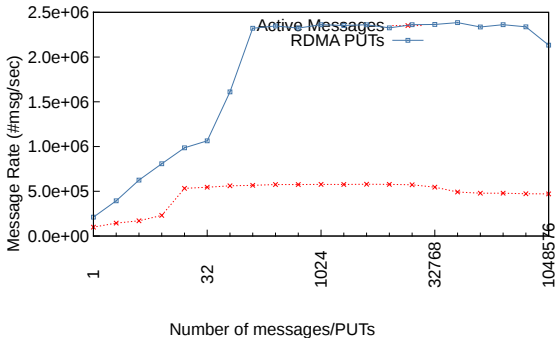
(a) Inter-node Unidirectional Bandwidth (bytes/sec)



(b) Inter-node Bidirectional Bandwidth (bytes/sec)



(c) Inter-node Unidirectional Message Rate (msg/sec)



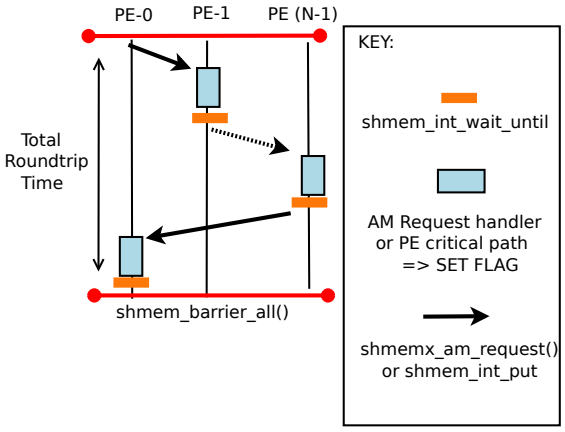
(d) Inter-node Bidirectional Message Rate (msg/sec)

Figure 13.3: Communication line diagrams and performance results for bandwidth and message rates

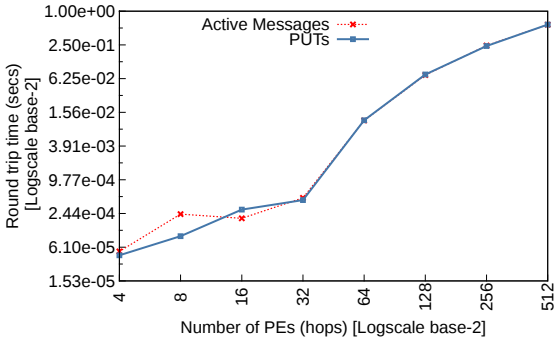
different compute nodes. The results for the token-ring based tests and the miniapp (Traveling Salesman Problem) were obtained by launching multiple number of PEs - 2 through 512 and 256 respectively, each bound to a specific core across multiple nodes.

13.4.3 Performance Study

This section presents a performance analysis of the prototype implementation. As noted in previous sections, the proposed AM interface enables transfer of data buffers



(a) Communication Line Diagram using Active Messages and standard OpenSHMEM PUTs



(b) Round Trip Latency (seconds)

Figure 13.4: Empirical study of Token Ring based communication pattern

in addition to the invocation of remote handlers. This section investigates the feasibility of using Active Messages instead of OpenSHMEM point-to-point operations to transfer data among PEs. It must be noted that the results presented as part of this study correspond to the prototype implementation of Active Messages and is meant to highlight the difference in behavior between the prototype and one-sided operations. The reader must bear in mind that fine-tuned implementations of Active Messages can exploit additional features of the underlying hardware stack to achieve better performance.

As part of this study, a microbenchmark test suite was designed to measure the achievable unidirectional and bidirectional bandwidth and message rate during data transfers using both the mechanisms[4]. The communication patterns within the microbenchmark suite use multiple PEs that communicate using either the proposed

[4]The microbenchmark test suite for OpenSHMEM AM is hosted as a git repository at `https://github.com/sidjana/shmem_am_testsuite`

AM interface (*shmemx_am_request() /shmemx_am_quiet()*) or point-to-point PUT operations (*shmem_putmem() / shmem_quiet()*). These benchmarks evaluate the unidirectional and bidirectional bandwidth and message rates. In addition they also measure the round-trip latency of a token-ring topology.

Bandwidth:

Test Design: The execution time of the communication pattern was monitored for different payload sizes from 1B through 2KB. We do not measure transfers beyond the 2KB size because we learned that Active Messages are not a good data transport mechanism for bulk payloads.

Test Results: The unidirectional and bidirectional bandwidth using the proposed AM interface and the standard OpenSHMEM point-to-point PUT operations are depicted in Figure 13.3a and 13.3b, respectively. The x-axis corresponds to the size of the data payload transferred (in log_2 scale) across the network. The value of the achievable bandwidth (in bytes/second) is plotted on the y-axis. From the figures, we observe that a higher bandwidth is achievable while using point-to-point PUT operations as compared to using the prototype implementation. This is not surprising since the AM request mechanism is associated with multiple cost factors. At the source, the PE is responsible for copying the contents of the data payload from the user's address space to a temporary buffer that gets packed along with additional information necessary for the target PE to respond. At the destination, the PE is responsible for detecting an incoming AM request, launching the corresponding AM handler and then notifying the source about the completion of the handler execution. It can be observed that the impact of these factors increases with the size of the data

payload being transferred. This leads to an important conclusion that the purpose of using an AM is not to transfer data payloads, but rather to trigger computation at the same location as the transferred payload.

Message Rate:

Test Design: The execution time of the pattern was monitored for different number of messages initiated consecutively with minimal payload (4 bytes).

Test Results: The unidirectional and bidirectional message rate using the proposed AM interface and standard OpenSHMEM point-to-point PUT operations are depicted in Figure 13.3c and 13.3d, respectively. The x-axis corresponds to the number of messages (PUT operations / AM requests) initiated before waiting for completion (in logscale, base 2)[5]. The value of the achievable message rate (in messages/second) is plotted on the y-axis. Similar to the bandwidth tests above, we observe that there is a negative impact on the message rate of the transfers. There is a significant impact when the number of consecutive AM requests increases beyond 32. The drop in message rate while using the AM interface is about 3X in case of unidirectional tests and 5X in case of bidirectional.

Token-ring Communication Pattern:

Launching an AM is similar to triggering an event on a remote destination. Therefore, incorporating the support for AM into a programming model enables

[5]Completion of a PUT operation / AM request is ensured by calling the functions - shmem_quiet() / shmemx_am_quiet(), respectively

applications to be built using communication patterns that rely on sending and responding to asynchronous events. It enables the design of patterns wherein a single AM request can be used to propagate a signal across other remote PEs. In order to ensure high performance, it is essential that implementations invest as few CPU cycles as possible between detecting an AM request and executing the AM handler. In order to study the impact on latency of an AM request as it hops across multiple PEs, two synthetic microbenchmarks were designed to mimic a token-ring based communication topology. The benchmark was designed such that the token was propagated using either standard OpenSHMEM point-to-point synchronization or the proposed AM interface.

Test Design: The line diagrams of these patterns are depicted in Figure 13.4a. As shown, the transfer of the token is achieved by transferring a single integer across consecutive pairs of PE in the ring topology. In an N-PE system, a PE k sends a signal (either via an AM or a PUT) to PE ((k+1)%N) which then propagates the same to the PE ((k+2)%N), and so on. PE (N-1) sends the signal back to PE-0 thereby completing a single round-trip. The motivation for such a design is to measure the total round-trip latency for different ring sizes.

Test Results: As part of this study, we study the impact on the time taken to complete a single round trip as a function of the number of hops (PEs) within the ring. In an implementation with minimal software overhead during AM handler management, the expectation is that the total round-trip time scales almost linearly with the number of hops. Figure 13.4b shows the empirical results for this test. The x-axis represents the number of hops (the number of PEs) in a single round-trip.

The y-axis corresponds to the total time taken for the token initiated by PE-0 (in the form of an AM request or a PUT) to pass through all the PEs before returning to PE-0. From the graph we observe that the latency for the round-trip latency for both the approaches is almost the same. This can be attributed to the fact that the difference between the latencies of transferring data payloads using AM and standard PUT is more tangible for large data payloads. Since this pattern used a single 4-byte integer to represent the token, the performance is similar.

Summary:

From the bandwidth and message rate plots, we learn that the purpose of using Active Messages is *not* to transfer data payloads. To achieve *closer-to-metal* bandwidth and message rates for data transfers, the OpenSHMEM programmer is better off using traditional point-to-point operations that are currently provided by the standard. From the token-ring experiment, it can be seen that Active Messages are better suited for triggering specific events on remote PEs with the added benefit of providing a means for productivity (due to its coding style) and no significant loss in performance.

13.4.4 The Traveling Salesman Problem (TSP)

In order to study the impact of the proposed AM interface, the Traveling Salesman Problem (TSP) miniapp was chosen as the target benchmark because the algorithm can be divided into multiple independent tasks. This gives an opportunity to exploit asynchronous computation within the algorithm.

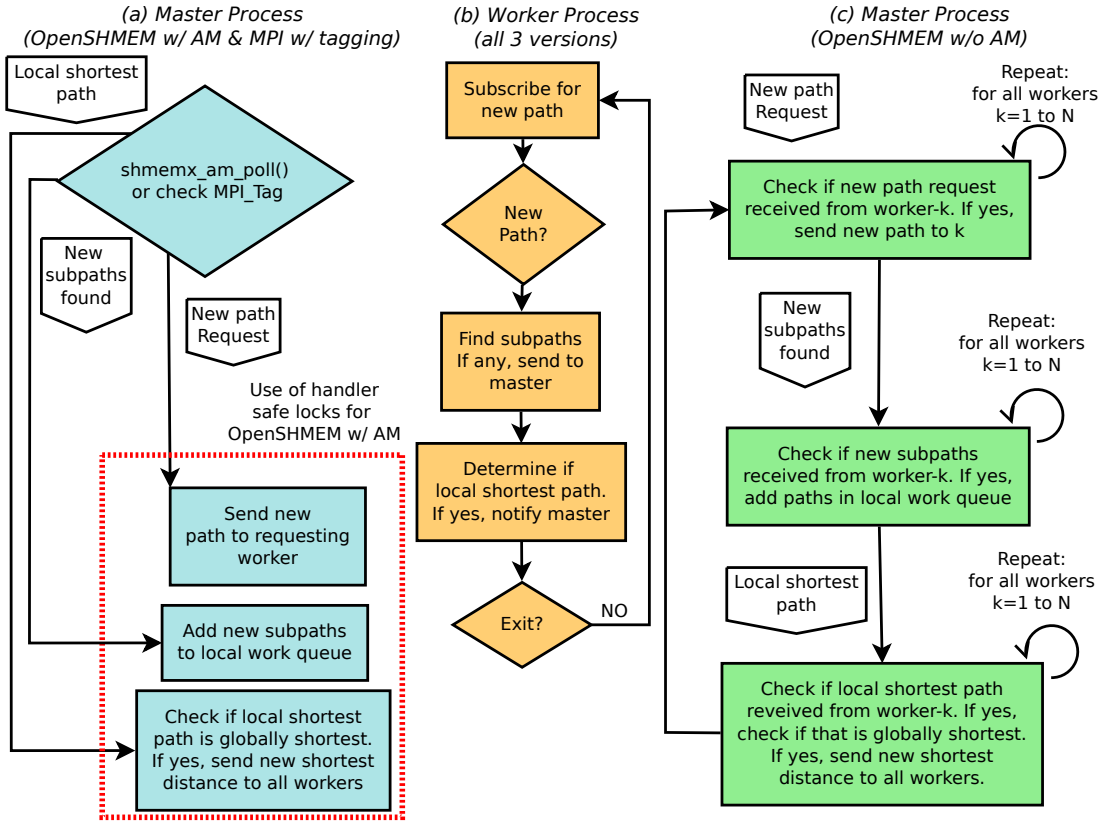
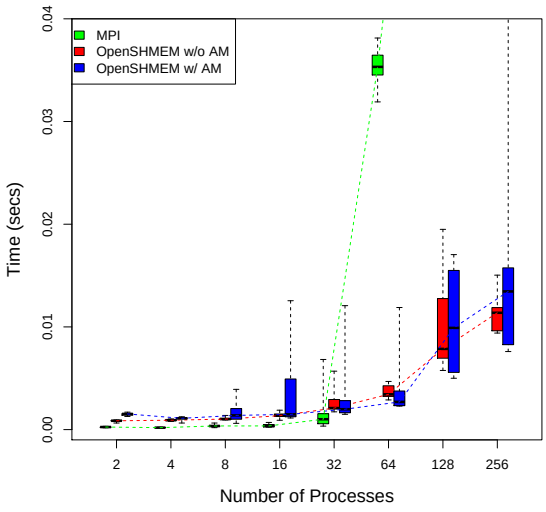
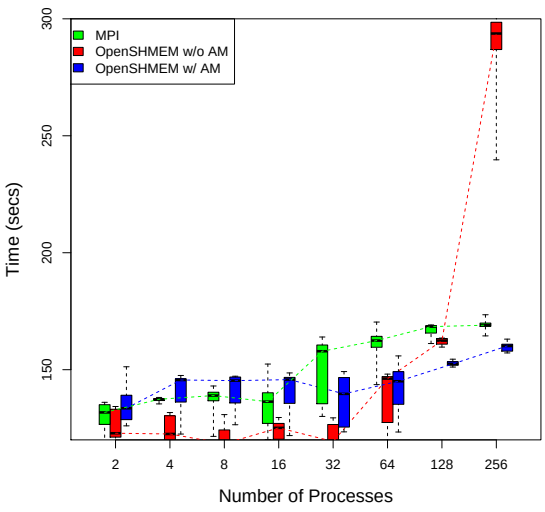


Figure 13.5: Flow diagram of the master and worker processes for all three versions of the Traveling Salesman Problem (TSP): (a) Master for both OpenSHMEM with AM and MPI, (b) Worker for all three versions, (c) Master for OpenSHMEM without AM.

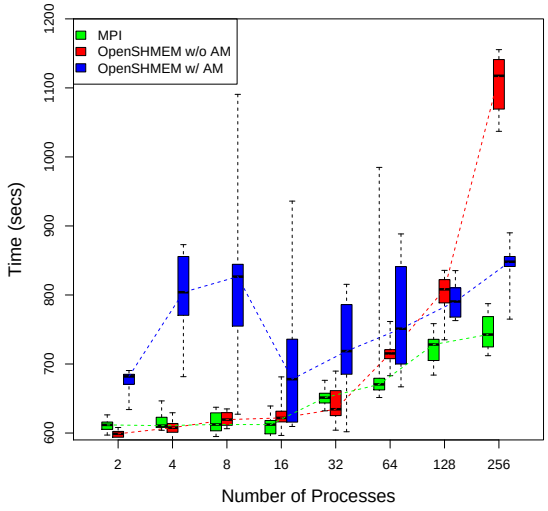
Miniapp Versions The TSP miniapp uses a master-worker communication pattern. The master PE is responsible for reading an input cost matrix and for assigning different paths to the worker PEs. The worker PEs in turn are responsible for either breaking down a path into smaller subpaths, determining the shortest distance for a given path, or requesting a new path from the master PE. As part of the experiment, the performance of three different versions of the miniapp were evaluated. The difference between the three is in the deign of the master process. Active Messages provide a mechanism to map a function handler to an identifier. We noted that



(a) Data size: 04 cities. Problem size: 16(4x4)



(b) Data size: 14 cities. Problem size: 196(14x14)



(c) Data size: 15 cities. Problem size: 255 (15x15)

Figure 13.6: Performance results of a traveling salesman problem written - MPI (in GREEN) v/s standard OpenSHMEM (in RED) v/s OpenSHMEM with the proposed AM interface (in BLUE). The dashed line connects all the medians of the box-plots that correspond to each of the versions.

this is similar to the message-tagging mechanism provided by MPI. Not surprisingly, the logical flow of the algorithms that used MPI tag-matching algorithm and the OpenSHMEM with AM was similar. This is depicted in Figure 13.5(a) and 13.5(b). The flow of the algorithm used to design the miniapp using standard OpenSHMEM without any AM interface is illustrated in Figure 13.5(b) and 13.5(c). The design details are described below:

(i) With AM interface / MPI Tag-matching: In the OpenSHMEM version that uses the AM interface, the worker PE communicates with the master PE using Active Messages[6]. Each request contains the id of the function handler which on detection is triggered by the master PE. Since the handler function is presented with a pointer to the contents of the message, it is not responsible for costs associated with memory management. The MPI implementation[7] exploits the availability of message tags to differentiate between different messages sent by the worker ranks. In this case, the worker rank communicates with the master by appending MPI messages with with tag-ids that correspond to different tasks. Because of this feature, the design of the master rank is similar to the master PE that uses the OpenSHMEM AM interface. One of the challenges in designing the OpenSHMEM version with AM is the need to share multiple data structures among different AM handlers. To ensure correctness and avoid race conditions, it becomes essential to use handler-safe locks to ensure exclusive access to these data structures. This in turn leads to a potential rise in lock contention, and hence performance degradation for small data sets.

[6]The version of the TSP miniapp using the proposed AM interface is hosted at https://github.com/sidjana/traveling_salesman_shmem_am/tree/master/shmem_MMPQ

[7]The version of the TSP miniapp using MPI-tagging approach is hosted at http://www.eecg.toronto.edu/ amza/ece1747h/homeworks/examples/MPI

(ii) Without AM interface: In this case, each worker PE remotely updates an assigned bucket stored on the master PE, using point-to-point communication operations[8]. The master PE is in charge of maintaining the remotely accessible buckets. Since the communication pattern relies on a single master and multiple workers, there is a need to assign a different bucket for each worker PE. This helps avoid network congestion at the master PE due to repeated use of the distributed locking interfaces or atomic operations provided by OpenSHMEM. The disadvantage of this approach though is that the master PE has to repeatedly scan through all the buckets to check for any updates by the PEs. The cost of this access takes a toll on the performance for large count of buckets / worker PEs. Here the cost associated with accessing the buckets increases linearly with the number of worker PEs, this design has the potential for severe performance degradation at large PE count.

Experiment Methodology Three different implementations of the TSP were chosen for the comparative study - two of which were designed using OpenSHMEM (as explained above) and the third, using the MPI two-sided model. Three different problem sizes were chosen (number of cities = 4, 14, 15). The results are shown in Figure 13.6(a), (b), and (c) respectively. Due to the highly irregular and dynamic nature of this miniapp, the execution time is prone to high variation. The results are therefore presented as a box plot distribution, where each plot for a given problem size and PE count corresponds to a distribution of 20 runs of the miniapp version. The X-axis plots the number of PEs used for execution. The Y-axis corresponds to the time taken (in seconds) to arrive at the solution (shortest path).

[8]The version of the TSP miniapp using standard OpenSHMEM interface is hosted at https://github.com/sidjana/traveling_salesman_shmem_am/tree/master/shmem_pure

Empirical Results The major observations are as follows:

- With a small input data set (Figure 13.6a), we see a severe performance degradation with the MPI version. This can be attributed to the fact that the implementation heavily relies on the traditional two-sided blocking communication to transfer data among the master and multiple worker processes. The use of either the proposed AM interface or the standard non-blocking one-sided communication both alleviate this penalty.

- With large data sets (Figure 13.6b and 13.6c), we see that the OpenSHMEM version that uses the standard interface suffers a significant performance loss when scaled beyond one node (number of PEs > 32). Since this version maintains a separate bucket for each worker, the master suffers a slowdown due to the cost associated with scanning multiple buckets iteratively. This cost is completely eliminated in case of the AM version where no CPU cycles are invested in determining the status of worker processes. Instead, the unordered incoming requests initiated by the worker processes are are asynchronously executed at the master process.

- The plots also show that for large data sets and higher process count, the performance between the MPI and the OpenSHMEM with AM versions are close to each other. This is because the MPI implementation relies on tag matching to detect the task to be executed. Functionally, this is similar to the underlying AM implementation where the handler functions are invoked by matching the handler-id embedded within the incoming AM request.

- There is an interesting behavior by the OpenSHMEM version that uses AM interfaces for the input data set with 15 cities (Figure 13.6c). We see a very high variation among execution time for small PE count. This high variation can be attributed to use of handler-safe locks among the AM request handlers, thereby leading to heavy lock contention. This variation reduces for higher PE count which can be explained by greater overlap of the computation at the worker with that of the AM handler at the master. Since the MPI version in synchronous, it does not rely on any locking mechanism thereby avoiding the high variation in execution time for this data set. The lesson learned here is that in order to exploit asynchronous execution of AM handlers, the use of shared data structures, and hence the use of handler-safe locks should be limited. Despite this, we observe that using Active Messages gives a high performance gain at scale over the version that uses the standard OpenSHMEM interfaces.

13.5 Related Work

Active Messages were first introduced by Eicken et al.[49]. The original motivation was to enable communication/computation overlap and shift the responsibility of tolerating latency from the underlying hardware to the programmers/compilers. The authors described a programming model called *Split-C* that enables remote one-sided communication to be executed using Active Messages.

Multiple low-level communication libraries that support Active Messages include GASNet[27], UCX[123], LAPI[138], and PAMI[97].

At a higher level in the software stack, the execution model of Active Messages

can be compared to programming models that enable explicit launching of tasks among processes in a distributed environment. These include ParalleX[87] (*parcels*), UPC++[163] (*function shipping*), Charm++[4] (*entry methods*), Chapel[61] (*begin-at*), CAF 2.0[133] (*spawn*), and GASPI[7] (*passive communication*).

Research efforts have been made to also introduce Active Messages within MPI[154, 26, 162, 36, 68]. Some of these approaches like AM++[154] and AMMPI[26] are designed on top of existing MPI libraries. Alternative approaches like Zhao et al.[162] describe techniques for incorporating Active Messages directly within the MPI runtime (e.g. by extension the semantics of *MPI_Accumulate* within MPICH).

Unlike Active Messages that enable inter-process parallelism using explicitly specified computation units, some programming models offer constructs that help exploit dynamic parallelism within a process. Programming models like X10[35], Titanium[158], Chapel[61], and those based on the Habanero framework (which in turn is based on X10's *finish-async* constructs) - Habanero Java[31], Habanero C[130], Habanero UPC[129], Habanero-C MPI[36], and Habanero-UPC++[98], all provide tasking mechanisms that incorporate dynamic load-balancing strategies by scheduling work across a dedicated pool of workers.

13.6 Chapter Summary

This chapter explores the feasibility of introducing Active Messages (AM) within the OpenSHMEM programming model. As part of this work, an API was proposed along with an empirical study of a prototype implementation within the OpenSHMEM reference implementation.

Synthetic microbenchmarks were used to compare the performance of data movement using the proposed AM interface and the existing standard OpenSHMEM remote write operations. The results show that the primary intent of using Active Messages should not be to transfer data to remote locations. Instead, the purpose is to facilitate the transfer of computation to a destination that hosts the data that needs to be computed upon. Nevertheless, a simple interface has been proposed that allows a process to attach a user buffer to the Active Message request. One potential approach to avoid the poor bandwidth costs of appending data payloads to an AM request maybe to instead perform a standard PUT operation followed by shmem_quiet and then the AM request with zero bytes of payload. This may help applications exploit the RDMA capabilities of underlying network.

Another noteworthy point in the proposed semantics is the lack of restriction on the size of the data payload that is appended to an AM request. One possible modification to this approach could be where the interface provides multiple variations for different sized data payloads while initiating Active Message. While this provides greater flexibility to the end user, there is also an increase of burden on the user to choose the right interface to achieve the expected performance. Examples of low-level communication libraries that do provide such interfaces include GASNet[27] (using *medium*, *long*, and *longasync* AM requests) and UCX[123] (using *short*, *buffered*, and *zero-copy* AM requests).

On comparing the performance of different implementations of a miniapp (the Traveling Salesman Problem), it was learned that while using Active Messages, sharing of data structures among different handlers of the same PE should be avoided,

otherwise there is a potential for performance loss due to contention among *handler-safe locks*. However, it was observed that despite such a design of the algorithm, the miniapp was able to achieve significant performance improvement over the version that solely relied on using the standard OpenSHMEM interfaces.

For more details about the topics discussed in this Chapter, the interested reader is directed to the literature documented by Jana et al. under [82].

Chapter 14

Future Work

There have been a handful of recent research efforts towards integrating hardware power management techniques with the software stack. As an initial step, the focus has been more on power monitoring infrastructure.

Three different software frameworks are undergoing development that cater to this need. The HPC Power API[77] led by Sandia National Laboratory is a proposed de facto standard that attempts to provide a power-management interface for all HPC software - from job schedulers, to operating systems, to user applications. The Global Extensible Open Power Manager (GEOPM)[48] led by Intel, is an open source framework that attempts to dynamically control the power consumption of MPI jobs launched in a distributed environment. It provides a plug-in architecture using which multiple power-control algorithms can be implemented. Redfish[14] led by DMTF Scalable Platforms Management Forum is an attempt to develop an open industry standard specification that provides interfaces to monitor various "IPMI-class" data

from different components of the system.

As a follow up for the work described in this thesis, it is crucial to leverage the lessons learned to the future development of the Power API and GEO-PM. Currently, these interfaces provide limited "gateways" for application and middleware developers to monitor and control the behavior of the application. The number of application design factors discussed in Chapters 5 through 9 can be incorporated within these frameworks to achieve power management at finer granularity.

Chapter 15

Conclusion

This dissertation presents an in-depth analysis of different factors that affect the energy consumption of distributed memory HPC applications. The factors discussed were broadly divided into three different categories on the basis of whether they relate to the communication model, memory model, execution model.

A great emphasis was put on the inter-process communication stack.

As part of this thesis, at first, empirical evidence was discussed that highlighted the fact that adopting a race-to-halt approach (*i.e. running your application at the fastest speed possible*), is not always the right approach. This was followed by an overview of multiple variables within the HPC runtime that have the potential to impact the energy costs of a distributed memory application. Factors corresponding to the memory model, communication model, and the execution model were discussed.

A number of factors within an application design were shown to impact energy

profiles of communication-intensive kernels. These include - the total size of the data payload being transferred, the number of explicit IPC calls, even the data-access patterns used within the kernels. It was shown that choosing the right access pattern can lead to 40% savings in Energy Delay Product of the kernel. Design factors within the transport layer were discussed. It was shown that using OpenIB over Infiniband can give up to 760x improvement in bytes/joules over TCP over Ethernet. Up to 25x improvement in bytes/joule can be achieved by choosing an eager communication protocol over rendezvous. In addition, design details of the middleware like the overhead of memory copy operations, use of pinned-down memory and the impact of using an additional service thread.

The common approach of using DVFS as a means to achieving energy efficiency was introduced. This was complemented with a discussion of how the extent of impact of using DVFS depend on factors like the hardware organization of the underlying processor. Negative impact of using DVFS within a communication intensive kernel was discussed.

As a solution of the above challenges, one proposed approach was to adopt execution models that deviate from the common distributed memory programming models like MPI and OpenSHMEM. We revisit a parallel programming construct from early 90's - Active Messages and incorporate it within the execution model of OpenSHMEM. We discuss a prototype implementations along with empirical results showing significant performance benefits at scale.

The work presented in this thesis is intended to act as a guidance to application programmers and system developers of current and future systems alike.

Appendix A

Test Platform

A.1 System-A at OLCF: RAPL monitoring

In order to monitor energy consumption by different components of a compute node (cores, socket, memory), we used Intel's Running Average Power Limiting (RAPL) interface[79]. Figure A.1 illustrates our experimental setup which incorporates this interface by monitoring the thermal and power management values of the model-specific registers (MSRs) exposed by the Intel Sandy Bridge processor, E5-2670. In order to read the RAPL counters in MSRs from the device file system (/dev/cpu/*/msr on *devfs*), we used the RAPL component provided by PAPI v5.1[116]. In addition, we used Vampir Trace[94] for fine-grained instrumentation of our synthetic microbenchmarks.

Verifications by David et al.[44], Hackenberg et al.[62], and Dongarra et al.[45] provide empirical evidence of a high correlation between the energy consumption

Table A.1: Test machine and environment details

Processor	Intel Xeon CPU E5-2670
Microarchitecture	Intel's Sandy Bridge
Maximum Thermal Design Power (TDP)	115 Watts
Hyperthreading support	Disabled
Sockets	2
Cores/socket	8
L1 cache size (per core)	32KB
L2 cache size (per core)	256KB
L3 cache size (shared - 1/socket)	20MB
Infiniband card	Mellanox MT26428 [ConnectX PCIe2.0 5GT/s]
Infiniband switch	InfiniScale 36-Port QSFP 40Gb/s, MTS3600
Compiler	gcc version 4.4.6
Compiler flags used	-O3
OpenSHMEM version	Mellanox OpenSHMEM ver. 2.2-23513

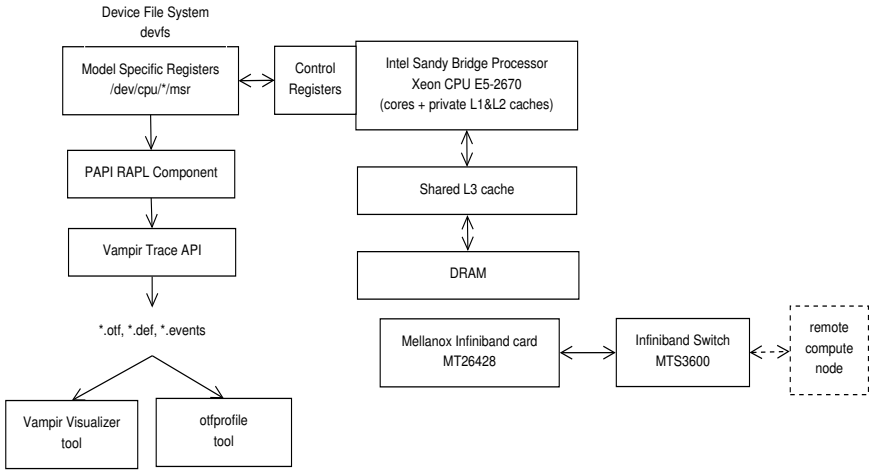


Figure A.1: Experimental Setup incorporating Intel's RAPL interface for fine-grained power monitoring

readings provided by the RAPL interface and direct power measurements. However, readings provided by this interface have certain shortcomings due to its model-based approach for estimating the metrics[62]. The fact that energy values of DRAM as reported by RAPL only take into account the memory accesses initiated by the CPU and not other I/O devices (e.g. the network card), was a major obstacle in this study. Nevertheless, these DRAM-specific values are a good estimation of the impact of the energy consumption due to data transfers between the CPU and the memory. Any direct memory accesses by the interconnect (without the participation of the CPU) would only lead to further increase in the impact of the power/energy consumption. Due to space constraints, we do not present these in this work.

Statistical evidence[44] indicates that the estimated energy consumption[1] by the cores and sockets, are sufficient to understand the impact of an application.

On our system the time-window for the RAPL interface was 0.046 seconds. We found this configuration acceptable for studying the behavior of the energy and power consumption patterns by different OpenSHMEM interfaces.

A discussion on the power management capabilities of Intel's Sandy Bridge processors by Rotem et al.[131], indicates that these modern architectures expose energy estimation and power capping ability by exposing the RAPL interface. The authors acknowledge that power conservation techniques like frequently triggering sleep states of processors affect performance of applications and in some cases leads to an increase in energy consumption. Using a *demotion* algorithm, these processors can override

[1]It must be noted that Intel's RAPL does not provide power readings. It estimates energy values, that need to be averaged over multiple updates in order to obtain a reliable measure.

the operating systems decision to enter into a low power state. Such fine grained estimation which incorporates the low powered C-states of processors indicate a reliable metric.

It is crucial to note some of the observations as outlined by Hackenberg et al.[62] while using RAPL on 1P and 2P Sandy bridge processors:

- RAPL readings add an overhead of about 1.4 microseconds for reading all the domains
- RAPL underestimates the effect of power consumption due to hyperthreading
- While RAPL provides estimates for energy consumption by the DRAM, it does so based on memory accesses about which the processor die is aware of. As a result, energy consumption due to memory accesses through the DMA controllers initiated via the NICs are remain unaccounted for.

Despite these, empirical studies indicate that the readings exposed via this interface have a high correlation with those obtained by external monitoring devices. Dongarra et al.[45] underline the importance of RAPL in analyzing the power consumption profile of applications, at a high sampling rate (this is in the light of about 1.4 microseconds overhead for monitoring the counter readings). This is reflected in experiments by Hackenberg et al.[62] who include in their work, empirical verification of RAPL by incorporating readings from external power monitoring instruments - ZES, iDRAC and PDUs. Additionally, RAPL provides energy readings on a per-component basis including Thus for the purpose of understanding the power

Table A.2: Test-Platform characteristics of SystemG

Processor	Intel Xeon CPU E5462
Microarchitecture	Intel's Sandy Bridge
Operating Frequency	2.8 GHz
Maximum Thermal Design Power (TDP)	80 Watts
Hyperthreading support	Disabled
Infiniband card	Mellanox MT26428, fw-ver:2.5.9
Linux kernel version	2.6.32 x86_64
Compiler	gcc version 4.4.4
Compiler flags used	-O3
Power Sampling rate	10ms

consumption signature during the entire run of an application, RAPL appears to be a good candidate.

Reducing Background Noise in Energy Readings To reduce OS noise and avoid other processes from being scheduled on the monitored socket, we used Linux *CPU shielding*[91]. This ensured that all unrelated processes/threads (including most OS service threads) were scheduled on the extra unmonitored socket on the compute node (refer to Table A.1 for the machine details). We verified this approach by observing a steady power consumption of 3.786 Watts when none of our experimental processes were scheduled on the monitored socket.

A.2 System-B at VirginiaTech: PowerPack monitoring

Our study was aimed at performing a fine-grained analysis of the impact on different components of a distributed system, namely, the cores, the socket, the motherboard, the memory unit, and the entire compute-node as a whole. The experiments were performed for two different implementations of MPI - Open MPI[51]

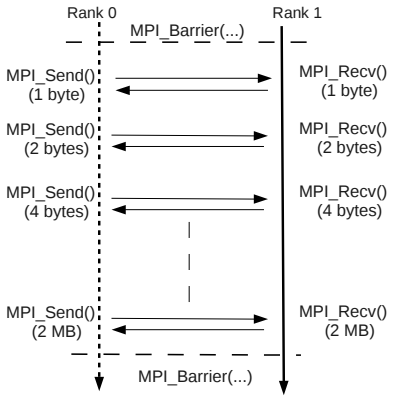


Figure A.2: Synthetic microbenchmark used for evaluating energy and power consumption by varying the total size of data payload and the number of fragments

and MVAPICH2[108]. We observed similar behavior between the two MPI implementations. Due to space constraints, we discuss the impact of only OpenMPI's implementation of data transfers on two major components that contribute to the total power consumption of a system, viz. the CPU cores and the memory. While the network card forms an important component of a distributed system, past study indicates that its impact on the total power consumption by a system is about 1%[50]. We therefore omit any further discussion on the impact of NIC from the rest of the text.

A.3 System-C at ZIH: HDEEM monitoring

The details of the experimental setup are listed in Table A.3. All experiments were conducted using two OpenSHMEM processes (PEs) where each process was launched and bound to one of the cores of an Intel Sandy Bridge processor on a separate node.

Table A.3: Characteristics of the power monitored node

Processor	Intel Xeon CPU E5-2690
Microarchitecture	Intel's Sandy Bridge
Hyperthreading support	Disabled
Main Memory	32 GB
Infiniband card	Mellanox MT27500, ConnectX-3
Linux kernel version	2.6.32 x86_64

These two nodes are prototypes for an upcoming installation at the University of Technology Dresden, which are instrumented for fine-grained accurate power measurement[2]. Each node has instrumented voltage regulators (VRs) that are sampled with a sampling frequency of 1 KHz for both sockets and the four voltage lanes of the DIMMs on board. With the help of an FPGA, a digital filter is applied to smooth the samples. Furthermore, a linear correction is applied to the measurement data coming from the VRs in order to ensure an error margin not exceeding 3 %. Our study was aimed at performing a fine-grained analysis of the impact on two main components that dictate the energy and power consumption of a system - the CPU and the DRAM. For studies on large scale systems, the contribution of the interconnect and the network topologies becomes crucial, as highlighted by recent study[99].

[2]More information about the High Definition Energy Efficiency Monitoring (HDEEM) project is available at `http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/projekte/hdeem`

Appendix B

Microbenchmark Design

This section lists the pseudo-codes for various synthetic microbenchmarks used throughout the experiments described in this work. The line diagrams corresponding to these snippets are depicted within the main text.

```
shmem_barrier_all ()
if (myid == sender)
  clflush (...)
  foreach (src_buffer[i])
  do
    shmem_put (src_buffer[i], dest_buffer[i])
  done
  shmem_quiet ()
endif
shmem_barrier_all ()
```

Listing B.1: "Code Snippet: Remote write implementation: Servicing PUTs with no participation by the receiver"

```
index = 0; clflush (...)
shmem_barrier_all ()
if (myid == sender)
  foreach (src_buffer[i])
  do
    /* Pack all buffers into stemp_buff */
    stemp_buff[index] <- src_buffer[i]
    size = sizeof(src_buffer[i])
    index = index + size
  done
  shmem_put (stemp_buff, dtemp_buff)
  shmem_quiet ()
  shmem_int_swap (flag...)
  shmem_quiet ()
else /* myid == receiver */
  shmem_int_wait_until (flag...)
  foreach (dest_buffer[i])
  do
  /* Unpack all buffers from dtemp_buff */
    dest_buffer[i] <- dtemp_buff[index]
    size = sizeof(dest_buffer[i])
    index = index + size
  done
endif
```

Listing B.2: "Code snippet: Remote write implementation: Servicing PUTs with active participation by the receiver"

```
shmem_barrier_all ()
```

```
if (myid == sender)
  clflush (...)
  foreach (src_buffer[i])
  do
    shmem_put (src_buffer[i],dtemp_buffer[i])
  done
  shmem_quiet ()
endif
....
/****** software agent *******/
while(true)
do
  if(new_packet_arrived() == true)
    /* Unpack all buffers from dtemp_buff */
      dest_buffer[i] <- dtemp_buff[index]
      size = sizeof(dest_buffer[i])
      index = index + size
  endif
done
```

Listing B.3: "Code snippet: Remote write implementation: Servicing PUTs with an additional thread supporting the receiver"

```
/*MAX_WRK_SIZE: is the maximum data
                payload to be transferred
                within a communication kernel
*/
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
for(j=1;j<=MAX_WRK_SIZE;j*=2)
{
 for(frag_cnt=1; frag_cnt<=j; frag_cnt*=2)
 {
  bytes_per_msg = j / frag_cnt;
  MPI_Barrier();
  // START monitoring
  for (it=0; it<frag_cnt; it++)
   if (rank==0)
    MPI_Send(...,bytes_per_msg,MPI_BYTE,1,..);
   else
    MPI_Recv(...,bytes_per_msg,MPI_BYTE,0,..);
  // STOP monitoring
 }
}
```

Listing B.4: "Code snippet for the synthetic microbenchmark used for evaluating energy and power consumption by varying the total size of data payload and the number of fragments"

Bibliography

[1] N. Aboughazaleh, B. Childers, R. Melhem, and M. Craven. Collaborative compiler-os power management for time-sensitive applications. Technical report, Department of Computer Science, University of Pittsburgh, Department of Computer Science, University of Pittsburgh, 2002.

[2] N. AbouGhazaleh, D. Mossé, B. Childers, and R. Melhem. In L. Benini, M. Kandemir, and J. Ramanujam, editors, *Compilers and Operating Systems for Low Power*, chapter Toward the Placement of Power Management Points in Real-time Applications, pages 37–52. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[3] N. AbouGhazaleh, D. Mossé, B. R. Childers, and R. Melhem. Collaborative operating system and compiler power management for real-time applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(1):82–115, 2006.

[4] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale. Parallel programming with migratable objects: Charm++ in practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 647–658, Piscataway, NJ, USA, 2014. IEEE Press.

[5] J. Aliaga, M. Dolz, A. Martin, R. Mayo, and E. Quintana-Orti. Leveraging task-parallelism in energy-efficient ILU preconditioners. In A. Auweter, D. Kranzlmueller, A. Tahamtan, and A. Tjoa, editors, *ICT as Key Technology against Global Warming*, volume 7453 of *Lecture Notes in Computer Science*, pages 55–63. Springer Berlin Heidelberg, 2012.

[6] P. Alonso, M. Dolz, F. Igual, R. Mayo, and E. Quintana-Orti. Reducing energy consumption of dense linear algebra operations on hybrid cpu-gpu platforms. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 56–62, July 2012.

[7] T. Alrutz, J. Backhaus, T. Brandes, V. End, T. Gerhold, A. Geiger, D. Grünewald, V. Heuveline, J. Jägersküpper, A. Knüpfer, O. Krzikalla, E. Kügeler, C. Lojewski, G. Lonsdale, R. Müller-Pfefferkorn, W. Nagel, L. Oden, F.-J. Pfreundt, M. Rahn, M. Sattler, M. Schmidtobreick, A. Schiller, C. Simmendinger, T. Soddemann, G. Sutmann, H. Weber, and J.-P. Weiss. *GASPI – A Partitioned Global Address Space Programming Interface*, pages 135–136. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[8] S. Amarasinghe, M. Hall, R. Lathin, K. Pingarli, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, and K. Yelick. ASCR Programming Challenges for Exasacle Computing. 2011.

[9] AMD. ACP The Truth About Power Consumption Starts Here.

[10] AMD. Linux tuning guide, amd opteron 6200 series processors. April 2012.

[11] M. Annavaram. Energy per instruction trends in. intel microprocessors. *Technology Intel Magazine*, 2006.

[12] K. Asanovic. Energy-exposed instruction set architectures. In *In Progress Session, Sixth International Symposium on High Performance Computer Architecture*, 2000.

[13] K. Asanović, M. Hampton, R. Krashinsky, and E. Witchel. *Energy-Exposed Instruction Sets*, pages 79–98. Springer US, Boston, MA, 2002.

[14] J. Autor. Introduction to Redfish. November 2015.

[15] A. Azevedo, R. Cornea, I. Issenin, R. Gupta, N. Dutt, A. Nicolau, and A. Veidenbaum. Architectural and compiler strategies for dynamic power management in the copper project. In *in the COPPER project. International Workshop on Innovative Architecture*, 2001.

[16] P. Balaprakash, L. A. B. Gomez, M.-S. Bouguerra, S. M. Wild, F. Cappello, and P. D. Hovland. Energy-performance tradeoffs in multilevel checkpoint strategies. 2014.

[17] S. Barrachina, M. Barreda, S. Catal, M. F. Dolz, R. Mayo, and E. S. Quintana-ort. An Integrated Framework for Power-Performance Analysis of Parallel Scientific Workloads. In *The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 114–119, 2013.

[18] M. Barreda, S. Cataln, M. Dolz, R. Mayo, and E. Quintana-Ort. Automatic detection of power bottlenecks in parallel scientific applications. *Computer Science - Research and Development*, 29(3-4):1–9, 2013.

[19] B. Barrett. OpenMPI Data Transfer, December 2012. Detailed overview of the OpenMPI data transfer system.

[20] B. Barrett. OpenMPI Data Transfer. http://www.open-mpi.org/video/internals/Sandia_BrianBarrett-1up.pdf, December 2012. Detailed overview of the OpenMPI data transfer system.

[21] J. M. Bearfield. Power Control Design Key to Realizing InfiniBand Benefits, Texas Instruments Inc. November 2001.

[22] F. Bellosa. The case for event-driven energy accounting. Technical report.

[23] Y. Ben-Itzhak, I. Cidon, and A. Kolodny. Performance and power aware cmp thread allocation modeling. In *Proceedings of the 5th international conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 232–246, Berlin, Heidelberg, 2010. Springer-Verlag.

[24] S. Benedict. Review: Energy-aware performance analysis methodologies for hpc architectures-an exploratory study. *J. Netw. Comput. Appl.*, 35(6):1709–1719, Nov. 2012.

[25] L. Benini, M. Kandemir, and J. Ramanujam, editors. *Compilers and Operating Systems for Low Power*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[26] D. Bonachea. AMMPI: Active Messages over MPI - Quick Overview.

[27] D. Bonachea. Gasnet specification, v1.1. Technical report, Berkeley, CA, USA, 2002.

[28] D. Bonachea. Active Messages - Extract from GASNET spec, 2006.

[29] T. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12, 2011.

[30] M. Castillo, J. Fernandez, R. Mayo, E. Quintana-Orti, and V. Roca. Analysis of strategies to save energy for message-passing dense linear algebra kernels. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 346–352, Feb 2012.

[31] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[32] J.-C. Chang, C.-Y. Lee, C.-J. Chen, and R.-G. Chang. Low power compiler optimization for pipelining scaling. In J.-S. Pan, C.-N. Yang, and C.-C. Lin, editors, *Advances in Intelligent Systems and Applications - Volume 2*, volume 21 of *Smart Innovation, Systems and Technologies*, pages 637–646. Springer Berlin Heidelberg, 2013.

[33] J.-M. Chang and M. Pedram. Register allocation and binding for low power. In *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, DAC '95, pages 29–35, New York, NY, USA, 1995. ACM.

[34] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA, 2010. ACM.

[35] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, Oct. 2005.

[36] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cav, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan. Integrating asynchronous task parallelism with mpi. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 712–725, May 2013.

[37] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc. A roofline model of energy. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 661–672, May 2013.

[38] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc. A theoretical framework for algorithm-architecture co-design. In *Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, Boston, MA, USA, May 2013.

[39] K. Choi, R. Soma, and M. Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, ISLPED '04, pages 174–179, New York, NY, USA, 2004. ACM.

[40] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(1):18–28, Jan 2005.

[41] P. Cicotti, A. Tiwari, and L. Carrington. Efficient speed (ES): adaptive DVFS and clock modulation for energy efficiency. In *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014, Madrid, Spain, September 22-26, 2014*, 2014.

[42] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. Identifying the optimal energy-efficient operating points of parallel workloads. In *2011 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2011, San Jose, California, USA, November 7-10, 2011*, pages 608–615, 2011.

[43] D. Culler, K. Keeton, C. Krumbein, L. T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, K. Wright, and C. Yoshikawa. Generic Active Message Interface Specification, 1994.

[44] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. RAPL: Memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194, 2010.

[45] J. Dongarra, H. Ltaief, P. Luszczek, and V. Weaver. Energy footprint of advanced dense numerical linear algebra using tile algorithms on multicore architectures. In *Cloud and Green Computing (CGC), 2012 Second International Conference on*, pages 274–281, 2012.

[46] J. Dongarra, H. Ltaief, P. Luszczek, and V. M. Weaver. Energy Footprint of Advanced Dense Numerical Linear Algebra Using Tile Algorithms on Multicore Architectures. *2012 Second International Conference on Cloud and Green Computing*, pages 274–281, 2012.

[47] Z. Du, H. Sun, Y. He, Y. He, D. A. Bader, and H. Zhang. Energy-efficient scheduling for best-effort interactive services to achieve high response quality. *Parallel and Distributed Processing Symposium, International*, 0:637–648, 2013.

[48] J. Eastep. An Overview of GEO (Global Extensible Open Power Manager).

[49] T. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. *[1992] Proceedings the*

19th Annual International Symposium on Computer Architecture, (May):256–266, 1992.

[50] X. Feng, R. Ge, and K. Cameron. Power and energy profiling of scientific applications on distributed systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 34–34, April 2005.

[51] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[52] M. Gamell, I. Rodero, M. Parashar, and R. Muralidhar. Exploring cross-layer power management for pgas applications on the scc platform. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 235–246, New York, NY, USA, 2012. ACM.

[53] 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, 2010.

[54] C. H. Gebotys. Low energy memory and register allocation using network flow. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 435–440, New York, NY, USA, 1997. ACM.

[55] Y. Georgiou, T. Cadeau, D. Glesser, D. Auble, M. Jette, and M. Hautreux. Energy accounting and control with slurm resource and job management system. In M. Chatterjee, J.-n. Cao, K. Kothapalli, and S. Rajsbaum, editors, *Distributed Computing and Networking*, volume 8314 of *Lecture Notes in Computer Science*, pages 96–118. Springer Berlin Heidelberg, 2014.

[56] S. Ghosh, S. Chandrasekaran, and B. Chapman. Poster: Statistical power and energy modeling of multi-gpu kernels. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1516–1516, 2012.

[57] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277–1284, Sep 1996.

[58] S. Gotz, T. Ilsche, J. Cardoso, J. Spillner, T. Kissinger, U. Assmann, W. Lehner, S. Götz, T. Ilsche, J. Cardoso, and J. Spillner. Software Energy-Efficiency with Sweet Spot Frequencies. 2014.

[59] R. L. Graham, P. Shamis, J. A. Kuehn, and S. W. Poole. Communication middleware overview. Tech Report ORNL/TM-2012/120, Oak Ridge National Laboratory (ORNL), 2012.

[60] H. P. C. T. group and E. S. S. C. a. O. at UH. Openshmem application programming interface, version 1.0. Technical report, University of Houston (UH), Oak Ridge National Laboratory (ORNL), 2012.

[61] B. Gu, W. Yu, and Y. Kwak. *Communication and Computation Overlap through Task Synchronization in Multi-locale Chapel Environment*, pages 285–292. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[62] D. Hackenberg, T. Ilsche, R. Schone, D. Molka, M. Schmidt, and W. Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 194–204, 2013.

[63] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, Jan. 2012.

[64] P. Hammarlund. 4th Generation Intel Core Processor , codenamed Haswell, 2013.

[65] M. Hampton. *Exposing Datapath Elements to Reduce Microprocessor Energy Consumption*. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2001.

[66] W. Heirman, S. Sarkar, T. E. Carlson, I. Hur, and L. Eeckhout. Power-aware multi-core simulation for early design stage hardware/software co-optimization. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 3–12, New York, NY, USA, 2012. ACM.

[67] T. Hoefler. Software and hardware techniques for power-efficient hpc networking. *Computing in Science Engineering*, 12(6):30–37, 2010.

[68] T. Hoefler and J. Willcock. *Active Messages for MPI*. 2009.

[69] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pages 8–11, Oct 1994.

[70] C. hsing Hsu and U. Kremer. Compiler-directed dynamic voltage scaling for memory-bound applications, 2002.

[71] C.-h. Hsu and W.-c. Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 1–, Washington, DC, USA, 2005. IEEE Computer Society.

[72] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 38–48, New York, NY, USA, 2003. ACM.

[73] C.-H. Hsu and U. Kremer. Single region vs. multiple regions: A comparison of different compiler-directed dynamic voltage scheduling approaches. In *Proceedings of the 2Nd International Conference on Power-aware Computer Systems*, PACS'02, pages 197–211, Berlin, Heidelberg, 2003. Springer-Verlag.

[74] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, ISLPED '01, pages 275–278, New York, NY, USA, 2001. ACM.

[75] S. Huang, Y. Luo, and W. Feng. Modeling and analysis of power in multicore network processors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, 2008.

[76] M. E. a. Ibrahim, M. Rupp, and H. S. E.-D. Compiler-based optimizations impact on embedded software power consumption. In *Circuits and Systems and TAISA Conference, 2009. NEWCAS-TAISA '09. Joint IEEE North-East Workshop on*, pages 1–4, June 2009.

[77] J. H. L. III. An Overview of Sandia National Laboratorys High Performance Computing Power Application Programming Interface (API) Specification.

[78] Intel. Intel® Data Direct I/O Technology (Intel® DDIO): A Primer, Revision 1.0. February 2012.

[79] Intel Corporation. Intel(R) 64 and IA-32 Architectures Software Developer's Manual Vol. 3B: System Programming Guide, Part-2. February 2014.

[80] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 93–, Washington, DC, USA, 2003. IEEE Computer Society.

[81] S. Jana and B. Chapman. Impact of Frequency Scaling on One Sided Remote Memory Accesses. In *9th International Conference on Partitioned Global Address Space Programming Models (PGAS 2015)*, September 2015.

[82] S. Jana, T. Curtis, D. Khaldi, and B. Chapman. Increasing Computational Asynchrony in OpenSHMEM with Active Messages. In *OpenSHMEM 2016: Third workshop on OpenSHMEM and Related Technologies*, August 2016.

[83] S. Jana, O. Hernandez, S. Poole, and B. Chapman. Power consumption due to data movement in distributed programming models. In F. Silva, I. Dutra, and V. Santos Costa, editors, *Euro-Par 2014 Parallel Processing*, volume 8632 of *Lecture Notes in Computer Science*, pages 366–378. Springer International Publishing, 2014.

[84] S. Jana, O. Hernandez, S. Poole, C.-H. Hsu, and B. Chapman. Analyzing the energy and power consumption of remote memory accesses in the openshmem model. In S. Poole, O. Hernandez, and P. Shamis, editors, *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, volume 8356 of *Lecture Notes in Computer Science*, pages 59–73. Springer International Publishing, 2014.

[85] S. Jana, J. Schuchart, and B. Chapman. Analysis of energy and performance of pgas-based data access patterns. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 15:1–15:10, New York, NY, USA, 2014. ACM.

[86] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras. Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 262:262–262:272, New York, NY, USA, 2014. ACM.

[87] H. Kaiser, M. Brodowicz, and T. Sterling. ParalleX An Advanced Parallel Execution Model for Scaling-Impaired Applications. In *Proceedings of the 2009*

International Conference on Parallel Processing Workshops, ICPPW '09, pages 394–401, Washington, DC, USA, 2009. IEEE Computer Society.

[88] M. Kandemir, A. Choudhary, P. Banerjee, J. Ramanujam, and N. Shenoy. Minimizing data and synchronization costs in one-way communication. *Parallel and Distributed Systems, IEEE Transactions on*, 11(12):1232–1251, Dec 2000.

[89] M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Power aware computing. chapter Compiler Optimizations for Low Power Systems, pages 191–210. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[90] 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 *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 33. IEEE Computer Society, 2005.

[91] M. Kerrisk. Linux programmer's manual. 2012.

[92] J. Kim, S. Yoo, and C. Kyung. Program phase-aware dynamic voltage scaling under variable computational workload and memory stall environment. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 30(1):110–123, 2011.

[93] M. Knobloch, M. Foszczynski, W. Homberg, D. Pleiter, and H. Bttiger. Mapping fine-grained power measurements to hpc application runtime characteristics on ibm power7. *Computer Science - Research and Development*, pages 1–9, 2013.

[94] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. Mller, and W. Nagel. The vampir performance analysis tool-set. In M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Tools for High Performance Computing*, pages 139–155. Springer Berlin Heidelberg, 2008.

[95] V. A. Korthikanti and G. Agha. Towards optimizing energy costs of algorithms for shared memory architectures. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 157–165, New York, NY, USA, 2010. ACM.

[96] V. A. Korthikanti and G. Agha. Towards optimizing energy costs of algorithms for shared memory architectures. *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures - SPAA '10*, page 157, 2010.

[97] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 763–773, May 2012.

[98] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar. Habaneroupc++: A compiler-free pgas library. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 5:1–5:10, New York, NY, USA, 2014. ACM.

[99] J. Laros, K. Pedretti, S. M. Kelly, W. Shu, K. Ferreira, J. Dyke, and C. Vaughan. *Energy-efficient high performance computing*. London: Springer, 2013.

[100] M. T.-C. Lee, M. Fujita, V. Tiwari, and S. Malik. Power analysis and minimization techniques for embedded dsp software. *IEEE Trans. Very Large Scale Integr. Syst.*, 5(1):123–135, Mar. 1997.

[101] S. Lee, A. Ermedahl, S. L. Min, and N. Chang. An accurate instruction-level energy consumption model for embedded risc processors. *SIGPLAN Not.*, 36(8):1–10, Aug. 2001.

[102] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis. Power management of datacenter workloads using per-core power gating. *IEEE Comput. Archit. Lett.*, 8(2):48–51, July 2009.

[103] D. Li, B. De Supinski, M. Schulz, K. Cameron, and D. Nikolopoulos. Hybrid mpi/openmp power-aware computing. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010.

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

[105] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu. Identifying Opportunities for Byte-Addressable Non-Volatile Memory in Extreme-Scale Scientific Applications. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 945–956, Washington, DC, USA, 2012. IEEE Computer Society.

[106] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. Adaptive, Transparent Frequency and Voltage Scaling of Communication Phases in MPI Programs. *ACM/IEEE SC 2006 Conference (SC'06)*, 2006.

[107] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance vmm-bypass i/o in virtual machines. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 3–3, Berkeley, CA, USA, 2006. USENIX Association.

[108] J. Liu, J. Wu, and D. Panda. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[109] C. Lively, V. Taylor, X. Wu, H.-C. Chang, C.-Y. Su, K. Cameron, S. Moore, and D. Terpstra. E-amom: an energy-aware modeling and optimization methodology for scientific applications. *Computer Science - Research and Development*, pages 1–14, 2013.

[110] A. Mainwaring and D. Culler. Active Message Applications Programming Interface and Communication Subsystem Organization. 1995.

[111] E. Markatos, M. Crovella, P. Das, C. Dubnicki, and T. LeBlanc. The effects of multiprogramming on barrier synchronization. In *Parallel and Distributed Processing, 1991. Proceedings of the Third IEEE Symposium on*, pages 662–669, 1991.

[112] Microsoft. Windows Power Management: Instant PC Availability and Energy Savings.

[113] T. M. Mintz and J. P. Davis. Low-power tradeoffs for mobile computing applications: embedded processors versus custom computing kernels. In *Proceedings of the 45th annual southeast regional conference*, ACM-SE 45, pages 144–149, New York, NY, USA, 2007. ACM.

[114] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: understanding the runtime effects of frequency scaling. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 35–44, New York, NY, USA, 2002. ACM.

[115] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *In Workshop on Compilers and Operating Systems for Low Power*, 2000.

[116] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[117] D. K. Newsom, S. F. Azari, A. Anbar, and T. El-Ghazawi. Locality-aware power optimization and measurement methodology for pgas workloads on smp clusters. In *Green Computing Conference (IGCC), 2013 International*, pages 1–10. IEEE, 2013.

[118] J. Nieplocha and B. Carpenter. Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In J. Rolim, F. Mueller, A. Zomaya, F. Ercal, S. Olariu, B. Ravindran, J. Gustafsson, H. Takada, R. Olsson, L. Kale, P. Beckman, M. Haines, H. ElGindy, D. Caromel, S. Chaumette, G. Fox, Y. Pan, K. Li, T. Yang, G. Chiola, G. Conte, L. Mancini, D. Mry, B. Sanders, D. Bhatt, and V. Prasanna, editors, *Parallel and Distributed Processing*, volume 1586 of *Lecture Notes in Computer Science*, pages 533–546. Springer Berlin Heidelberg, 1999.

[119] J. Nieplocha and B. Carpenter. Armci: A portable remote memory copy libray for ditributed array libraries and compiler run-time systems. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 533–546, London, UK, UK, 1999. Springer-Verlag.

[120] NSF, SRC. Energy-Efficient Computing: from Devices to Architectures (E2CDA). March 2016.

[121] NVIDIA. Nvml api reference manual, ver.5.319.43. August 2013.

[122] OpenFabrics Alliance. The Case for Open Source - RDMA. August 2011.

[123] OpenUCX. Unified Communication X (UCX) API Documentation.

[124] J. Pallister, S. J. Hollis, and J. Bennett. Identifying compiler options to minimise energy consumption for embedded platforms. *CoRR*, abs/1303.6485, 2013.

[125] J. Pan. Rapl (running average power limit) driver.

[126] A. Parikh, S. Kim, M. Kandemir, N. Vijaykrishnan, and M. Irwin. Instruction scheduling for low power. *Journal of VLSI signal processing systems for signal, image and video technology*, 37(1):129–149, 2004.

[127] S. F. Rahman, J. Guo, and Q. Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 107–116, New York, NY, USA, 2011. ACM.

[128] T. Rauber and G. Rünger. Modeling the energy consumption for concurrent executions of parallel tasks. In *Proceedings of the 14th Communications and Networking Symposium*, CNS '11, pages 11–18, San Diego, CA, USA, 2011. Society for Computer Simulation International.

[129] Rice University. Habanero UPC. https://github.com/habanero-rice/habanero-upc.

[130] Rice University. Habanero-C Overview. 2013.

[131] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *Micro, IEEE*, 32(2):20–27, 2012.

[132] B. Rountree, D. H. Ahn, B. R. de Supinski, D. K. Lowenthal, and M. Schulz. Beyond dvfs: A first look at performance under a hardware-enforced power bound. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 947–953, Washington, DC, USA, 2012. IEEE Computer Society.

[133] W. N. Scherer, III, L. Adhianto, G. Jin, J. Mellor-Crummey, and C. Yang. Hiding latency in coarray fortran 2.0. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 14:1–14:9, New York, NY, USA, 2010. ACM.

[134] R. Schöne, D. Hackenberg, and D. Molka. Memory performance at reduced CPU clock speeds: an analysis of current x86_64 processors. *Proceedings of the USENIX Workshop on Power-Aware Computing and Systems (HotPower)*, 2012.

[135] R. Schne and D. Molka. Integrating performance analysis and energy efficiency optimizations in a unified environment. *Computer Science - Research and Development*, pages 1–9, 2013.

[136] R. Schne, R. Tschter, T. Ilsche, and D. Hackenberg. The vampirtrace plugin counter interface: Introduction and examples. In M. Guarracino, F. Vivien, J. Trff, M. Cannatoro, M. Danelutto, A. Hast, F. Perla, A. Knpfer, B. Martino, and M. Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*,

volume 6586 of *Lecture Notes in Computer Science*, pages 501–511. Springer Berlin Heidelberg, 2011.

[137] J. S. Seng and D. M. Tullsen. The effect of compiler optimizations on pentium 4 power consumption. In *Proceedings of the Seventh Workshop on Interaction Between Compilers and Computer Architectures*, INTERACT '03, pages 51–, Washington, DC, USA, 2003. IEEE Computer Society.

[138] G. Shah and C. Bender. Performance and experience with lapi – a new high-performance communication library for the ibm rs/6000 sp. In *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, IPPS '98, pages 260–, Washington, DC, USA, 1998. IEEE Computer Society.

[139] J. Shalf, S. Dosanjh, and J. Morrison. Exascale Computing Technology Challenges. pages 1–25, 2011.

[140] P. Shamis, M. G. Venkata, J. A. Kuehn, S. W. Poole, and R. L. Graham. Universal common communication substrate (uccs) specification. version 0.1. Tech Report ORNL/TM-2012/339, Oak Ridge National Laboratory (ORNL), 2012.

[141] R. Sohan, A. Rice, A. W. Moore, and K. Mansley. Characterizing 10 gbps network interface energy consumption. In *LCN*, pages 268–271. IEEE, 2010.

[142] C.-L. Su, C.-Y. Tsui, and A. M. Despain. Saving power in the control path of embedded processors. *IEEE Des. Test*, 11(4):24–30, Oct. 1994.

[143] C.-Y. Su, D. Li, D. S. Nikolopoulos, K. W. Cameron, B. R. de Supinski, and E. A. Leon. Model-based, memory-centric performance and power optimization on numa multiprocessors. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 164–173, San Diego, CA, Nov. 2012.

[144] V. Taylor, X. Wu, C. W. Lee, K. Cameron, H.-C. Chang, D. Terpstra, and S. Moore. Combined Performance and Power Consumption Modeling and Optimization with MuMMI 1. pages 1–23.

[145] The White House, Office of the Press Secretary. Executive Order – Creating a National Strategic Computing Initiative. July 2015.

[146] A. Tiwari, M. Laurenzano, L. Carrington, and A. Snavely. Modeling power and energy usage of hpc kernels. In *Parallel and Distributed Processing Symposium*

Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International, pages 990–998, 2012.

[147] V. Tiwari, S. Malik, A. Wolfe, and M.-C. Lee. Instruction level power analysis and optimization of software. In *VLSI Design, 1996. Proceedings., Ninth International Conference on*, pages 326–328, 1996.

[148] V. Tiwari, S. Malik, A. Wolfe, and M. Tien-Chien Lee. Instruction level power analysis and optimization of software. In *VLSI Design, 1996. Proceedings., Ninth International Conference on*, pages 326–328, Jan 1996.

[149] TOP500.org. TOP500. The List.

[150] J. Torrellas, D. Quinlan, A. snavely, and W. Pinfold. Thrifty: An exascale architecture for energy-proportional computing.

[151] A. Venkatesh, K. Kandalla, and D. Panda. Evaluation of energy characteristics of mpi communication primitives with rapl. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 938–945, May 2013.

[152] A. Vishnu, S. Song, A. Marquez, K. Barker, D. Kerbyson, K. Cameron, and P. Balaji. Designing energy efficient communication runtime systems for data centric programming models. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 229–236, 2010.

[153] A. Vishnu, S. Song, A. Marquez, K. Barker, D. Kerbyson, K. Cameron, and P. Balaji. Designing energy efficient communication runtime systems: a view from pgas models. *The Journal of Supercomputing*, 63(3):691–709, 2013.

[154] J. J. Willcock, S. W. Ave, N. G. Edmonds, and A. Lumsdaine. AM ++ : A Generalized Active Message Framework. 2010.

[155] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanović. Direct addressed caches for reduced power consumption. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 124–133, Washington, DC, USA, 2001. IEEE Computer Society.

[156] J. Wu, P. Wyckoff, and D. K. Panda. PVFS over infiniband: Design and performance evaluation. In *IN THE 2003 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING (ICPP) 03*, pages 125–132, 2003.

[157] X. Wu, H.-C. Chang, S. Moore, V. Taylor, C.-Y. Su, D. Terpstra, C. Lively, K. Cameron, and C. W. Lee. Mummi: multiple metrics modeling infrastructure for exploring performance and power modeling. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, XSEDE '13, pages 36:1–36:8, New York, NY, USA, 2013. ACM.

[158] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. In *In ACM*, pages 10–11, 1998.

[159] J. Zambreno, M. T. Kandemir, and A. N. Choudhary. Enhancing compiler techniques for memory energy optimizations. In *Proceedings of the Second International Conference on Embedded Software*, EMSOFT '02, pages 364–381, London, UK, UK, 2002. Springer-Verlag.

[160] T. Zhang, W. Shi, and S. Pande. Static techniques to improve power efficiency of branch predictors. In *Proceedings of the 11th International Conference on High Performance Computing*, HiPC'04, pages 274–285, Berlin, Heidelberg, 2004. Springer-Verlag.

[161] W. Zhang, J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Reducing instruction cache energy consumption using a compiler-based strategy. *ACM Trans. Archit. Code Optim.*, 1(1):3–33, Mar. 2004.

[162] X. Zhao, D. Buntinas, J. Zounmevo, J. Dinan, D. Goodell, P. Balaji, R. Thakur, A. Afsahi, and W. Gropp. Toward asynchronous and MPI-interoperable active messages. *Proceedings - 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013*, pages 87–94, 2013.

[163] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: A PGAS Extension for C++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1105–1114, May 2014.