SEVENTH FRAMEWORK PROGRAMME
THEME ICT-2013.3.4
Advanced Computing, Embedded and Control Systems

EXCESS

Execution Models for Energy-Efficient Computing Systems
Project ID: 611183

D2.3
Power models, energy models and libraries for energy-efficient concurrent data structures and algorithms

Phuong Ha, Vi Tran, Ibrahim Umar, Aras Atalar, Anders Gidenstam, Paul Renaud-Goud, Philippas Tsigas, Ivan Walulya

SEVENTH FRAMEWORK PROGRAMME

Date of preparation (latest version): 29.02.2016

DOCUMENT INFORMATION

Deliverable Number	D2.3
Deliverable Name	Power models, energy models and libraries for energy-efficient concurrent data structures and algorithms
Authors	Phuong Ha Vi Tran Ibrahim Umar Aras Atalar Anders Gidenstam Paul Renaud-Goud Philippas Tsigas Ivan Walulya
Responsible Author	Phuong Ha e-mail: `phuong.hoai.ha@uit.no` Phone: +47 776 44032
Keywords	High Performance Computing; Energy Efficiency
WP/Task	WP2/Task 2.1, 2.2, 2.3
Nature	R
Dissemination Level	PU
Planned Date	29.02.2016
Final Version Date	28.02.2016
Reviewed by	
MGT Board Approval	

DOCUMENT HISTORY

Partner	**Date**	**Comment**	**Version**
UiT (P. Ha, V. Tran)	14.12.2015	Deliverable skeleton	0.1
Chalmers (P. Renaud-Goud)	22.1.2016	Input - energy model and energy evaluation	0.2
UiT (P. Ha, V. Tran, I . Umar)	29.1.2016	Input - energy/power models and libraries	0.3
UiT (I . Umar)	19.2.2016	Revised chapter 6	0.4
UiT (V. Tran)	19.2.2016	Revised chapter 1, 2, 3	0.5
Chalmers (P. Renaud-Goud)	22.2.2016	Revised chapter 1, 4, 5	0.6
UiT (P. Ha, V. Tran)	26.2.2016	Minor fixes	0.7

Abstract

This deliverable reports the results of the power models, energy models and libraries for energy-efficient concurrent data structures and algorithms as available by project month 30 of Work Package 2 (WP2). It reports i) the latest results of Task 2.2-2.4 on providing programming abstractions and libraries for developing energy-efficient data structures and algorithms and ii) the improved results of Task 2.1 on investigating and modeling the trade-off between energy and performance of concurrent data structures and algorithms. The work has been conducted on two main EXCESS platforms: Intel platforms with recent Intel multicore CPUs and Movidius Myriad platforms.

Regarding modeling the trade-off between energy-efficiency and performance of concurrent data structures and algorithms, we report in this deliverable four energy/power model evaluation studies, including: an improved power model for EXCESS platforms compared to Deliverable D2.2 (i.e., Movidius Myriad1), a new energy complexity model (namely, EPEM - Energy-aware Parallel External Memory) for multi-threaded algorithms, the modeling of the performance and the energy consumption of data structures on a CPU platform, as well as an investigation on the optimization of streaming applications on Myriad2, from three points of view, which are performance, energy consumption, and space.

Regarding developing novel programming abstractions and libraries, we have collected additional performance profiles using CPU counters and a cycle accurate simulator (i.e., GEM5), which are useful to gain insight into the relation between reduced data movements and energy efficiency. Moreover, we have implemented DeltaTree, the locality-aware data structures, and a fast concurrent B-Tree on Myriad2 platform, and have shown that a specialized ultra low-power embedded platform such as Movidius Myriad2 can also benefit from the locality-aware data structures.

Executive Summary

Work package 2 (WP2) aims to develop libraries for energy-efficient inter-process communication and data sharing on the EXCESS platforms. In order to set the stage for these tasks, WP2 needs to investigate and model the trade-offs between energy consumption and performance of data structures and algorithms for inter-process communication. WP2 also provides concurrent data structures and algorithms that support energy-efficient massive parallelism while minimizing inter-component communication. The developed data structures and algorithms are locality- and heterogeneity-aware.

This Deliverable D2.3 includes the results of four on-going tasks (i.e., Task 2.1-Task 2.4). In summary, Task 2.1 (PM1 - PM36) aims for modeling the trade-off between energy and performance in concurrent data structures and algorithms. Task 2.2 (PM7 - PM36) objective is to identify essential concurrent data structures and algorithms for inter-process communication. Task 2.3 (PM7 - PM36) develops locality- and heterogeneity-aware concurrent data structures and Task 2.4 (PM7 - PM36) develops locality- and heterogeneity-aware memory access algorithms.

The latest results of Task 2.1 on investigating and modeling the trade-off between energy and performance [46] are presented in the Sections 2, 3, 4, 5 (power/energy model studies). The latest results of Tasks 2.2-2.4 on energy-efficient and concurrent programming abstractions and libraries [49] available by project month 30 are also summarized in this report in Section 6.

Power/Energy models

The studies on proposing new power and energy models help to investigate the trade-off between energy-efficiency and performance of concurrent data structures and algorithms. The new power and energy models provide the understanding of energy consumption of concurrent data structures and algorithms.

- We have improved the power models for EXCESS platforms (i.e., Movidius Myriad1) from the power models presented in Deliverable D2.2. The latest RTHpower models have modeled the power consumption of Myriad1 both when computation and data transfer are performed in parallel and separately.

- We have proposed EPEM, a new energy complexity model for multithreaded algorithms. This new general and validated energy complexity model for parallel (multithreaded) algorithms abstracts away possible multicore platforms by their static and dynamic energy of a computational operation and data access, and derives the energy complexity of a given algorithm from its *work*, *span* and *I/O* complexity. The new model is validated by different sparse matrix vector multiplication (SpMV) algorithms (e.g., compressed sparse column (CSC) and compressed sparse block (CSB)) running on high performance computing (HPC) platforms (e.g., Intel Xeon and Xeon Phi) and using nine sparse matrix types from Florida Matrix Collection. The new energy complexity model is able to characterize and compare the energy consumption of SpMV kernels according to three aspects: different algorithms, different input matrix types and different platforms.

- We have continued the modelling of the performance and the energy consumption of data structures on a CPU platform. In Deliverable D2.2, we have successfully modelled lock-free queues: we have been able to predict the throughput, the power dissipated by the chip and the energy per operation of six different implementations of lock-free queues by measuring only a very few points of the studied domain. In the present deliverable, we target the same metrics, which are performance- and/or energy-related, and take some more steps: we present the prediction of those metrics on a larger set of data structures, namely stack, shared counter, queue and priority queue on a single-socket processor. To obtain these estimates of the metrics, we need even less measurement points than previously.

- We have investigated on the optimization of streaming applications on Myriad2, from three points of view, which are performance, energy consumption, and space. To do this, we have focused our evaluation on an operator that is widely used in data streaming applications: the multiway aggregator. Several implementations of this aggregator has been tested, employing several queue implementations.

Libraries of concurrent data structures and algorithms

We describe a set of implemented concurrent search trees as well as their energy and performance analyses.

- In the previous deliverable, we have shown that locality-aware concurrent search trees were able to consume less energy while able to maintain better throughput than the locality-oblivious concurrent search trees. Recently, we have collected additional performance profiles using CPU counters and a cycle accurate simulator (i.e., GEM5), which are useful to gain insight on the relation between reduced data movements and energy efficiency.

- We have implemented DeltaTree and a fast concurrent B-Tree on Myriad2 platform, and have shown that a specialized ultra low-power embedded platform such as Movidius Myriad2 can also benefit from the fine-grained locality data structures. The experimental results show that GreenBST has up to 100% better energy efficiency and more operations/second on the x86, ARM, and Xeon Phi platforms.

This report is organized as follows. Section 1 provides the background and motivations of the work presented in this deliverable. Sections 2, 3, 4, 5 discuss power and energy models studies. Section 6 describes the second prototype with latest updates of EXCESS libraries including numerous concurrent search tree implementations as well as their performance and energy analyses. Section 7 concludes the report by summarizing the latest results and future works.

Contents

1 Introduction

1.1 Purpose

The goal of Work package 2 (WP2) is to develop programming abstraction and libraries for inter-process communication and data sharing on EXCESS platforms, along with investigating and modeling the trade-offs between energy consumption and performance of data structures and algorithms for inter-process communication. WP2 also concerns supporting energy-efficient massive parallelism through scalable concurrent data structures and algorithms that strive for the energy limit, and minimizing inter-component communication through locality- and heterogeneity-aware data structures and algorithms.

This report summarizes i) the latest results of Task 2.1 on investigating and modeling the trade-off between energy and performance of concurrent data structures and algorithms ii) the improved results of Task 2.2 on providing essential concurrent data structures and algorithms for inter-process communication and well as results of Task 2.3 on developing novel concurrent data structures that are locality- and heterogeneity-aware.

1.2 Power and Energy Models

This section explains the motivations of four energy/power model studies, including: an improved power model for EXCESS platforms compared to Deliverable D2.2 (i.e., Movidius Myriad1), a new energy complexity model (namely, EPEM - Energy-aware Parallel External Memory) for multi-threaded algorithms, the modeling of the performance and the energy consumption of data structures on a CPU platform, as well as an investigation on the optimization of streaming applications on Myriad2.

1.2.1 Power Models for Ultra-low Power Embedded Systems

Devising accurate power models is crucial to gain insights into how a computer system consumes power and energy. Significant efforts have been devoted to devising power and energy models, resulting in several seminal papers in the literature such as [9, 63, 22, 21, 67, 68, 60, 81, 94]. The models are either platform specific [63] or application specific [9]. Jacobson et al. [63] provided an insightful analysis in power modeling methodologies via a range of abstraction levels. They also proposed accurate power modeling methodologies for POWER-family processors. Alonso et al. [9] proposed energy models for three key dense-matrix factorizations. Roofline model of energy [22, 21] considers both algorithmic and platform properties. However, the Roofline model does not consider the number of cores running applications as a model parameter (i.e., coarse-grained models). Theoretical models by Korthikanti et al. [68, 67] are based on strong theoretical assumptions and are not yet validated on real platforms. Imes et al. [60] provided a portable approach to make real-time decision and run the chosen configuration to minimize energy consumption. However, the approach requires systems supporting hardware resources (e.g., model-specific register) to expose energy data to the software during run-time. Mishra et al. [81] used a probabilistic approach to find the most energy-efficient configuration by combining online and offline machine-learning approaches. However, this approach collects a significant amount of data to feed to its probabilistic network. RTHpower models proposed in this study are lightweight and applicable to any ultra-low power embedded systems as long as there is a means to measure the energy consumption of micro-benchmarks on the targeted platform.

Recently, ultra-low power (ULP) embedded systems have become popular in the scientific community and industry, especially in media and wearable computing. ULP systems can achieve low energy per instruction down to a few pJ [7]. Alioto [7] mentioned that techniques such as pipe-lining, hardware replication, ultra-low-voltage memory design and leakage-reducing make a system ultra-low power. In order to model ULP systems where energy per instruction can be as low as few pJ, more accurate fine-grained approaches are needed. However, to the best of our knowledge, there are no application-general, fine-grained and validated models yet that provide insights into how an application running on an ULP embedded system consumes energy and, particularly, whether the race-to-halt (RTH) strategy (i.e, system is run as fast as possible, and then switched to idle state to save energy) that is widely used in high-performance computing (HPC) systems is still applicable to ULP embedded systems.

1.2.2 Energy Complexity Model for Multithreaded Algorithms

Like time complexity models that have significantly contributed to the analysis and development of fast algorithms, energy complexity models for parallel algorithms are desired as crucial means to develop energy efficient algorithms for ubiquitous multicore platforms. Ideal energy complexity models should be validated on real multicore platforms and applicable to a wide range of parallel algorithms. However, existing energy complexity models [9, 67, 68] for parallel algorithms are either theoretical without model validation or algorithm-specific without being applicable to a wide range of algorithms. This work presents a new general validated energy complexity model for parallel (multithreaded) algorithms.

1.2.3 Energy Model on CPU for Lock-free Data-structures with Low-level of Disjoint-access Parallelism

We consider the modeling and the analysis of the performance of lock-free concurrent data structures. Lock-free designs employ an optimistic conflict control mechanism, allowing several threads to access the shared data object at the same time. They guarantee that at least one concurrent operation finishes in a finite number of its own steps regardless of the state of the operations. Our analysis considers such lock-free data structures that can be represented as linear combinations of fixed size retry loops.

Our main contribution is a new way of modeling and analyzing a general class of lock-free algorithms (including stack, shared counter, queue, priority queue), achieving predictions of throughput that are close to what we observe in practice. We emphasize two kinds of conflicts that shape the performance: (i) hardware conflicts, due to concurrent calls to atomic primitives; (ii) logical conflicts, caused by simultaneous operations on the shared data structure. We show how to deal with these hardware and logical conflicts separately, and how to combine them, so as to calculate the throughput of lock-free algorithms.

We propose also a common framework that, in addition to providing a better understanding of the performance impacting factors, enables a fair comparison between lock-free implementations by covering the whole contention domain. This comparison translates into the ability to choose the best implementation at hand, with respect to the actual application that uses the data structure. This part of our analysis comes with a method for calculating a good back-off strategy to finely tune the performance of a lock-free algorithm. Our experimental results, based on a set of widely used concurrent data structures and on abstract lock-free designs, show that our analysis follows closely the actual code behavior.

1.2.4 Energy Evaluation on Myriad2 for Multiway Aggregation on Streaming Applications

The transition from uniprocessor to multiprocessor designs in embedded systems is not as straightforward as in general purpose machines due to the fact that limitations in terms of space availability and energy consumption are introduced depending on the application's purpose and target environment along with the hardware used.

Developing this kind of systems in a way that it uses memory and energy in an optimal way, and such that performance is not affected at a big percentage, is of utmost importance and data structures used within an application play a significant role towards this goal.

The work presented here aims to exhibit data structures that are suited for embedded systems, and to investigate trade-offs between different implementations in terms of energy consumption, memory utilization and performance. Through this investigation the focus will be on data streaming applications implementing multiway aggregation of the received data.

Although efficient data structures for a concurrent environment have been studied extensively, the issue of appropriate data structures for data streaming applications has been neglected. Concurrent data structures play a major role between aggregation stages, because they are in charge of the regulation of the parallelism and the load balancing in this streaming applications. For this reason we have developed such a streaming application, which is based on the research conducted on concurrent data structures. An efficient solution providing lower latency, bigger throughput and energy efficiency at the data aggregation function of the application is then achieved.

1.3 Energy-efficient and Concurrent Data Structures and Algorithms

Recent research has suggested that improving fine-grained data-locality is one of the main approaches to improving energy efficiency and performance. However, no previous research has investigated the effect of the approach on these metrices in the case of concurrent data structures.

Our study investigates how fine-grained data locality influences energy efficiency and performance in concurrent search trees, a crucial data structure that is widely used in several important systems. We conduct a set of experiments on three lock-based concurrent search trees: GreenBST, a portable fine-grained locality-aware concurrent search tree; CBTree, a coarse-grained locality-aware concurrent B+tree; LFBST, a locality-oblivious non-blocking binary search tree; and citrus, a locality-oblivious read-copy-update (RCU) concurrent search tree. We run the experiments on a commodity x86 platform, an embedded ARM platform, and an accelerator board based on the Intel Xeon Phi. The experimental results show that GreenBST has up to 100% better energy efficiency and more operations/second on the x86, ARM, and Xeon Phi platforms. The results also confirm that portable fine-grained locality can improve energy efficiency and performance in concurrent search trees.

1.4 Contributions

The main achievements in this report are summarized as follows.

1.4.1 Power/Energy models

- We have improved the power model for one EXCESS platform (Movidius Myriad1) initially described in Deliverable D2.2. The power models characterize applications by their operational intensity that can be extracted from any application. The models are validated with three application kernels, namely dense matrix multiplication, sparse matrix vector multiplication and breadth first search. Based on the models, we propose a framework to predict whether it is energy-efficient to apply race-to-halt (RTH) strategy (i.e. running an application with a maximum number of cores). For the application kernels, the proposed framework is able to predict when to use RTH and when not to use RTH precisely. The experimental results show that we can save up to 60% energy for dense matrix multiplication, 61% energy for sparse matrix vector multiplication by using RTH and 5% energy for breadth first search by *not* using RTH.

- We have proposed a new energy complexity model for multithreaded algorithms. This new general and validated energy complexity model for parallel (multithreaded) algorithms abstracts away possible multicore platforms by their static and dynamic energy of a computational operation and data access, and derives the energy complexity of a given algorithm from its *work*, *span* and *I/O* complexity. The new model is validated by different sparse matrix vector multiplication (SpMV) algorithms (e.g., compressed sparse column (CSC) and compressed sparse block (CSB)) running on high performance computing (HPC) platforms (e.g., Intel Xeon and Xeon Phi) and using nine sparse matrix types from Florida Matrix Collection [28]. The new energy complexity model is able to characterize and compare the energy consumption of SpMV kernels according to three aspects: different algorithms, different input matrix types and different platforms.

- We have continued the modelling of the performance and the energy consumption of data structures on a CPU platform. In Deliverable D2.2, we have successfully modelled lock-free queues: we have been able to predict the throughput, the power dissipated by the chip and the energy per operation of six different implementations of lock-free queues by measuring only a very few points of the studied domain. In the present deliverable, we targeted the same metrics, which are performance- and/or energy-related, and take some more steps: we present the prediction of those metrics on a larger set of data structures, namely stack, shared counter, queue and priority queue on a single-socket processor. To obtain these estimates of the metrics, we need even less measurements points than previously.

- We have investigated on the optimization of streaming applications on Myriad2, from three points of view, which are performance, energy consumption, and space. To do this, we have focused our evaluation on an operator that is widely used in data streaming applications: the multiway aggregator. Several implementations of this aggregator has been tested, employing several queue implementations.

1.4.2 Libraries of concurrent data structures and algorithms

- We have added two state-of-the art concurrent search trees into the concurrent search tree library. In the previous deliverable, we have shown that locality-aware concurrent search trees were able to consume less energy lead while able to maintain better throughput than the locality-oblivious concurrent search trees. Recently, we have collected additional performance profiles using CPU counters and a cycle accurate simulator (i.e., GEM5), which are useful to gain insight on the relation between reduced data movements and energy efficiency.

- We have implemented DeltaTree and a fast concurrent B-Tree on Myriad2 platform, and have shown that a specialized ultra low-power embedded platform such as Movidius Myriad2 can also benefit from the fine-grained locality data structures.

2 Power Models for Ultra-low Power Embedded Systems

This section presents the improvements on power models for Myriad1 platform from the model proposed in EXCESS Deliverable D2.2 [49]. The study is summarized to three main works as follows.

- We propose new application-general fine-grained power models (namely, RTHpower) that provide insights into how a given application consumes power when running on an ultra-low power embedded system [97]. The RTHpower models consider three parameter groups: platform properties, application properties (e.g. operational intensity and scalability) and execution settings (e.g., the number of cores executing a given application). The models consider both platform and application properties, which give more insights into how to design applications to achieve better energy efficiency. (cf. Section 2.1)

- We train the new RTHpower models on an ultra-low power embedded system, namely Movidius Myriad using different sets of micro-benchmarks and validate the models using two computation kernels from Berkeley dwarfs [13] and one data-intensive kernel from Graph500 benchmarks [95]. The three chosen application kernels are dense matrix multiplication (Matmul), sparse matrix vector multiplication (SpMV) and breadth first search (BFS). The model fitting has percentage error at most 7% for micro-benchmarks and 10% for application benchmarks (cf. Section 2.2).

- We investigate the RTH strategy on an ultra-low power embedded platform using the new RTHpower models. We propose a framework that is able to predict precisely when to and when not to apply the RTH strategy in order to minimize energy consumption. We validate the framework using micro-benchmarks and application kernels. From our experiments, we show real-world scenarios when to use RTH and when not to use RTH. We can save up to 61% energy for dense matrix multiplication, 59% energy for SpMV by using RTH and up to 5% energy for BFS by not using RTH. (cf. Section 2.3)

2.1 RTHpower - Analytical Power Models

For the flow of reading, we first summarize a power model for operation units described in Deliverable D.2.2 and then develop it to the improved RTHpower models considering application properties in this Deliverable.

2.1.1 A Power Model for Operation Units

The experimental results of the micro-benchmarks suite for operation units show that the power consumption of Movidius Myriad1 platform is ruled by Equation 1. In the equation, the static power P^{sta} is the required power when the Myriad chip is on, including memory

Table 1: $P^{dyn}(op)$ of SHAVE Operation Units

Operation	Description	P^{dyn} (mW)
SAUXOR	Perform bitwise exclusive-OR on scalar	14.68
SAUMUL	Perform scalar multiplication	17.69
VAUXOR	Perform bitwise exclusive-OR on vector	34.34
VAUMUL	Perform vector multiplication	51.98
IAUXOR	Perform bitwise exclusive-OR on integer	15.91
IAUMUL	Perform integer multiplication	18.48
CMUCPSS	Copy scalar to scalar	12.62
CMUCPIVR	Copy integer to vector	18.84
LSULOAD	Load from a memory address to a register	29.87
LSUSTORE	Store from a register to a memory address	37.49

storage power; the active power P^{act} is the power consumed when a SHAVE core is on and actively performing computational work; the dynamic power $P^{dyn}(op)$ is the power consumed by each operation unit such as arithmetic units (e.g., IAU, VAU, SAU, CMU) or load/store units (e.g., LSU0, LSU1) in one SHAVE. The experimental results show that different operation units have different $P^{dyn}(op)$ values as listed in Table 1. The total dynamic power of a SHAVE core is the sum of all dynamic power from involved units. If benchmarks or programs are executed with n SHAVE cores, the active and dynamic power needs to be multiplied with the number of used SHAVE cores. By using regression fitting techniques, the average value of P^{sta} and P^{act} from all micro-benchmarks are computed in Equation 2 and Equation 3. Table 2 provides the description of parameters in the proposed models.

$$P^{units} = P^{sta} + n \times \left(P^{act} + \sum_i P_i^{dyn}(op) \right) \tag{1}$$

$$P^{sta} = 61.81 \text{ mW} \tag{2}$$

$$P^{act} = 29.33 \text{ mW} \tag{3}$$

2.1.2 RTHpower Models for Applications

Since typical applications require both computation and data movement, we use the concept of operational intensity proposed by Williams et al.[105] to characterize applications. An application can be characterized by the amount of computational work W and data transfer Q. W is the number of operations performed by an application. Q is the number of transferred bytes required during the program execution. Both W and Q define the operational

Table 2: Model Parameter List

Parameter	Description
P^{sta}	Static power of a whole chip
P^{act}	Active power of a core
$P^{dyn}(op)$	Dynamic power of an operation unit
P^{LSU}	Dynamic power of Load Store Unit
P^{ctn}	Contention power of a core waiting for data
m	Average number of active cores accessing data
n	Number of assigned cores to the program
I	Operational intensity of an application
α	Time ratio of data transfer to computation
β	Tuning parameter of an application

intensity I of applications as in Equation 4.

$$I = \frac{W}{Q} \tag{4}$$

As the time required to perform one operation is different from the time required to transfer one byte of data, we introduce a parameter to the models: time ratio α of transferring one byte of data to performing one arithmetic operation. Ratio α is the property of an application on a specific platform and its value depends on the application. Since the time to access data and time to perform computation work can be overlapped, during a program execution, the SHAVE core can be in one of the three states: performing computation, performing data transfer or performing both computation and data transfer in parallel. An application either has data transfer time longer than computation time or vice versa. Therefore, there are two models for the two cases for higher accuracy (as compared to Deliverable D2.2, only one model represents both cases).

- If data transfer time is longer than computation time, the model follows Equation 5. The execution can be modeled as two (composed) periods: one is when computation and data transfer are performed in parallel and the other is when only data transfer is performed. Fraction $\frac{W}{\alpha \times Q}$ represents the overlapped time of computation and data transfer. Fraction $\frac{\alpha \times Q - W}{\alpha \times Q}$ represents the remaining time for data transfer.

$$P = P^{comp||data} \times \frac{W}{\alpha \times Q} + P^{data} \times \frac{\alpha \times Q - W}{\alpha \times Q} \tag{5}$$

- If computation time is longer than data transfer time, estimated power follows Equation 6. The execution can be modeled as two periods: one is when computation and data transfer are performed in parallel and the other is when only computation is performed.

$$P = P^{comp||data} \times \frac{\alpha \times Q}{W} + P^{comp} \times \frac{W - \alpha \times Q}{W} \tag{6}$$

After converting W and Q to I by using Equation 4, the final models are simplified as Equation 7 and Equation 8,

$$P = P^{comp||data} \times \frac{I}{\alpha} + P^{data} \times \frac{\alpha - I}{\alpha} \tag{7}$$

$$P = P^{comp||data} \times \frac{\alpha}{I} + P^{comp} \times \frac{I - \alpha}{I} \tag{8}$$

where P^{data}, P^{comp} and $P^{comp||data}$ are explained below:

2.1.2.1 Data transfer power

P^{data} is the power consumed by the whole chip when only data transfer is performed. P^{data} is computed by Equation 9. In Equation 9, P^{sta} is the static power; P^{act} is the active power; n is the number of active cores assigned to run the application; m is the average number of SHAVE cores accessing data in parallel during the application execution; contention power P^{ctn} is the power overhead occurring when a SHAVE core waits for accessing data because of the limited memory ports (or bandwidth) or cache size in the platform architecture. Therefore, $n - m$ is the average number of SHAVE cores waiting for memory access during the application execution.

$$\begin{aligned} P^{data} &= P^{sta} + min(m, n) \times (P^{act} + P^{LSU}) \\ &\quad + max(n - m, 0) \times P^{ctn} \end{aligned} \tag{9}$$

2.1.2.2 Computation power

P^{comp} is the power consumed by the whole chip when only computation is performed. P^{comp} is computed by Equation 10. Each core runs its arithmetic units (e.g. IAU, SAU, VAU) to perform computation work. There is no contention power due to no memory access. Therefore, all assigned cores are active and contribute to the power consumption.

$$P^{comp} = P^{sta} + n \times (P^{act} + \sum_i P_i^{dyn}(op)) \tag{10}$$

2.1.2.3 Computation and data transfer power

$P^{comp||data}$ is the power consumed by the whole chip when computation and data transfer are performed in parallel. $P^{comp||data}$ is computed by Equation 11. In this case, there is contention power due to data waiting. $P^{comp||data}$ is different from P^{data} in the aspect that the active cores also run arithmetic units that contribute to total power as $\sum_i P_i^{dyn}(op)$.

$$\begin{aligned} P^{comp||data} &= P^{sta} \\ &+ min(m,n) \times (P^{act} + P^{LSU} + \sum_i P_i^{dyn}(op)) \\ &+ max(n-m,0) \times P^{ctn} \end{aligned} \tag{11}$$

2.2 Model Training and Validation

This section presents the experimental results including two sets of micro-benchmarks and three application kernels (i.e., *matmul*, SpMV and BFS) that are used for training and validating the models .

2.2.1 Model Training with Micro-benchmarks

Analyses of experimental results are performed based on two sets of micro-benchmarks: 22 micro-benchmarks for operation units called *unit-suite* and 9 micro-benchmarks for different operational intensities called *intensity-suite*. Each micro-benchmark is executed with different numbers of SHAVE cores to measure its power consumption.

2.2.1.1 Micro-benchmarks for Operation Units

We assess the fitting of the power model for operation units (Equation 1) using data from *unit-suite*. The micro-benchmarks of *unit-suite* are listed in Table 3. We calculate the percentage errors of the model fitting and plot them in Figure 1. Percentage error is calculated as $PE = \frac{measurement-estimation}{measurement}$. The *absolute* percentage error is the absolute value of the percentage error. For this model, the absolute percentage errors are at most 6%. Figure 1 shows the percentage error of the worst cases in all three categories: one unit, two pipe-lined units and three pipe-lined units. These results prove that the model is applicable to micro-benchmarks using either a single (e.g., performing bit-wise exclusive-OR on scalar unit: SauXor) or pipe-lined arithmetic units in parallel (e.g., performing Xor on scalar and integer units, in parallel with copying from scalar to scalar unit: SauXorCmuCpssIauXor). The model also shows the compositionality of the power consumption not only for multiple SHAVE cores but also for multiple operation units within a SHAVE core.

2.2.1.2 Micro-benchmarks for Application Intensities

Since any application requires both computation and data movement, we design intensity-based micro-benchmarks which execute both arithmetic units (e.g., SAU) and two data transfer units (e.g., LSU0, LSU1) in a parallel manner. They are implemented with parallel instruction pipeline supported by the platform. In order to validate the RTHpower models,

Table 3: Micro-benchmarks for Operation Units

Description	Micro-benchmark Name
10 micro-benchmarks using one unit (cf. Table 2)	SAUXOR, SAUMUL, IAUXOR, IAUMUL, VAUXOR, VAUMUL, CMUCPSS, CMUCPIVR, LSULOAD, LSUSTORE
11 micro-benchmarks using two units	SAUXOR-CMUCPSS, SAUXOR-CMUCPIVR, SAUXOR-IAUMUL, SAUXOR-IAUXOR, SAUXOR-VAUMUL, SAUXOR-VAUXOR, SAUMUL-IAUXOR, IAUXOR-VAUXOR, IAUXOR-VAUMUL, IAUXOR-CMUCPSS, LSULOAD-STORE
1 micro-benchmark using three units	SAUXOR-IAUXOR-CMUCPSS

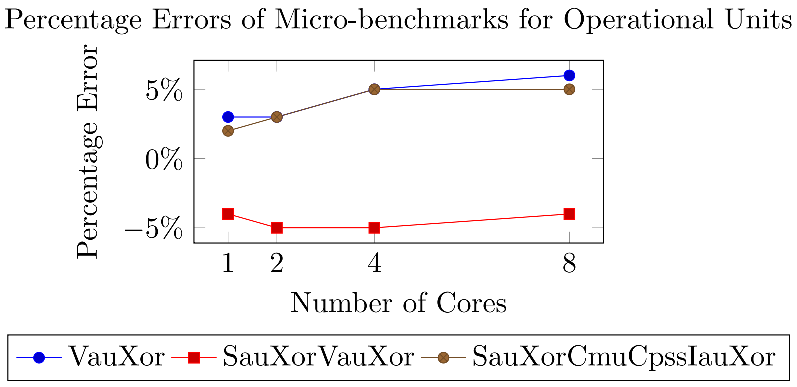


Figure 1: The upper-/ lower-bounds on percentage errors of the model fitting for *unit-suite* shown by the worst cases of three categories: one unit (e.g., SauXor), two units (e.g., IauXorVauXor) and three units (e.g., SauXorCmuIauXor). The absolute percentage errors of micro-benchmarks for operation units are at most 6%.

this *intensity-suite* indicates different values of operation intensities (from 0.25 to 64). Operational intensity I is retrieved from the assembly code by counting the number of arithmetic instructions and the number of load/store instructions.

In the models, there are platform-dependent parameters such as α, m and P^{ctn}. The parameter values for each application operational intensity are derived from experimental results by using Matlab function $lsqcurvefit$. For the application intensities from 0.25 to 1, α is found bigger than operational intensity I meaning that data transfer time is longer than computation time. The estimated power model follows Equation 7. For operational intensity from 2 to 64, α is less than I meaning that data transfer time is less than computation time. The estimated power follows Equation 8. We plot the percentage errors of the model fitting for intensity-based micro-benchmarks in Figure 2. In order to obtain a full range of estimated power with any values of intensities and numbers of cores, a fuzzy logic approach, namely Takagi Sugeno Kang (TSK) mechanism [96], is applied to the RTHpower models. Each intensity has a parameter set, including α, P^{ctn} and m. Based on the RTHpower models, each parameter set provides an individual function to estimate the power of an application based on its intensity value and a number of cores. The TSK mechanism considers the individual functions as membership functions and combines them into a general function that can be used for any input (i.e., intensity I and number of cores n). The membership

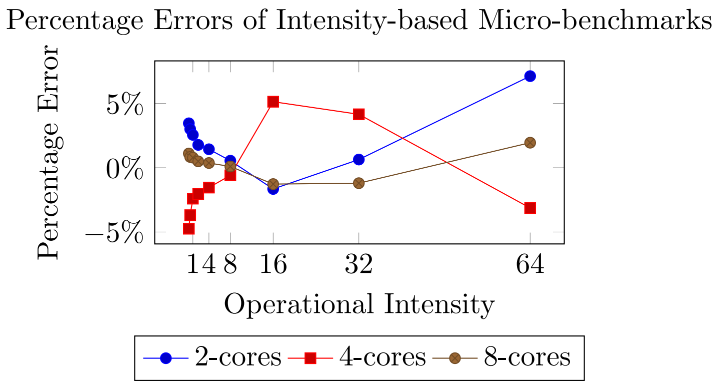


Figure 2: The absolute percentage errors of RTHpower model fitting for *intensity-suite* (operational intensity I from 0.25 to 64) are at most 7%.

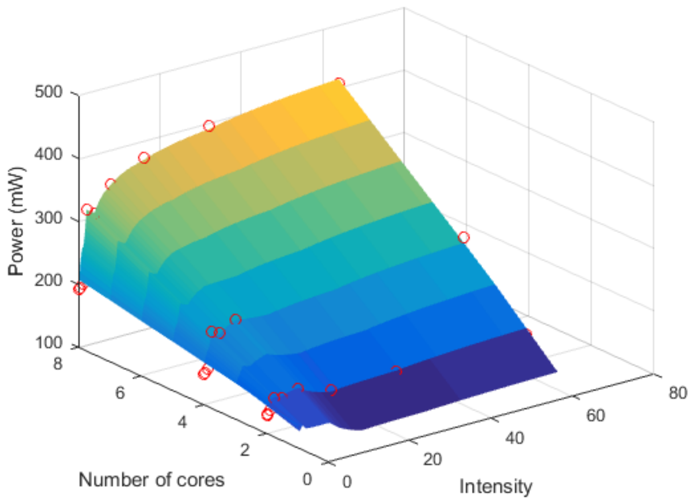


Figure 3: The estimated power range of varied intensities and numbers of cores from RTH-power models. The dots in the figure represent measurement data.

functions of the fuzzy sets are triangular[86]. After implemented the approach using Matlab Fuzzy Logic toolbox, the full range of estimated power is obtained and presented in Figure 3. It is observed that when the intensity value increases, the power-up (i.e., the power consumption ratio of the application executed with n cores to the application executed with 1 core) is also increased. The small dip in the diagram is due to the switch from Equation 7 to Equation 8 at the intensity $I = 2$.

2.2.2 Model Validation with Application Kernels

The following application kernels have been chosen to implement and validate the RTHpower models on Myriad1: *matmul* (a computation-intensive kernel), SpMV (a kernel with dynamic

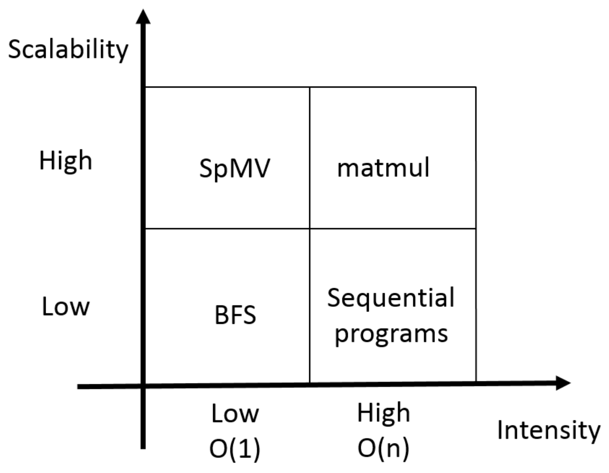


Figure 4: Application Categories

access patterns), and BFS (a data-intensive kernel of Graph500 benchmarks[95]). All three kernels belong to the list of Berkeley dwarfs [13] and are able to cover the two dimensions of operational intensity and performance speed-up as shown in Figure 4. *matmul* is proved to have high intensity and scalability [83]. SpMV has low operational intensity and high speed-up due to its parallel scalability [104]. BFS, on the other hand, has low operational intensity and saturated low scalability [23]. Since the available benchmark suites in literature are not executable on Myriad1 platform, the three mentioned kernels have been implemented by the authors using the Movidius Development Kit for Myriad1. As the RTHpower models will be used to predict whether the RTH strategy is an energy efficient approach for an given application, we focus mainly on two settings: the 8-core setting representing the RTH strategy (i.e., using all available cores of Myriad1) and the 1-core setting representing the other extreme (i.e., using a minimum number of cores).

2.2.2.1 Dense Matrix Multiplication

Matmul has been implemented on Myriad1 by using both C and assembly languages. The *matmul* algorithm computes matrix C based on two input matrices A and B $C = A \times B$. All three matrices in this benchmark are stored in DDR RAM. Matrix elements are stored with float type equivalent to four bytes. The number of operations and accessed data are calculated based on matrix size n as: $W = 2 \times N^3$ and $Q = 16 \times N^2$ [83]. Intensity of *matmul* is also varied with matrix size as: $I = \frac{W}{Q} = \frac{N}{8}$. The experiments are conducted until matrix size 1024x1024, the largest size that Myriad1 RAM memory can accommodate. Figure 5 shows that the percentage error of *matmul* estimated power compared to measured power is -25% on average for 1 core and 12% on average for 8 cores.

We observe that operational intensity is not enough to capture other factors such as the communication pattern and potential performance/power overheads due to the implementation. E.g., although a sequential version and a parallel version of a *matmul* algorithm have

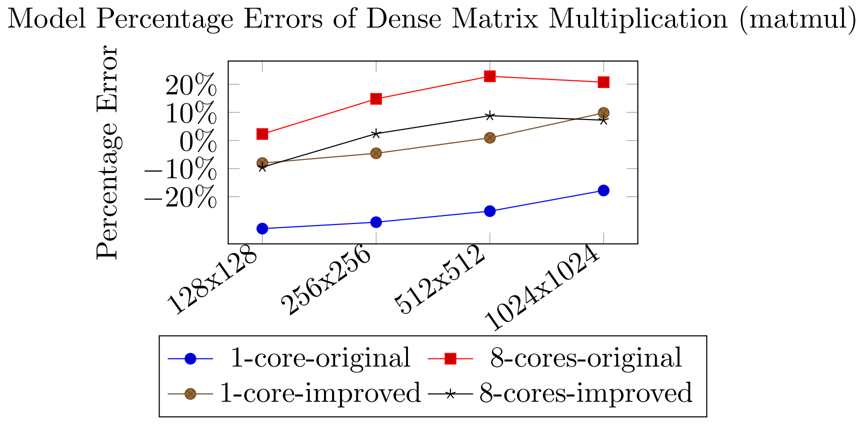


Figure 5: Absolute percentage errors of estimated power from measured power of *matmul*. After incorporating the tuning parameter, the absolute percentage errors of *matmul* are at most 10%.

the same intensity, it is obvious that they have different communication pattern (intuitively, the sequential version doesn't have communication between cores). Since different parallel versions for different number of cores have different communication patterns (e.g., sequential version vs. 8-core version), ignoring the mentioned factors contributes to the percentage errors. Therefore, we improve the models by introducing a tunning parameter β to the models in Equation 13 and Equation 14, where β is computed in Equation 12. Note that the tuning parameter β for each sequential/parallel version (e.g., 1-core version or 8-core version) is fixed across problem sizes and therefore it can be obtained during kernel installation and then saved as meta-data for each version in practice. E.g., with 1 core, $\beta = \frac{1}{1-25\%}$. With 8 cores, $\beta = \frac{1}{1+12\%}$. After the improvement, the percentage errors are at most 10% as shown in Figure 5.

$$\beta = \frac{1}{1 + \overline{PE}}. \tag{12}$$

$$P_{improved} = (P^{comp||data} \times \frac{I}{\alpha} + P^{data} \times \frac{\alpha - I}{\alpha}) \times \beta \tag{13}$$

$$P_{improved} = (P^{comp||data} \times \frac{\alpha}{I} + P^{comp} \times \frac{I - \alpha}{I}) \times \beta \tag{14}$$

2.2.2.2 Sparse Matrix Vector Multiplication

SpMV implementation on Myriad1 is written in C language. All input matrix and vector of this benchmark reside in DDR RAM. This implementation uses the common data layout of SpMV which is compressed sparse row (csr) format [90]. There is no random generator supported in the RISC core so a 5 non-zero elements per row is fixed in all experiments. Each element of matrix and vector is stored with float type of four bytes. From our implementation

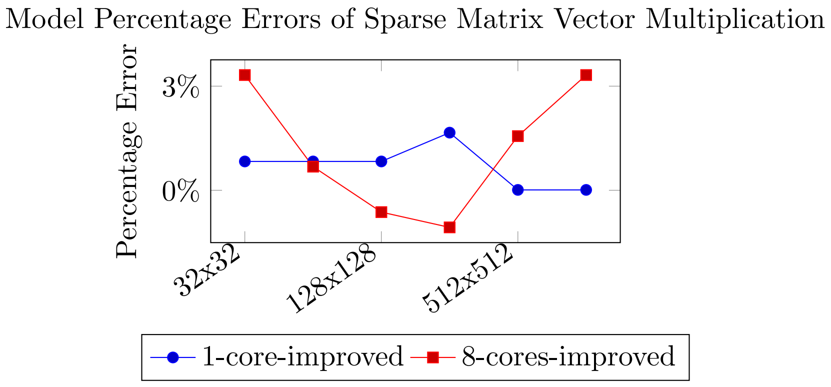


Figure 6: Absolute percentage errors of estimated power from measured power of SpMV. After incorporating the tuning parameter, the absolute percentage errors of SpMV are at most 4%.

analysis, the number of operations and accessed data are proportional to the size of a matrix dimension N as: $W = 5 \times 2 \times N$ and $Q = 5 \times 2 \times 4 \times N$. Operational intensity of SpMV therefore, does not depend on matrix size and is a fixed value: $I = \frac{W}{Q} = 0.25$.

Figure 6 shows the percentage error of SpMV estimated power using Equation 13 and 14 compared to measured power. The β values for 1-core and 8-core versions of SpMV are $\frac{1}{1+14\%}$ and $\frac{1}{1-9\%}$, respectively. The absolute percentage errors are at most 4% as shown in Figure 6. SpMV has lower modeling errors than *matmul* since SpMV has a fixed intensity value on different matrix sizes.

2.2.2.3 Breadth First Search

We also implemented BFS - a data-intensive Graph500 kernel, on Myriad1. BFS is the graph kernel to explore the vertices and edges of a directed graph from a starting vertex. We use the implementation of current Graph500 benchmark (omp-csr) and port it to Myriad1. The output BFS graphs after running BFS implementation on Myriad1 are verified by the verification step of original Graph500 code to ensure the output graphs are correct.

The size of a graph is defined by its scale and edgefactor. In our experiments, we mostly use the default edgefactor of 16 from the Graph500 so that each vertex of the graph has 16 edges in average. The graph scales are varied from 14 to 17 and the graphs has from 2^{14} to 2^{17} vertices. It is noted that graph scale 17 is the largest scale that the DDR RAM of Myriad1 can accommodate. From the implementation analysis, the operational intensity of BFS is a fixed value: $I = \frac{W}{Q} = 0.257$ and does not depend on edgefactor or scale.

Figure 7 shows the percentage error of BFS estimated power using Equation 13 and 14 compared to measured power . The β values for 1-core and 8-core versions of BFS are $\frac{1}{1+8\%}$ and $\frac{1}{1-19\%}$, respectively. The absolute percentage errors are at most 2% as shown in Figure 7.

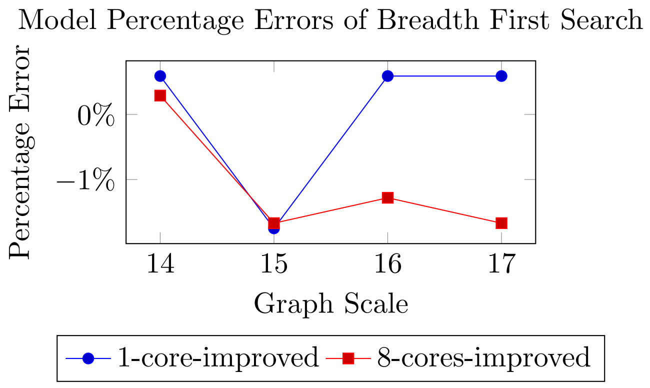


Figure 7: Absolute percentage errors of estimated power from measured power of BFS. After incorporating the tuning parameter, the absolute percentage errors are at most 2%.

2.3 Race-to-halt Prediction Framework

With the RTHpower models, we want to identify how many cores the system should use to run an application to achieve the least energy consumption. In order to answer the question, we need to consider the performance speed-up and power-up of an application on a specific platform.

From Amdahl's Law [58] the theoretical maximum speed-up of an application running on a multicore system is derived as Equation 15, where p denotes the fraction of the application that can be parallelized and n is the number of cores:

$$\textit{speed-up} \leq \frac{1}{(1-p) + \frac{p}{n}} \tag{15}$$

2.3.1 Framework Description

The purpose of this framework is to identify when to and when not to use RTH for a given application. The two required inputs for making decision are power-up and performance speed-up of the application executed with n cores, where n is the maximum number of cores.

- Step 1: Identify meta-data, including speed-up and operational intensity, of a given application by one of the three main approaches listed: i) doing theoretical analysis to find the amount of computation work W, data transfer Q and operational intensity I as well as identify the maximum speed-up of a given application; ii) executing the application on a targeted platform (e.g., Myriad1) to measure its performance speed-up and extract its operational intensity I; iii) using profiling tools [77] to extract the number of operations W and the amount of data transferred Q as well as the performance speed-up of an application on a common platform (e.g., Intel platform).

- Step 2: Compute power consumption of an application running with one core and with a maximum number of cores by the RTHpower models. Note that the RTHpower models are able to estimate power consumption for any number of cores by changing parameter n in the models. For verifying the RTH strategy, we only need to apply the model for a single core and all cores.

- Step 3: Compare the energy consumption of the application between using 1 core and using a maximum number of cores to identify whether running a maximum number of cores is the most energy-efficient.

2.3.2 Framework Validation

The framework is validated with three micro-benchmarks and three application kernels. In this validation, the values of operational intensity I are extracted from theoretical analysis of the implementations and performance speed-up is identified by executing the micro-benchmarks or application kernels with different numbers of cores.

2.3.2.1 Race-to-halt for Micro-benchmarks

We first validate the framework with micro-benchmarks. In this validation, we measure the power-up and performance speed-up of three micro-benchmarks: one with 60% parallel code, one with 100% parallel code and a small-size micro-benchmarks which has high overhead. All three micro-benchmarks have operational intensity $I = 0.25$. Namely, in the micro-benchmarks, each SauXor instruction is followed by a LsuLoad instruction which loads 4 bytes.

All three micro-benchmarks have the same assembly code wrapped inside a loop. The number of iterations to repeat the code are the difference among them. We run the micro-benchmarks on one SHAVE for 1 000 000 times. If the micro-benchmark has 100% parallel code, running it on n SHAVEs requires each SHAVE performing $\frac{1}{n}$ of the amount of work (e.g., if performing the micro-benchmark on 8 SHAVEs, each SHAVE needs to run 125 000 times). Similarly, if the micro-benchmark has a parallel fraction of 60%, then running the program on n SHAVEs requires each SHAVE to perform $(1 - 0.6) + (\frac{0.6}{n})$ of the amount of work (e.g.,if performing the micro-benchmark on 8 SHAVEs, each SHAVE needs to run 475 000 times). For small-size micro-benchmark, the code is executed 8 times with 1 core and once with 8 cores. Since the amount of computation is small, the relative overhead of initializing the platform and executing the small-size micro-benchmark is high.

Figure 8 shows that the power-up of running n SHAVEs to the program running 1 SHAVE varies from 1 (1 core) to 1.71 (8 cores) for operational intensity $I = 0.25$. If the performance speed-up is bigger than the power-up, RTH is an energy-saving strategy. If the speed-up is less than the power-up, running the program with the maximum number of cores consumes more energy than running it with 1 core. Note that when this happens, assigning one core to run the program is more energy-efficient and race-to-halt is no longer applicable for saving energy. For all three micro-benchmarks in this validation, the performance speed-up is identified by running them over different numbers of cores. The energy consumption of the three micro-benchmarks is shown in Figure 9. All three micro-benchmarks achieve the least energy consumption when executed with one core, from both measured and estimated

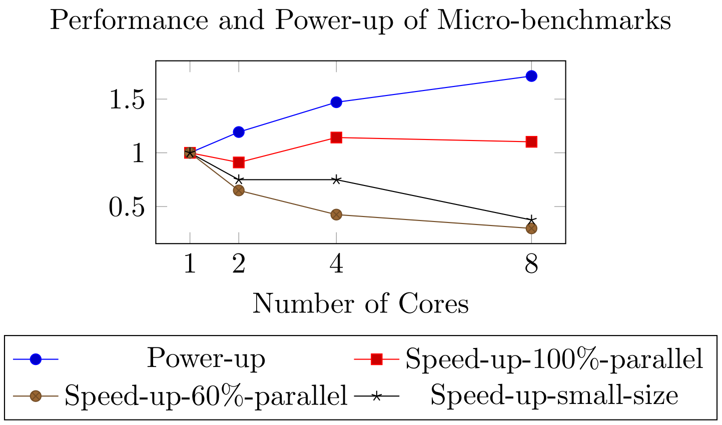


Figure 8: Performance and power-up of micro-benchmarks with operational intensity $I = 0.25$. All three reported micro-benchmarks have performance speed-up less than platform power-up.

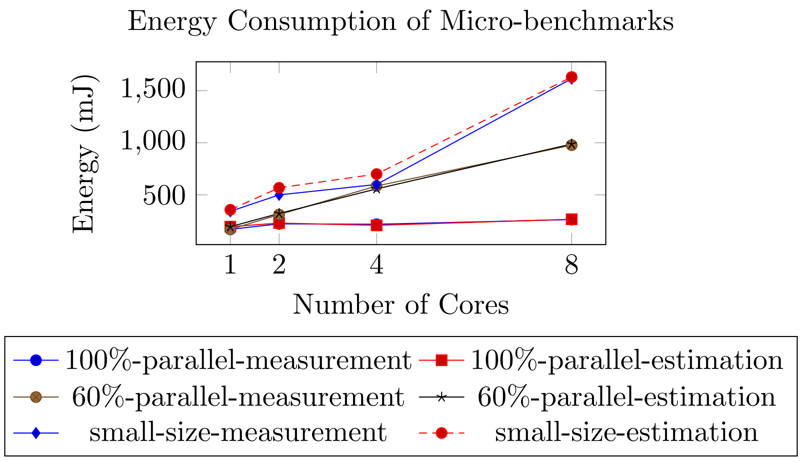


Figure 9: Energy consumption of micro-benchmarks with operational intensity $I = 0.25$. For all three reported micro-benchmarks, the programs executed with 1 core consume the least energy, compared to 2, 4, 8 cores, from both measured data and estimated data.

data. The model estimation and actual measurement show that RTH is not applicable to the three micro-benchmarks.

2.3.2.2 Race-to-halt for Dense Matrix Multiplication

The *matmul* application has increasing values of operational intensity over input sizes and its performance speed-up is higher than its power-up on Myriad1. Therefore, running *matmul* with the 8 cores is more energy-efficient than running it with one core. Figure 10 shows percentages of energy-saving if executing *matmul* with 8 cores instead of 1 core, from both measured and estimated data. The energy saving percentage is computed based on the

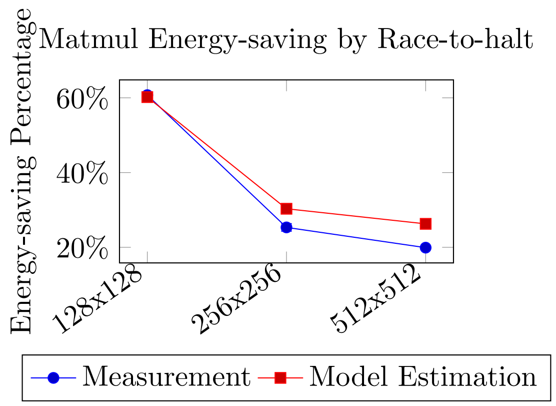


Figure 10: *matmul* Energy-saving by Race-to-halt. This diagram shows how many percentages of energy-saving if executing *matmul* with 8 cores instead of 1 core. Since the energy-saving percentage is positive over different matrix sizes, RTH is an energy-saving strategy for *matmul*. Energy-saving percentage from model estimation for *matmul* has standard deviation less than 3%.

energy gap of running 1 core and 8 cores divided by energy consumed by running 1 core as in Equation 16.

$$ES = \frac{E^{1core} - E^{8cores}}{E^{1core}} \tag{16}$$

The framework predict that RTH should be applied to *matmul* over different matrix sizes. By using RTH for *matmul*, we can save from 20% to 61% of *matmul* energy consumption. RTH is a good strategy for *matmul*. We observe that the energy saving reduces when matrix size increases due to the decrease of performance speed-up from size 128x128. The reason is that a matrix size bigger than 128x128 makes the data set no longer fit in the last level cache (or L2 cache of 64KB) and thereby lowers performance (in flops).

2.3.2.3 Race-to-halt for Sparse Matrix Vector Multiplication

SpMV has a fixed value of operational intensity over input sizes. From the RTHpower models as well as measurement data, the power-up of SpMV is relatively constant. However, SpMV has performance speed-up higher than its power-up. Therefore, running SpMV with the maximum number of cores is more energy-efficient than running it with one core. Figure 11 shows how many percentages of energy-saving if executing SpMV with 8 cores instead of 1 core, from both measured and estimated data. The framework can predict that RTH should be applied to SpMV over different matrix sizes. By using RTH for SpMV, we can save from 45% to 59% of SpMV energy consumption. RTH is a good strategy for SpMV. The energy saving increases from size 32x32 to 128x128 since the data fits in L1 cache.

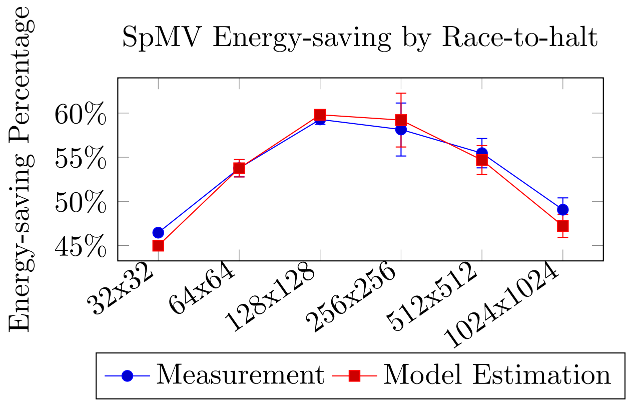


Figure 11: SpMV Energy-saving by Race-to-halt. This diagram shows how many percentages of energy-saving if execute SpMV with 8 cores instead of 1 core. Since the energy-saving percentage is positive over different matrix sizes, RTH is a energy-saving strategy for SpMV.

2.3.2.4 Race-to-halt for Breadth First Search

In our set of application kernels implemented on Myriad1, BFS is the application kernel able to prove that running with a maximum number of cores does not always give the least energy consumption. From both measured data and estimated data, the total energy consumed by running with one core is less than the total energy by running with 8 cores at scale 16 and 17. There are negative values of -5% and -3% in Figure 12 if running BFS with 8 cores instead of 1 cores at scale 16 and 17, respectively. The RTHpower models can predict when to apply RTH precisely for different scales.

The result can be explained by the relation between BFS power-up and performance speed-up. Since BFS has a fixed value of operational intensities across graph scale, from RTHpower models (cf. Equation 13 and 14), it is understood that BFS power consumption does not depend on graph scales and its power-up is a fixed value. From the measurement results, we also observe that BFS power-up is relatively constant over the graph scales. However, BFS speed-up in our experiments decreases when scale increases. The reason is that with the same graph degree, when scale increases, the graph becomes more sparse and disconnected. Compared to the Graph500 implementation, BFS search on Myriad1 are performed from a chosen subset of source nodes. The speed-up then, becomes less than power-up at scale 16 and 17. Therefore, running BFS with 8 cores at bigger graph scales (i.e., 16 and 17 in our experiments) consumes more energy than running BFS with one core.

2.4 Conclusion

In this study, new fine-grained power models have been proposed to provide insights into how a given application consumes energy when executing on an ultra-low power embedded system. In the models, applications are represented by their operational intensity. The models have

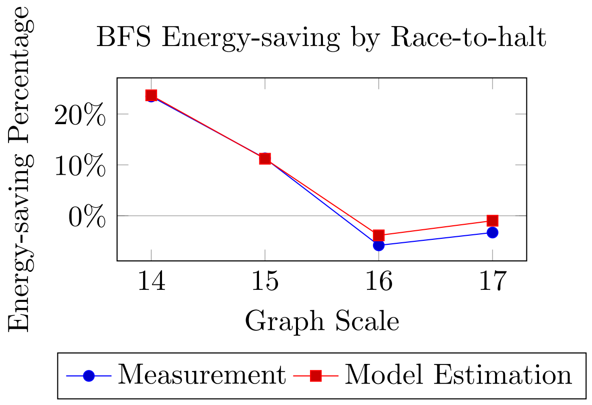


Figure 12: BFS Energy-saving by Race-to-halt. This diagram shows how many percentages of energy-saving if executing BFS with 8 cores instead of 1 core. The positive percentages at scale 14 and 15 mean that RTH should be applied. The negative percentages at scale 16 and 17 mean that RTH should not be applied. The standard deviation of BFS energy-saving percentage is less than 3%, from scale 14 to 17.

been validated on Movidius Myriad1, an ultra-low power embedded platform. Experimental results on 31 micro-benchmarks and three application kernels have shown high accuracy of estimated data by the model. Based on the models, we have devised a new framework to predict whether RTH is applicable to a given application. The framework has been validated by both micro-benchmarks and real application kernels, showing a prediction accuracy that is good enough for the purpose of deciding about RTH. Improving and applying the models and framework to other embedded platforms (e.g. ARM) and application kernels (e.g. other Berkeley dwarfs) are parts of our future work.

3 Energy Complexity Model for Multithreaded Algorithms

3.1 Introduction

Understanding the energy complexity of algorithms is crucially important to improve the energy efficiency of algorithms and reducing energy consumption of computing systems. By knowing the energy complexity of algorithms, the algorithm designers can choose one algorithm over the others to achieve energy optimization. Devising energy models is one of the main approaches to characterize the energy complexity of algorithms on computing systems. Energy models help to investigate the trade-offs between energy consumption and performance of algorithms as well as their inter-process communication.

Significant efforts have been devoted to develop power and energy models in literature [9, 22, 21, 67, 68, 60, 81, 94]. However, there are no analytic models for multithreaded algorithms that are both applicable to every algorithms and comprehensively validated yet. The existing *parallel* energy models are either theoretical studies without validation or only applicable for specific algorithms. Modeling energy consumption of *parallel* algorithms is difficult since the energy models need to model both algorithm characteristics and platform properties. Algorithm characteristics include computational workload, memory workload, data-accessing patterns and scalable parallelism. Platform properties include static and dynamic energy of memory accesses and computational operations. The previous energy model studies have not considered all the mentioned algorithm and platform properties.

The existing models and their classification are summarized in Table 4. The previous studies have not covered all listed aspects: ability to analyze the energy complexity of parallel algorithms (i.e. Energy complexity analysis for parallel algorithms), whether applicable to general algorithm (i.e., Algorithm Generality), whether the model is validated (i.e., Validation). Table 4 also shows how this work is different from the other studies.

The energy complexity model EPEM (Energy-aware Parallel External Memory) proposed in this study is for general multithreaded algorithms and validated on three aspects: different algorithms for a given problem, different input types types on different platforms. The proposed model is an analytic model which characterizes both algorithms (e.g., representing algorithm by *work*, *span* and *I/O* complexity) and platform properties (e.g., representing platforms with static and dynamic energy of memory accesses and computational operations). By considering *work* and *span* complexity, the new energy model can applied to any multithreaded algorithms.

Since the new EPEM energy model focuses on analyzing the energy complexity of algorithms, the model does not give the estimation of absolute energy consumption. The new model, however, provides the algorithms designers the understanding of how an algorithms consumes energy and give the insight on how to choose one algorithms over the others on different input types and platforms. This methodology has been validated for two SpMV algorithms running on two high performance platforms (Intel Xeon and Xeon Phi), computing nine matrix input types from Florida matrix collection [28]. The validation results prove

Table 4: Energy Model Summary

Study	**Energy complexity analysis for parallel algorithms**	**Algorithm generality**	**Validation**
LEO [81]	No	General	Yes
POET [60]	No	General	Yes
Koala [94]	No	General	Yes
Roofline [22, 21]	No	General	Yes
Energy scalability [67, 68]	Yes	General	No
Sequential energy complexity [89]	No	General	Yes
Alonso et al. [9]	Yes	Algorithm-specific	Yes
Malossi et al. [78]	Yes	Algorithm-specific	Yes
EPEM model	**Yes**	**General**	**Yes**

the practicability and applicability the EPEM energy complexity model when comparing the energy consumption of different algorithms as well as comparing different input types on different platforms. In this work, the following contributions have been made.

- Devise a new energy model EPEM for analyzing the energy complexity of multithreaded algorithms based on their *work*, *span* and *I/O* complexity (cf. Section 3.4).
- Conduct a case study to demonstrate the methodology how to apply the EPEM model to find energy complexity of three sparse matrix vector multiplication (SpMV) algorithms (i.e., Compressed Sparse Column(CSC) and Compressed Sparse Block(CSB) and Compressed Sparse Row(CSR))(cf. Section 3.5).
- Validate the EPEM energy complexity model according to three aspects: different algorithms, different input types and different platforms. The results show the precise prediction on which validated SpMV algorithm (i.e., CSB or CSC) consumes more energy when computing different matrix input types from Florida matrix collection [28] (cf. Section 3.6). The model platform-related parameters for 11 platforms, including x86, ARM and GPU, are also provided for further uses of the proposed energy complexity model.

3.2 Related Work - Overview of energy models

Devising accurate power models is crucial to gain insights into how a computer system consumes energy. Significant efforts have been devoted to predict energy, resulting in several energy model studies in the literature including analytic models [9, 22, 21, 67, 68] and

Table 5: Energy Model Details

Study	Parallel-Algorithm Support	Applicability	Validation	Communication model	Pre-run Overhead	Application properties
LEO [81]	parallel	Yes	Yes	No	Yes	None
POET [60]	parallel	Yes	Yes	No	No	None
Koala [94]	parallel	Yes	Yes	No	Yes	None
Roofline [22, 21]	sequential	Yes	Yes	Von Neumann shared cached	No	Operational intensity
Energy scalability [67]	parallel	Yes	No	Message passing	No	No. of messages No. of computations
Energy scalability [68]	parallel	Yes	No	CREW PEM	No	No. of mem-accesses No. of computations
Sequential energy complexity [89]	sequential	Yes	Yes	Uni-processor with parallel memory-bank	No	Work complexity I/O complexity
Alonso et al. [9]	parallel	No(Dense matrix factorization)	Yes	No	Yes	Application tasks
Malossi et al. [78]	parallel	No(Algebraic kernels)	Yes	Shared memory	Yes	No. of arithmetic, barrier mem-accesses, reduction
EPEM model	parallel	Yes	Yes	EPEM	No	Work, Span, I/O Input types

energy models to find energy-optimized system configurations [60, 81, 94]. We present the summary of existing modeling studies in Table 4. The characteristics of each approaches are extracted as the list of categories, including: whether the models support parallel algorithms (i.e., Parallel Algorithm Support), whether the model is applicable to general algorithms (i.e., Applicability), whether the model is validated (i.e., Validation), the communication model (i.e., Communication model), whether there is pre-run overhead before estimating energy consumption of applications (i.e., Pre-run Overhead) and how the model represents applications (i.e., App-properties). This summary is not an exhaustive survey on the topic of energy models. However, we believe the Table 5 represents the most current studies on energy models.

Energy models for finding energy-optimized system configurations for a given application have been recently reported [12, 16, 19]. Imes et al. [60] used controller theory and linear programming to find energy-optimized configurations for an application with soft real-time constraints at runtime. Mishra et al. [81] used hierarchical Bayesian model in machine learning to find energy-optimized configurations. They used offline learning to train the Bayesian model with a training set of applications with different patterns, and used online learning to quickly estimate the optimal configuration for a given application. Snowdon et al. [94] developed a power management framework called Koala which models the energy consumption of the platform and monitors an application' energy behavior. By matching an application's behavior with the system policy, an energy-optimized configuration is determined at runtime. Although the energy models for finding energy-optimized system configurations have resulted in energy saving in practice, they focus on characterizing system platforms rather than applications and therefore are not appropriate for analyzing the energy complexity of application algorithms.

Another direction of energy modeling study is to predict the energy consumption of applications by analyzing applications without actual execution on real platforms which we classify as analytic models. The analytic modeling approaches also provide algorithm designers the understanding about how their applications consume energy, helping them to improve the energy complexity of their algorithms.

Energy roofline models [22, 21] are some of the comprehensive energy models that abstract away possible algorithms in order to analyze and characterize different multicore platforms in terms of energy consumption. The models abstract possible algorithms by their operational intensity, the ratio of computation to communication (i.e., flop/byte), and characterize a platform's properties by running a set of micro-benchmarks on the platform. Our new energy model, which abstracts away possible multicore platform and characterize the energy complexity of algorithms based on their *work, span* and *I/O* complexity, complements the energy roofline models.

Validated energy models for *specific* algorithms have been reported recently [9, 78]. Alonso et al. [9] provided an accurate energy model for three key dense matrix factorizations. Malossi et al. [78] focused on basic linear-algebra kernels and characterized the kernels by the number of arithmetic operations, memory accesses, reduction and barrier steps. Although the energy models for specific algorithms are accurate for the target algorithms, they are not applicable for other algorithms and therefore cannot be used as general energy complexity models for parallel algorithms.

The *energy scalability* of a parallel algorithm has been investigated by Korthikanti et al. [67, 68]. The energy scalability studies are to find the optimal number of cores for a given algorithm with a real-time constraint which minimizes energy consumption. Our studies complement the energy scalability studies by addressing the following *energy complexity* question: *Given two parallel algorithms A and B for a given problem, which algorithm consumes less energy analytically?*. Unlike the energy scalability studies that have not been validated on real platforms, our new energy complexity model is validated on HPC and accelerator platforms, confirming its usability and accuracy.

The energy complexity of *sequential* algorithms on a *uniprocessor* machine with *several memory banks* has been studied by Roy et al. [89]. Our energy complexity studies complement Roy et al.'s studies by investigating the energy complexity of *parallel* algorithms on a *multiprocessor* machine with *a shared memory bank* and private caches, a machine model that has been widely adopted to study parallel algorithms [38, 12, 68].

Our new energy complexity model EPEM for multithreaded algorithms complements the aforementioned seminal studies on energy models. The EPEM model enables algorithm designers to analyze the energy complexity of their multithreaded algorithms without implementing and benchmarking the algorithms on a platform. We prove the model usability and accuracy by demonstrating how to apply the model for different SpMV algorithms and validating the results on HPC and accelerator platforms (i.e., Intel Xeon and Xeon Phi) using different sparse matrix types from Florida Matrix Collection.

3.3 EPEM Shared Memory Machine Model

Generally speaking, the energy consumption of a parallel algorithm is the sum of i) static energy (or leakage) E_{static}, ii) dynamic energy of computation E_{comp} and iii) dynamic energy of memory accesses E_{mem}. The static energy E_{static} is proportional to the execution time of the algorithm while the dynamic energy of computation and the dynamic energy of memory accesses are proportional to the number of computational operations and the number of memory accesses of the algorithm, respectively [68]. As a result, in the new EPEM energy complexity model the energy complexity of a multithreaded algorithm is analyzed based on its *span complexity* [24] (for the static energy), *work complexity* [24] (for the dynamic energy of computation) and *I/O complexity* (for the dynamic energy of memory accesses) (cf. Section 3.4).

This section describes shared-memory machine models supporting I/O complexity analysis for parallel algorithms. We first describe the parallel external memory (PEM) model [12] used for analyzing the energy scalability of parallel algorithms on shared memory multicore platforms [67] and explain why the PEM model is not appropriate for analyzing the energy complexity of multithreaded algorithms. We then describe the ideal distributed cache (IDC) model [39] that is used in the EPEM energy complexity model.

3.3.1 The PEM Model

The PEM model [12] is an extension of the Parallel Random Access Machine (PRAM) model that includes a two-level memory hierarchy. In the PEM model , there are n cores (or processors) each of which has its own *private* cache of size Z (in bytes) and shares the main memory with the other cores (cf. Figure 13). Unlike other I/O models for multicore platforms [16, 17], the PEM model enables analyzing the I/O complexity of parallel algorithms without additional assumption on how the cores are connected nor how the algorithm tasks are scheduled. In the PEM model, data is transferred between the shared memory and the cache in the form of blocks of size B (i.e., cache lines). The number of *parallel* block transfers between the shared memory and the caches is defined as *I/O complexity*. Namely, when n cores access n distinct blocks from the shared memory *simultaneously*, the I/O complexity in the PEM model is $O(1)$ instead of $O(n)$.

Like the PRAM shared-memory parallel model, the PEM model has three variations according to how multiple cores access the *same* block of shared memory, namely: Concurrent Read, Concurrent Writes (CRCW); Concurrent Read, Exclusive Write (CREW) and Exclusive Read, Exclusive Write (EREW). In the cases of exclusive write (i.e., CREW and EREW), there are write conflicts between n simultaneous writes to the same block in the main memory. A solution to the conflict is to serialize the n writes, resulting in n I/Os. The I/O complexity of n conflicting writes can be improved to $O(\log n)$ by using extra memory to combine the writes in a binary tree fashion [12].

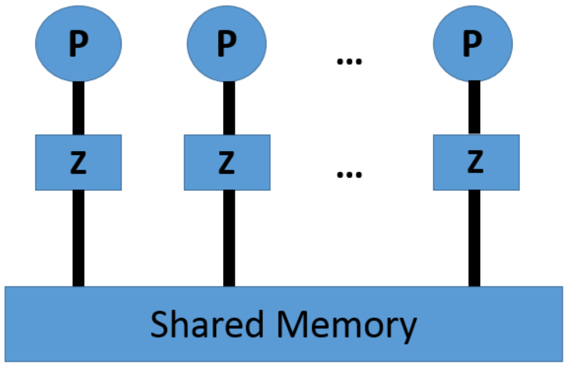


Figure 13: A Shared Memory Machine Model with Private Caches

3.3.2 The IDC Model

Although the PEM model is appropriate for analyzing the I/O complexity of parallel algorithms in terms of time performance [12], we have found that the PEM model is not appropriate for analyzing parallel algorithms in terms of the dynamic energy of memory accesses. In fact, even when the n cores can access data from the main memory simultaneously, the *dynamic* energy consumption of the access is proportional to the number n of accessing cores (because of the load-store unit activated within each accessing core and the energy compositionality of parallel computations [46, 74]), rather than a constant as implied by the PEM model.

As a result, we choose the ideal distributed cache (IDC) model [39] to analyze I/O complexity of multithreaded algorithms in terms of dynamic energy consumption. Like the PEM model, the IDC model has n cores and a two-level memory hierarchy as shown in Figure 13. Each core has its own private cache of size Z, which cannot be accessed by the other cores, and shares the main memory with the other cores. All the inter-core communication is conducted through writing to and reading from the main memory. The core must have data in its cache in order to operate on the data and the data is transferred between the main memory and its cache in blocks of size B (i.e., cache line size).

Unlike the PEM model, the IDC model defines I/O complexity (or cache complexity) of a computation as the number of *cache misses* caused by the computation on an *ideal cache* starting and ending with an empty cache. An ideal cache is a fully associative cache that uses optimal offline cache replacement policy. If a core does not have the data word it wants to access in its private cache, it incurs a cache miss to bring the data from the main memory to its private cache. The private caches are non-interfering, namely the number of cache misses incurred by a core can be analyzed independently of the other cores' actions. Since the cache complexity of m misses is $O(m)$ regardless of whether or not the cache misses are incurred simultaneously by the cores, the IDC model reflects the aforementioned dynamic energy consumption of memory accesses by the cores.

However, the IDC model is mainly designed for analyzing the cache complexity of divide-

and-conquer algorithms, making it difficult to apply to general multi-threaded algorithms targeted by our new EPEM energy model. Constraining the new EPEM energy model to the IDC model would limit the applicability of the EPEM model to a wide range of multithreaded algorithms.

In order to make our new EPEM energy model applicable to a wide range of multithreaded algorithms, we show that the cache complexity analysis using the traditional (sequential) ideal cache (IC) model [37] can be used to find an upper bound on the cache complexity of the same algorithm using the IDC model (cf. Lemma 1). As the sequential execution of multithreaded algorithms is a valid execution regardless of whether they are divide-or-conquer algorithms, the ability to analyze the cache complexity of multithreaded algorithms via their sequential execution in the EPEM energy model improves the usability of the EPEM model.

Let $Q_1(Alg, B, Z)$ and $Q_P(Alg, B, Z)$ be the cache complexity of a parallel algorithm Alg analyzed in the (uniprocessor) ideal cache (IC) model [37] with block size B and cache size Z (i.e, running Alg with a single core) and the cache complexity analyzed in the (multicore) IDC model with P cores each of which has a private cache of size Z and block size B, respectively. We have the following lemma:

Lemma 1. *The cache complexity $Q_P(Alg, B, Z)$ of a parallel algorithm Alg analyzed in the ideal distributed cache (IDC) model with P cores is bounded from above by the product of P and the cache complexity $Q_1(Alg, B, Z)$ of the same algorithm analyzed in the ideal cache (IC) model. Namely,*

$$Q_P(Alg, B, Z) \leq P * Q_1(Alg, B, Z) \tag{17}$$

Proof. (Sketch) Let $Q^i_P(Alg, B, Z)$ be the number of cache misses incurred by core i during the parallel execution of algorithm Alg in the IDC model. Because caches do not interfere with each other in the IDC model, the number of cache misses incurred by core i when executing algorithm Alg in parallel by P cores is not greater than the number of cache misses incurred by core i when executing the whole algorithm Alg only by core i. That is,

$$Q^i_P(Alg, B, Z) \leq Q_1(Alg, B, Z) \tag{18}$$

or

$$\sum_{i=1}^{P} Q^i_P(Alg, B, Z) \leq P * Q_1(Alg, B, Z) \tag{19}$$

On the other hand, since the number of cache misses incurred by algorithm Alg when it is executed by P cores in the IDC model is the sum of the numbers of cache misses incurred by each core during the Alg execution, we have

$$Q_P(Alg, B, Z) = \sum_{i=1}^{P} Q^i_P(Alg, B, Z) \tag{20}$$

From Equations 19 and 20, we have

$$Q_P(Alg, B, Z) \leq P * Q_1(Alg, B, Z) \tag{21}$$

$\square$

We also make the following assumptions regarding platforms.

- Algorithms are executed with the best configuration (e.g., maximum number of cores, maximum frequency) following the race-to-halt strategy.
- The I/O parallelism is bounded from above by the computation parallelism. Namely, each core can issue a memory request only if its previous memory requests have been served. Therefore, the work and span (i.e., critical path) of an algorithm represent the parallelism for both I/O and computation.

3.4 Energy Complexity in EPEM model

This section describes two energy complexity models, a platform-supporting energy complexity model considering both platform and algorithm characteristics and platform-independent energy complexity model considering only algorithm characteristics. The platform-supporting model is used when platform parameters in the model are available while platform-independent model analyses energy complexity of algorithms without considering platform characteristics.

3.4.1 Platform-supporting Energy Complexity Model

This section describes a methodology to find energy complexity of algorithms. The energy complexity model considers three groups of parameters: machine-dependent, algorithm-dependent and input-dependent parameters. The reason to consider all three parameter-categories is that only operational intensity [105] is insufficient to capture the characteristics of algorithms. Two algorithms with the same values of operational intensity might consume different levels of energy. The reasons are their differences in data accessing patterns leading to performance scalability gap among them. For example, although the sequential version and parallel version of an algorithm may have the same operational intensity, they may have different energy consumption since the parallel version would have less static energy consumption because of shorter execution time.

$$E = \epsilon_{op} \times Work + \epsilon_{I/O} \times I/O + P^{sta} \times max(T^{comp}, T^{mem}) \tag{22}$$

$$E = \epsilon_{op} \times Work + \epsilon_{I/O} \times I/O + max(\pi_{op} \times Span, \pi_{I/O} \times \frac{I/O \times Span}{Work}) \tag{23}$$

The energy consumption of a parallel algorithm is the sum of i) static energy (or leakage) E_{static}, ii) dynamic energy of computation E_{comp} and iii) dynamic energy of memory accesses E_{mem}: $E = E_{static} + E_{comp} + E_{mem}$. The static energy E_{static} is the product of the execution time of the algorithm and the static power of the whole processor. The dynamic energy of

Table 6: Platform Parameter Description

Machine	Description
P^{sta}	Static power of a whole chip
P^{op}	Dynamic power of an operation
$P^{I/O}$	Power to transfer one cache line
N	Maximum number of cores in the platform
M	Number of cycles per cache line transfer
F	Number of cycles per operation
$Freq$	Platform frequency
Z	Cache size of a single processor

Table 7: EPEM Model Parameter Description

Machine	Description
ϵ_{op}	dynamic energy of one operation (1 core)
$\epsilon_{I/O}$	dynamic energy of a random access (1 core)
π_{op}	static energy when performing one operation
$\pi_{I/O}$	static energy of a random data access
B	cache block size
Algorithm	Description
$Work$	Number of work in flops of the algorithm
$Span$	The critical path of the algorithm
I/O	Number of cache line transfer
SpMV Input	Description
n	Number of rows
nz	Number of nonzero elements
nr	Maximum number of nonzero in a row
nc	Maximum number of nonzero in a column
β	Size of a block

Table 8: Platform parameter summary. The parameters of the first nine platforms are derived from [21] and the parameters of the two new platforms are found in this study.

Platform	Processor	ϵ_{op}(nJ)	π_{op}(nJ)	$\epsilon_{I/O}$(nJ)	$\pi_{I/O}$(nJ)
Nehalem i7-950	Intel i7-950	0.670	2.455	50.88	408.80
Ivy Bridge i3-3217U	Intel i3-3217U	0.024	0.591	26.75	58.99
Bobcat CPU	AMD E2-1800	0.199	3.980	27.84	387.47
Fermi GTX 580	NVIDIA GF100	0.213	0.622	32.83	45.66
Kepler GTX 680	NVIDIA GK104	0.263	0.452	27.97	26.90
Kepler GTX Titan	NVIDIA GK110	0.094	0.077	17.09	32.94
XeonPhi KNC	Intel 5110P	0.012	0.178	8.70	63.65
Cortex-A9	TI OMAP 4460	0.302	1.152	51.84	174.00
Arndale Cortex-A15	Samsung Exynos 5	0.275	1.385	24.70	89.34
Xeon	2xIntel E5-2650l v3	0.263	0.108	8.86	23.29
Xeon-Phi	Intel 31S1P	0.006	0.078	25.02	64.40

computation and the dynamic energy of memory accesses are proportional to the number of computational operations $Work$ and the number of memory accesses of the algorithm I/O, respectively [68]. Since computation time and memory-access time can be overlapped, the execution time of the algorithm is the maximum value of computation time and memory-access time. Therefore, the energy consumption of algorithms is computed by Equation 22.

The computation time of parallel algorithms is proportional to the span complexity of the algorithm, which is $T^{comp} = \frac{Span \times F}{Freq}$ where $Freq$ is the processor frequency, and F is the number of cycles per operation. The memory-access time of parallel algorithms in the EPEM model is is proportional to the I/O complexity of the algorithm divided by its I/O parallelism. As I/O parallelism is bounded by the computation parallelism (cf. Section 3.3), I/O parallelism is divided by $\frac{Work}{Span}$. The memory-access time T^{mem} becomes: $T^{mem} = \frac{I/O \times Span \times M}{Work \times Freq}$ where M is the number of cycles per cache line transfer. If an algorithm has T^{comp} greater than T^{mem}, the algorithm is a CPU-bound algorithm. Otherwise, it is a memory-bound algorithm.

The summary of platform parameters are listed in Table 6. The EPEM energy complexity model in Equation 22 is simplified to Equation 23, where the mathematical meaning of ϵ_{op}, $\epsilon_{I/O}$, π_{op}, and $\pi_{I/O}$ are described in the Equation 24, 25, 26, and 27. The model considers the parameters listed in Table 7. The parameter values of recent computing platforms are summarized in Table 8. How to obtain the platform parameters is discussed in Section 3.6.3.

The dynamic energy of one operation by one core ϵ_{op} is the product of the consumed power of one operation by one active core P^{op} and the time to perform one operation. Equation 24 shows how ϵ_{op} relates to frequency $Freq$ and time per operation F. Similarly, the dynamic energy of a random access by one core $\epsilon_{I/O}$ is the product of the consumed power by one active core performing one I/O (i.e., cache-line transfer) $P^{I/O}$ and the time to perform one cache line transfer computed as $M/Freq$ (cf. Equation 25). The static energy of operations π_{op} is the product of the whole chip static power P^{sta} and time per operation. The static energy of one I/O $\pi_{I/O}$ is the product of the whole chip static power and time per I/O, shown by Equation 26 and 27.

$$\epsilon_{op} = P^{op} \times \frac{F}{Freq} \tag{24}$$

$$\epsilon_{I/O} = P^{I/O} \times \frac{M}{Freq} \tag{25}$$

$$\pi_{op} = P^{sta} \times \frac{F}{Freq} \tag{26}$$

$$\pi_{I/O} = P^{sta} \times \frac{M}{Freq} \tag{27}$$

3.4.2 CPU-bound Algorithms

If an algorithm has computation time longer than time for accessing data (i.e., CPU-bound algorithms): $T^{comp} \geq T^{mem}$, the EPEM energy complexity model becomes Equation 28 and 29.

$$E = \epsilon_{op} \times Work + \epsilon_{I/O} \times I/O + P^{sta} \times \frac{Span \times F}{Freq} \tag{28}$$

or

$$E = \epsilon_{op} \times Work + \epsilon_{I/O} \times I/O + \pi_{op} \times Span \tag{29}$$

3.4.3 Memory-bound Algorithms

If an algorithm has data-accessing time longer than computation time (i.e., memory-bound algorithms): $T^{mem} \geq T^{comp}$, energy complexity becomes Equation 30 and 31.

$$E = \epsilon_{op} \times Work + \epsilon_{I/O} \times I/O + P^{sta} \times \frac{I/O \times Span \times M}{Work \times Freq} \tag{30}$$

or

$$E = \epsilon_{op} \times Work + \epsilon_{I/O} \times I/O + \pi_{I/O} \times \frac{I/O \times Span}{Work} \tag{31}$$

3.4.4 Platform-independent Energy Complexity Model

This section describes the energy complexity model that is platform-independent and considers only algorithm characteristics. When the platform parameters (i.e., ϵ_{op}, $\epsilon_{I/O}$, π_{op}, and $\pi_{I/O}$) are unavailable, the energy complexity model is derived from Equation 23. The platform parameters are constants and can be removed from the Equation 23. Assuming

$\pi_{max} = max(\pi_{op}, \pi_{I/O})$, after removing platform parameters, the platform-independent energy complexity model are shown in Equation (32).

$$E = O(Work + I/O + max(Span, \frac{I/O \times Span}{Work})) \tag{32}$$

3.5 A Case Study - SpMV Energy Complexity

SpMV is one of the most common application kernels in Berkeley dwarf list[13]. It computes a vector result y by multiplying a sparse matrix A with a dense vector x: $y = A \times x$. SpMV is a data-intensive kernel and has irregular memory-access patterns. The data access patterns for SpMV is defined by its sparse matrix format and matrix input types. There are several sparse matrix formats and SpMV algorithms in literature. To name a few, they are Coordinate Format (COO), Compressed Sparse Column (CSC), Compressed Sparse Row (CSR), Compressed Sparse Block (CSB), Recursive Sparse Block (RSB), Block Compressed Sparse Row (BCSR) and so on. Three popular SpMV algorithms, namely CSC, CSB and CSR are chosen to validate the proposed energy complexity model. They have different data-accessing patterns leading to different values of I/O, work and span complexity. Since SpMV is a memory-bound application kernel, Equation 29 is applied.

3.5.1 Compressed Sparse Row

CSR is a standard storage format for sparse matrices which reduces the storage of matrix compared to the tuple representation [69]. This format enables row-wise compression of A with size $n \times n$ (or $n \times m$) to store only the non-zero nz elements. Let nz be the number of non-zero elements in matrix A. The work complexity of CSR SpMV is $\Theta(nz)$ where $nz >= n$ and span complexity is $O(nr + \log n)$ [20], where nr is the maximum number of non-zero elements in a row. The I/O complexity of CSR in the sequential I/O model of row-major layout is $O(nz)$ [15] namely, scanning all non-zero elements of matrix A costs $O(\frac{nz}{B})$ I/Os with B is the cache block size. However, randomly accessing vector x causes the total of $O(nz)$ I/Os. Applying the proposed model on CSR SpMV, their total energy complexity are computed as Equation 33.

$$E_{CSR} = O(\epsilon_{op} \times nz + \epsilon_{I/O} \times nz + \pi_{I/O} \times (nr + \log n)) \tag{33}$$

3.5.2 Compressed Sparse Column

CSC is the similar storage format for sparse matrices as CSR. However, it compresses the sparse matrix in column-wise manner to store the non-zero elements. The work complexity of CSC SpMV is $\Theta(nz)$ where $nz >= n$ and span complexity is $O(nc + \log n)$, where nc is the maximum number of non-zero elements in a column. The I/O complexity of CSC in the sequential I/O model of column-major layout is $O(nz)$ [15]. Similar to CSR, scanning all non-zero elements of matrix A in CSC format costs $O(\frac{nz}{B})$ I/Os. However, randomly

$$E_{CSB} = O(\epsilon_{op} \times (\frac{n^2}{\beta^2} + nz) + \epsilon_{I/O} \times (\frac{n^2}{\beta^2} + \frac{nz}{B}) + \pi_{I/O} \times \frac{(\frac{n^2}{\beta^2} + \frac{nz}{B}) \times (\beta \times \log \frac{n}{\beta} + \frac{n}{\beta})}{(\frac{n^2}{\beta^2} + nz)}) \quad (35)$$

Table 9: SpMV Complexity Analysis

Complexity	CSC	CSB	CSR
Work	$\Theta(nz)$ [20]	$\Theta(\frac{n^2}{\beta^2} + nz)$ [20]	$\Theta(nz)$ [20]
I/O	$O(nz)$ [15]	$O(\frac{n^2}{\beta^2} + \frac{nz}{B})$ [this report]	$O(nz)$ [15]
Span	$O(nc + \log n)$ [20]	$O(\beta \times \log \frac{n}{\beta} + \frac{n}{\beta})$ [20]	$O(nr + \log n)$ [20]

updating vector y causing the bottle neck with total of $O(nz)$ I/Os. Applying the proposed model on CSC SpMV, their total energy complexity are computed as Equation 34.

$$E_{CSC} = O(\epsilon_{op} \times nz + \epsilon_{I/O} \times nz + \pi_{I/O} \times (nc + \log n)) \quad (34)$$

3.5.3 Compressed Sparse Block

Given a sparse matrix A, while CSR has good performance on SpMV $y = A \times x$, CSC has good performance on transpose sparse matrix vector multiplication $y = A^T \times x$, Compressed sparse blocks (CSB) format is efficient for computing either Ax or $A^T x$. CSB is another storage format for representing sparse matrices by dividing the matrix A and vector x, y to blocks. A block-row contains multiple chunks, each chunks contains consecutive blocks and non-zero elements of each block are stored in Z-Morton-ordered [20]. From Beluc et al. [20], CSB SpMV computing a matrix with nz non-zero elements, size $n \times n$ and divided by block size $\beta \times \beta$ has span complexity $O(\beta \times \log \frac{n}{\beta} + \frac{n}{\beta})$ and work complexity as $\Theta(\frac{n^2}{b^2} + nz)$.

I/O complexity for CSB SpMV is not available in the literature. We do the analysis of CSB manually by following the master method [24]. The I/O complexity is analyzed for the algorithm CSB_SpMV(A,x,y) from Beluc et al. [20]. The I/O complexity of CSB is similar to work complexity of CSB $O(\frac{n^2}{\beta^2} + nz)$, only that non-zero accesses in a block is divided by B: $O(\frac{n^2}{\beta^2} + \frac{nz}{B})$. The reason is that non-zero elements in a block are stored in Z-Morton order which only requires $\frac{nz}{B}$ I/Os. The energy complexity of CSB SPMV is shown in Equation 35.

From the complexity analysis of SpMV algorithms using different layouts, the complexity of CSR, CSC and CSB are summarized in Table 9.

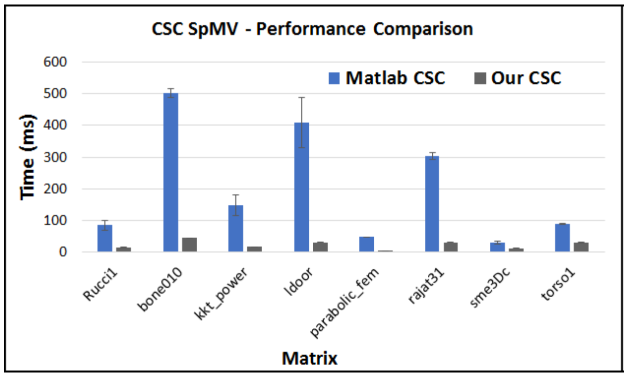


Figure 14: Performance (time) comparison of two parallel CSC SpMV implementations. For a set of different input matrices, the parallel CSC SpMV using Cilk out-performs Matlab parallel CSC.

3.6 Validation of EPEM Model

This section describes the experimental study to validate the EPEM model, including: describing SpMV implementation and sparse matrix types used in this validation (cf. Section 3.6.1), introducing the two experimental platforms and how to obtain their parameters for the EPEM model(cf. Section 3.6.3) and discussing the validation results.

3.6.1 SpMV Implementation

We want to conduct complexity analysis and experimental study with two SpMV algorithms, namely CSB and CSC. Parallel CSB and sequential CSC implementations are available thanks to the study from Buluç et al. [20]. Since the optimization steps of available parallel SpMV kernels (e.g., pOSKI [3], LAMA[36]) might affect the work complexity of the algorithms, we decided to implement a pure parallel CSC using Cilk and *Pthread*. To validate the correctness of our parallel CSC implementation, we compare the vector result y from $y = A * x$ of CSC and CSB implementation. The comparison shows the equality of the two vector results y. Moreover, we compare the performance of the our parallel CSC implementation with Matlab parallel CSC-SpMV implementation. Matlab also uses CSC layout as the format for their sparse matrix [41]. Our CSC implementation has out-performed Matlab parallel CSC kernel when computing the same targeted input matrices. Figure 14 shows the performance comparison of our CSC SpMV implementation and Matlab CSC SpMV kernel. The experimental study of SpMV energy consumption is then conducted with CSB SpMV implementation from Buluç et al. [20] and our CSC SpMV parallel implementation.

3.6.2 SpMV Matrix Input Types

We conducted the experiments with nine different matrix-input types from Florida sparse matrix collection [28]. Each matrix input has different properties, including size of the matrix $n \times m$, the maximum number of non-zero of the sparse matrix nz, the maximum number

Table 10: Sparse matrix input types. The maximum number of non-zero elements in a column *nc* is derived from [20].

Matrix type	**n**	**m**	**nz**	**nc**
bone010	986703	986703	47851783	63
kkt_power	2063494	2063494	12771361	90
ldoor	952203	952203	42493817	77
parabolic_fem	525825	525825	3674625	7
pds-100	156243	517577	1096002	7
rajat31	4690002	4690002	20316253	1200
Rucci1	1977885	109900	7791168	108
sme3Dc	42930	42930	3148656	405
torso1	116158	116158	8516500	1200

of non-zero elements in one column *nc*. Table 10 lists the matrix types in this experimental validation.

3.6.3 Experiment Set-up

For the validation of the EPEM model, we conduct the experiments on two HPC platforms: one platform with two Intel Xeon E5-2650l v3 processors and one platform with Xeon Phi 31S1P processor. The Intel Xeon platform has two processors Xeon E5-2650l v3 with 2×12 cores, each processor has the frequency 1.8 GHz. The Intel Xeon Phi platform has one processor Xeon Phi 31S1P with 57 cores and its frequency is 1.1 GHz. To measure energy consumption of the platforms, we read the PCM MSR counters for Intel Xeon and MIC power reader for Xeon Phi.

3.6.4 Identifying Platform Parameters

We apply the energy roofline approach [22, 21] to find the platform parameters for the two new experimental platforms, namely Intel Xeon E5-2650l v3 and Xeon Phi 31S1P. The energy roofline study [21] has also provided a list of other platforms including CPU, GPU, embedded platforms with their parameters considered in the Roofline model. Thanks to authors Choi et al. [21], we extract the parameters required for the EPEM energy complexity model from their platform data. Along with the two HPC platforms used in this validation, we provide parameters required in the energy complexity model for a list of available platforms. The parameter values of recent computing platforms for further uses are listed in Table 8.

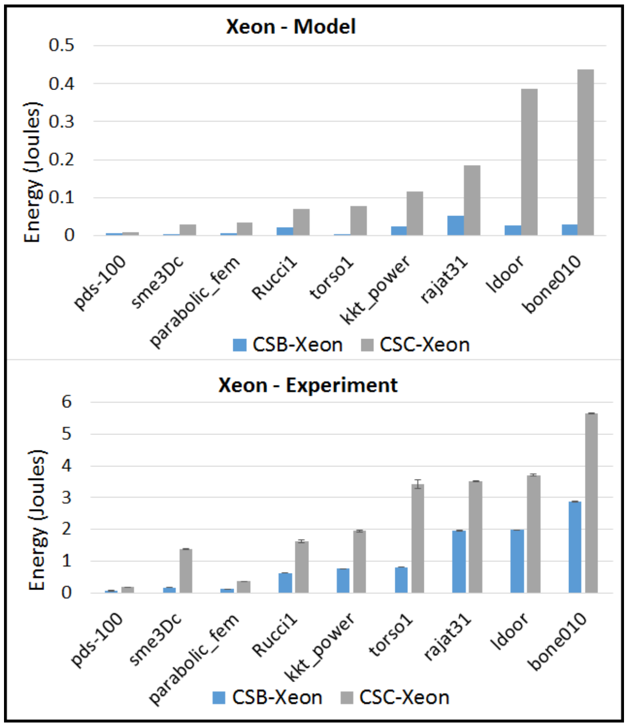


Figure 15: Algorithm comparison of CSB and CSC SpMV energy consumption from the EPEM model estimation and experimental measurement on Intel Xeon platform. The EPEM model is able to predict that the CSC SpMV algorithm consumes more energy than the CSB SpMV algorithm, on different matrix input types.

3.6.5 Validating EPEM Using Different SpMV Algorithms

Figure 15 and 16 show the energy prediction and measurement of CSB SpMV and CSC SpMV algorithms on two platforms Xeon and Xeon Phi. From the model-estimated data, CSB SpMV consumes less energy than CSC SpMV on both platforms. Even though CSB has higher work complexity than CSC, CSB SpMV has less I/O complexity than CSC SpMV. Firstly, the dynamic energy cost of one I/O is much greater than the energy cost of one operation (i.e., $\epsilon_{I/O} >> \epsilon_{op}$) on both platforms. Secondly, CSB has better parallelism than CSC, computed by $\frac{Work}{Span}$, which results in shorter execution time. Both reasons contribute to the less energy consumption of CSB SpMV.

The measurement data confirms that CSB SpMV algorithm consumes less energy than CSC SpMV algorithm, shown in the Figure 15 and 16. For all input matrices, the model has predicted precisely that CSB SpMV consumes less energy than CSC SpMV algorithm.

3.6.6 Validating EPEM Using Different Input Types

To validate the EPEM model regarding input types, the experiments have been conducted with nine matrix types listed in Table 10. The model can capture the energy-consumption

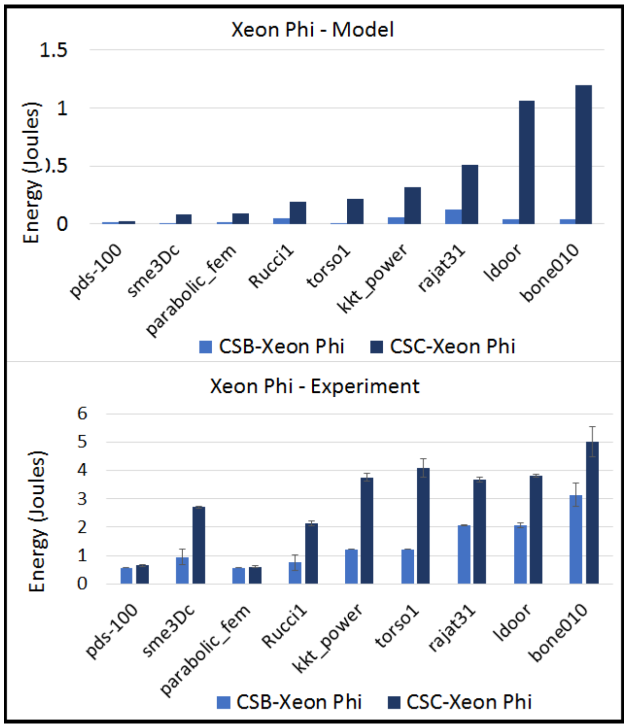


Figure 16: Algorithm comparison of CSB and CSC SpMV energy consumption from the EPEM model estimation and experimental measurement on Intel Xeon Phi platform. The EPEM model is able to predict that the CSC SpMV algorithm consumes more energy than the CSB SpMV algorithm, on different matrix input types.

relation among different inputs. The increasing order of energy consumption of different matrix-input types are shown in Table 11, from both model estimation and experimental study. For instance, in order to validate the comparison of energy consumption for different input types, a validated table as Table 13 is created for CSC SpMV on Xeon to compare model prediction and experimental measurement. For nine input types, there are $\frac{9\times9}{2}-9=36$ input relations. If the relation is correct, meaning both experimental data and model data are the same, the relation value in the table of two inputs is 1. Otherwise, the relation value is 0. From Table 13, there are 34 out of 36 relations are the same for both model and experiment, which gives 94% accuracy on the relation of the energy consumption of different inputs. Similarly, the input validation for CSC and CSB on both Xeon and Xeon Phi platforms is provided in Table 12.

3.6.7 Validating The Applicability of EPEM on Different Platforms

The energy comparison of CSB and CSC SpMV is predicted for eleven platforms listed in Table 8. Like two Xeon and Xeon Phi 31S1P platforms used in experiments, Figure 17 shows the prediction that CSB SpMV consumes less energy than CSC SpMV, on all platforms listed in Table 8. This confirms the applicability of EPEM model to compare energy consumption

Table 11: Comparison of Energy Consumption of Different Matrix Input Types.

Algorithm	CSB	CSB	CSC	CSC	CSB	CSB	CSC	CSC
Platform	model-X	exprmt-X	model-X	exprmt-X	model-XP	exprmt-XP	model-XP	exprmt-XP
Increasing	sme3Dc	pds-100	pds-100	pds-100	sme3Dc	pds-100	pds-100	parabolic
Energy	torso1	parabolic	sme3Dc	parabolic	torso1	parabolic	sme3Dc	pds-100
Consumption	pds-100	sme3Dc	parabolic	sme3Dc	pds-100	Rucci1	parabolic	Rucci1
Order	parabolic	Rucci1	Rucci1	Rucci1	parabolic	sme3Dc	Rucci1	sme3Dc
	Rucci1	kkt	torso1	kkt	ldoor	kktr	torso1	rajat31
	kkt	torso1	kkt	torso1	bone010	torso1	kkt	kkt
	ldoor	rajat31	rajat31	rajat31	Rucci1	rajat31	rajat31	ldoor
	bone010	ldoor	ldoor	ldoor	kkt	ldoor	ldoor	torso1
	rajat31	bone010	bone010	bone010	rajat31	bone010	bone010	bone010

Table 12: Comparison accuracy of SpMV energy consumption computing different input matrix types

Algorithm	CSB	CSC
Xeon	75%	94%
Xeon Phi	63.8%	80.5%

Table 13: CSC Energy Comparison of Different Input Matrix Types on Xeon

Correctness	pds-100	parabolic	sme3Dc	Rucci1	kkt	torso1	rajat31	ldoor	bone010
pds-100	x	1	1	1	1	1	1	1	1
parabolic		x	0	1	1	1	1	1	1
sme3Dc			x	1	1	1	1	1	1
Rucci1				x	1	1	1	1	1
kkt					x	0	1	1	1
torso1						x	1	1	1
rajat31							x	1	1
ldoor								x	1
bone010									x

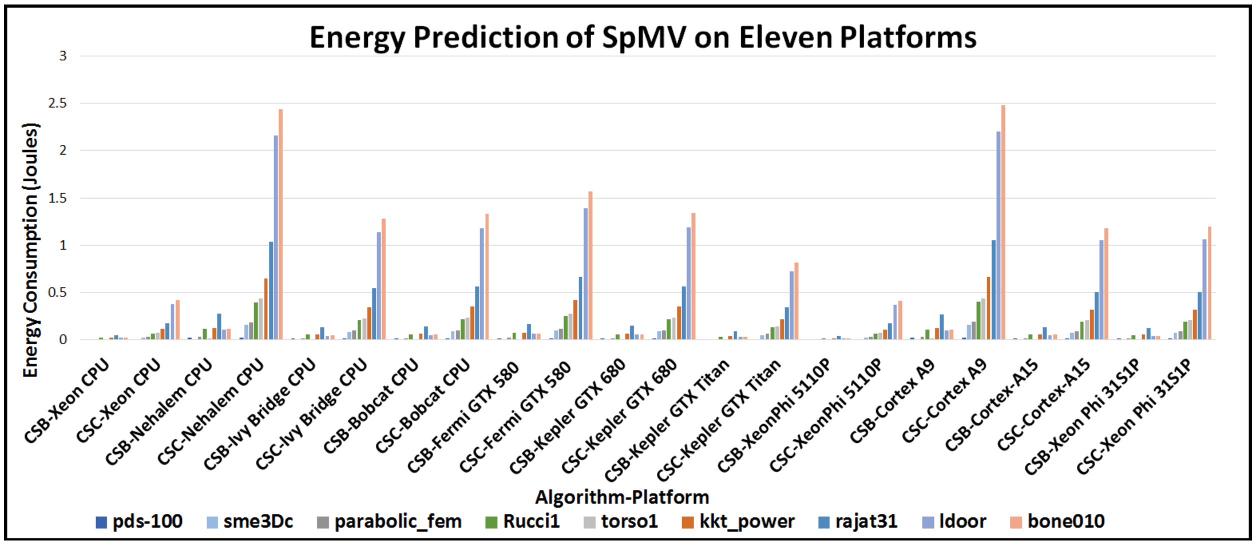


Figure 17: Energy Comparison of CSB and CSC SpMV on eleven different platforms.

of algorithms for different input types on different platforms.

3.6.8 Validating the Platform-independent Energy Complexity Model

From Equation 34 and 35, the platform-independent energy complexity for CSC and CSB SpMV are derived as Equation 36 and 37, respectively.

$$E_{CSC} = O(2 \times nz + (nc + \log n)) \tag{36}$$

$$E_{CSB} = O(2 \times \frac{n^2}{\beta^2} + nz \times (1 + \frac{1}{B}) + \beta \times \log \frac{n}{\beta} + \frac{n}{\beta}) \tag{37}$$

We validate the platform-independent energy complexity of CSC and CSB SpMV with experimental results. The platform-independent energy complexity also shows the accurate comparison of CSC and CSB SpMV computing different matrix types shown in Figure 18. Both platform-independent and platform-supporting models can predict which SpMV algorithm consumes more energy. The difference between the energy complexity of CSC and CSB using the platform-independent model is not very clear for all the input types except "ldoor" while in the platform-supporting model, the difference is clear for each input types and consistent with the experiment results in terms of which algorithm consumes less energy for the input types. Comparing energy consumption of different input types requires more detailed information of the platforms. Therefore, the platform-independent model is only applicable to predict which algorithm consumes more energy.

3.7 Conclusion

In this study, energy models for algorithms to predict energy consumption of an application have been devised. Based on the analysis of application complexity such as *work* complexity,

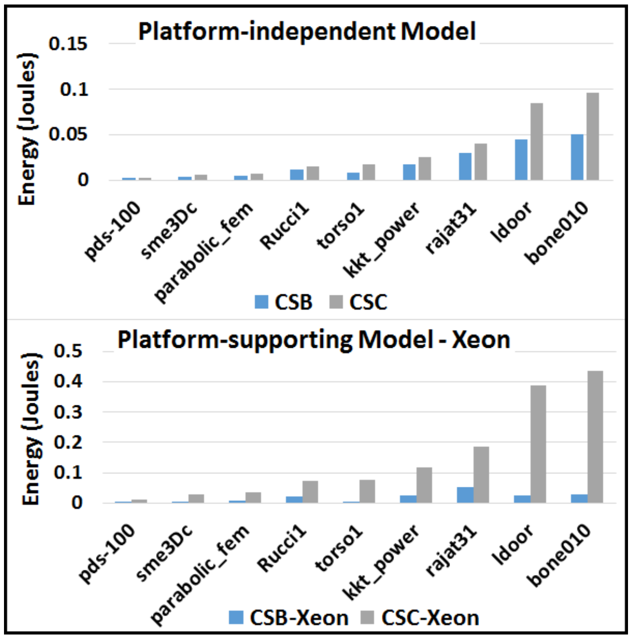


Figure 18: Comparison of platform-dependent and platform-supporting energy complexity model. Both models can predict that CSC SpMV consumes more energy than CSB SpMV.

I/O complexity and *span* complexity, the energy complexity of applications is predicted. Moreover, a case study is conducted to demonstrate how to use the model and predict energy consumption of sparse matrix vector multiplication (SpMV) on three different layouts (i.e., CSC, CSR and CSB). This prediction is validated for the two SpMV algorithms on two HPC platform with nine different input matrix types from Florida matrix collection. The results show the precise prediction on which algorithm and which platform consumes more energy. The EPEM energy complexity model gives the algorithm designers the insight to choose which design of algorithm to use for their application to minimize energy consumption.

In the future, we would extend our work in two directions:

- We want to develop a run-time framework which can choose the least energy-consumption implementations among the available kernels at run-time using the proposed energy model. Selecting the most suitable implementations helps to minimize energy consumption.
- Nowadays, there are executable frameworks that connect different platforms to one task scheduler. Selecting the most energy-efficient platforms or system configurations to run applications is one of the techniques to achieve energy optimization. In order to do so, the energy models need to be able to model the details of each platform. We want to improve and use the energy models to compare the energy consumption of applications on different platforms.

4 Energy Model for Lock-free Data-structures with Low-level of Disjoint-access Parallelism

4.1 Introduction

Lock-free programming provides highly concurrent access to data and has been increasing its footprint in industrial settings, *e.g.* Intel's Threading Building Blocks Framework [62], the Java concurrency package [1] and the Microsoft .NET Framework [2]. Providing a modeling and an analysis framework capable of describing the practical performance of lock-free algorithms is an essential, missing resource necessary to the parallel programming and algorithmic research communities in their effort to build on previous intellectual efforts. The definition of lock-freedom mainly guarantees that at least one concurrent operation on the data structure finishes in a finite number of its own steps, regardless of the state of the operations. On the individual operation level, lock-freedom cannot guarantee that an operation will not starve. The analysis frameworks that currently exist in the literature focus on such worst-case behavior and are far from capturing the behavior observed in practice.

The goal of this section is to provide a way to model and analyze the practically observed performance of lock-free data structures. In the literature, the common performance measure of a lock-free data structure is the throughput, *i.e.* the number of successful operations per unit of time. It is obtained while threads are accessing the data structure according to an access pattern that interleaves local work between calls to consecutive operations on the data structure. Although this access pattern to the data structure is significant, there is no consensus in the literature on what access to be used when comparing two data structures. So, the amount of local work could be constant ([80, 92]), uniformly distributed ([56, 30]), exponentially distributed ([102, 32]), null ([66, 76]), *etc.*, and more questionably, the average amount is rarely scanned, which leads to a partial covering of the contention domain.

We propose here a common framework enabling a fair comparison between lock-free data structures, while exhibiting the main phenomena that drive performance, and particularly the contention, which leads to different kinds of conflicts. As this is the first step in this direction, we want to deeply analyze the core of the problem, without impacting factors being diluted within a probabilistic smoothing. Therefore, we choose a constant local work, hence implying a constant access rate to the data structures. In addition to the prediction of the data structure performance, our model provides a good back-off strategy, that achieves the peak performance of a lock-free algorithm.

Two kinds of conflict appear during the execution of a lock-free algorithm, both of them leading to additional work. Hardware conflicts occur when concurrent operations call atomic primitives on the same data: these calls collide and conduct to stall time, that we name here *expansion*. Logical conflicts take place if concurrent operations overlap: because of the lock-free nature of the algorithm, several concurrent operations can run simultaneously, but only one can logically succeed. We show that the additional work produced by the failures is not necessary harmful for the system-wise performance.

We then show how throughput, that we consider here as the performance criterion, of a general class of lock-free algorithms can be computed by connecting these two key factors in an iterative way. We start by estimating the expansion probabilistically, and emulate the effect of stall time introduced by the hardware conflicts as extra work added to each thread. Then we estimate the number of failed operations, that in turn lead to additional extra work, produced by the failed retries. We continue our computation by computing again the expansion on a system setting where those two new amounts of work have been incorporated, and reiterate the process; the convergence is ensured by a fixed-point search.

We consider the class of lock-free algorithms that can be modeled as a linear composition of fixed size retry loops. This class covers numerous extensively used lock-free designs such as stacks [98] (`Pop`, `Push`), queues [80] (`Enqueue`, `Dequeue`), counters [30] (`Increment`, `Decrement`) and priority queues [76] (`DeleteMin`).

To evaluate the accuracy of our model and analysis framework, we performed experiments both on synthetic tests, that capture a wide range of possible abstract algorithmic designs, and on several reference implementations of extensively studied lock-free data structures, namely stacks, queues, counters and priority queues. Our evaluation results reveal that our model is able to capture the behavior of all the synthetic and real designs for all different numbers of threads and sizes of parallel work (consequently also contention). Our model follows the performance behavior of the data structures exactly in low contention, when our lower and upper bounds meet in one line with the observed behavior; and follows closely also the performance in high contention. We also evaluate the use of our analysis as a tool for tuning the performance of lock-free code by selecting the appropriate back-off strategy that will maximize throughput by comparing our method with against widely known back-off policies, namely linear and exponential.

The rest of this section is organized as follows. We discuss related work in Section 4.2, then the problem is formally described in Section 4.3. We consider the logical conflicts in the absence of hardware conflicts in Section 4.4, while in Section 4.5, we firstly show how to compute the expansion, then combine hardware and logical conflicts to obtain the final throughput estimate. We describe the experimental results in Section 4.6.

4.2 Related Work

Many studies have been conducted to estimate retry loop interferences and shared memory contention, which are two main components of our model. Anderson *et al.* [10] evaluate the performance of lock-free objects in real-time system by emphasizing the impact of retry loop interferences. Tasks can be preempted during the retry loop execution which can lead to interference, thus to an inflation in retry loop execution due to retries. They obtain upper bounds for the number of interferences under various priority based scheduling schemes in the uniprocessor setting for periodic real-time tasks. Also, conflicts in transactional memory and critical section contention reveal significant conceptual similarities with the retry loop interferences. Eyerman *et al.* [35] provide a probabilistic model which estimates critical section contention. Assuming total randomness in the critical section entry times, they formulate the contention in terms of the parallel, critical section granularities and probability contention for any two critical sections. In their model, Yu *et al.* [106] represent execution as a Markov chain and formulate state transition probabilities by considering arrival rate and service time for transactions together with other parameters to come up with the conflict rate.

Furthermore, performance implications of shared memory contention have been explored in various studies. Dwork *et al.* [33] provide lower bounds for the contention on shared resources for well-known problems. Intel [61] conduct an empirical study to illustrate performance and scalability of locks. It is shown that the critical section size, the time interval between releasing and re-acquiring the lock and number of threads contending the lock are vital parameters. Experiments reveal the increasing significance of contention with decreasing critical section size. Empirical analysis of atomic instructions, such as *CAS*, is done by David *et al.* [27], where latencies, which depend on the cache line state, are illustrated. Also, they experimentally compare various locks and atomic instructions under different levels of contention to highlight the impact of memory contention.

Failed retries do not only lead to useless effort but also degrade the performance of successful ones by contending the shared resources. By pointing out this fact, Alemany *et al.* [6] design non-blocking algorithms with operating system support.

Alistarh *et al.* [8] introduce a model for a class of lock-free structures, which is same as the structure targeted in our study. Their work is more oriented towards introducing a new methodology to analyze lock-free structures; in contrast our model targeting to predict throughput. They model execution as a Markov chain by considering per-process states under a stochastic scheduler. In order to relate per-process performance with system wide performance, they lift the Markov chain which composes per-process states using the assumption of process uniformity. Under stochastic scheduler assumption, they reveal the unfairness among processes in terms of number of steps taken in short intervals, which in turn leads to increase in success rate, as one process takes enough steps to complete its operation. Based on this, they provide upper bounds on number of steps, system-wise and thread-wise, to complete an operation. The difference between scheduling assumptions makes the comparison between their and our bounds not trivial, not to say incongruous.

4.3 Problem Statement

4.3.1 Running Program and Targeted Platform

We aim at evaluating the throughput of a multi-threaded algorithm that is based on the utilization of a shared lock-free data structure. Such a program can be abstracted by the Procedure AbstractAlgorithm (see Figure 19) that represents the skeleton of the function which is called by each spawned thread. It is decomposed in two main phases: the *parallel section*, represented on line 3, and the *retry loop*, from line 4 to line 7. A *retry* starts at line 5 and ends at line 7.

As for line 1, the function `Initialization` shall be seen as an abstraction of the delay between the spawns of the threads, that is expected not to be null, even when a barrier is used. We then consider that the threads begin at the exact same time, but have different initialization times.

Procedure AbstractAlgorithm

```
Initialization();
while ! done do
    Parallel_Work();
    while ! success do
        current ← Read(AP);
        new ← Critical_Work(current);
        success ← CAS(AP, current, new);
```

Figure 19: Thread procedure

🟦 parallel section
🟩 successful retry
🟥 failed retry
⬛ initialization

Figure 20: Legend of Figures 21, 22, 24, 25, 26.

The parallel section is the part of the code where the thread does not access the shared data structure; the work that is performed inside this parallel section can possibly depend on the value that has been read from the data structure, *e.g.* in the case of processing an element that has been dequeued from a FIFO (First-In-First-Out) queue.

In each retry, a thread tries to modify the data structure, and does not exit the retry loop until it has successfully modified the data structure. It does that by firstly reading the access point AP of the data structure, then according to the value that has been read, and possibly to other previous computations that occurred in the past, the thread prepares the new desired value as an access point of the data structure. Finally, it atomically tries to perform the change through a call to the *Compare-And-Swap* (*CAS*) primitive. If it succeeds, *i.e.* if the access point has not been changed by another thread between the first *Read* and the *CAS*, then it goes to the next parallel section, otherwise it repeats the process. The retry loop is composed of at least one retry, and we number the retries starting from 0, since the first iteration of the retry loop is actually not a retry, but a try.

We analyze the behavior of AbstractAlgorithm from a throughput perspective, which is defined as the number of successful data structure operations per unit of time. In the context of Procedure AbstractAlgorithm, it is equivalent to the number of successful *CAS*s.

The throughput of the lock-free algorithm, that we denote by T, is impacted by several parameters.

- *Algorithm parameters*: the amount of work inside a call to `Parallel_Work` (resp. `Critical_Work`) denoted by pw (resp. cw).

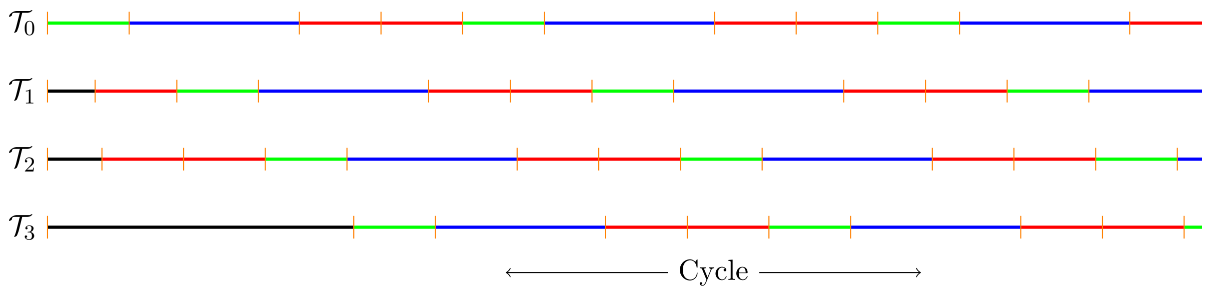


Figure 21: Execution with one wasted retry, and one inevitable failure

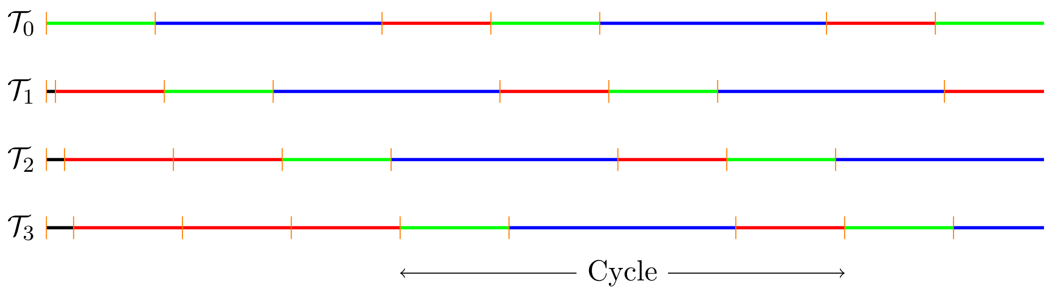


Figure 22: Execution with minimum number of failures

- *Platform parameters*: *Read* and *CAS* latencies (rc and cc respectively), and the number P of processing units (cores). We assume homogeneity for the latencies, *i.e.* every thread experiences the same latency when accessing an uncontended shared data, which is achieved in practice by pinning threads to the same socket.

4.3.2 Examples and Issues

We first present two straightforward upper bounds on the throughput, and describe the two kinds of conflict that keep the actual throughput away from those upper bounds.

4.3.2.1 Immediate Upper Bounds

Trivially, the minimum amount of work $rlw^{(-)}$ in a given retry is $rlw^{(-)} = rc + cw + cc$, as we should pay at least the memory accesses (hence *Read* latency rc and *CAS* latency cc) and the critical work cw in between.

Thread-wise: A given thread can at most perform one successful retry every $pw + rlw^{(-)}$ units of time. In the best case, P threads can then lead to a throughput of $P/(pw + rlw^{(-)})$.

System-wise: By definition, two successful retries cannot overlap, hence we have at most 1 successful retry every $rlw^{(-)}$ units of time.

Altogether, the throughput T is bounded by

$$T \leq \min\left(\frac{1}{rc + cw + cc}, \frac{P}{pw + rc + cw + cc}\right),$$

which can be rewritten as

$$T \leq \begin{cases} \frac{1}{rc+cw+cc} & \text{if } pw \leq (P-1)(rc + cw + cc) \\ \frac{P}{pw+rc+cw+cc} & \text{otherwise.} \end{cases} \tag{38}$$

4.3.2.2 Conflicts

Logical conflicts Equation 38 expresses the fact that when pw is small enough, *i.e.* when $pw \leq (P-1)rlw^{(\text{-})}$, we cannot expect that every thread performs a successful retry every $pw + rlw^{(\text{-})}$ units of time, since it is more than what the retry loop can afford. As a result, some logical conflicts, hence unsuccessful retries, will be inevitable, while the others, if any, are called *wasted*.

However, different executions can lead to different numbers of failures, which end up with different throughput values. Figures 21 and 22 depict two executions, where the black parts are the calls to `Initialization`, the blue parts are the parallel sections, and the retries can be either unsuccessful — in red — or successful — in green (the legend is displayed in Figure 20). We experiment different initialization times, and observe different synchronizations, hence different numbers of wasted retries. After the initial transient state, the execution depicted in Figure 22 comprises only the inevitable unsuccessful retries, while the execution of Figure 21 contains one wasted retry.

We can see on those two examples that a cyclic execution is reached after the transient behavior; actually, we show in Section 4.4 that, in the absence of hardware conflicts, every execution will become periodic, if the initialization times are spaced enough. In addition, we prove that the shortest period is such that, during this period, every thread succeeds exactly once. This finally leads us to define the additional failures as wasted, since we can directly link the throughput with this number of wasted retries: a higher number of wasted retries implying a lower throughput.

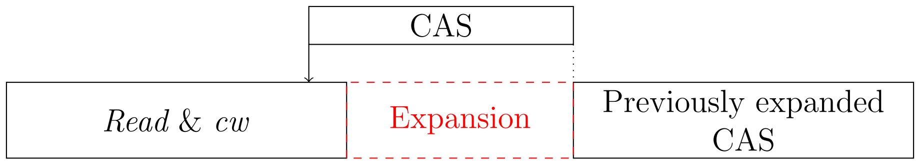


Figure 23: Expansion

Hardware conflicts The requirement of atomicity compels the ownership of the data in an exclusive manner by the executing core. This fact prohibits concurrent execution of atomic instructions if they are operating on the same data. Therefore, overlapping parts of atomic instructions are serialized by the hardware, leading to stalls in subsequently issued ones. For our target lock-free algorithm, these stalls that we refer to as expansion become an important slowdown factor in case threads interfere in the retry loop. As illustrated in Figure 23, the latency for CAS can expand and cause remarkable decreases in throughput since the CAS of a successful thread is then expanded by others; for this reason, the amount of work inside a retry is not constant, but is, generally speaking, a function depending on the number of threads that are inside the retry loop.

4.3.2.3 Process

We deal with the two kinds of conflicts separately and connect them together through the fixed-point iterative convergence.

In Section 4.5.1, we compute the expansion in execution time of a retry, noted e, by following a probabilistic approach. The estimation takes as input the expected number of threads inside the retry loop at any time, and returns the expected increase in the execution time of a retry due to the serialization of atomic primitives.

In Section 4.4, we are given a program without hardware conflict described by the size of the parallel section $pw^{(+)}$ and the size of a retry $rlw^{(+)}$. We compute upper and lower bounds on the throughput T, the number of wasted retries w, and the average number of threads inside the retry loop P_{rl}. Without loss of generality, we can normalize those execution times by the execution time of a retry, and define the parallel section size as $pw^{(+)} = q + r$, where q is a non-negative integer and r is such that $0 \leq r < 1$. This pair (together with the number of threads P) constitutes the actual input of the estimation.

Finally, we combine those two outcomes in Section 4.5.2 by emulating expansion through work not prone to hardware conflicts and obtain the full estimation of the throughput according to the model parameters that have been described in Section 4.3.1.

4.4 Execution without hardware conflict

We show in this section that, in the absence of hardware conflicts, the execution becomes periodic, which eases the calculation of the throughput. We start by defining some concepts: (f, P)-cyclic executions are a special kind of periodic executions such that within the shortest period, each thread performs exactly f unsuccessful retries and 1 successful retry. The *well-formed seed* is a set of events that allows us to detect an *(f, P)-cyclic execution* early, and the *gaps* are a measure of the quality of the synchronization between threads. The idea is to iteratively add threads into the execution and show that the periodicity is maintained. Theorem 1, on page 60, establishes a fundamental relation between gaps and well-formed seeds, while Theorem 2, on page 63, proves the periodicity, relying on the disjoint cases depicted on Figures 24, 25 and 26. We recall that the complete version of the proofs can be found in [14], together with additional Lemmas. Finally, we exhibit upper and lower bounds on throughput and number of failures, along with the average number of threads inside the retry loop.

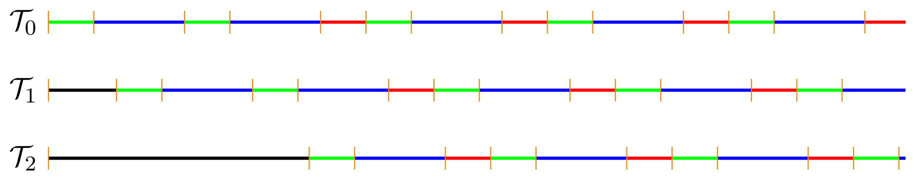


Figure 24: New thread does not lead to a reordering

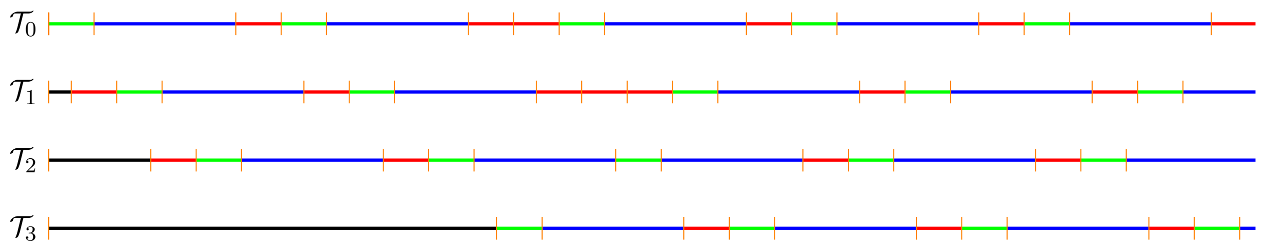


Figure 25: Reordering and immediate new seed

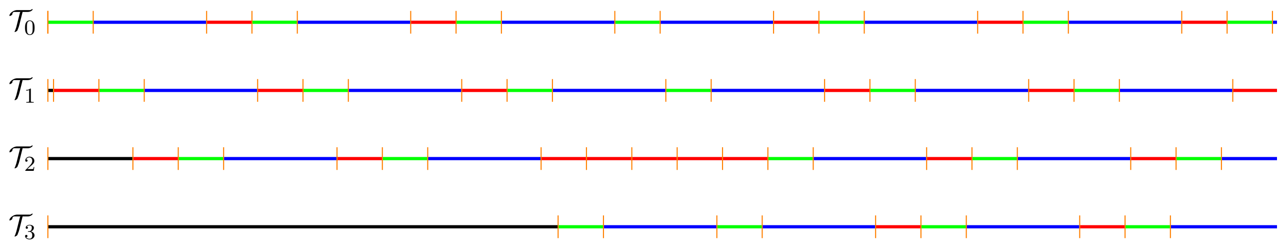


Figure 26: Reordering and transient state

4.4.1 Setting

In preamble, note that the events are strictly ordered (according to their instant of occurrence, with the thread id as a tie-breaker). As for correctness, *i.e.* to decide for the success or the failure of a retry, we need instants of occurrence for *Read* and *CAS*; we consider that the entrance (resp. exit) time of a retry is the instant of occurrence of the *Read* (resp. *CAS*).

4.4.1.1 Notations and Definitions

We recall that P threads are executing the pseudo-code described in Procedure AbstractAlgorithm, one retry is of unit-size, and the parallel section is of size $pw^{(+)} = q + r$, where q is a non-negative integer and r is such that $0 \leq r < 1$. Considering a thread $\mathcal{T}_n$ which succeeds at time S_n; this thread completes a whole retry in 1 unit of time, then executes the parallel section of size $pw^{(+)}$, and attempts to perform again the operation every unit of time, until one of the attempt is successful. We note R_n^k the k^{th} retry so that $R_n^k = S_n + 1 + q + r + k$. Also, at a given time t where not any thread is currently succeeding, the next successful attempt will be at time

$$\min_{n \in \{0,\dots,P-1\}} \{R_n^k = S_n + 1 + q + r + k > t \ ; \ S_n \text{ is the last success of } \mathcal{T}_n\},$$

and n gives the thread number of the successful thread.

Definition 1. An execution with P threads is called (C, P)-cyclic execution if and only if (i) the execution is periodic, *i.e.* at every time, every thread is in the same state as one period before, (ii) the shortest period contains exactly one successful attempt per thread, (iii) the shortest period is $1 + q + r + C$.

Definition 2. Let $\mathcal{S} = (\mathcal{T}_i, S_i)_{i \in \{0,\dots,P-1\}}$, where $\mathcal{T}_i$ are threads and S_i ordered times, *i.e.* such that $S_0 < S_1 < \cdots < S_{P-1}$. $\mathcal{S}$ is a *seed* if and only if for all $i \in \{0, \dots, P-1\}$, $\mathcal{T}_i$ does not succeed between S_0 and S_i, and starts a retry at S_i.

We define $f(\mathcal{S})$ as the smallest non-negative integer such that $S_0 + 1 + q + r + f(\mathcal{S}) > S_{P-1} + 1$, *i.e.* $f(\mathcal{S}) = \max(0, \lceil S_{P-1} - S_0 - q - r \rceil)$. When $\mathcal{S}$ is clear from the context, we denote $f(\mathcal{S})$ by f.

Definition 3. $\mathcal{S}$ is a *well-formed seed* if and only if for each $i \in \{0, \dots, P-1\}$, the execution of thread $\mathcal{T}_i$ contains the following sequence: firstly a success beginning at S_i, the parallel section, f unsuccessful retries, and finally a successful retry.

Those definitions are coupled through the two natural following properties:

Property 1. *Given a (C, P)-cyclic execution, any seed $\mathcal{S}$ including P consecutive successes is a well-formed seed, with $f(\mathcal{S}) = C$.*

Proof. Choosing any set of P consecutive successes, we are ensured, by the definition of a (f, P)-cyclic execution, that for each thread, after the first success, the next success will be obtained after f failures. The order will be preserved, and this shows that a seed including our set of successes is actually a well-formed seed. □

Property 2. *If there exists a well-formed seed in an execution, then after every thread has succeeded once, the execution coincides with an (f, P)-cyclic execution.*

Proof. By the definition of a well-formed seed, we know that the threads will first succeed in order, fails f times, and succeed again in the same order. Considering the second set of successes in a new well-formed seed, we observe that the threads will succeed a third time in the same order, after failing f times. By induction, the execution coincides with an (f, P)-cyclic execution. □

Together with the seed concept, we define the notion of *gap* that we will use extensively in the next subsection. The general idea of those gaps is that within an (f, P)-cyclic execution, the period is higher than $P \times 1$, which is the total execution time of all the successful retries within the period. The difference between the period (that lasts $1 + q + r + f$) and P, reduced by r (so that we obtain an integer), is referred as *lagging time* in the following. If the threads are numbered according to their order of success (modulo P), as the time elapsed between the successes of two given consecutive threads is constant (during the next period, this time will remain the same), this lagging time can be seen in a circular manner: the threads are represented on a circle whose length is the lagging time increased by r, and the length between two consecutive threads is the time between the end of the successful retry of the first thread and the begin of the successful retry of the second one. More formally, for all $(n, k) \in \{0, \dots, P-1\}^2$, we define the gap $G_n^{(k)}$ between $\mathcal{T}_n$ and its k^{th} predecessor based on the gap with the first predecessor:

$$\begin{cases} \forall n \in \{1, \dots, P-1\} \quad ; \quad G_n^{(1)} = S_n - S_{n-1} - 1 \\ G_0^{(1)} = S_0 + q + r + f - S_{P-1} \end{cases},$$

which leads to the definition of higher order gaps:

$$\forall n \in \{0, \dots, P-1\}; \forall k > 0; G_n^{(k)} = \sum_{j=n-k+1}^{n} G_{j \bmod P}^{(1)}.$$

For consistency, for all $n \in \{0, \dots, P-1\}$, $G_n^{(0)} = 0$.

Equally, the gaps can be obtained directly from the successes: for all $k \in \{1, \dots, P-1\}$,

$$G_n^{(k)} = \begin{cases} S_n - S_{n-k} - k & \text{if } n > k \\ S_n - S_{P+n-k} + 1 + q + r + f - k & \text{otherwise} \end{cases} \tag{39}$$

Note that, in an (f, P)-cyclic execution, the lagging time is the sum of all first order gaps, reduced by r.

4.4.2 Cyclic Executions

Theorem 1. *Given a seed $\mathcal{S} = (\mathcal{T}_i, S_i)_{i \in \{0,\dots,P-1\}}$, $\mathcal{S}$ is a well-formed seed if and only if for all $n \in \{0, \dots, P-1\}$, $0 \le G_n^{(f)} < 1$.*

Proof. Let $\mathcal{S} = (\mathcal{T}_i, S_i)_{i\in\{0,\dots,P-1\}}$ be a seed.
($\Leftarrow$) We assume that for all $n \in \{0, \dots, P-1\}$, $0 < G_n^{(f)} < 1$, and we first show that the first successes occur in the following order: $\mathcal{T}_0$ at S_0, $\mathcal{T}_1$ at S_1, ..., $\mathcal{T}_{P-1}$ at S_{P-1}, $\mathcal{T}_0$ again at R_0^f. The first threads that are successful executes their parallel section after their success, then enters their second retry loop: from this moment, they can make the first attempt of the threads, that has not been successful yet, fail. Therefore, we will look at which retry of which already successful threads could have an impact on which other threads.

We can notice that for all $n \in \{0, \dots, P-1\}$, if the first success of $\mathcal{T}_n$ occurs at S_n, then its next attempts will potentially occur at $R_n^k = S_n + 1 + q + r + k$, where $k \geq 0$. More specifically, thanks to Equation 39, for all $n \leq f$, $R_n^k = S_{P+n-f} + G_n^{(f)} + k$. Also, for all $k \leq f - n$,

$$\begin{aligned} R_n^k - S_{P+n-f+k} &= -\left(S_{P+n-f+k} - S_{P+n-f} - k\right) + G_n^{(f)} \\ &= G_n^{(f)} - G_{P+n-f+k}^{(k)} \\ R_n^k - S_{P+n-f+k} &= G_n^{(f-k)}, \end{aligned} \tag{40}$$

and this implies that if $k > 0$,

$$S_{P+n-f+k} - R_n^{k-1} = 1 - G_n^{(f-k)}. \tag{41}$$

We know, by hypothesis, that $0 < G_n^{(f-k)} < 1$, equivalently $0 < 1 - G_n^{(f-k)} < 1$. Therefore Equation 40 states that if a thread $\mathcal{T}_{n'}$ starts a successful attempt at $S_{P+n-f+k}$, then this thread will make the k^{th} retry of $\mathcal{T}_n$ fail, since $\mathcal{T}_n$ enters a retry while $\mathcal{T}_{n'}$ is in a successful retry. And Equation 41 shows that, given a thread $\mathcal{T}_{n'}$ starting a new retry at $S_{P+n-f+k}$, the only retry of $\mathcal{T}_n$ that can make $\mathcal{T}_{n'}$ fail on its attempt is the $(k-1)^{\text{th}}$ one. There is indeed only one retry of $\mathcal{T}_n$ that can enter a retry before the entrance of $\mathcal{T}_{n'}$, and exit the retry after it.

$\mathcal{T}_0$ is the first thread to succeed at S_0, because no other thread is in the retry loop at this time. Its next attempt will occur at R_0^0, and all thread attempts that start before S_{P-f} (included) cannot fail because of $\mathcal{T}_0$, since it runs then the parallel section. Also, since all gaps are positive, the threads $\mathcal{T}_1$ to $\mathcal{T}_{P-f}$ will succeed in this order, respectively starting at times S_1 to S_{P-f}.

Then, using induction, we can show that $\mathcal{T}_{P-f+1}$, ..., $\mathcal{T}_{P-1}$ succeed in this order, respectively starting at times S_{P-f+1}, ..., S_{P-1}. For $j \in \{0, \dots, f-1\}$, let $(\mathcal{P}_j)$ be the following property: for all $n \in \{0, \dots, P-f+j\}$, $\mathcal{T}_n$ starts a successful retry at S_n. We assume that for a given j, $(\mathcal{P}_j)$ is true, and we show that it implies that $\mathcal{T}_{P-f+j+1}$ will succeed at $S_{P-f+j+1}$. The successful attempt of $\mathcal{T}_{P-f+j}$ at S_{P-f+j} leads, for all $j' \in \{0, \dots, j\}$, to the failure of the j'^{th} retry of $\mathcal{T}_{j-j'}$ (explanation of Equation 40). But for each $\mathcal{T}_{j'}$, this attempt was precisely the one that could have made $\mathcal{T}_{P-f+j+1}$ fail on its attempt at $S_{P-f+j+1}$ (explanation of Equation 40). Given that all threads $\mathcal{T}_n$, where $n > P - f + j + 1$, do not start any retry loop before $S_{P-f+j+1}$, $\mathcal{T}_{P-f+j+1}$ will succeed at $S_{P-f+j+1}$. By induction, $(\mathcal{P}_j)$ is true for all $j \in \{0, \dots, f-1\}$.

Finally, when $\mathcal{T}_{P-1}$ succeeds, it makes the $(f-1-n)^{\text{th}}$ retry of $\mathcal{T}_n$ fail, for all $n \in \{0, \dots, f-1\}$; also the next potentially successful attempt for $\mathcal{T}_n$ is at R_n^{f-n}. (Naturally, for all $n \in \{f, \dots, P-1\}$, the next potentially successful attempt for $\mathcal{T}_n$ is at R_n^0.)

We can observe that for all $n < P$, $j \in \{0, \dots, P-1-n\}$, and all $k \geq j$,

$$\begin{aligned} R_{n+j}^{k-j} - R_n^k &= S_{n+j} + k - j - (S_n + k) \\ R_{n+j}^{k-j} - R_n^k &= G_{n+j}^{(j)}, \end{aligned} \tag{42}$$

hence for all $n \in \{1, \dots, f\}$, $R_n^{f-n} - R_0^f = G_n^{(n)} > 0$.

As we have as well, for all $n \in \{f+1, \dots, P-1\}$, $R_n^0 > R_f^0$, we obtain that among all the threads, the earliest possibly successful attempt is R_0^f. Following $\mathcal{T}_{P-1}$, $\mathcal{T}_0$ is consequently the next successful thread in its f^{th} retry.

To conclude this part, we can renumber the threads ($\mathcal{T}_{n+1}$ becoming now $\mathcal{T}_n$ if $n > 0$, and $\mathcal{T}_0$ becoming $\mathcal{T}_{P-1}$), and follow the same line of reasoning. The only difference is the fact that $\mathcal{T}_{P-1}$ (according to the new numbering) enters the retry loop f units of time before S_{P-1}, but it does not interfere with the other threads, since we know that those attempts will fail.

There remains the case where there exists $n \in \{0, \dots, P-1\}$ such that $G_n^{(f)} = 0$. This implies that $f = 0$, thus we have a well-formed seed.

($\Rightarrow$) We prove now the implication by contraposition; we assume that there exists $n \in \{0, \dots, P-1\}$ such that $G_n^{(f)} > 1$ or $G_n^{(f)} < 0$, and show that $\mathcal{S}$ is not a well-formed seed.

We assume first that an f^{th} order gap is negative. As it is a sum of 1st order gaps, then there exists n' such that $G_{n'}^{(1)}$ is negative; let n'' be the highest one.

If $n'' > 0$, then either the threads $\mathcal{T}_0, \dots, \mathcal{T}_{n''-1}$ succeeded in order at their 0th retry, and then $\mathcal{T}_{n''-1}$ makes $\mathcal{T}_{n''}$ fail at its 0th retry (we have a seed, hence by definition, $S_{n''-1} < S_{n''}$, and $G_{n''}^{(1)} < 0$, thus $S_{n''-1} < S_{n''} < S_{n''-1} + 1$), or they did not succeed in order at their first try. In both cases, $\mathcal{S}$ is not a well-formed seed.

If $n'' = 0$, let us assume that $\mathcal{S}$ is a well-formed seed. Let also a new seed be $\mathcal{S}' = (\mathcal{T}_i, S'_i)_{i \in \{0, \dots, P-1\}}$, where for all $n \in \{0, \dots, P-2\}$, $S'_{n+1} = S_n$, and $S'_0 = S_{P-1} - (q+1+f+r)$. Like $\mathcal{S}$, $\mathcal{S}'$ is a well-formed seed; however, $G_1^{(1)}$ is negative, and we fall back into the previous case, which shows that $\mathcal{S}'$ is not a well-formed seed. This is absurd, hence $\mathcal{S}$ is not a well-formed seed.

We assume now that every gap is positive and choose n_0 defined by: $n_0 = \min\{n \ ; \ \exists k \in \{0, \dots, P-1\} / G_{n+k}^{(k)} > 1\}$, and $f_0 = \min\{k \ ; \ G_{n_0+k}^{(k)} > 1\}$: among the gaps that exceed 1, we pick those that concern the earliest thread, and among them the one with the lowest order.

Let us assume that threads $\mathcal{T}_0$, ..., $\mathcal{T}_{P-1}$ succeed at their 0th retry in this order, then $\mathcal{T}_0$, ..., $\mathcal{T}_{n_0}$ complete their second successful retry loop at their f^{th} retry, in this order. If this is not the case, then $\mathcal{S}$ is not a well-formed seed, and the proof is completed. According to Equation 42, we have, on the one hand, $R_{n_0+1}^{f_0-1} - R_{n_0}^{f_0} = G_{n_0+1}^{(1)}$, which implies $R_{n_0+1}^{f_0} - 1 - R_{n_0}^{f_0} = G_{n_0+1}^{(1)}$, thus $R_{n_0+1}^{f} - (R_{n_0}^{f} + 1) = G_{n_0+1}^{(1)}$; and on the other hand, $R_{n_0+f_0}^{0} - R_{n_0}^{f_0} = G_{n_0+f_0}^{(f_0)}$

implying $R^{f-f_0}_{n_0+f_0} - \left(R^f_{n_0} + 1\right) = G^{(f_0)}_{n_0+f_0} - 1$. As we know that $G^{(f_0)}_{n_0+f_0} - G^{(1)}_{n_0+1} = G^{(f_0-1)}_{n_0+f_0} < 1$ by definition of f_0 (and n_0), we can derive that $R^f_{n_0+1} - (R^f_{n_0} + 1) > R^{f-f_0}_{n_0+f_0} - (R^f_{n_0} + 1)$. We have assumed that $\mathcal{T}_{n_0}$ succeeds at its f^{th} retry, which will end at $R^f_{n_0} + 1$. The previous inequality states then that $\mathcal{T}_{n_0+1}$ cannot be successful at its f^{th} retry, since either a thread succeeds before $\mathcal{T}_{n_0+f_0}$ and makes both $\mathcal{T}_{n_0+f_0}$ and $\mathcal{T}_{n_0+1}$ fail, or $\mathcal{T}_{n_0+f_0}$ succeeds and makes $\mathcal{T}_{n_0+1}$ fail. We have shown that $\mathcal{S}$ is not a well-formed seed.

□

Theorem 2. *Assuming $r \neq 0$, if a new thread is added to an $(f, P-1)$-cyclic execution, then all the threads will eventually form either an (f, P)-cyclic execution, or an $(f+1, P)$-cyclic execution.*

Proof. We only give the sketch of the proof, that decomposes the Theorem into three Lemmas which we describe here graphically:

- If all gaps of $(f+1)^{\text{th}}$ order are less than 1, then every existing thread will fail once more, and the new steady-state is reached immediately. See Figure 24.
- Otherwise
  - Either: everyone succeeds once, whereupon a new (f, P)-cyclic execution is formed. See Figure 25.
  - Or: before everyone succeeds again, a new (f, P')-cyclic execution, where $P' \leq P$, is formed, which finally leads to an (f, P)-cyclic execution. See Figure 26. □

4.4.3 Throughput Bounds

Firstly we calculate the expression of throughput and the expected number of threads inside the retry loop (that is needed when we gather expansion and wasted retries). Then we exhibit upper and lower bounds on both throughput and the number of failures, and show that those bounds are reached. Finally, we give the worst case on the number of wasted retries.

Lemma 2. *In an (f, P)-cyclic execution, the throughput is*

$$T = \frac{P}{q + r + 1 + f}. \tag{43}$$

Proof. By definition, the execution is periodic, and the period lasts $q + r + 1 + f$ units of time. As P successes occur during this period, we end up with the claimed expression. □

Lemma 3. *In an (f, P)-cyclic execution, the average number of threads P_{rl} in the retry loop is given by*

$$P_{rl} = P \times \frac{f+1}{q + r + f + 1}.$$

Proof. Within a period, each thread spends $f+1$ units of time in the retry loop, among the $q+r+f+1$ units of time of the period, hence the Lemma. □

Lemma 4. *The number of failures is not less than $f^{(-)}$, where*

$$f^{(-)} = \begin{cases} P-q-1 & \textit{if } q \le P-1 \\ 0 & \textit{otherwise} \end{cases}, \textit{ and accordingly,} \qquad T \le \begin{cases} \frac{P}{P+r} & \textit{if } q \le P-1 \\ \frac{P}{q+r+1} & \textit{otherwise.} \end{cases} \tag{44}$$

Proof. According to Equation 43, the throughput is maximized when the number of failures is minimized. In addition, we have two lower bounds on the number of failures: (i) $f \ge 0$, and (ii) P successes should fit within a period, hence $q+1+f \ge P$. Therefore, if $P-1-q<0$, $T \le P/(q+r+1+0)$, otherwise,

$$T \le \frac{P}{q+r+1+P-1-q} = \frac{P}{P+r}.$$

□

Remark 1. We notice that if $q > P-1$, the upper bound in Equation 44 is actually the same as the immediate upper bound described in Section 4.3.2.1. However, if $q \le P-1$, Equation 44 refines the immediate upper bound.

Lemma 5. *The number of failures is bounded by*

$$f \le f^{(+)} = \left\lfloor \frac{1}{2}\left((P-1-q-r) + \sqrt{(P-1-q-r)^2+4P} \right) \right\rfloor, \textit{ and accordingly,}$$

the throughput is bounded by

$$T \ge \frac{P}{q+r+1+f^{(+)}}.$$

Proof. We show that a necessary condition so that an (f,P)-cyclic execution, whose lagging time is ℓ, exists, is $f \times (\ell + r) < P$. According to Properties 1 and 2, any set of P consecutive successes is a well-formed seed with P threads. Let $\mathcal{S}$ be any of them. As we have f failures before success, Theorem 1 ensures that for all $n \in \{0,\dots,P-1\}$, $G_n^{(f)} < 1$. We recall that for all $n \in \{0,\dots,P-1\}$, we also have $G_n^{(P)} = \ell + r$.

On the one hand, we have

$$\begin{aligned} \sum_{n=0}^{P-1} G_n^{(f)} &= \sum_{n=0}^{P-1} \sum_{j=n-f+1}^{n} G_{j \bmod P}^{(1)} \\ &= f \times \sum_{n=0}^{P-1} G_n^{(1)} \\ \sum_{n=0}^{P-1} G_n^{(f)} &= f \times (\ell + r). \end{aligned}$$

On the other hand, $\sum_{n=0}^{P-1} G_n^{(f)} < \sum_{n=0}^{P-1} 1 = P$.

Altogether, the necessary condition states that $f \times (\ell + r) < P$, which can be rewritten as $f \times (q + 1 + f - P + r) < P$. The proof is complete since minimizing the throughput is equivalent to maximizing the number of failures. $\square$

Lemma 6. *For each of the bounds defined in Lemmas 4 and 5, there exists an (f, P)-cyclic execution that reaches the bound.*

Proof. According to Lemmas 4 and 5, if an (f, P)-cyclic execution exists, then the number of failures is such that $f^{(-)} \leq f \leq f^{(+)}$.
We show now that this double necessary condition is also sufficient. We consider f such that $f^{(-)} \leq f \leq f^{(+)}$, and build a well-formed seed $\mathcal{S} = (\mathcal{T}_i, S_i)_{i \in \{0,\dots,P-1\}}$.

For all $n \in \{0, \dots, P-1\}$, we define S_i as

$$S_n = n \times \left(\frac{q + 1 + f - P + r}{P} + 1 \right).$$

We first show that $f(\mathcal{S}) = f$. By definition, $f(\mathcal{S}) = \max(0, \lceil S_{P-1} - S_0 - q - r \rceil)$; we have then

$$\begin{aligned} f(\mathcal{S}) &= \max\left(0, \left\lceil (P-1) \times \left(\frac{q+1+f-P+r}{P} + 1 \right) - q - r \right\rceil \right) \\ &= \max\left(0, \left\lceil (P-1-q-r) + (q+1+f-P+r) - \frac{q+1+f-P+r}{P} \right\rceil \right) \\ f(\mathcal{S}) &= \max\left(0, \left\lceil f - \frac{q+1+f-P+r}{P} \right\rceil \right). \end{aligned}$$

Firstly, we know that $q + 1 + f - P \geq 0$, thus if $f = 0$, then the second term of the maximum is not positive, and $f(\mathcal{S}) = 0 = f$. Secondly, if $f > 0$, then according to Lemma 4, $(q + 1 + f - P + r)/P < 1/f \leq 1$. As we also have $(q + 1 + f - P + r)/P \geq 0$, we conclude that $f(\mathcal{S}) = \lceil f - \frac{q+1+f-P+r}{P} \rceil = f$.

Additionally, for all $n \in \{0, \dots, P-1\}$,

$$\begin{aligned} G_n^{(f)} &= \begin{cases} S_n - S_{n-f} - f & \text{if } n > f \\ S_n - S_{P+n-f} + 1 + q + r & \text{otherwise} \end{cases} \\ &= \begin{cases} n \times \left(\frac{q+1+f-P+r}{P} + 1\right) - (n-f) \times \left(\frac{q+1+f-P+r}{P} + 1\right) - f \\ n \times \left(\frac{q+1+f-P+r}{P} + 1\right) - (P+n-f) \times \left(\frac{q+1+f-P+r}{P} + 1\right) + 1 + q + r \end{cases} \\ &= \begin{cases} f \times \frac{q+1+f-P+r}{P} \\ -(P-f) - (q+1+f-P+r) + f \times \frac{q+1+f-P+r}{P} + 1 + q + r \end{cases} \\ G_n^{(f)} &= f \times \frac{w+r}{P} \end{aligned}$$

As $w \leq 0$ and $f \leq 0$, $G_n^{(f)} > 0$. Since $f \leq f^{(+)}$, $G_n^{(f)} < 1$. Theorem 1 implies that $\mathcal{S}$ is a well-formed seed that leads to an (f, P)-cyclic execution.

We have shown that for all f such that $f^{(-)} \leq f \leq f^{(+)}$ there exists an (f, P)-cyclic execution; in particular there exist an $(f^{(+)}, P)$-cyclic execution and an $(f^{(-)}, P)$-cyclic execution. □

Corollary 1. *The highest possible number of wasted repetitions is* $\left\lceil \sqrt{P} - 1 \right\rceil$ *and is achieved when* $P = q + 1$.

Proof. The highest possible number of wasted repetitions $\tilde{w}(P)$ with P threads is given by

$$\tilde{w}(P) = f^{(+)} - f^{(-)} = \left\lfloor \frac{1}{2}\left(-a(P) + \sqrt{a(P)^2 + 4P}\right) - f^{(-)} \right\rfloor.$$

Let a and h be the functions respectively defined as $a(P) = q + 1 - P + r$, which implies $a'(P) = -1$, and $h(P) = (-a(P) + \sqrt{a(P)^2 + 4P})/2 - f^{(-)}$, so that $\tilde{w}(P) = \lfloor h(P) \rfloor$.

Let us first assume that $a(P) > 0$. In this case, $q \leq P - 1$, hence $f^{(-)} = 0$. We have

$$2h'(P) = 1 + \frac{-2a(P) + 4}{2\sqrt{a(P)^2 + 4P}}$$
$$2h'(P) = 2 \times \frac{2 - a(P) + \sqrt{a(P)^2 + 4P}}{2\sqrt{a(P)^2 + 4P}}$$

Therefore, $h'(P)$ is negative if and only if $\sqrt{a(P)^2 + 4P} < a(P) - 2$. It cannot be true if $a(P) < 2$. If $a(P) \geq 2$, then the previous inequality is equivalent to $a(P)^2 + 4P < a(P)^2 - 4a(P) + 4$, which can be rewritten in $q + 1 + r < 1$, which is absurd. We have shown that h is increasing in $]0, q + 1]$.

Let us now assume that $a(P) \leq 0$. In this case, $q > P - 1$, hence $f^{(-)} = P - q - 1$, and $h'(P) = \left(a(P) + \sqrt{a(P)^2 + 4P}\right)/2 - r$. Assuming $h'(P)$ to be positive leads to the same absurd inequality $q + 1 + r < 1$, which proves that h is decreasing on $[q + 2, +\infty[$.

Also, the maximum number of wasted repetitions is achieved as $P = q + 1$ or $P = q + 2$. Since

$$h(q+1) = \frac{1}{2}\left(-r + \sqrt{r^2 + 4P}\right) > \frac{1}{2}\left(-(r+1) + \sqrt{r^2 + 4P}\right) = h(q+2),$$

the maximum number of wasted repetitions is $\tilde{w}(q + 1)$. In addition,

$$\begin{aligned}
\frac{1}{2}\left(-r + \sqrt{4P}\right) &< h(q+1) < \frac{1}{2}\left(-r + \sqrt{r^2} + \sqrt{4P}\right) \\
\sqrt{P} - \frac{r}{2} &< h(q+1) < \sqrt{P} \\
\sqrt{P} - 1 &\leq h(q+1) < \sqrt{P}
\end{aligned}$$

We conclude that the maximum number of wasted repetitions is $\left\lceil \sqrt{P} - 1 \right\rceil$. □

4.5 Expansion and Complete Throughput Estimation

4.5.1 Expansion

Interference of threads does not only lead to logical conflicts but also to hardware conflicts which impact the performance significantly.

We model the behavior of the cache coherency protocols which determine the interaction of overlapping *Read*s and *CAS*s. By taking MESIF [42] as basis, we come up with the following assumptions. When executing an atomic *CAS*, the core gets the cache line in exclusive state and does not forward it to any other requesting core until the instruction is retired. Therefore, requests stall for the release of the cache line which implies serialization. On the other hand, ongoing *Read*s can overlap with other operations. As a result, a *CAS* introduces expansion only to overlapping *Read* and *CAS* operations that start after it, as illustrated in Figure 23. As a remark, we ignore memory bandwidth issues which are negligible for our study.

Furthermore, we assume that *Read*s that are executed just after a *CAS* do not lead to expansion (as the thread already owns of the data), which takes effect at the beginning of a retry following a failing attempt. Thus, read expansions need only to be considered before the 0th retry. In this sense, read expansion can be moved to parallel section and calculated in the same way as *CAS* expansion is calculated.

To estimate expansion, we consider the delay that a thread can introduce, provided that there is already a given number of threads in the retry loop. The starting point of each *CAS* is a random variable which is distributed uniformly within an expanded retry. The cost function d provides the amount of delay that the additional thread introduces, depending on the point where the starting point of its *CAS* hits. By using this cost function we can formulate the expansion increase that each new thread introduces and derive the differential equation below to calculate the expansion of a *CAS*.

Lemma 7. *The expansion of a CAS operation is the solution of the following system of equations:*

$$\begin{cases} e'\left(P_{rl}\right) & = & cc \times \dfrac{\frac{cc}{2} + e\left(P_{rl}\right)}{rc + cw + cc + e\left(P_{rl}\right)} \\ e\left(P_{rl}^{(0)}\right) & = & 0 \end{cases}, \textit{ where } P_{rl}^{(0)} \textit{ is the point where expansion begins.}$$

Proof. We compute $e\left(P_{rl} + h\right)$, where $h \leq 1$, by assuming that there are already P_{rl} threads in the retry loop, and that a new thread attempts to *CAS* during the retry, within a probability h.

$$e\left(P_{rl}+h\right)=e\left(P_{rl}\right)+h\times\int_{0}^{rlw^{(+)}}\frac{d\left(t\right)}{rlw^{(+)}}\,dt$$
$$=e\left(P_{rl}\right)+\Big(\int_{0}^{rc+cw-cc}\frac{d\left(t\right)}{rlw^{(+)}}\,dt+\int_{rc+cw-cc}^{rc+cw}\frac{d\left(t\right)}{rlw^{(+)}}\,dt+\int_{rc+cw}^{rc+cw+e(P_{rl})}\frac{d\left(t\right)}{rlw^{(+)}}\,dt+\int_{rc+cw+e(P_{rl})}^{rlw^{(+)}}\frac{d\left(t\right)}{rlw^{(+)}}\,dt\Big)h$$
$$=e\left(P_{rl}\right)+\Big(\int_{rc+cw-cc}^{rc+cw}\frac{t}{rlw^{(+)}}\,dt+\int_{rc+cw}^{rc+cw+e(P_{rl})}\frac{cc}{rlw^{(+)}}\,dt\Big)h=e\left(P_{rl}\right)+h\times\frac{\frac{cc^2}{2}+e\left(P_{rl}\right)\times cc}{rlw^{(+)}}.$$

This leads to $\dfrac{e\left(P_{rl}+h\right)-e\left(P_{rl}\right)}{h}=\dfrac{\frac{cc^2}{2}+e\left(P_{rl}\right)\times cc}{rlw^{(+)}}$. When making h tend to 0, we finally obtain

$$e'\left(P_{rl}\right)=cc\times\frac{\frac{cc}{2}+e\left(P_{rl}\right)}{rc+cw+cc+e\left(P_{rl}\right)}.$$ □

4.5.2 Throughput Estimate

There remains to combine hardware and logical conflicts in order to obtain the final upper and lower bounds on throughput. We are given as an input an expected number of threads P_{rl} inside the retry loop. We firstly compute the expansion accordingly, by solving numerically the differential equation of Lemma 7. As explained in the previous subsection, we have $pw^{(+)}=pw+e$, and $rlw^{(+)}=rc+cw+e+cc$. We can then compute q and r, that are the inputs (together with the total number of threads P) of the method described in Section 4.4. Assuming that the initialization times of the threads are spaced enough, the execution will superimpose an (f,P)-cyclic execution. Thanks to Lemma 3, we can compute the average number of threads inside the retry loop, that we note by $h_f(P_{rl})$. A posteriori, the solution is consistent if this average number of threads inside the retry loop $h_f(P_{rl})$ is equal to the expected number of threads P_{rl} that has been given as an input.

Several (f,P)-cyclic executions belong to the domain of the possible outcomes, but we are interested in upper and lower bounds on the number of failures f. We can compute them through Lemmas 4 and 5, along with their corresponding throughput and average number of threads inside the retry loop. We note by $h^{(+)}(P_{rl})$ and $h^{(-)}(P_{rl})$ the average number of threads for the lowest number of failures and highest one, respectively. Our aim is finally to find $P_{rl}^{(-)}$ and $P_{rl}^{(+)}$, such that $h^{(+)}(P_{rl}^{(+)})=P_{rl}^{(+)}$ and $h^{(-)}(P_{rl}^{(-)})=P_{rl}^{(-)}$. If several solutions exist, then we want to keep the smallest, since the retry loop stops to expand when a stable state is reached.

Note that we also need to provide the point where the expansion begins. It begins when we start to have failures, while reducing the parallel section. Thus this point is $(2P-1)rlw^{(-)}$ (resp. $(P-1)rlw^{(-)}$) for the lower (resp. upper) bound on the throughput.

Theorem 3. *Let (x_n) be the sequence defined recursively by $x_0=0$ and $x_{n+1}=h^{(+)}(x_n)$. If $pw\geq rc+cw+cc$, then $P_{rl}^{(+)}=\lim_{n\to+\infty}x_n$.*

Proof. First of all, the average number of threads belongs to $]0, P[$, thus for all $x \in [0, P]$, $0 < h^{(+)}(x) < P$. In particular, we have $h^{(+)}(0) > 0$, and $h^{(+)}(P) < P$, which proves that there exist one fixed point for $h^{(+)}$.

In addition, we show that $h^{(+)}$ is a non-decreasing function. According to Lemma 3,

$$h^{(+)}(P_{rl}) = P \times \frac{1 + f^{(-)}}{q + r + f^{(-)} + 1},$$

where all variables except P depend actually on P_{rl}. We have

$$q = \left\lfloor \frac{pw + e}{rlw^{(-)} + e} \right\rfloor \text{ and } r = \frac{pw + e}{rlw^{(-)} + e} - q,$$

hence, if $pw \geq rlw^{(-)}$, q and r are non-increasing as e is non-decreasing, which is non-decreasing with P_{rl}. Since $f^{(-)}$ is non-decreasing as a function of q, we have shown that if $pw \geq rlw^{(-)}$, $h^{(+)}$ is a non-decreasing function.

Finally, the proof is completed by the theorem of Knaster-Tarski. □

The same line of reasoning holds for $h^{(-)}$ as well. As a remark, we point out that when $pw < rlw^{(-)}$, we scan the interval of solution, and have no guarantees about the fact that the solution is the smallest one; still it corresponds to very extreme cases.

4.5.3 Several Retry Loops

We consider here a lock-free algorithm that, instead of being a loop over one parallel section and one retry loop, is composed of a loop over a sequence of alternating parallel sections and retry loops. We show that this algorithm is equivalent to an algorithm with only one parallel section and one retry loop, by proving the intuition that the longest retry loop is the only one that fails and hence expands.

4.5.3.1 Problem Formulation

In this subsection, we consider an execution such that each spawned thread runs Procedure Combined in Figure 27. Each thread executes a linear combination of S independent retry loops, *i.e.* operating on separate variables, interleaved with parallel sections. We note now as $rlw_i^{(+)}$ and $pw_i^{(+)}$ the size of a retry of the i^{th} retry loop and the size of the i^{th} parallel section, respectively, for each $i \in \{1, \ldots, S\}$. As previously, q_i and r_i are defined such that $pw_i^{(+)} = (q_i + r_i) \times rlw_i^{(+)}$, where q_i is a non-negative integer and r_i is smaller than 1.

The Procedure Combined executes the retry loops and parallel sections in a cyclic fashion, so we can normalize the writing of this procedure by assuming that a retry of the 1st retry loop is the longest one. More precisely, we consider the initial algorithm, and we define i_0 as

$$i_0 = \min \operatorname{argmax}_{i \in \{1, \ldots, S\}} rlw_i^{(+)}.$$

We then renumber the retry loops such that the new ordering is $i_0, \dots, S, 1, \dots, i_0 - 1$, and we add in `Initialization` the first parallel sections and retry loops on access points from 1 to i_0 — according to the initial ordering.

One success at the system level is defined as one success of the last CAS, and the throughput is defined accordingly. We note that in steady-state, all retry loops have the same throughput, so the throughput can be computed from the throughput of the 1st retry loop instead.

4.5.3.2 Wasted Retries

Lemma 8. *Unsuccessful retry loops can only occur in the* 1st *retry loop.*

Proof. We note $(t_n)_{n\in[1,+\infty[}$ the sequence of the thread numbers that succeeds in the 1st retry loop, and $(s_n)_{n\in[1,+\infty[}$ the sequence of the corresponding time where they exit the retry loop. We notice that by construction, for all $n \in [1, +\infty[$, $s_n < s_{n+1}$. Let, for $i \in \{2, \dots, S\}$ and $n \in [1, +\infty[$, $(\mathcal{P}_{i,n})$ be the following property: for all $i' \in \{2, \dots, i\}$, and for all $n' \in \{1, \dots, n\}$, the thread $\mathcal{T}_{t_{n'}}$ succeeds in the i^{th} retry loop at its first attempt.

We assume that for a given (i, n), $(\mathcal{P}_{i+1,n})$ and $(\mathcal{P}_{i,n+1})$ is true, and show that $(\mathcal{P}_{i+1,n+1})$ is true. As the threads $\mathcal{T}_{t_n}$ and $\mathcal{T}_{t_{n+1}}$ do not have any failure in the first i retry loops, their entrance time in the $i+1^{\text{th}}$ retry loop is given by

$$s_n + \sum_{i'=1}^{i}(rlw_{i'}^{(+)} + pw_{i'}^{(+)}) + pw_{i+1}^{(+)} = X_1 \text{ and } s_{n+1} + \sum_{i'=1}^{i}(rlw_{i'}^{(+)} + pw_{i'}^{(+)}) + pw_{i+1}^{(+)} = X_2,$$

respectively. Thread $\mathcal{T}_{t_n}$ does not fail in the $i+1^{\text{th}}$ retry loop, hence exits at

$$X_1 + rlw_{i+1}^{(+)} < X_1 + rlw_1^{(+)} = s_n + X_2 - s_{n+1} < X_2.$$

As the previous threads $\mathcal{T}_{n-1}, \dots, \mathcal{T}_1$ exits the i^{th} retry loop before $\mathcal{T}_n$, and next threads $\mathcal{T}_{n'}$, where $n' > n+1$, enters this retry loop after $\mathcal{T}_{n+1}$, this implies that the thread $\mathcal{T}_{t_{n+1}}$ succeeds in the $i+1^{\text{th}}$ retry loop at its first attempt, and $(\mathcal{P}_{i+1,n+1})$ is true.

Procedure Combined

```
Initialization();
while not(done) do
    for i ← 1 to S do
        Parallel_Work(i);
        while not(success) do
            current ← Read(AP[i]);
            new ← Critical_Work(i,current);
            success ← CAS(AP, current, new);
```

Figure 27: Thread procedure with several retry loops

Regarding the first thread that succeeds in the first retry loop, we know that he successes in any retry loop since there is no other thread to compete with. Therefore, for all $i \in \{2, \ldots, S\}$, $(\mathcal{P}_{i,1})$ is true. Then we show by induction that all $(\mathcal{P}_{2,n})$ is true, then all $(\mathcal{P}_{3,n})$, *etc.*, until all $(\mathcal{P}_{S,n})$, which concludes the proof. □

Theorem 4. *The multi-retry loop Procedure Combined is equivalent to the Procedure AbstractAlgorithm, where*

$$pw^{(+)} = pw_1^{(+)} + \sum_{i=2}^{S} \left(pw_i^{(+)} + rlw_i^{(+)} \right) \quad \text{and} \quad rlw^{(+)} = rlw_1^{(+)}.$$

Proof. According to Lemma 8 there is no failure in other retry loop than the first one; therefore, all retry loops have a constant duration, and can thus be considered as parallel sections. □

4.5.3.3 Expansion

The expansion in the retry loop starts as threads fail inside this retry loop. When threads are launched, there is no expansion, and Lemma 8 implies that if threads fail, it should be inside the first retry loop, because it is the longest one. As a result, there will be some stall time in the memory accesses of this first retry loop, *i.e.* expansion, and it will get even longer. Failures will thus still occur in the first retry loop: there is a positive feedback on the expansion of the first retry loop that keeps this first retry loop as the longest one among all retry loops. Therefore, in accordance to Theorem 4, we can compute the expansion by considering the equivalent single-retry loop procedure described in the theorem.

4.6 Experimental Evaluation

We validate our model and analysis framework through a set of successive steps, from synthetic tests, capturing a wide range of possible abstract algorithmic designs, to several reference implementations of extensively studied lock-free data structure designs that include cases with non constant parallel section and retry loop.

4.6.1 Setting

4.6.1.1 Single retry loop

We have conducted experiments on an Intel ccNUMA workstation system. The system is composed of two sockets, that is equipped with Intel Xeon E5-2687W v2 CPUs with frequency band 1.2-3.4 GHz. The physical cores have private L1, L2 caches and they share an L3 cache, which is 25 MB. In a socket, the ring interconnect provides L3 cache accesses and core-to-core communication. Due to the bi-directionality of the ring interconnect, uncontended latencies for intra-socket communication between cores do not show significant variability.

Our model assumes uniformity in the *CAS* and *Read* latencies on the shared cache line. Thus, threads are pinned to a single socket to minimize non-uniformity in *Read* and *CAS*

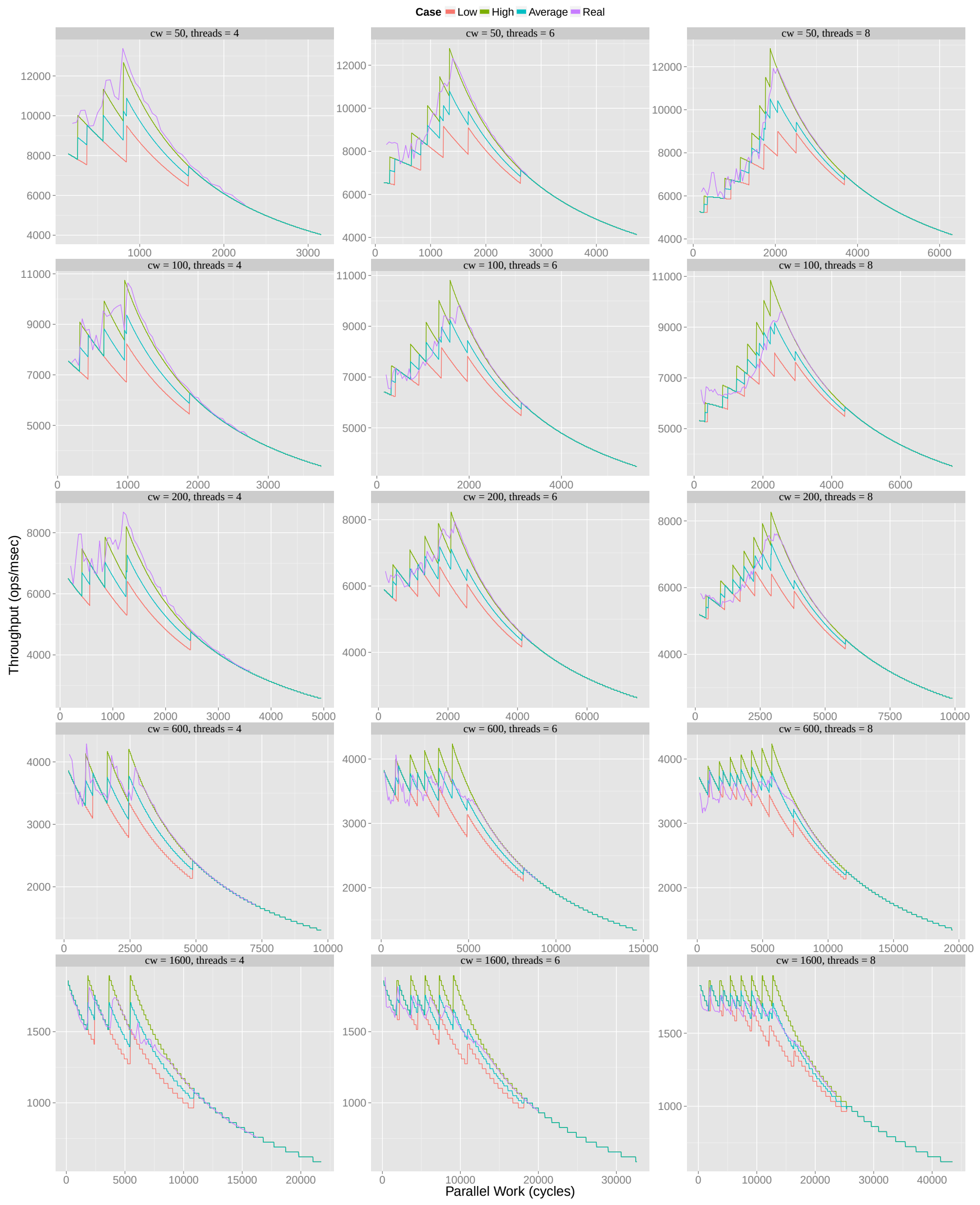


Figure 28: Synthetic program

latencies. In the experiments, we vary the number of threads between 4 and 8 since the maximum number of threads that can be used in the experiments are bounded by the number of physical cores that reside in one socket.

In all figures, y-axis provides the throughput, which is the number of successful operations completed per millisecond. Parallel work is represented in x-axis in cycles. As mentioned in Section 4.4, the graphs contain the high and low estimates, corresponding to the lower and upper bound on the wasted retries, respectively, and an additional curve that shows the average of them.

As mentioned before, the latencies of *CAS* and *Read* are parameters of our model. We used the methodology described in [27] to measure latencies of these operations in a benchmark program by using two threads that are pinned to the same socket. The aim is to bring the cache line into the state used in our model. Our assumption is that the *Read* is conducted on an invalid line. For *CAS*, the state of the cache line could be exclusive, forward, shared or invalid. Regardless of the state of the cache line, *CAS* requests it for ownership, that compels invalidation in other cores, which in turn incurs a two-way communication and a memory fence afterwards to assure atomicity. Thus, the latency of *CAS* does not show negligible variability with respect to the state of the cache line, as also revealed in our latency benchmarks.

As for the computation cost, the work inside the parallel section is implemented by a dummy for-loop of *Pause* instructions.

4.6.2 Synthetic Tests

For the evaluation of our model, we first create synthetic tests that emulate different design patterns of lock-free data structures (value of cw) and different application contexts (value of pw). As described in the previous subsection, in the Procedure AbstractAlgorithm, the amount of work in both the parallel section and the retry loop are implemented as dummy loops, whose costs are adjusted through the number of iterations in the loop.

Generally speaking, in Figure 28, we observe two main behaviors: when pw is high, the data structure is not contended, and threads can operate without failure. When pw is low, the data structure is contended, and depending on the size of cw (that drives the expansion) a steep decrease in throughput or just a roughly constant bound on the performance is observed.

The position of the experimental curve between the high and low estimates, depends on cw. It can be observed that the experimental curve mostly tends upwards as cw gets smaller, possibly because the serialization of the *CAS*s helps the synchronization of the threads.

For the cases with considerable expansion, it is expected to have unfairness among threads. This fact loosens the validity of our deterministic model that assumes uniformity and presumably leads to underestimation of throughput.

Another interesting fact is the waves appearing on the experimental curve, especially when the number of threads is low or the critical work big. This behavior is originating because of the variation of r with the change of parallel work, a fact that is captured by our analysis.

4.6.2.1 Several retry loops

We have created experiments by combining several retry loops, each operating on an independent variable which is aligned to a cache line. In Figure 29, results are compared with the model for single retry loop case where the single retry loop is equal to the longest retry

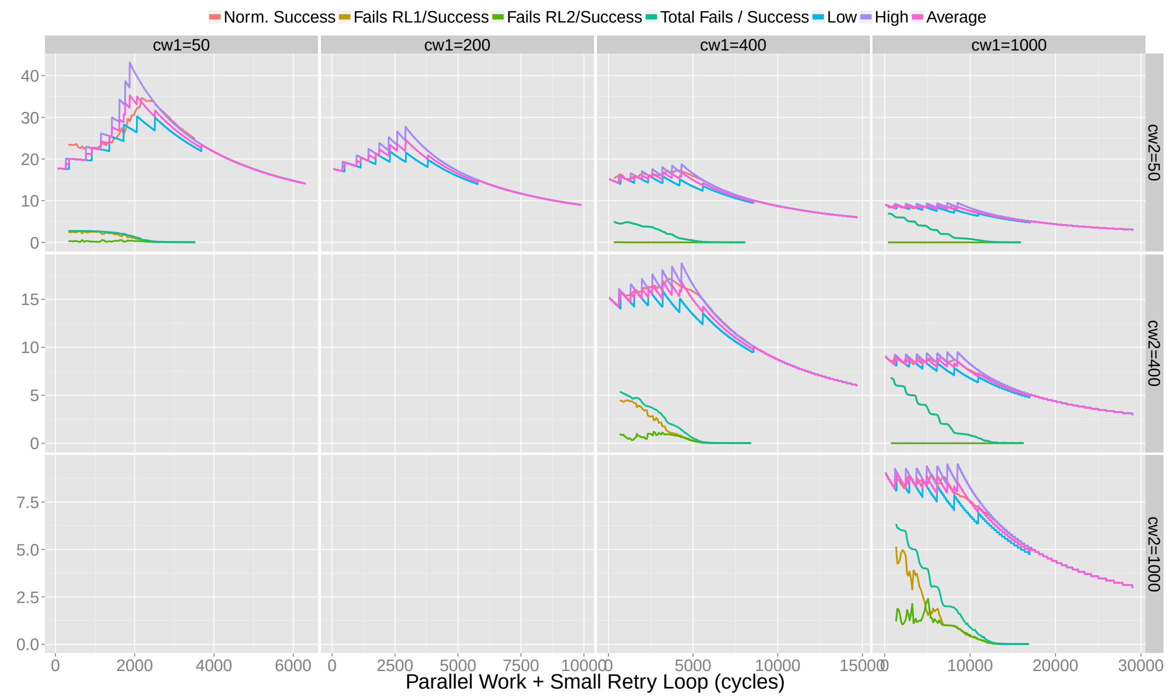


Figure 29: Multiple retry loops with 8 threads

loop, while the other retry loops are part of the parallel section. The distribution of fails in the retry loops are illustrated and all throughput curves are normalized with a factor of 175 (to be easily seen in the same graph). Fails per success values are not normalized and a success is obtained after completing all retry loops.

4.6.3 Treiber's Stack

The lock-free stack by Treiber [98] is one of the most studied efficient data structures. `Pop` and `Push` both contain a retry loop, such that each retry starts with a *Read* and ends with *CAS* on the shared top pointer. In order to validate our model, we start by using `Pop`s. From a stack which is initiated with 50 million elements, threads continuously pop elements for a given amount of time. We count the total number of pop operations per millisecond. Each `Pop` first reads the top pointer and gets the next pointer of the element to obtain the address of the second element in the stack, before attempting to *CAS* with the address of the second element. The access to the next pointer of the first element occurs in between the *Read* and the *CAS*. Thus, it represents the work in cw. This memory access can possibly introduce a costly cache miss depending on the locality of the popped element.

To validate our model with different cw values, we make use of this costly cache miss possibility. We allocate a contiguous chunk of memory and align each element to a cache line. Then, we initialize the stack by pushing elements from contiguous memory either with

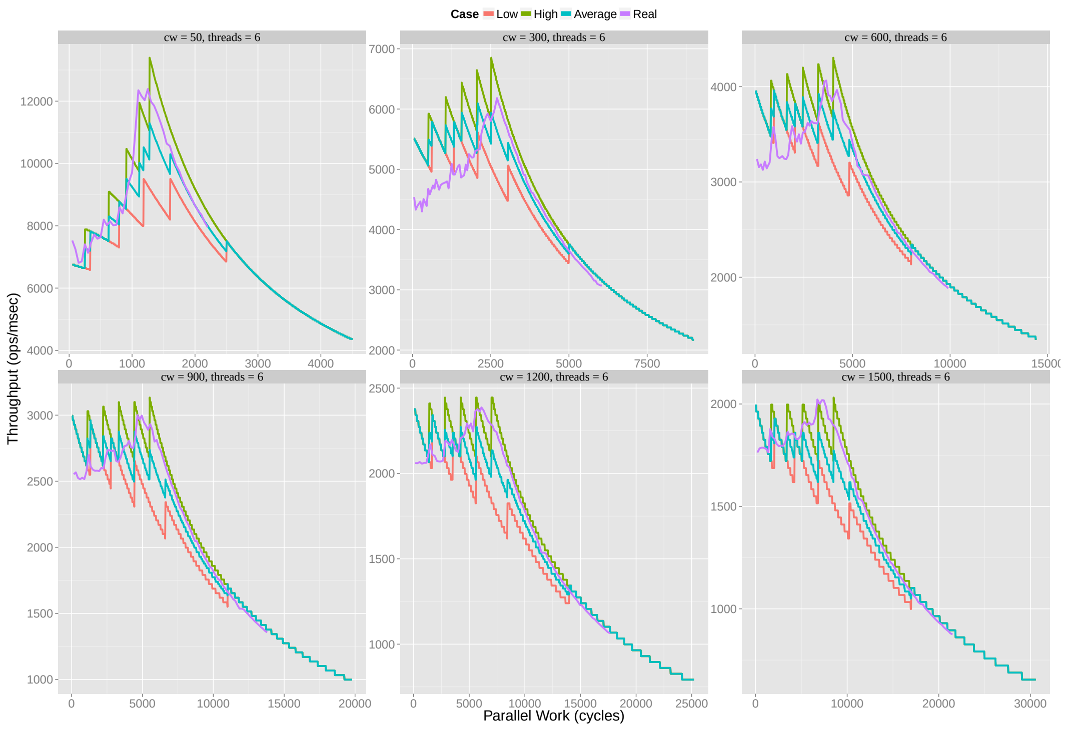


Figure 30: `Pop` on Treiber's stack

a single or large stride to disable the prefetcher. When we measure the latency of *cw* in `Pop` for single and large stride cases, we obtain the values that are approximately 50 and 300 cycles, respectively. As a remark, 300 cycles is the cost of an L3 miss in our system when it is serviced from the local main memory module. To create more test cases with larger *cw*, we extended the stack implementation to pop multiple elements with a single operation. Thus, each access to the next element could introduce an additional L3 cache miss while popping multiple elements. By doing so, we created cases in which each thread pops 2, 3, *etc.* elements, and *cw* goes to 600, 900, *etc.* cycles, respectively. In Figure 30, comparison of the experimental results from Treiber's stack and our model is provided.

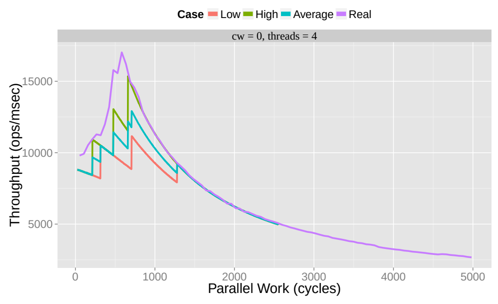
As a remark, we did not implemented memory reclamation for our experiments but one can implement a stack that allows pop and push of multiple elements with small modifications using hazard pointers [79]. Pushing can be implemented in the same way as single element case. A `Pop` requires some modifications for memory reclamation. It can be implemented by making use of hazard pointers just by adding the address of the next element to the hazard list before jumping to it. Also, the validity of top pointer should be checked after adding the pointer to the hazard list to make sure that other threads are aware of the newly added hazard pointer. By repeating this process, a thread can jump through multiple elements and pop all of them with a *CAS* at the end.

Algorithm 1: Multiple Pop

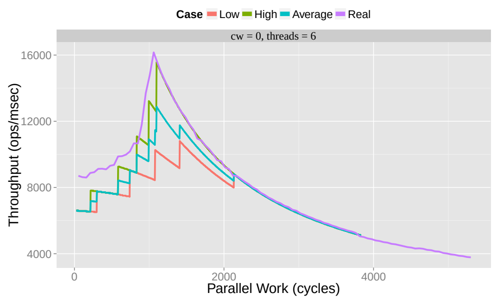
```
1  Pop (multiple)
2  while true do
3      t = Read(top);
4      for multiple do
5          if t = NULL then
6              return EMPTY;
7          hp* = t;
8          if top != t then
9              break;
10         hp++;
11         next = t.next;
12     if CAS(&top, t, next) then
13         break;
14 RetireNodes (t, multiple);
```

4.6.4 Shared Counter

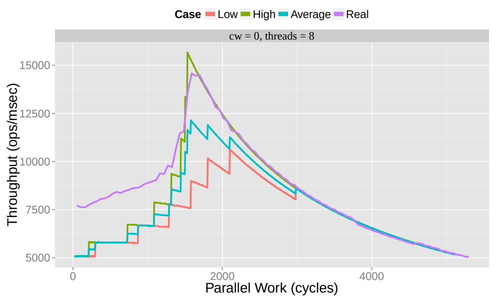
In [30], the authors have implemented a "scalable statistics counters" relying on the following idea: when contention is low, the implementation is a regular concurrent counter with a CAS;



(a) 4 threads



(b) 6 threads

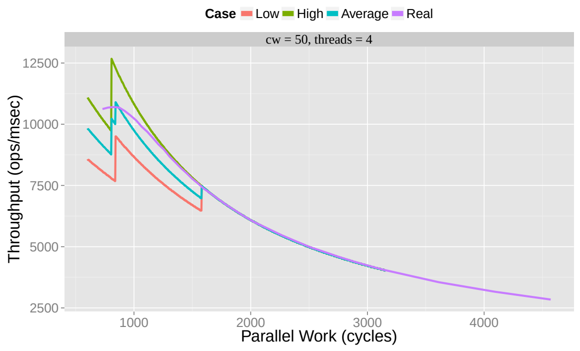


(c) 8 threads

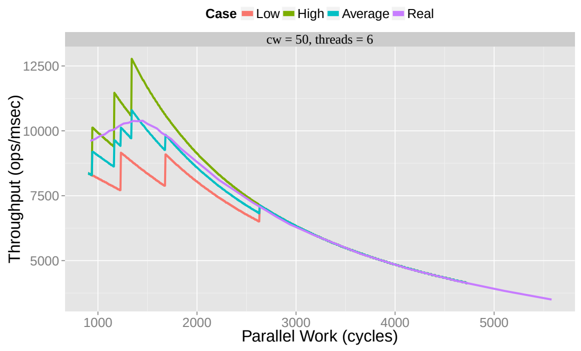
Figure 31: `Increment` on a shared counter

when the counter starts to be contended, it switches to a statistical implementation, where the counter is actually incremented less frequently, but by a higher value. One key point of this algorithm is the switch point, which is decided thanks to the number of failed increments; our model can be used by providing the peak point of performance of the regular counter implementation as the switch point. We then have implemented a shared counter which is basically a *Fetch-and-Increment* using a *CAS*, and compared it with our analysis. The result is illustrated in Figure 31, and shows that the parallel section size corresponding to the peak point is correctly estimated using our analysis.

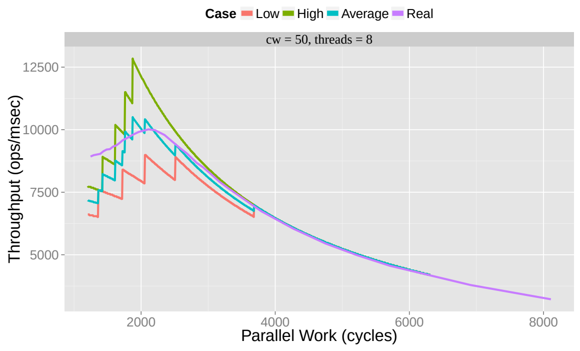
4.6.5 `DeleteMin` in Priority List



(a) 4 threads



(b) 6 threads



(c) 8 threads

Figure 32: `DeleteMin` on a priority list

We have applied our model to `DeleteMin` of the skiplist based priority queue designed in [76]. `DeleteMin` traverses the list from the beginning of the lowest level, finds the first node that is not logically deleted, and tries to delete it by marking. If the operation does not succeed, it continues with the next node. Physical removal is done in batches when reaching a threshold on the number of deleted prefixes, and is followed by a restructuring of the list by updating the higher level pointers, which is conducted by the thread that is successful in redirecting the head to the node deleted by itself.

We consider the last link traversal before the logical deletion as critical work, as it continues with the next node in case of failure. The rest of the traversal is attributed to the parallel section as the threads can proceed concurrently without interference. We measured the average cost of a traversal under low contention for each number of threads, since traversal becomes expensive with more threads. In addition, average cost of restructuring is also included in the parallel section since it is executed infrequently by a single thread.

We initialize the priority queue with a large set of elements. As illustrated in Figure 32, the smallest *pw* value is not zero as the average cost of traversal and restructuring is intrinsically included. The peak point is in the estimated place but the curve does not go down sharply under high contention. This presumably occurs as the traversal might require more than one steps (link access) after a failed attempt, which creates a back-off effect.

4.6.6 `Enqueue-Dequeue` on a Queue

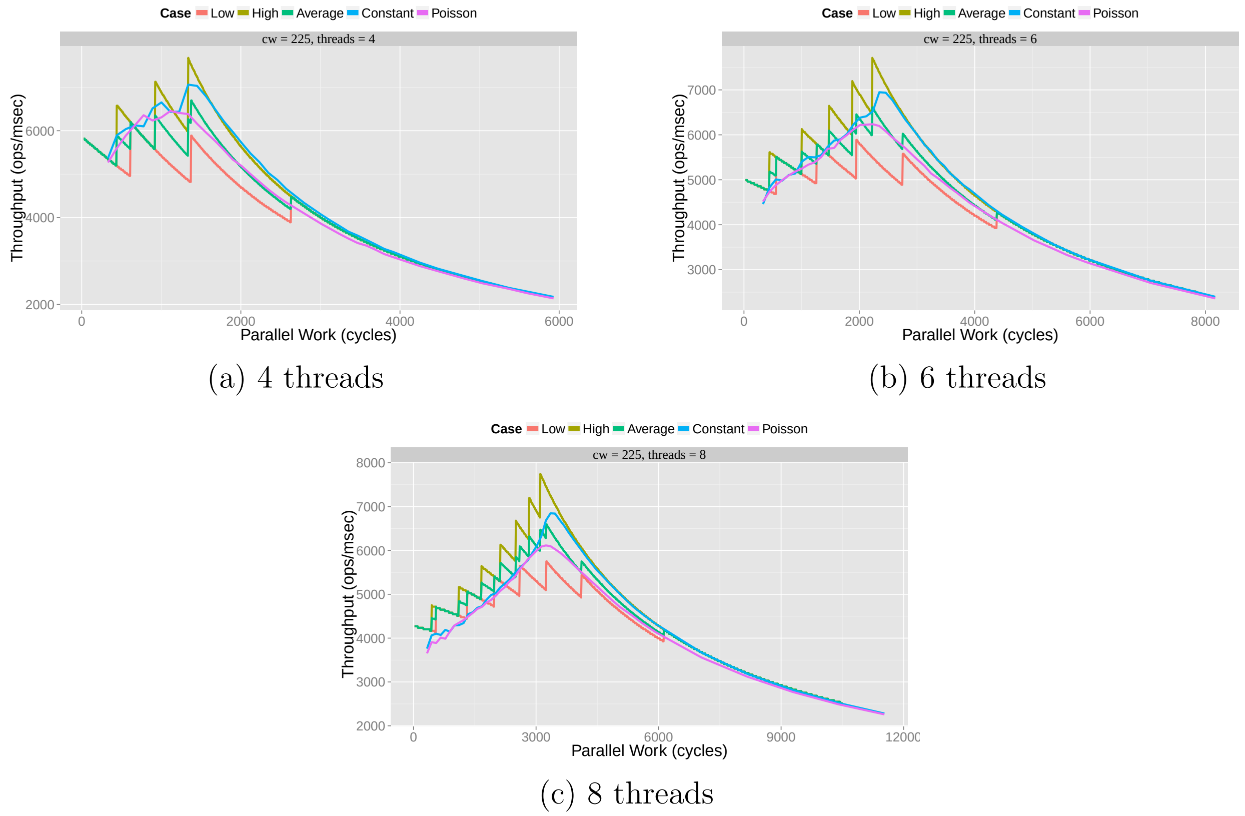


Figure 33: `Enqueue-Dequeue` on Michael and Scott queues

In order to demonstrate the validity of the model with several retry loops (see Section 4.5.3), and that the results covers a wider spectrum of application and designs from the ones we focused in our model, we studied the following setting: the threads share a queue, and each thread enqueues an element, executes the parallel section, dequeues an element, and reiterates. We consider the queue implementation by Michael and Scott [80], that is usually viewed as the reference queue while looking at lock-free queue implementations.

`Dequeue` operations fit immediately into our model but `Enqueue` operations need an adjustment due to the helping mechanism. Note that without this helping mechanism, a simple queue implementation would fit directly, but we also want to show that the model is malleable, *i.e.* the fundamental behavior remains unchanged even if we divert slightly from the initial assumptions. We consider an equivalent execution that catches up with the model, and use it to approximate the performance of the actual execution of `Enqueue`.

`Enqueue` is composed of two steps. Firstly, the new node is attached to the last node of the queue via a *CAS*, that we denote by $\texttt{CAS}_\texttt{A}$, leading to a transient state. Secondly, the tail is redirected to point to the new node via another *CAS*, that we denote by $\texttt{CAS}_\texttt{B}$, which brings back the queue into a steady state.

A new `Enqueue` can not proceed before the two steps of previous success are completed. The first step is the linearization point of operation and the second step could be conducted by a different thread through the helping mechanism. In order to start a new `Enqueue`, concurrent `Enqueue`s help the completion of the second step of the last success if they find the queue in the transient state. Alternatively, they try to attach their node to the queue if the queue is in the steady state at the instant of check. This process continues until they manage to attach their node to the queue via a retry loop in which state is checked and corresponding `CAS` is executed.

The flow of an `Enqueue` is determined by this state checks. Thus, an `Enqueue` could execute multiple $\texttt{CAS}_\texttt{B}$ (successful or failing) and multiple $\texttt{CAS}_\texttt{A}$ (failing) in an interleaved manner, before succeeding in $\texttt{CAS}_\texttt{A}$ at the end of the last retry. If we assume that both states are equally probable for a check instant which will then end up with a retry, the number of `CAS` s that ends up with a retry are expected to be distributed equally among $\texttt{CAS}_\texttt{A}$ and $\texttt{CAS}_\texttt{B}$ for each thread. In addition, each thread has a successful $\texttt{CAS}_\texttt{A}$ (which linearizes the `Enqueue`) and a $\texttt{CAS}_\texttt{B}$ at the end of the operation which could either be successful or failed by a concurrent helper thread.

We imitate such an execution with an equivalent execution in which threads keep the same relative ordering of the invocation, return from `Enqueue` together with same result. In equivalent execution, threads alternate between $\texttt{CAS}_\texttt{A}$ and $\texttt{CAS}_\texttt{B}$ in their retries, and both steps of successful operation is conducted by the same thread. The equivalent execution can be obtained by thread-wise reordering of `CAS` s that leads to a retry and exchanging successful $\texttt{CAS}_\texttt{B}$ s with the failed counterparts at the end of an `Enqueue`, as the latter ones indeed fail because of this success of helper threads. The model can be applied to this equivalent execution by attributing each $\texttt{CAS}_\texttt{A}$-$\texttt{CAS}_\texttt{B}$ couple to a single iteration and represent it as a larger retry loop since the successful couple can not overlap with another successful one and all overlapping ones fail. With a straightforward extension of the expansion formula, we accomodate the $\texttt{CAS}_\texttt{A}$ in the critical work which can also expand, and use $\texttt{CAS}_\texttt{B}$ as the *CAS* of our model.

In addition, we take one step further outside the analysis by including a new case, where the parallel section follows a Poisson distribution, instead of being constant. pw is chosen as the mean to generate Poisson distribution instead of taking it constant. The results are illustrated in Figure 33. Our model provides good estimates for the constant pw and also reasonable results for the Poisson distribution case, although this case deviates from (/extends) our model assumptions. The advantage of regularity, which brings synchronization to threads, can be observed when the constant and Poisson distributions are compared. In the Poisson distribution, the threads start to fail with larger pw, which smoothes the curve around the peak of the throughput curve.

4.6.7 Discussion

In this subsection we discuss the adequacy of our model, specifically the cyclic argument, to capture the behavior that we observe in practice. Figure 34 illustrates the frequency of occurrence of a given number of consecutive fails, together with average fails per success values and the throughput values, normalized by a constant factor so that they can be seen on

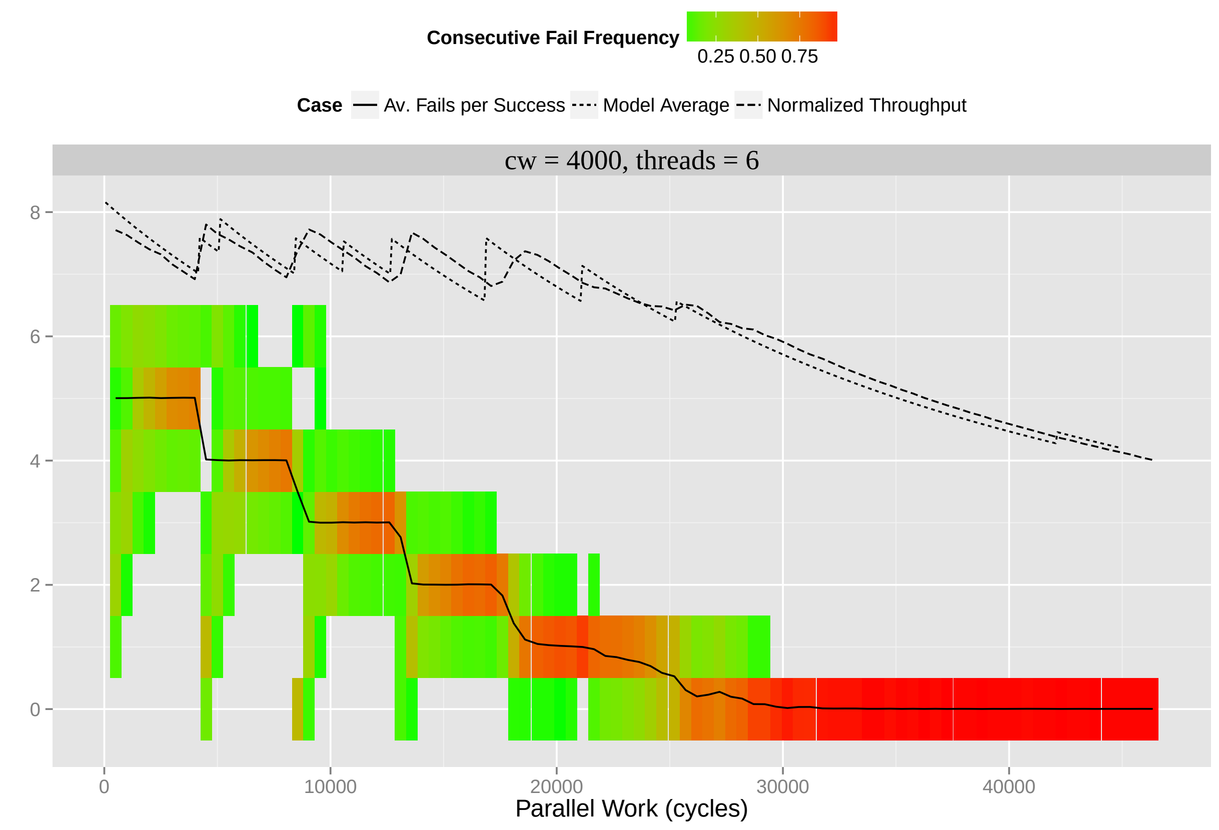
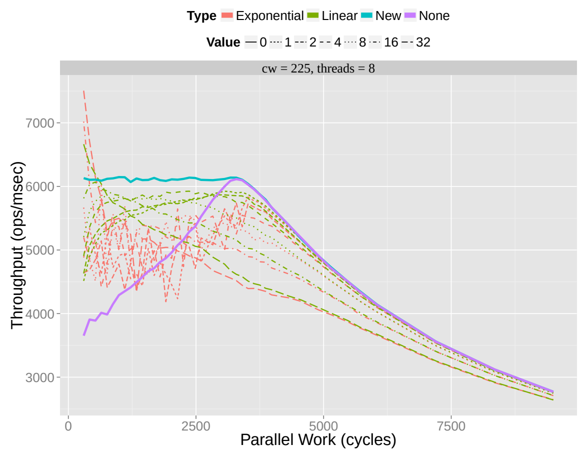


Figure 34: Consecutive Fails Frequency

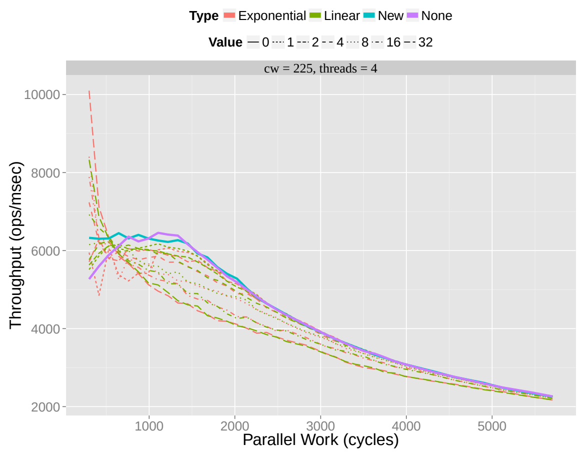
the graph. In the background, the frequency of occurrence of a given number of consecutive fails before success is presented. As a remark, the frequency of 6+ fails is gathered with 6. We expect to see a frequency distribution concentrated around the average fails per success value, within the bounds computed by our model.

While comparing the distribution of failures with the throughput, we could conjecture that the bumps come from the fact that the failures spread out. However, our model captures correctly the throughput variations and thus strips down the right impacting factor. The spread of the distribution of failures indicates the violation of a stable cyclic execution (that takes place in our model), but in these regions, r actually gets close to 0, as well as the minimum of all gaps. The scattering in failures shows that, during the execution, a thread is overtaken by another one. Still, as gaps are close to 0, the imaginary execution, in which we switch the two thread IDs, would create almost the same performance effect. This reasoning is strengthened by the fact that the actual average number of failures follows the step behavior, predicted by our model. This shows that even when the real execution is not cyclic and the distribution of failures is not concentrated, our model that results in a cyclic execution remains a close approximation of the actual execution.

4.6.8 Back-Off Tuning



(a) 8 threads



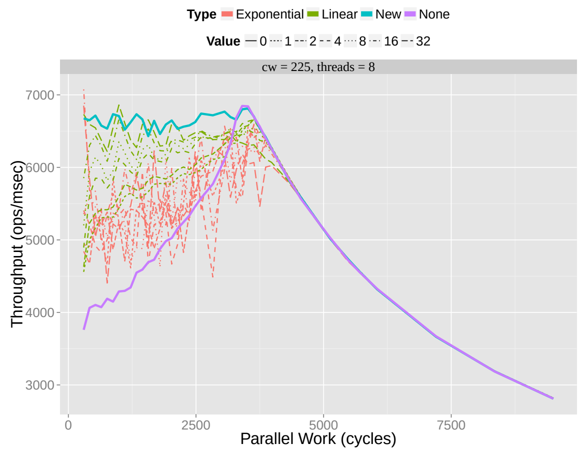
(b) 4 threads

Figure 35: Comparison of back-off schemes for Poisson Distribution

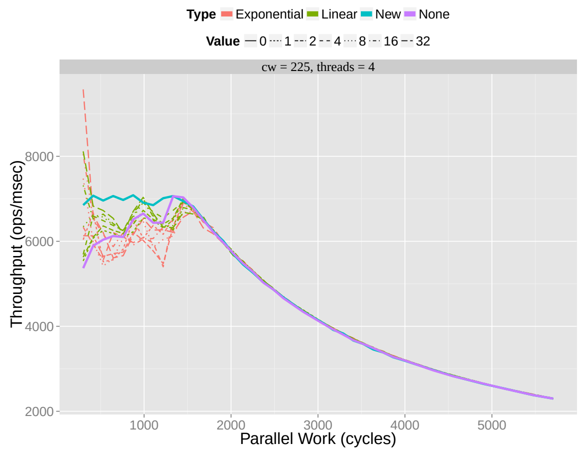
Together with the analysis comes a natural back-off strategy: we estimate the pw corresponding to the peak point of the average curve, and when the parallel section is smaller than the corresponding pw, we add a back-off in the parallel section, so that the new parallel section is at the peak point.

We have applied exponential, linear and our back-off strategy to the `Enqueue`/`Dequeue` experiment specified above. Our back-off estimate provides good results for both types of distribution. In Figure 35 (where the values of back-off are steps of 115 cycles), the comparison is plotted for the Poisson distribution, which is likely to be the worst for our back-off. Our back-off strategy is better than the other, except for very small parallel sections, but other back-off strategies should be tuned for each value of pw.

We obtained the same shapes while removing the distribution law and considering constant values. The results are illustrated in Figure 36.



(a) 8 threads



(b) 4 threads

Figure 36: Comparison of back-off schemes for constant pw

4.7 Conclusion on Throughput Modeling

We have modeled and analyzed the performance of a general class of lock-free algorithms. Thanks to this analysis, we have been able to predict the throughput of such algorithms, on actual executions. The analysis relies on the estimation of two impacting factors that lower the throughput: on the one hand, the expansion, due to the serialization of the atomic primitives that take place in the retry loops; on the other hand, the wasted retries, due to a non-optimal synchronization between the running threads. We have derived methods to calculate those parameters, along with the final throughput estimate, that is calculated from a combination of these two previous parameters. As a side result of our work, this accurate prediction enables the design of a back-off technique that performs better than other well-known techniques, namely linear and exponential back-offs.

As a future work, we envision to enlarge the domain of validity of the model, in order to cope with data structures whose operations do not have constant retry loop, as well as the framework, so that it includes more various access patterns. The fact that our results extend outside the model allows us to be optimistic on the identification of the right impacting factors. Finally, we also foresee studying back-off techniques that would combine a back-off in the parallel section (for lower contention) and in the retry loops (for higher robustness).

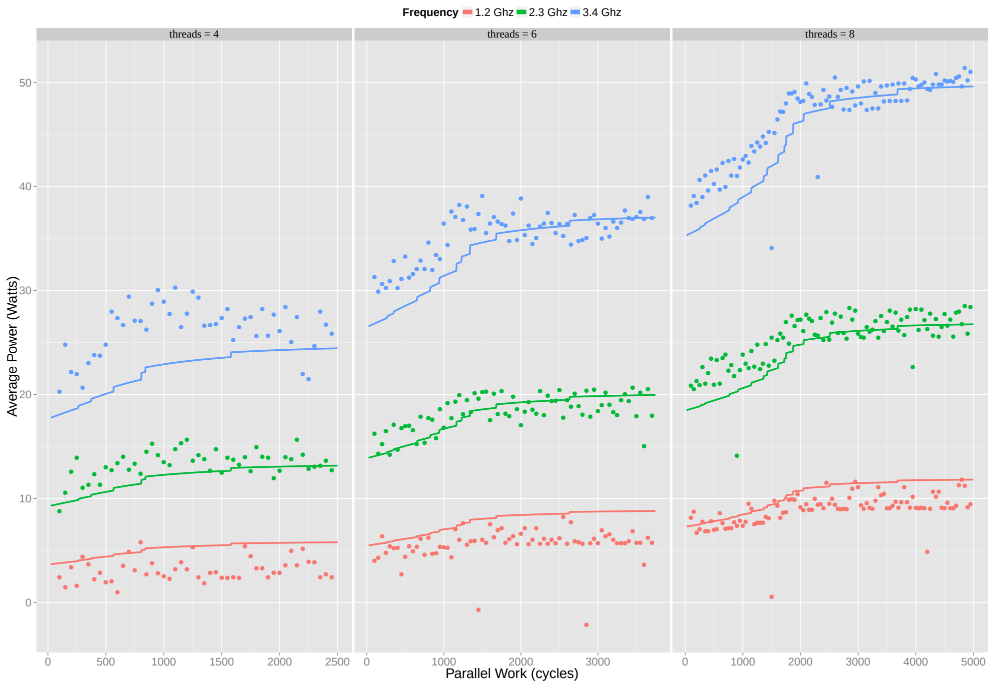


Figure 37: Average Power Consumption for Treiber's Stack (Pop operation)

4.8 Energy Modelling and Empirical Evaluation

We introduced our power model and the power impacting factors in D2.1 [50]. Here, we combine them with our performance model that is illustrated in Section 4. By doing so, we aim to come up with the average power consumption predictions for the parallel programs that use fundamental concurrent lock-free data structures.

In D2.1, we decompose the power into two orthogonal bases, each base having three dimensions. On the one hand, we define the model base by separating the power into static, active and dynamic power. On the other hand, the measurement base corresponds to the components that actually dissipates the power,*i.e.* CPU, memory and uncore, in accordance with RAPL energy counters. We recall that we are interested only in the dynamic component of power, since we determine the static power and the activation power, that do not depend on the data structure implementation or the application that uses the concurrent data structure. Our performance model does not cover the cases where the inter-socket communication takes place. Here, we do not present the dynamic memory and uncore power evaluations because they are insignificant (*i.e.* close to 0 for all cases) when there are not memory accesses (parallel work is composed of multiplication instructions) or inter-socket communication (threads are pinned to the same socket).

In D2.1, we illustrate that CPI (cycles per instruction) is the main impacting factor for the dynamic CPU power. And, CPI reaches to a high value during the execution of data structure operations. This is because the retry loops are composed of read accesses and *CAS*s

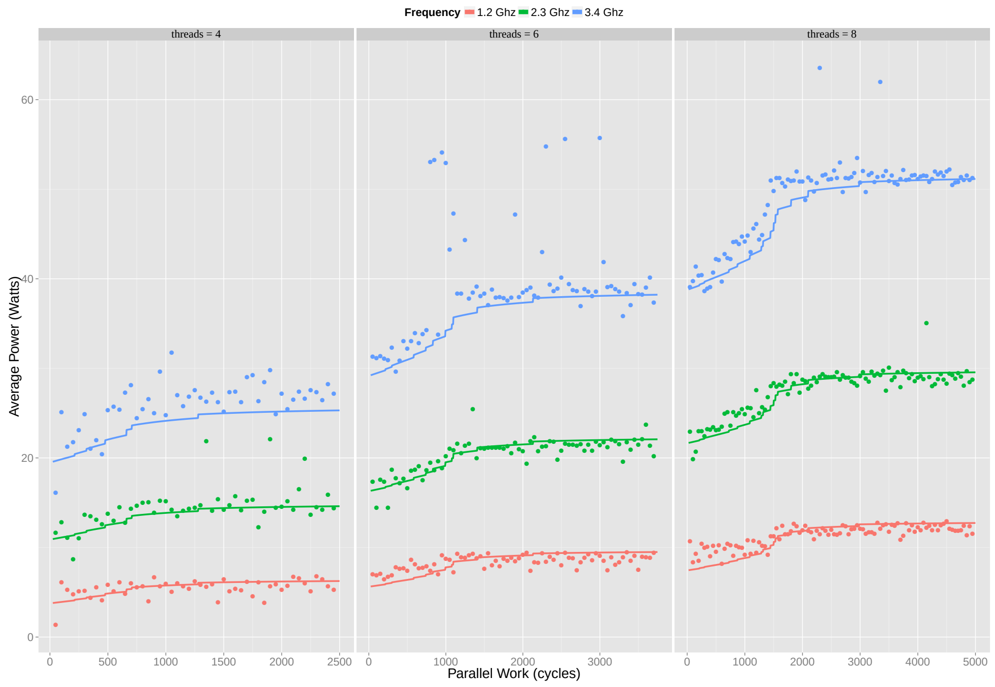


Figure 38: Average Power Consumption for Shared Counter (Increment operation)

which typically lead to costly cache misses in a concurrent environment. In contrast, one could expect to observe a lower value of CPI in the application specific parts of the parallel program, *i.e.* parallel work. This is because computations presumably are executed in the application specific part and the concurrent data structures are used for the communication. That is why, we emulate the parallel work with a for-loop of multiplication instructions. In this region of the parallel program, CPI gets low as a multiplication instruction can be executed within almost a cycle so we observe an increase in the dynamic CPU power during the execution of the parallel work. As we expect two different average power behaviours in these two regions, we build our reasoning over the ratio of execution time that the threads spend in the retry loops. Thanks to our performance model, we can predict this ratio for each value of the parallel work, the number of threads and the data structure operation. We can also generalize our performance predictions to a whole clock frequency domain with a straightforward evaluation of the model parameters for each frequency (*i.e.* pw, cw, CAS and rc).

Dynamic CPU power does not scale linearly with the frequency, instead shows a super-linear behaviour (see D2.1). We handle this issue by taking power measurements for each frequency. For each data structure operation and frequency, we run the parallel program for two values of pw that corresponds to a low and a high contention case. respectively (*i.e.* where the ratio of time spent in retry loop is 0.05 and 0.5) with 8 threads for a given duration. Using RAPL energy counters, we measure the energy consumption for these cases.

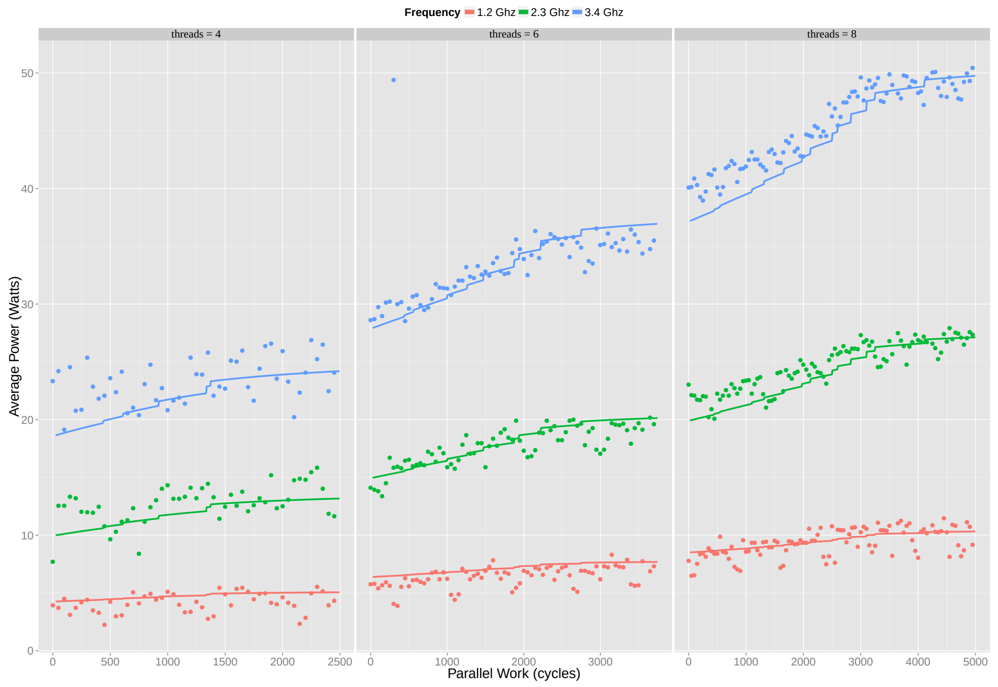


Figure 39: Average Power Consumption for MS Queue (Enqueue operation)

Then, we use these values to extract the average power consumption for each phase (retry loop and parallel work) of the parallel program. Recall that we know the static and activation component of power so we can extract the dynamic component for memory, uncore and CPU power.

The average power consumption (for each of the memory, CPU and uncore components) for the retry loop (Pow_{RL}) and the parallel work (Pow_{PW}) can be obtained as follows: (by using the two energy measurements for pw values corresponding to two different ratios of execution time spent in the retry loops):

$$Pow_{Average} = Threads \times (Pow_{RL} \times Ratio_{RL} + Pow_{PW} \times (1 - Ratio_{RL}))$$

Based on the estimation of Pow_{PW} and Pow_{RL}, for each frequency and data structure operation, we provide average power predictions that span the whole parallel work and the number of threads domain. This is simply done by plugging the $Ratio_{RL}$, that is provided by our performance model, to the formula above. In Figures 37, 38, 39, 40, we present the results for a set of fundamental lock-free data structure operations, namely for Micheal and Scott Queue (Enqueue and Dequeue operations), Treiber's Stack (Pop operation) and Shared Counter (Increment operation). In the figures, lines and points represent predictions and actual measurements, respectively.

In all the figures, we observe a similar behaviour. Dynamic CPU power decreases when pw decreases. We know that pw is a key aspect that influences the contention on the data

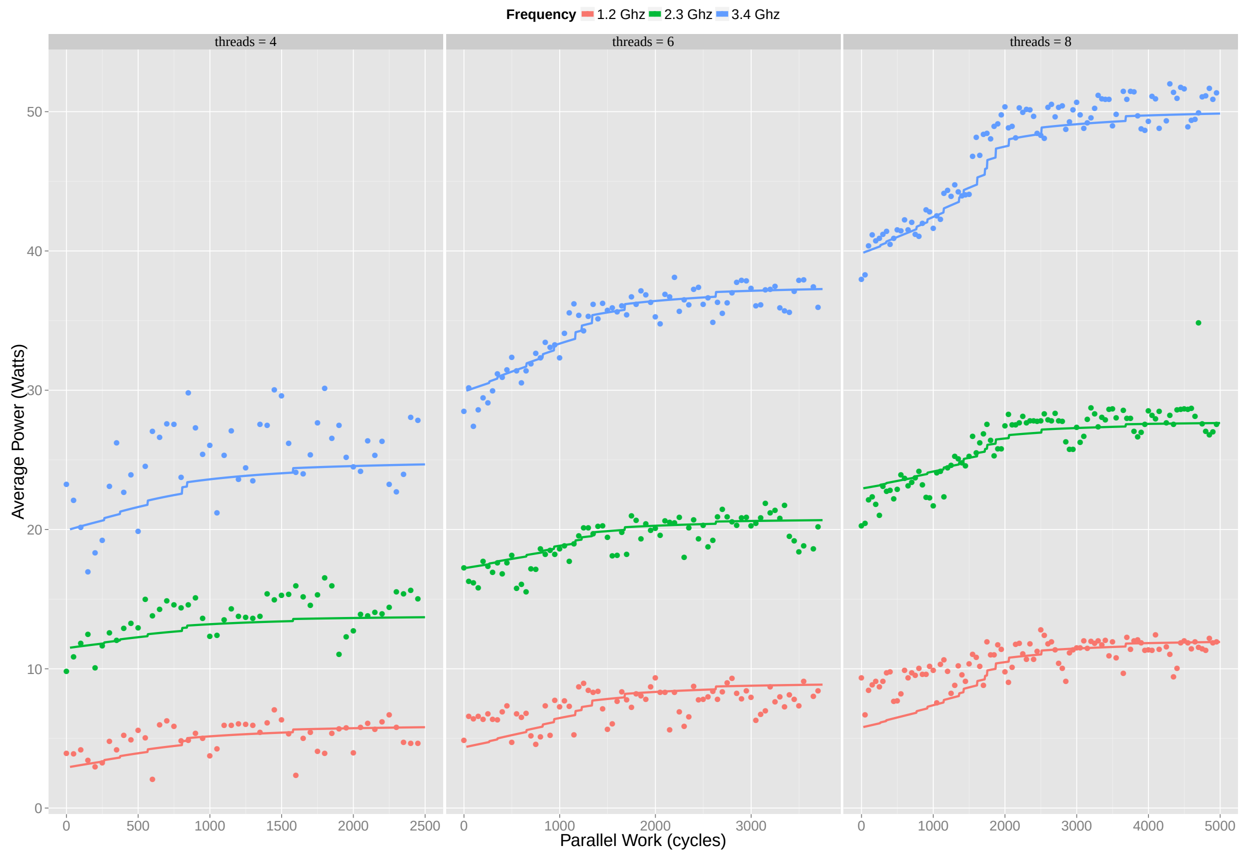


Figure 40: Average Power Consumption for MS Queue (Dequeue operation)

structure, equally with the the ratio of time that threads spend in the retry loop. With the decrease of pw, CPI increases and dynamic CPU power reduces.

5 Energy Evaluation on Myriad2 for Multiway Aggregation on Streaming Applications

5.1 Introduction

5.1.1 Background

Nowadays due to the fact that multiprocessors can reach high levels of performance and efficiency, they are employed in almost all areas of engineering. The increase in number of transistors was until recently a widely used approach in order to improve performance. This trend introduced power consumption issues, additional design complexity of computing systems and wire delays because of complicated designs adopted. Due to all these aforementioned issues arising, a shift has been observed towards multicore systems, improving in this way parallelism and performance [40].

Having multiple processors working in parallel on the same or different tasks, translates into multiple processes trying to communicate and synchronize, a trend that provides an interesting challenge especially when space availability and energy consumption are limited resources, as is the case for embedded systems.

The data structures that are used play essential role for the communication to become more efficient. In the context of streaming applications, data flows from one stage to the other via data structures in order to be processed and used further by an application. The way this communication takes place defines the effectiveness of the application, the level of difficulty it can deal with and its applicability to crucial projects with major impact for the society.

Extensive research on how to increase performance of a computing system has already been conducted within the High Performance Computing (HPC) community and the results and solutions obtained match the needs of the embedded systems field [84]. Furthermore power consumption becomes a major issue in HPC as the energy cost of a supercomputer facility's operation after some years almost equals the cost of the hardware infrastructure.

Due to the fact that power consumption consists one of the main constraints the HPC community will have to deal with in future systems and as this is an important issue that embedded systems try to confront, it is the turn of HPC to adopt solutions utilized by embedded systems [85].

So far embedded platforms utilize mutual exclusion or interrupt handling in order to achieve synchronization, while lock-based and lock-free approaches are adopted by the HPC community. As there is a convergence between the two fields and one can benefit from the other, it is interesting to search how solutions provided by the HPC field can be adjusted to embedded multicore platforms in this context and how much improvement we can get by such an initiative. Additionally the improvement in power consumption that we can get out of such an investigation can be useful to HPC in the future.

The work presented here aims to make a research on data structures suited for embedded systems and investigate trade-offs between different implementations in terms of energy consumption, memory utilization and performance. Through this investigation the focus will be on data streaming applications implementing multiway aggregation of the received data.

Although efficient data structures for a concurrent environment have been studied extensively, the issue of appropriate data structures for data streaming applications has been neglected [44]. Concurrent data structures play a major role between aggregation stages, through the parallelism and the load balancing role that they can offer in this kind of applications. For this reason such an application will be developed and based on the research conducted on concurrent data structures, an efficient solution providing lower latency, bigger throughput and energy efficiency at the data aggregation function of the application will try to be achieved.

We examin already existing algorithms and solutions of concurrent data structures, with specific interest in energy, space and performance trade-offs. The shared data structure is the queue as it consists one of the most widely used data structures in embedded applications. The algorithms analyzed provide solutions for the Single-Producer-Single-Consumer (SPSC) problem.

The data streaming application runs on Myriad 2 platform and evaluation is realized in terms of how many messages containing application data can be processed each millisecond for given fixed workloads. Furthermore power consumption is measured in order to evaluate how many Joules are needed for the processing of a message.

Next section will introduce latest researches conducted on the subject. Section 5.2 explains the type of application this work takes into consideration and how concurrent data structures are used within such kind of applications. Following, Section 5.3 gives a brief overview of the platform on which evaluation will be conducted, together with the issues concerning memory handling when cache memory is used. Subsequently Section 5.4 contains theory concerning algorithms for concurrent data structures, essential to keep up with the rest of the document. Section 5.7 continues with the evaluation results of the chosen algorithms in order to observe their performance on the platform before they are used in the final application whose evaluation comes right after in Section 5.8. Finally, a discussion on the results and some final thoughts are provided in Section 5.9.

5.1.2 Related Work

In 2013 Cederman *et al.* [44] investigate about the neglected field of concurrent data structures on the context of efficiency in data streaming aggregation. As it has already been mentioned, data structures play a major role between aggregation stages and parallelism in some of the stages is a challenging issue to address. Through this work it is shown that for this type of applications lock-based or lock-free approaches do not matter as much as the data structure itself that is used. New types of data structures have been implemented giving better throughput and less latency than already existing queue based approaches.

In 2014 Papadopoulos *et al.* [84] try to look into the future where the number of cores per chip will be increased. As currently used lock based techniques for synchronization between contending threads over shared data has some disadvantages and do not scale well as the number of cores increases (due to increased contention), lock-free solutions introduced by the HPC field are adjusted for use in a multicore embedded platform in order to find out how much we can improve in terms of performance depending on the solution tested. The shared data structure used for this work is the queue. As a result of the investigation 29.6% increase in the performance of the platform was obtained.

Again in 2014 Papadopoulos *et al.* [85] conduct an investigation on concurrent queue implementations inspired by the HPC domain adjusted to an embedded multicore platform. This time along with performance evaluation, power consumption is taken into consideration and 6.8% less power dissipation is achieved while the lock free implementations provide lower execution times by 28.2% compared to lock based approaches. As mentioned to this work, as the number of cores per chip will increase in the future, lock free solutions will be more attractive and there is plenty of room for improvement.

5.2 Data streaming aggregation

In this section multiway data streaming aggregation is explained in order to further understand the type of problem we aim to investigate and give a solution to.

5.2.1 Data streaming

Data streaming constitutes a new paradigm which processes incoming information of a system in real time instead of following the classic way of storing the received data in order to process them later on. This need is imperative nowadays as the amount of data that needs to be processed on a daily basis by a contemporary system can be really big rendering solutions that abide by the store-and-process paradigm non-practical.

5.2.2 A Stream and Multiway Streams

A stream is a flow of incoming tuples containing fields related to data that need to be communicated by an application along with a timestamp provided by the producer of a tuple [45]. In the context of a multiway aggregation, multiple input streams are sent to an aggregator which processes them in order to produce a deterministic output depending on what the application wants to extract from the given data.

5.2.3 Aggregation

An example of an aggregation application would be smart metering data out of which an aggregator sums up the amount of Watts that have been consumed by a house or a whole district. An aggregator may be stateful or stateless depending on the nature of an application and whether it needs an aggregator to hold some kind of state throughout the whole processing phase.

In case of stateful aggregators, time is divided in windows within which state is held. Windows are set either according to time or according to tuples. They have a certain size declaring the amount of time or incoming tuples for which they are valid and they also have an advance parameter which sets the boundaries of the next window depending on the previous one.

As an example, if tuples of last ten minutes are grouped together every 5 minutes this means that we have windows of size ten and advance five. The windows that would be created in this way would be $[0, 10)$, $[5, 15)$, $[10, 20)$ and so on. The same holds for tuple based approaches where for example ten last tuples are grouped together every five incoming tuples.

According to [45] the functionality of an aggregator consists of four main stages :

1. *Add stage*: Fetch tuples from each input stream.
2. *Merge stage* : Merge and sort fetched tuples according to timestamp.
3. *Update stage* : Update the state of windows a tuple contributes to.
4. *Output stage* : Forward output tuples to the next aggregation stage.

In this work, the focus is on the third and fourth stages of an aggregator. The first and second stages are out of scope and are not taken into consideration as the application deployed simulates already merged and sorted data.

5.2.4 Concurrent Data Structures

Under the context of multiway data streaming aggregation, concurrent data structures are used between the different stages of the aggregation process in order for the communication between different parties to be achieved.

The data structures used need to provide as much parallelism as possible, ease the communication of tuples between different stages of the process and load balance the workload so that all processes deal with similar amount of workload and none is potentially choked creating a bottleneck. Lock-free approaches prove to increase throughput and start arising research interest more and more, something that led to mainstream programming languages to incorporate implementations in their standard libraries [45].

Two typical stream processing engines are Borealis [4] and StreamCloud [43]. For multiway aggregation to be achieved in these implementations, as depicted in Figure 41, tuples from each input stream are placed in queues by multiple threads. On the other side there is one consumer thread performing the merge, update and output stages of aggregation by dequeuing each time the first tuple of each queue in order to make a decision on which tuple should be processed next.

5.3 Myriad2 Hardware Platform

The complete specifications of Myriad2 Movidius processor can be found in Deliverable D4.2 [?]. We recall here important information needed in our context.

The SHAVEs are processors designed for handling efficiently VLIW instructions, containing registers and functional units that enable SIMD operations and provide high parallelism

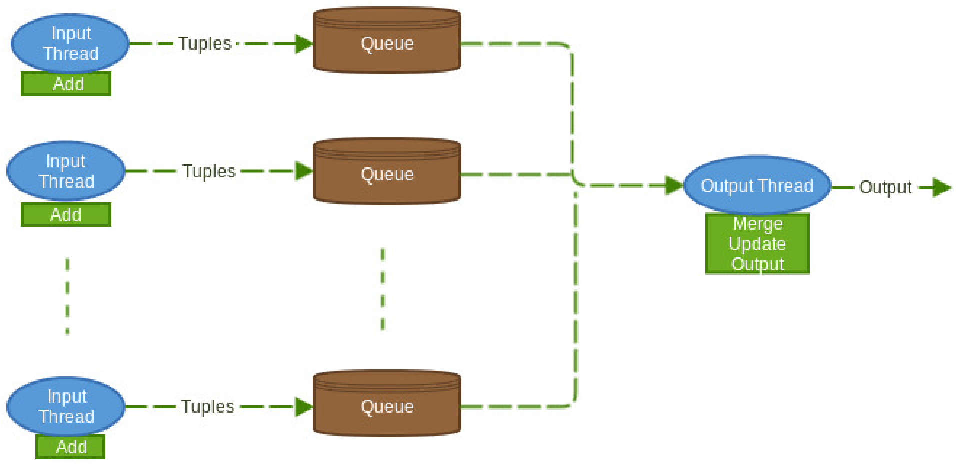


Figure 41: Aggregation example

and throughput. Used for performing intensive computational tasks. SHAVES also have access to a 2KB L1 instruction cache which enables running code residing in DDR without much delay and a 1KB L1 data cache. Additionally shaves have access to a 256KB L2 cache memory following a write-back policy and featuring a 64-byte cache line. Applications on SHAVES can be written in Assembly or C/C++.

CMX memory allows LEON and SHAVE processors to have low memory access cost in contrast to DDR memory which depending on whether retrieved data are already cached or not may incur high cost. For this reason CMX should be preferred for application data manipulated by SHAVES.

CMX is divided in sixteen 128KB parts. Each SHAVE has a preferential port attached to one of these parts, thus $128 \times 12 = 1536$KB are preferentially used by SHAVES. The remaining 512KB can be used for other general purposes such as LEON_OS timing critical code.

A SHAVE may have access to any CMX part with the same cost. On the other hand the resources used for routing between different CMX parts are finite. Furthermore an access to a local CMX slice happens with lower energy consumption and in order to achieve optimal performance, memory should be manipulated in such a way so that a SHAVE mostly accesses data residing in its local part.

There exists a cache coherency protocol that handles the movement of data between the DDR memory and the L1 and L2 caches of every SHAVE. However, due to the fact that using DDR is much slower, CMX should be preferred. As CMX is not cached, cache coherency is not an issue any more so the overhead incured by cache coherency protocols is avoided. Furthermore all SHAVES have very fast access to all CMX memory.

In case DDR should be used (e.g too much data to be held by CMX) a better alternative would be to use the DMA engine which allows whole blocks of memory to be fetched from DDR to CMX much faster than when DDR is accessed directly and data get cached. The way DMA engine can be used is depicted in Figure 42.

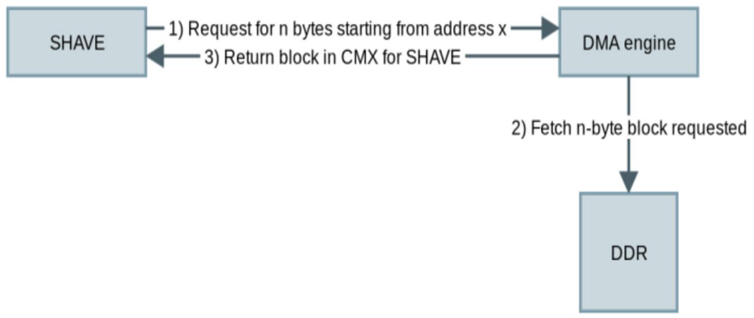

SHAVE
1) Request for n bytes starting from address x
3) Return block in CMX for SHAVE
DMA engine
2) Fetch n-byte block requested
DDR

Figure 42: DMA usage

5.4 Single-Producer-Single-Consumer algorithms

We present here some widely-used implementations of a FIFO queue, where a single entity can enqueue elements and another single entity can dequeue elements.

A bounded buffer with a fixed number of slots is used and the producer can only insert elements when the buffer is not full (in case the buffer is static) while the consumer may consume when the buffer is not empty. Furthermore the FIFO property needs to be guaranteed, through which the elements removed by the consumer appear in the same order as inserted by the producer.

In this case synchronization is a less complicated issue which can be overcome through primitive linearizable stores which are faster than hardware synchronization primitives. As CAS is not needed and there is only one producer and one consumer, the ABA problem, which is a factor contributing to the previous algorithms being slower, is not taken into consideration anymore.

5.4.1 Lamport's algorithm

In 1983 Lamport introduces a lock-free algorithm that solves the SPSC problem [70]. The approach uses a cyclic array as the data structure, is pretty straightforward and is depicted below in Listing 1.

Listing 1: Lamport's algorithm

```
int head = 0;
int tail = 0;
enqueue (data) {
    while (AFTER(tail) == head); // wait, queue is full
    queue[tail] = data;
```

```
        tail = AFTER(tail);
        return 0;
    }
    dequeue (data) {
        while (head == tail); // wait, queue is empty
        data = queue[head];
        head = AFTER(head);
        return 0;
    }
}
```

At the software level the proposed solution avoids explicit synchronization between the processes, which communicate indirectly through atomic read and write operations on control variables, namely head and tail according to Listing 1. On the other hand, the algorithm does not take into consideration memory and cache coherency, a fact which renders the solution the slowest one compared to the algorithms presented in this section.

Due to the fact that the control variables are shared, there are a lot of cache line transfers induced as the producer and the consumer need to manipulate these variables (either read them or write them) at every operation, generating a lot of invalidations and cache coherency traffic. To make things worse, in case the control variables are not placed in separate cache lines, there is a possibility their values get invalidated even when they are not actually changed because of changes to a nearby memory location residing in the same cache line [59].

Furthermore, in case the producer and the consumer work at the same speed to nearby memory locations operations will be happening on the same cache line, obviously leading to even more invalidations and cache line thrashing. For this phenomenon to be avoided, the consumer should avoid manipulating elements in cache lines that will possibly change (see temporal slip in Section 5.4.2) [59]. Finally the approach does not work for systems running on weaker memory consistency models than sequential consistency and additional fence operations would be needed if this would be the case, introducing more overhead.

5.4.2 Fast Forward algorithm

In 2008 Giaconomi *et al.* [40], design an efficient algorithm in order to provide better throughput in a pipeline parallel application where packet processing is realized. In order for the algorithm to be applicable to some weaker memory consistency models a memory barrier needs to be placed before an enqueue takes place. A pseudocode of the algorithm is provided below in Listing 2.

Listing 2: Fast Forward algorithm

```
int head = 0;
int tail = 0;
enqueue (data) {
    while (queue[tail] != NULL); // wait, queue is full
```

```
        queue[tail] = data;
        tail = AFTER(tail);
        return 0;
    }
    dequeue (data) {
        do {
            data = queue[head];
          }while (data == NULL); // wait, queue is empty
        queue[head] == NULL;
        head = AFTER(head);
        return 0;
    }
}
```

The approach is array based again and manages to decrease the amount of cache thrashing, in comparison with Lamport's algorithm in Listing 5.4.1. This is achieved through a special element introduced, which indicates whether an array cell is or is not available for an operation to occur. By synchronizing the processes in that way, head and tail are manipulated only by the producer and the consumer respectively so no invalidations occur because of these variables.

By introducing this special element though, a constraint is imposed, due to the fact that the datatypes of elements that can be used are limited and the value used as a special element is not possible to be used by the application [72]. A solution for this issue the designers propose, is to have an additional array containing pointers indicating whether a slot is empty or not. In case this solution is adopted, the issue that arises is that one more access is necessary in order for an operation on the data structure to occur (i.e. one access to the array holding the pointers and one access to the array holding actual data) and the memory needs of the algorithm increase.

Cache thrashing may still occur though when the producer and the consumer operate on array elements residing in the same cache line. For this reason temporal slip is suggested by the designers, according to which the consumer is delayed for as much time as needed, in order for the producer to fill one cache line. At this point it has to be noted that as the algorithm is destined for pipeline parallel applications, in case stages of the pipeline differ significantly in duration, temporal slip needs to be checked more often, increasing the overhead incurred and eventually introducing cache line thrashing which the solution tries to avoid in the first place [59].

Compared to Lamport's algorithm, the approach proves to be approximately 3.7 times more efficient performance wise and exhibits more stable behaviour. The algorithm's performance does not seem to be affected in cases where memory fences and variations to the queue size, the workload, the core allocation or the pipeline stage duration are introduced. As far as temporal slip is concerned, the overhead is minimal (1 ns). Finally the algorithm scales well for processes that reside on the same or different chips.

5.5 Batch Queue algorithm

In 2010 Preud'Homme *et al.* [59], design an array based algorithm, which as the name implies, conscripts batch processing. The algorithm proves to be fast and takes into consideration memory usage and cache coherency. A pseudocode demonstrating the logic of the algorithm is given below in Listing 3.

Listing 3: Batch queue

```
int enq_index = 0;
int deq_index = 0;
bool isFull = false;
enqueue (data) {
    queue[enq_index++] = data;
    enq_index = enq_index mod (2*N) // N equals half size of the array
    if (enq_index mod N == 0) { // half queue filled
        while (isFull);
        isFull = true; // allow consumer to consume
    }
}
dequeue (data) {
    int i;
    while(!isFull); // wait for producer to fill half queue
    for (i = deq_index; i < deq_index + N; i++)
        copy_buf[i - deq_index] = queue[i];
    deq_index = (deq_index + N) mod (2*N);
    isFull = false; // notify producer half array is emptied
}
```

The algorithm processes a whole batch of elements at a time. The queue is split in two sub-queues where either the producer or the consumer works at a time. After the producer has finished enqueuing in one sub-queue it signals this to the consumer who can now dequeue the elements of the sub-queue while the producer keeps enqueuing to the other sub-queue.

Synchronization is achieved through one shared flag (the isFull variable in Listing 3) in order for the swapping of the sub-queues to happen. When the producer finishes with filling one sub-queue, it sets the flag in order to notify its part is full. Subsequently it waits for the consumer to signal it emptied the other half of the queue by unsetting the flag, notifying it is finished dequeuing and the producer can start filling the sub-queue again .

In order to achieve optimal performance two full cache lines are needed and the two variables holding the indexes along with the flag, need to be placed in different cache lines, otherwise a lot of unnecessary invalidations and cache coherency traffic will take place decreasing performance at a substantial level. The flag ensures the two processes work in different cache lines, playing the role temporal slip does in Listing 2.

The algorithm provides high throughput and has small memory needs. On the other hand it exhibits increased latency due to batch processing which increases overhead and requires high level of communication through a shared flag in order to work. So the algorithm is suitable for applications with high L1 cache usage and relaxed latency requirements.

Performance wise an improvement of up to 10 times is observed in relation to Lamport's algorithm and a word sized data element can be sent through the queue within 19 − 125 cycles and 12.5 − 40.6 nanoseconds depending on L1 cache pressure.

5.6 MCRingBuffer

Again in 2010, Patrick PC Lee *et al.* [72], propose an algorithm, aiming to keep up with the bandwidth of the communication link for packet processing purposes (line rate packet processing). The algorithmic design is presented below in pseudocode 4.

Listing 4: MCRingBuffer

```
/* Variable definitions */
char cachePad0[CACHE LINE];

/* shared control variables */
volatile int read;
volatile int write;
char cachePad1[CACHE LINE - 2 * sizeof(int)];

/* consumer local variables */
int localWrite;
int nextRead;
int rBatch;
char cachePad2[CACHE LINE - 3 * sizeof(int)];

/* producer local variables */
int localRead;
int nextWrite;
int wBatch;
char cachePad3[CACHE LINE - 3 * sizeof(int)];

/* constants */
int batchSize;
char cachePad4[CACHE LINE -  sizeof(int)];

/* Enqueue function */
enqueue (data) {
    int afterNextWrite = AFTER(nextWrite);
    if (afterNextWrite == localRead) {
        while (afterNextWrite == read); // wait queue is full
        localRead = read;
    }
    queue[nextWrite] = data;
    nextWrite = afterNextWrite;
    wBatch++;
    if (wBatch >= batchSize) {
        write = nextWrite;
        wBatch = 0;
    }
    return SUCCESS;
}

/* Dequeue function */
dequeue (data) {
    if (nextRead == localWrite) {
        while (nextRead == write); // wait queue is empty
        localWrite = write;
    }
    data = queue[nextRead];
    nextRead = AFTER(nextRead);
    rBatch++;
    if (rBatch >= batchSize) {
        read = nextRead;
        rBatch = 0;
    }
    return SUCCESS;
}
```

The algorithm is lock-free and tries to take advantage of cache locality in order for access in shared data to become more efficient. As it is shown in pseudocode 4, control variables are carefully laid out in memory, separated by paddings wherever needed in order for cache invalidations to be avoided. The approach of placing variables in memory in such a way is called *cache line protection*.

By employing this strategy false sharing between the producer and the consumer is avoided. As a consequence, when one of the processes invalidates a whole cache line by changing one variable, the variables local to the other one are not affected as it is ensured that they reside in different cache lines.

Aside from cache line protection which reduces cache coherency traffic, another important design choice with impact on the performance of the algorithm is the *batch updates* of shared control variables. Through the local variables to each process, the shared control variables (read and write in the pseudocode provided) are sparsely changed. Each process consults its variable that holds what the process thinks about the progress of the other process and only reads the shared variable holding the actual progress of the other process when no more progress can be made. The shared variables are updated by each process after a predetermined number of steps which separate the buffer in smaller parts.

As it is obvious this approach of batch updates can lead to wrongly not inserting data to the buffer even if the buffer is not full (and not dequeuing data from the buffer even if data are available) due to the fact that the shared variables are not updated yet. This may happen when the incoming rate of data elements is too small. In order for this scenario to be avoided, the authors suggest that the producer periodically inserts unused elements to the queue, discarded by the consumer, used to make the shared variables be updated and have progress. By making use of batch updates, the same effect as in the temporal slip approach (see Section 5.4.2) or the flag used in BatchQueue (see Section 5.5) can be achieved without needing any scheduling of the producer and the consumer.

As control and data elements are not merged like in the fast forward algorithm, the approach supports generic datatypes and does not impose any limitation on the datatypes than an application may use, rendering it completely independent. On the other hand, compared to the other algorithms, the memory needs are increased as six variables are used by the processes in order to synchronise. The approach works with systems supporting the sequential consistency memory model while in order to be adopted to platforms with weaker memory consistency, memory barriers need to be used.

The algorithm outperforms Lamport's approach especially when batch updates are introduced. The same holds for the fast forward algorithm for which though temporal slip has not been used. It does not seem to perform well in case the buffer size is small as it is probable that the producer cannot enqueue elements. In general, through this approach, throughput of conventional lock-free solutions is improved by up to 5 times and cache misses are considerably reduced. Furthermore processing throughput is augmented by up to 5.2 times for the single-threaded case and 1.9 times for the multi-threaded case.

5.7 Algorithms Evaluation

Due to constraints of the Myriad2 platform (not providing CAS functionality) and the pipelined nature of the aggregation phase of a data streaming application, the algorithms this evaluation focuses on are the four SPSC algorithms that have already been discussed in the previous section. From now on FastForward is mentioned as FF, BatchQueue as BQ and MCRingBuffer as MCR. The performance evaluation of these algorithms along with Lamport's design, is conducted in order to get an insight on the way they behave on the platform.

In general, in order to take advantage of the platform specifications and achieve optimal performance, issues already discussed in Section 5.3 are taken into consideration. Thus, if possible, queues are placed in CMX (particularly in the CMX slice of one of the SHAVES that manipulate the queue) and each shared variable is placed in the CMX slice of the SHAVE that makes changes to it. If the queue size exceeds CMX capacity, the queue is placed into DDR.

Leon OS boots and starts Leon RT from which it awaits a signal in order to initiate a task that performs the energy measurement of the platform. Leon RT sends this signal right before it turns on the SHAVES used for the evaluation. After signaling the beginning of the power measurement it starts measuring execution time and turns on the SHAVES. Once the SHAVES finish their executions Leon RT stops measuring the execution time and signals to Leon OS to stop the power measurement.

The first two SHAVES (SHAVE0 and SHAVE1) are used for the evaluations discussed in this section. SHAVE0 plays the role of the producer while SHAVE1 runs the application of the consumer. In case the queue used for communication is placed in CMX it is stored in the slice of the producer (SHAVE0).

For the purposes of the evaluation, by one operation a pair of enqueue/dequeue is meant and the percentages used for comparison, provide an approximation of the differences between the algorithms resulted from the average calculation over all different cases studied. For each case, an algorithm is evaluated five times and the average of these evaluations is used as the result in the final computation of the percentages provided.

5.7.1 Buffer Size Evaluation

The behaviour of the algorithms depending on the size of the queue at question is examined in order to investigate how performance is affected.

In the following evaluation in case the buffer size is greater than 2048 elements the queue is placed in DDR memory as CMX's capacity is not enough. Furthermore DMA engine is not used and data are directly accessed from DDR.

Prior to running the tests the buffers are already filled with data. In case buffer capacity is 128 elements, the buffer is half filled while for all the other occasions buffers are filled with 150 elements. In this way temporal slip is achieved (producer begins 128 or 150 elements ahead of the consumer) for all the algorithms as the producer and the consumer are working on different cache lines when buffer is placed in DDR.

Batches are used by BQ (by design batches are equal to half the size of the queue) but not for MCR. By not applying batches to MCR it means that the shared variables are updated at every operation but still they are sparsely consulted by the processes (every 128 or 150 operations). Lamport and FF do not incorporate batches in their implementations by design.

Because of the nature of CMX memory, temporal slip does not need to be taken into consideration when queues are placed in CMX as cache line thrashing is not an issue anymore. Each element of the queue is 12 bytes (evaluation on the impact of the size of a data element is investigated later in Section 5.7.3).

BQ proves to provide better throughput than any other algorithm for every single occasion (Fig. 43). Specifically, the algorithm achieves better throughput by 24% than MCR, 40% than FF and 42% than Lamport. This leads us to the conclusion that processing data in batches and not immediately when they are available provides much better performance. As the producer and the consumer are constantly working independently, needing to synchronize only when they are done with their batch (a synchronization made through just a single variable), a much more efficient design is achieved compared to all other algorithms which perform checks on synchronization variables much more frequently.

On the down side of using batches, in case the queue is placed in DDR, it seems that BQ is the only algorithm whose throughput is affected as the size increases. This happens because the bigger the queue the more the latency gets increased. A SHAVE has access to a 1 KB L1 data cache. As a consequence performance is affected by how long data can be kept in the L1 cache. As cache misses occur, processing delay is augmented and due to the fact that long batches are used (half the size of the queue) the processing of the tuples gets slowed down as well. As a consequence it can clearly be seen that throughput is decreased as the size of the queue increases. For all other algorithms that do not make use of batches, throughput does not seem to be affected.

It also needs to be noted that the input rate of elements is constant and the producer runs at the same speed as the consumer. This is not the case though for the different stages of aggregation that we investigate and it would not be surprising if the results shown here would not apply later under the new circumstances.

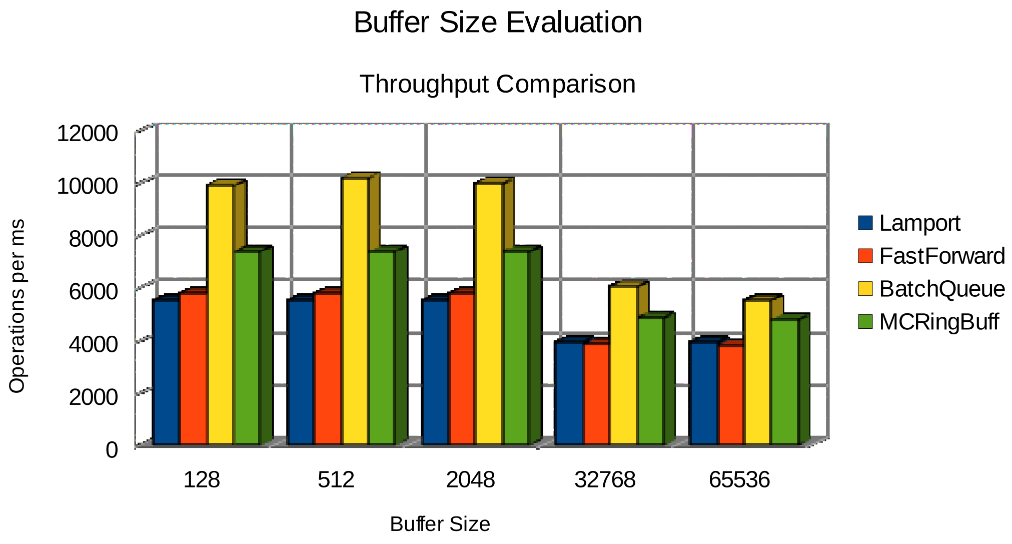


Figure 43: Throughput comparison depending on the size of the queue used

Additionally it can be seen that FF is less efficient than any other algorithm when the queue is placed in DDR. It performs even worse than Lamport's algorithm even though it surpasses it when the queue is placed in CMX. This happens as FF couples control with data elements (i.e. a data element is checked in order to decide whether an enqueue or a dequeue should happen). While all other algorithms synchronize processes through variables residing in CMX, FF needs to access DDR or a cache in order to synchronize the producer and the consumer, rendering it slower.

Although FF is slower than Lamport when queue is placed in DDR it is more energy efficient by a slight difference although this is not always the case when the queue resides in CMX. For all the test cases, the differences between the two are minor as when CMX is used no caches are involved for cache thrashing to occur from Lamport and when DDR is used and caches are involved, due to the fact that the queues are already filled with data prior to execution, Lamport avoids cache thrashing as well.

BQ also prevails in the energy consumption evaluation, constituting the most energy efficient solution (Fig. 44). It is more energy efficient by a percentage of 23% from MCR and 43% from Lamport and FF. As for the throughput evaluation earlier, it can be seen that energy efficiency is affected by the size of the queue when it resides in DDR for the same reason that throughput is affected as well. Cache misses mean slower processing of tuples thus more busy waiting by the processes and consequently increased energy needs. The rest of the algorithms that do not make use of batches do not seem to be affected and demonstrate a stable behaviour either when CMX or DDR is used.

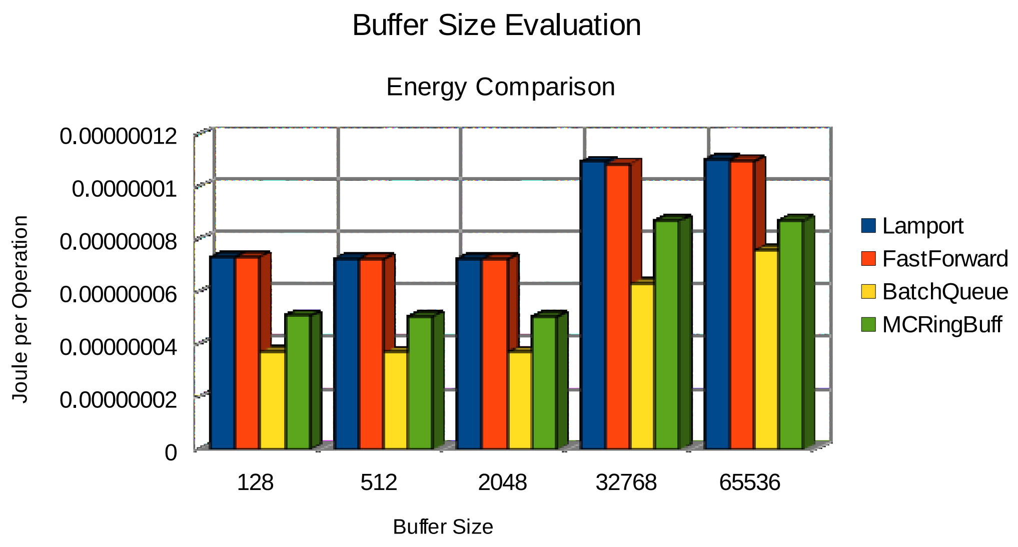


Figure 44: Energy comparison depending on the size of the queue used

Overall, with the exception of BQ when the queue is placed in DDR, the algorithms exhibit a stable behaviour on the platform that is not affected by the size of the buffer used for communication. Furthermore the inferiority of DDR over CMX is clearly shown as throughput decreases when queues are placed in DDR while energy consumption increases. BQ seems to be influenced the most when the queue is placed in DDR as its throughput is decreased by 42% while its energy consumption is increased by 47%. On the other hand Lamport is the algorithm affected the least, as its throughput is decreased by 29% while its energy needs are augmented by 34%. Throughput for FF and MCR is affected at a similar degree (around 34% slower) while for the former, energy needs increase by 33% and for the latter energy consumption increases by 42%.

5.7.2 DMA engine test

Subsequently DMA engine is used when queues are placed in DDR in order to decide whether it is suitable to use it for the algorithms tested, compared to accessing directly data from DDR. The way DMA is designed to be used is already discussed in Figure 42.

Tests were run for FF and MCR in order to decide what kind of applications DMA engine favours the most (batch updates or single element update).

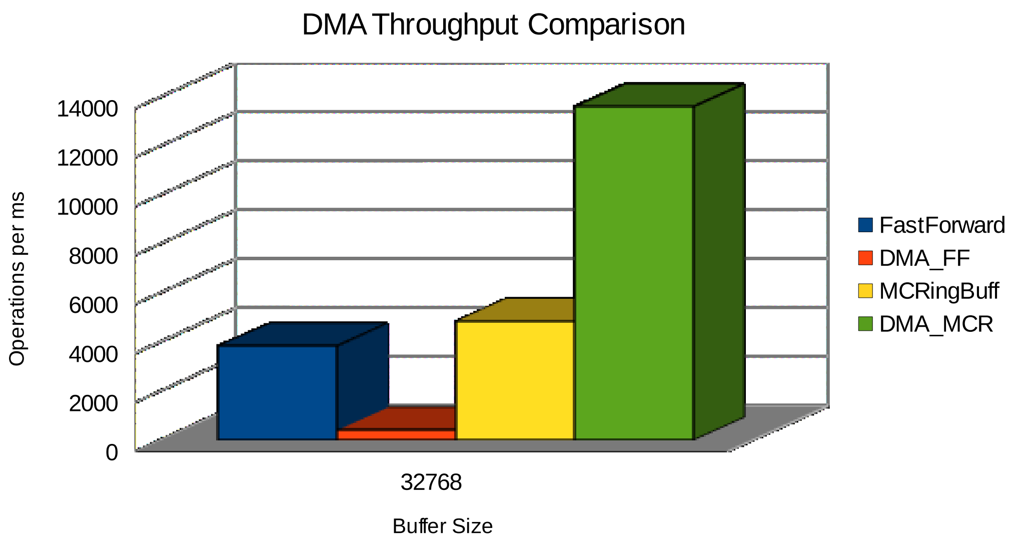


Figure 45: Throughput comparison when DMA used

DMA is not practical when used for single data element access (i.e Lamport, FastForward), as it is invoked every single time an element needs to be read in order to decide whether an enqueue or a dequeue should happen (FastForward). This is a use case for which DMA is not destined to be used in the first place and the results provided in Figures 45 and 46 enforce this fact. FF algorithm's throughput is decreased by 89% when DMA is used while energy consumption is increased by 91%.

On the other hand when batches of elements are fetched (for MCRingBuffer where 128 elements are fetched at a time), MCR design using DMA outperforms the case where data are accessed directly from DDR, performance and energy wise (Fig. 45 and 47). Throughput is increased by 66% while energy needs are diminished by 70%. The larger the blocks requested by DMA are, the more efficiency is increased.

5.7.3 Data Element Size Evaluation

The performance of the algorithms is evaluated depending on the size of a data element they store. For these tests the buffer size is fixed to 128 elements and buffers are half filled prior to execution. Buffers always reside in CMX (the producer's slice) so temporal slip is not needed as the memory manipulated is not cached. Batches are only used by BQ as it incorporates them by design (a batch equals half the capacity of the queue thus 64 elements for these experiments). Batches are not used for MCR.

Once more BQ is superior than all other algorithms, MCR comes second, with FF and Lamport following (Fig. 48). On average, BQ provides performance enhancement of the size

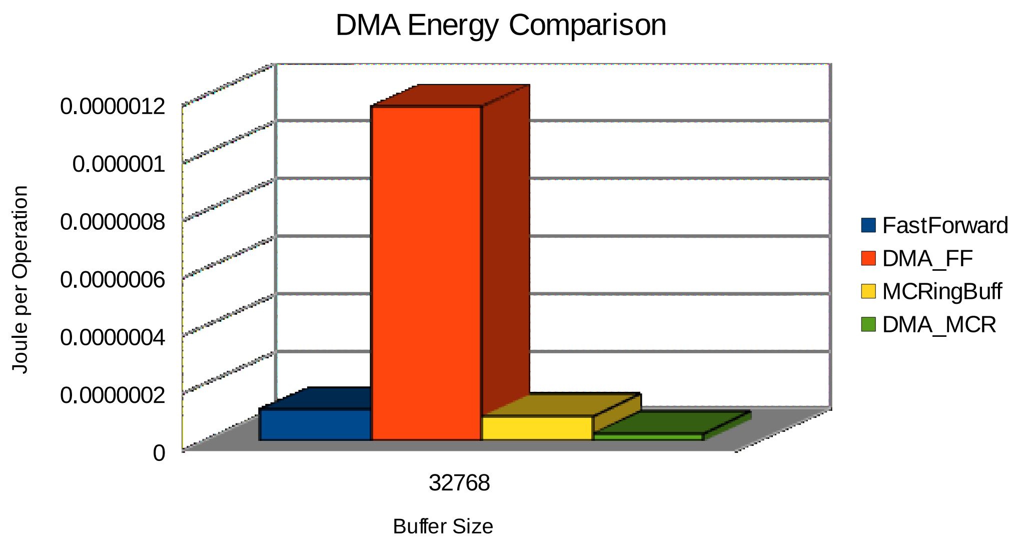


Figure 46: Energy comparison when DMA used

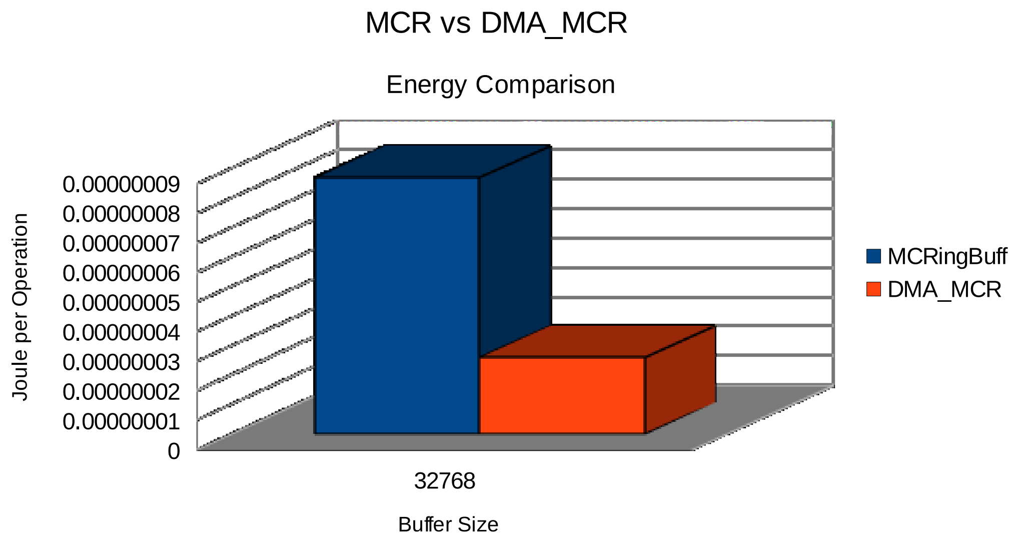


Figure 47: Energy comparison between MCR and DMA_MCR

of 27% compared to Lamport, 21% compared to FF and 16% compared to MCR. It is obvious that as the size of a data element increases performance of the algorithms converges as more calculations are performed before operations take place. As a consequence differences in performance are smaller compared to the buffer size evaluation previously.

With respect to energy, FF is the only algorithm that seems to be affected as the element size increases. Until the 64-byte element test, it is more energy efficient than MCR something

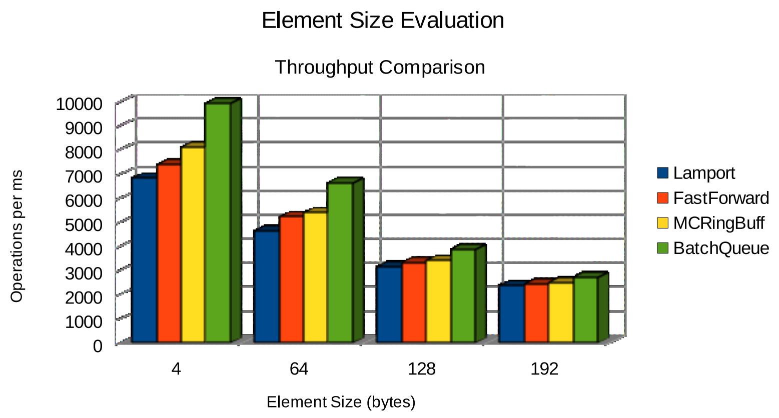


Figure 48: Throughput comparison for variable data element size

that does not hold for the 128-byte element test with the margin increasing even more for the 192-byte element test. All other algorithms exhibit stable behaviour as the element size increases (Fig. 49). In particular BQ is more energy efficient than Lamport by 21%, FF by 15% and MCR by 12%. Again the small difference in energy consumption comes from the fact that the algorithms converge as the size of the data elements increases.

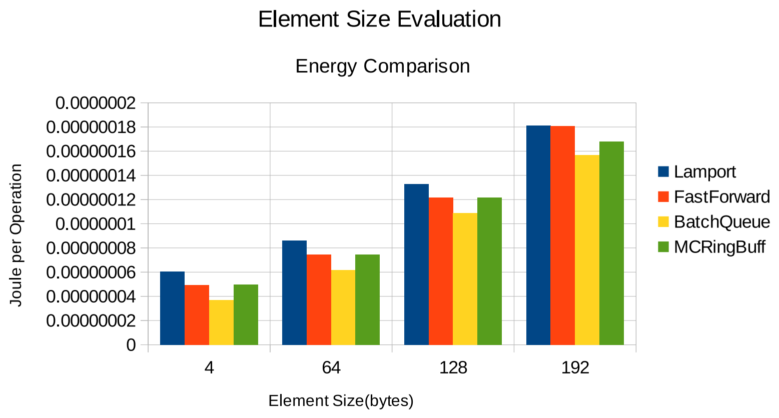


Figure 49: Energy comparison for variable data element size

5.7.4 Algorithms evaluation as the number of tuples to be processed increases

The processing speed and energy consumption of the algorithms is compared as the number of elements that need to be processed by the system increases. Buffer capacity is fixed to 128 elements while a data element is 12 bytes. As for the previous evaluation, batches are only incorporated in the BQ implementation (64 elements batches). Again the queue resides in the CMX slice of the producer, thus temporal slip does not constitute an issue for these tests.

As it is expected from previous evaluations, BQ proves to be the fastest algorithm in this evaluation as well (Fig. 50). MCR follows next, with FF and Lamport coming third and fourth respectively. BQ exhibits better throughput than MCR by 26%, FF by 42% and Lamport by 45%.

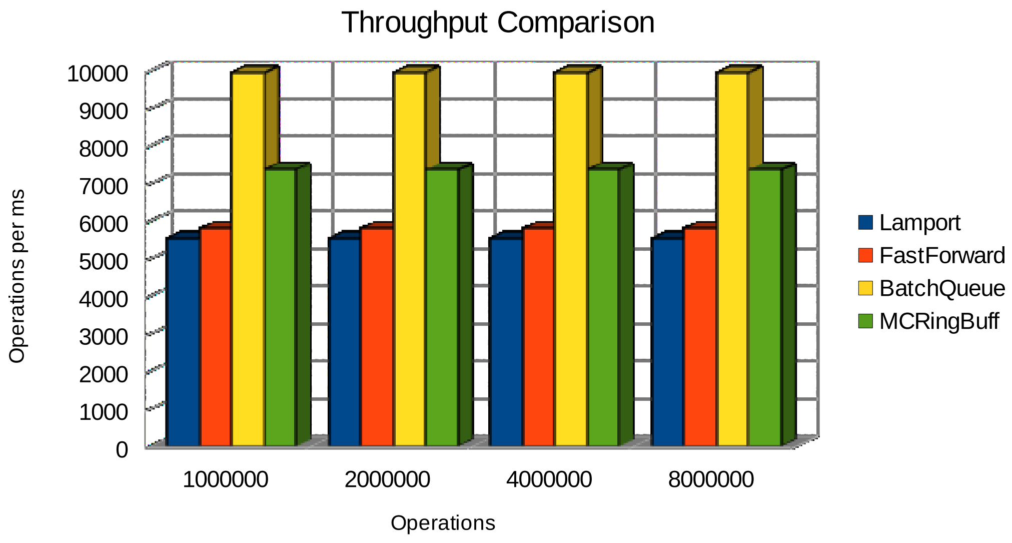


Figure 50: Throughput evaluation as the number of tuples in the system increases

The same holds for the energy footprint of the algorithms, where BQ is more efficient than MCR by 29%, FF by 43% and Lamport by 46% (Fig. 51).

5.8 Data streaming Aggregation Evaluation

The algorithms evaluated in the previous section are now used between the stages of a data streaming aggregation process in order to decide on their suitability and performance under the new circumstances. As previous evaluation has shown that use of data structures is most effective when they reside in CMX, throughout this evaluation the data structures are placed in CMX in order to benefit from a simpler and more efficient model of communication.

The same process described at Section 5.7 is used for execution time and energy measurements while all twelve shaves are used for the tests performed. The queues connecting

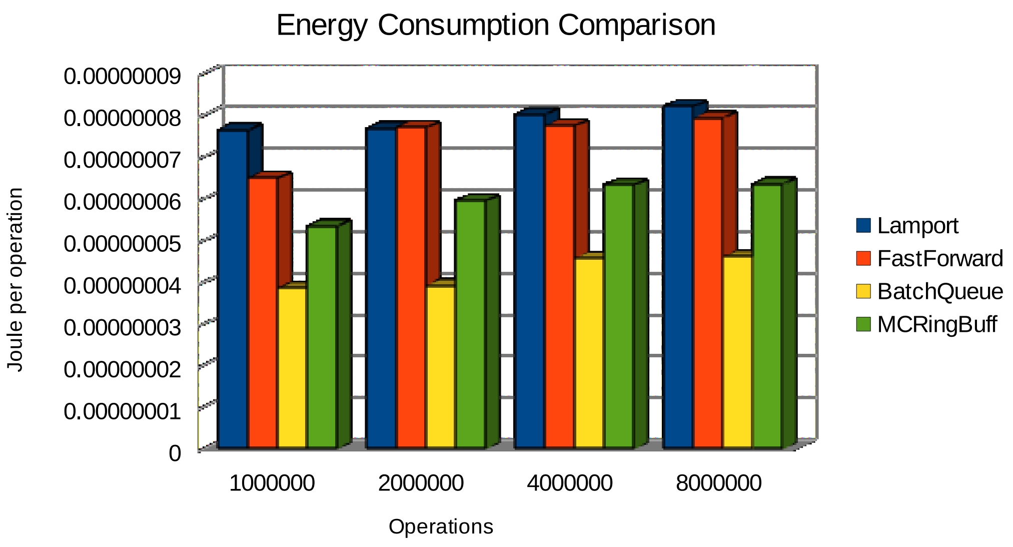


Figure 51: Energy evaluation as the number of tuples in the system increases

an aggregator with producers and the final aggregator can store 128 elements and they are not filled with elements prior to execution. Batches are used only for the BQ case.

5.8.1 Approach

The aggregation process simulates smart metering data that have already been merged and sorted by a node which plays the role of the producer of the tuples that will be emitted to aggregators. The producer constitutes the first stage of the process.

The tuples produced contain a timestamp and a value representing the amount of Watts that have been reported by a smart meter. Once aggregators receive the tuples, they aggregate the values obtained according to timestamp. The aggregators constitute the second stage within the process.

The time is divided in windows of four time units with two time units of advance. Windows containing the aggregated values are reported to a final aggregator who reports the total sum of each window and represents the third and final stage of the process.

The communication between the different stages of the process and their functionality is implemented according to the following description :

Producer-Aggregator Communication

A producer communicates with each aggregator through a dedicated SPSC queue. It waits until all the tuples sent have been received and once notified through a shared variable that everything has been consumed, it informs aggregators that production is finished in order for them to report last windows that have not yet expired.

Aggregators - Last Aggregator Communication

The aggregators while consuming tuples, report for windows expired to the last aggregator through a SPSC queue (same type of queue used by a producer to communicate with an aggregator). They count how many messages are sent over to the last aggregator in order to inform him when he is finished processing everything that has been sent to him. Once this is true they state their inactivity to the last aggregator.

Last Aggregator
The last aggregator, aggregates values sent to him by every aggregator for each window. Each window in the aggregator's window list encompasses a contribution list stating which second stage aggregator has reported on the specific window.

In order for a window to be reported a list holding which aggregator is still active or inactive is checked in order to decide whether there will be somebody else that has not yet reported on the window and will potentially do so.

The queues are checked in a round-robin fashion and the aggregator finishes only when all second stage aggregators are inactive (which means everything has been processed). Once everything has been processed, final windows that may have not yet been reported are eventually reported.

The algorithm that needs to be treated with special care in this type of application is the BQ. When using this data structure, it is likely that a consumer waiting for a producer to fill a batch, does not eventually consume everything when the producer is finished. Such an event may occur when a producer has not completely filled a batch. Thus a consumer will never be informed for leftovers residing in the last batch which has not yet been processed.

As a consequence a producer cannot wait for consumers to consume everything in order to inform them to make their final report like in all other algorithms. For this reason a flag is introduced by a producer which informs that leftovers exist for a consumer when the former states completion. Under this occasion, the index of the producer is consulted by a corresponding aggregator in order to know which data elements to consume. In case a producer has completely filled a batch before completion, it informs that no leftovers exist so an aggregator may proceed straight to the final report.

This could have been the case for MCR as well as it separates the buffer in multiple batches. As already discussed in the previous section though, when data structures reside in CMX, it is not so beneficial to incorporate batches in the algorithm. As a consequence shared variables are updated immediately and the above scenario is avoided.

5.8.2 Single producer variation

In this variation, one producer feeds tuples to ten aggregators in a round robin fashion and one final aggregator is used (Fig. 52). All processes are run on SHAVES. SHAVE0 acts as the producer while SHAVE1-SHAVE10 run the applications for the aggregators. SHAVE11 runs the application of the final aggregator. The data structures reside on the CMX slices of the second stage aggregators (i.e every aggregator holds the queues that make it communicate with the producer and the final aggregator).

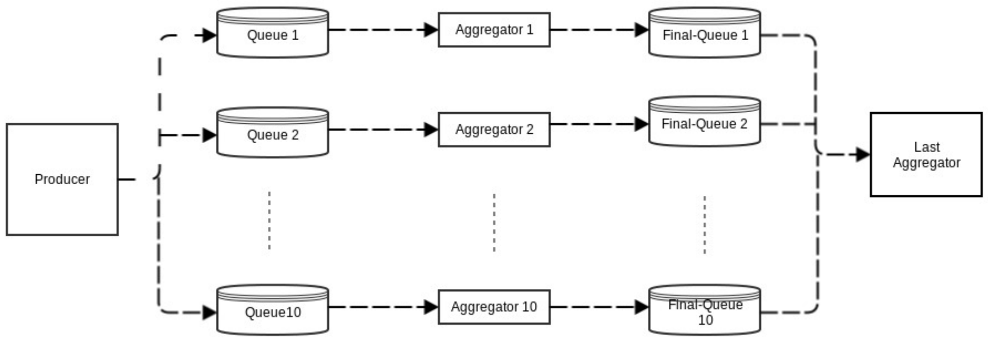


Figure 52: Single producer aggregation

When used within aggregation, BQ performs much worse than every other algorithm despite the fact that it is the most efficient in the evaluation of Section 5.7 (Fig. 53 and 54). Specifically MCR provides better throughput than all other algorithms, surpassing BQ by 73%, Lamport by 7% and FF by 5%. Energy wise MCR is the most energy efficient solution as well with BQ exhibiting the highest energy consumption needs. BQ needs 71% more power than Lamport, 72% more power than FF and 76% more power than MCR.

Superiority of MCR holds due to synchroniztion being more efficient as shared variables are consulted more sparsely than in other algorithms. The producer, as both of the processes start from index zero, will consult the shared variable manipulated by the consumer only when he fills the buffer and from then on every time it reaches the index at which it thinks the consumer is currently working. The consumer also consults the shared variable manipulated by the producer more sparsely as the producer is faster and fills some elements before the consumer finishes the aggregation algorithm (producer only executes a for loop while a

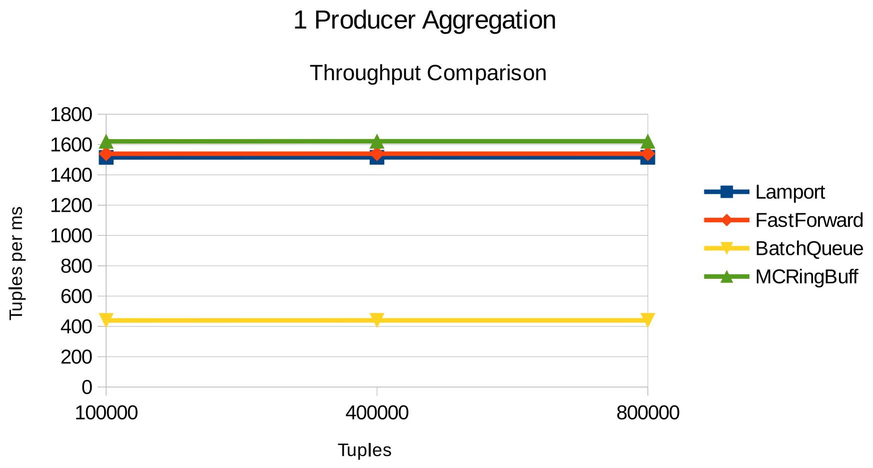


Figure 53: Single producer aggregation throughput evaluation

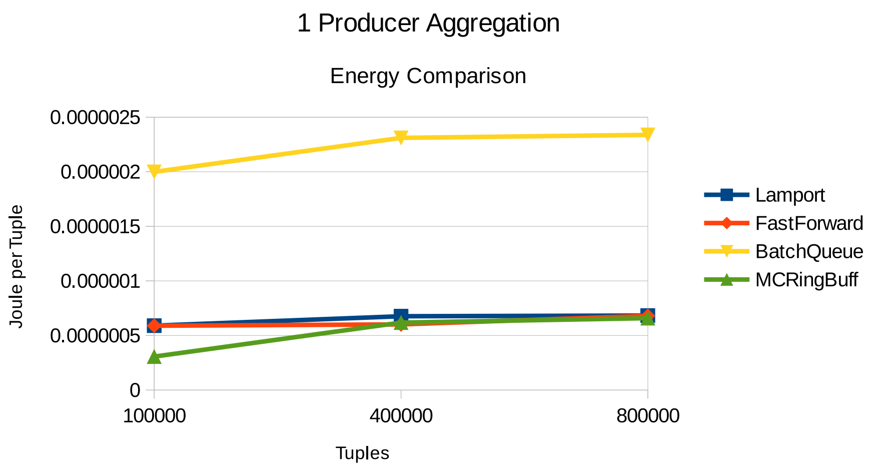


Figure 54: Single producer aggregation energy evaluation

consumer executes the aggregation algorithm).

Furthermore, the size of the queues used for communication seems to affect BQ's performance and energy consumption, as the bigger the capacity of a queue the longer the consumer should wait to consume and thus more busy waiting is involved. For this reason evaluation of differences between different sizes of queues is conducted in order to visualize its effect in the algorithm's performance in the context of aggregation.

As expected the capacity of the communication queues used have a major impact on the algorithm's performance (Fig. 55 and 56). When buffer size is doubled to 256 elements, a decrease by 43% is observed in throughput while energy consumption is increased by 46%.

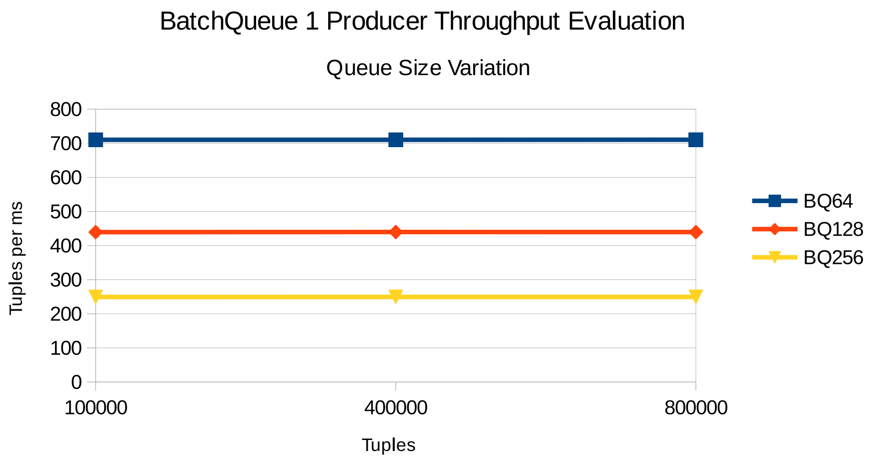


Figure 55: BQ buffer variation throughput evaluation

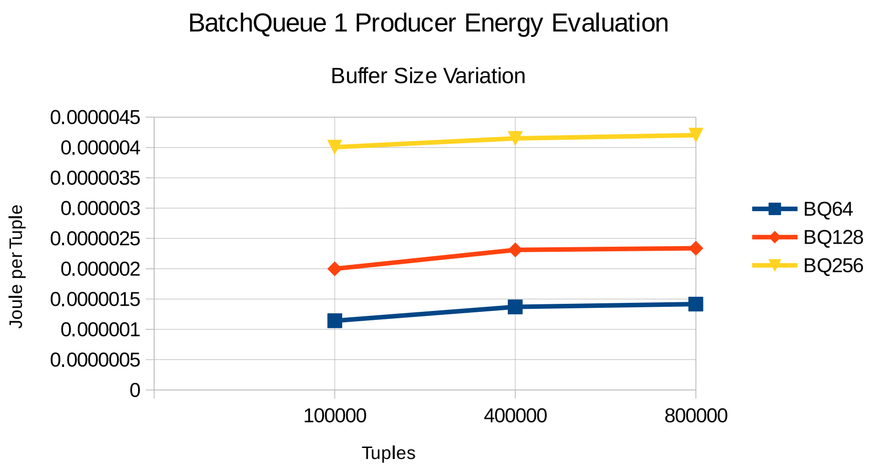


Figure 56: BQ buffer variation energy evaluation

On the contrary when buffer size for communication between the stages of the aggregation is reduced to half (64 elements), an increase in throughput by 38% is observed while energy consumption is decreased by 41%.

5.8.3 Three producers variation

In this variation as depicted by Figure 57, three producers feed tuples to eight aggregators which communicate their windows to a final aggregator. The same approach as for the single producer variation concerning the placement of the data structures in memory holds (see Section 5.8.2). Each producer feeds tuples to assigned consumers. In this way more pressure

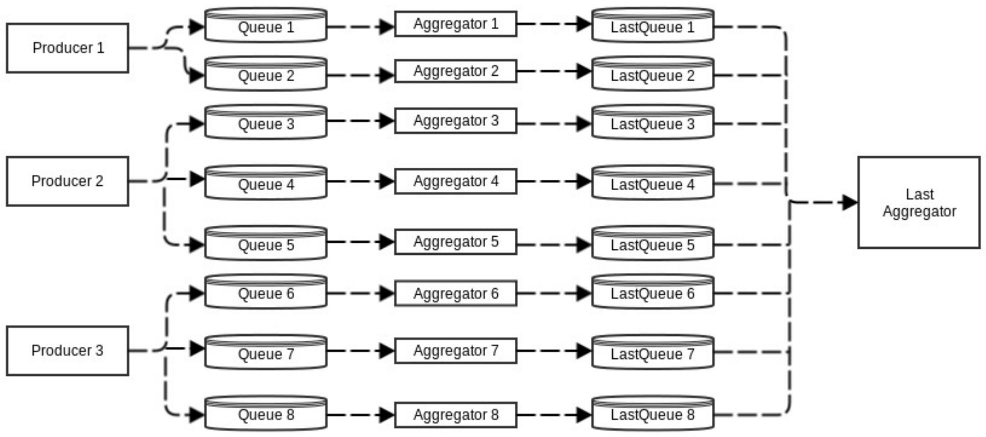


Figure 57: Three producers aggregation

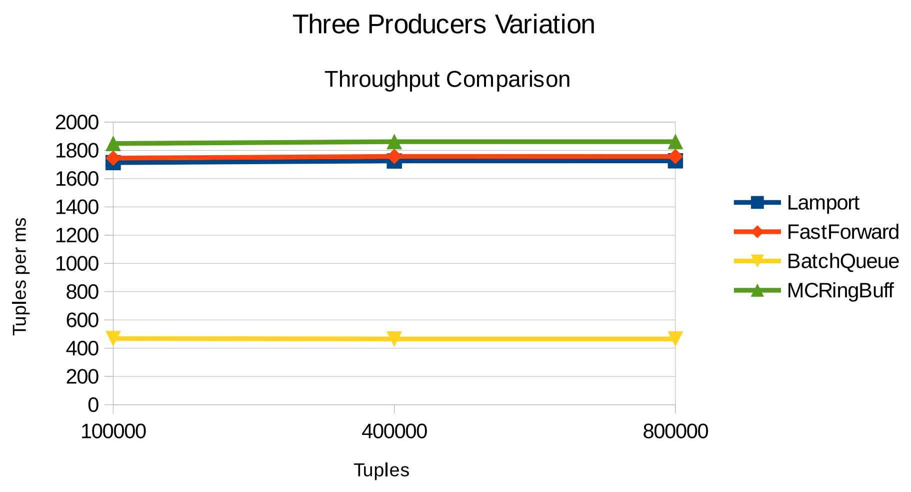


Figure 58: Three producers aggregation - throughput comparison

on the queues is achieved and the initial workload is equally shared among the SHAVES. Due to the fact that the producing phase of the process is faster than before, this approach is expected to provide better performance.

The pattern observed in Figure 58 is the same as for the single producer variation in Figure 53. The difference lies in that the algorithms, as expected, have become faster now that three producers are used. MCR is more efficient by 75% from BQ, by 7% from Lamport and by 6% from FF.

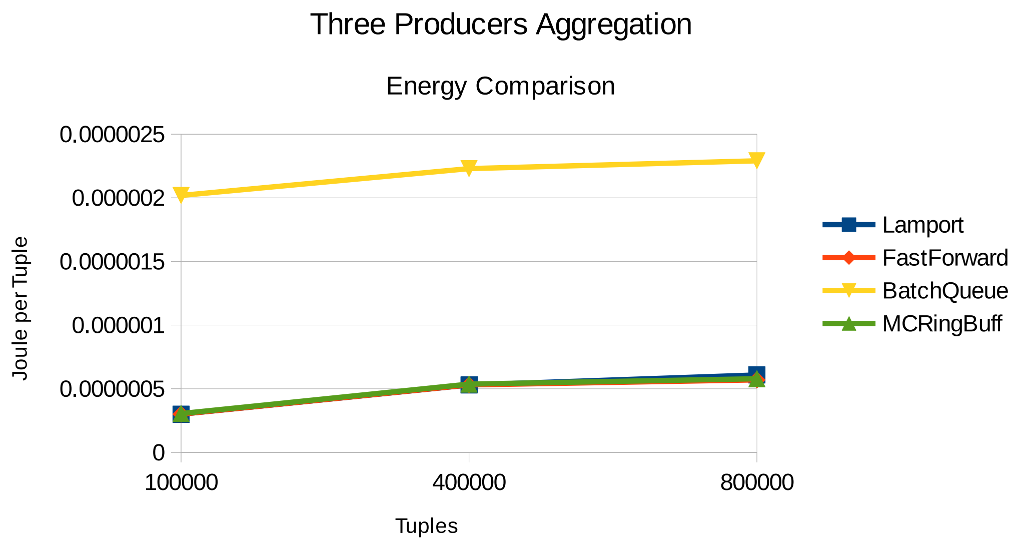


Figure 59: Three producers aggregation - energy comparison

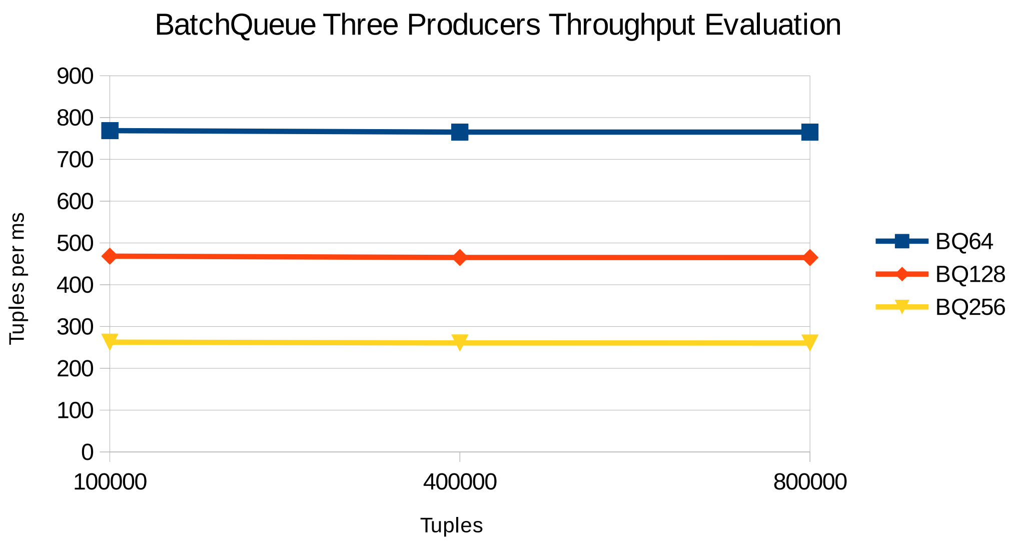


Figure 60: BQ buffer variation throughput evaluation

Energy wise, with the exception of BQ which exhibits 77% higher energy needs from all other contenders, the rest of the algorithms exhibit similar behaviour (Fig. 59).

As for the single producer aggregation, BQ's performance is dependent on the size of the queues used (Fig. 60 and 61). When buffer capacity is doubled, throughput is reduced by 44% while energy consumption is increased by 45%. On the contrary, when buffer size is

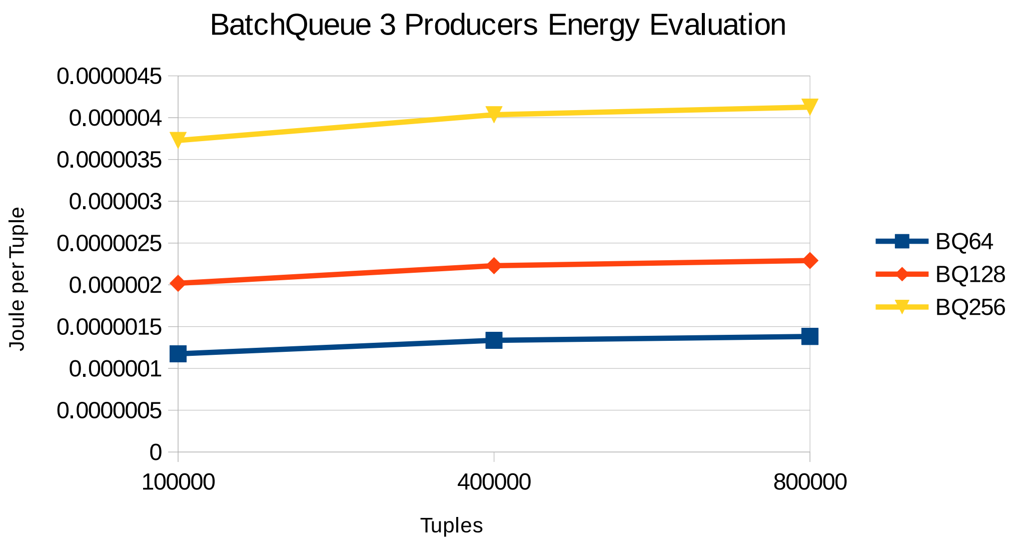


Figure 61: BQ buffer variation energy evaluation

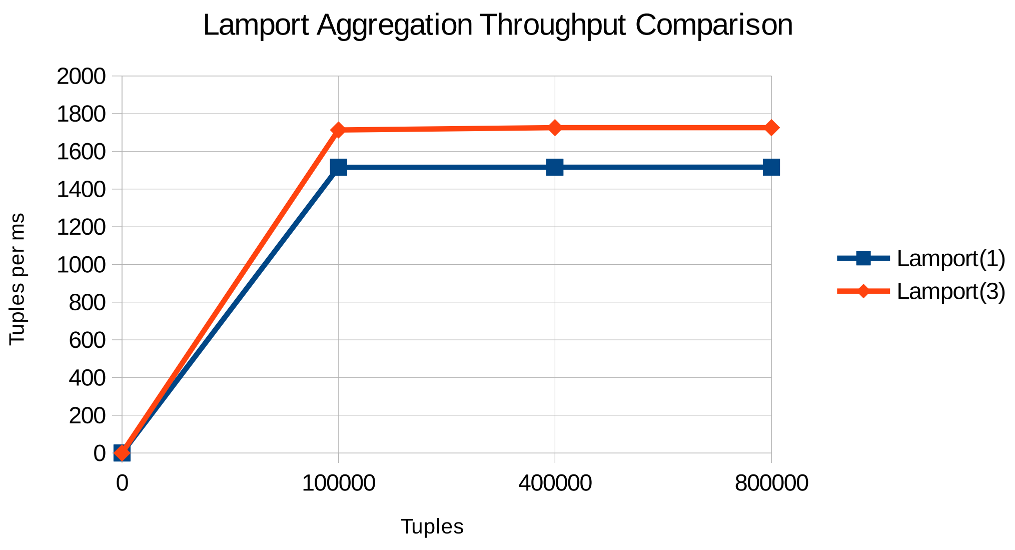


Figure 62: Lamport aggregation - throughput comparison

reduced to half, throughput is increased by 39% while energy consumption is decreased by 40%.

Figures 62, 63, 64, 65, 66, 67, 68 and 69 demonstrate the improvement gained through the second variation for every algorithm separately. The name of the algorithm followed by the number one, corresponds to the evaluation of the algorithm for the single producer

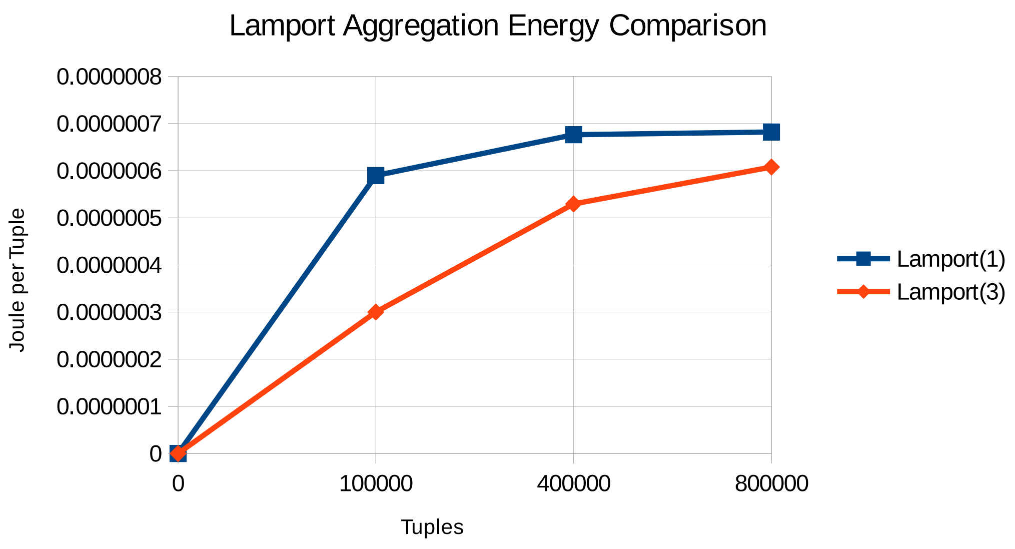


Figure 63: Lamport aggregation - energy comparison

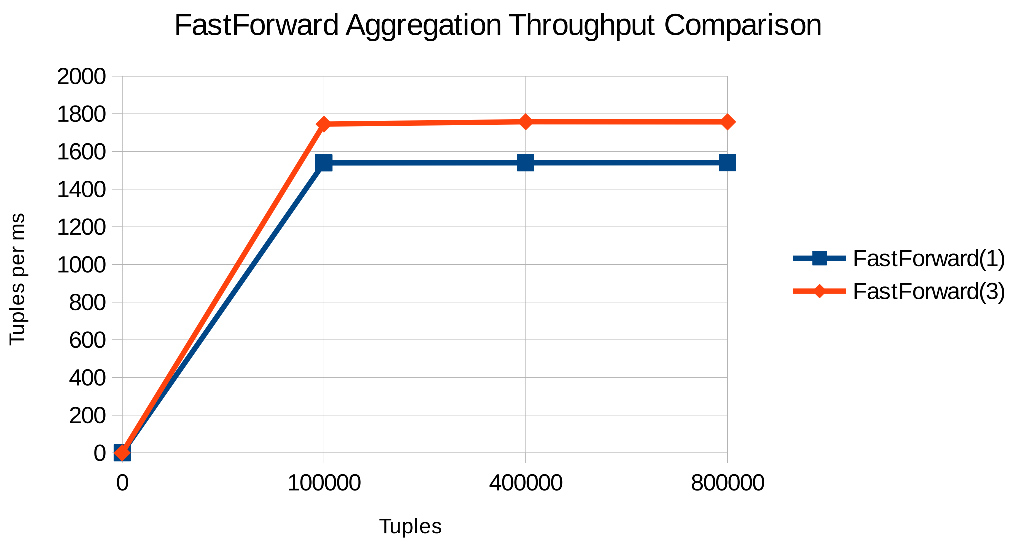


Figure 64: FF aggregation - throughput comparison

aggregation. The name of the algorithm followed by the number three, corresponds to the evaluation of the algorithm for the aggregation where three producers are used.

BQ's performance and energy consumption do not have that much big of a difference for the two different scenarios. When using three producers, although tuples may be communicated faster to and from the second stage aggregators at the beginning, the pace of the last

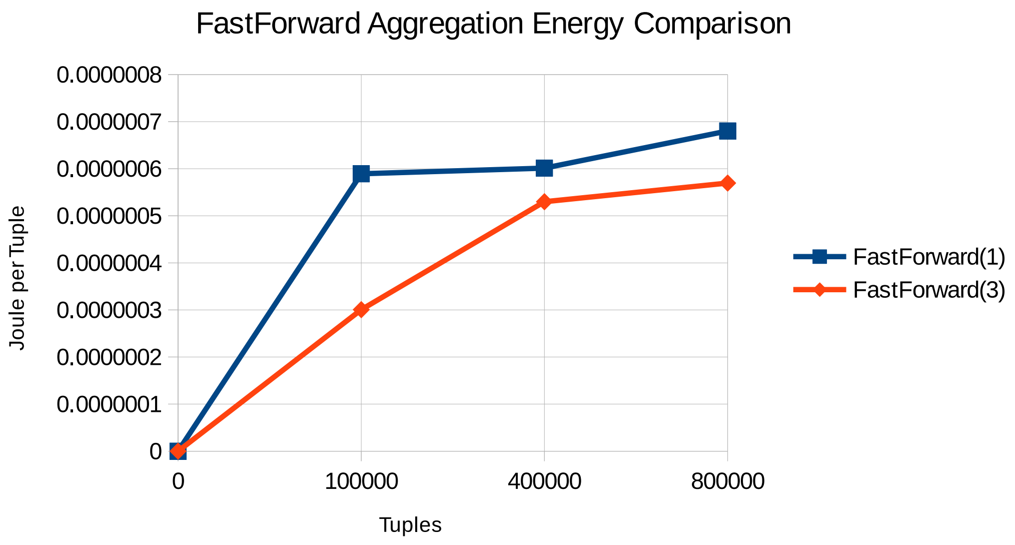


Figure 65: FF aggregation - energy comparison

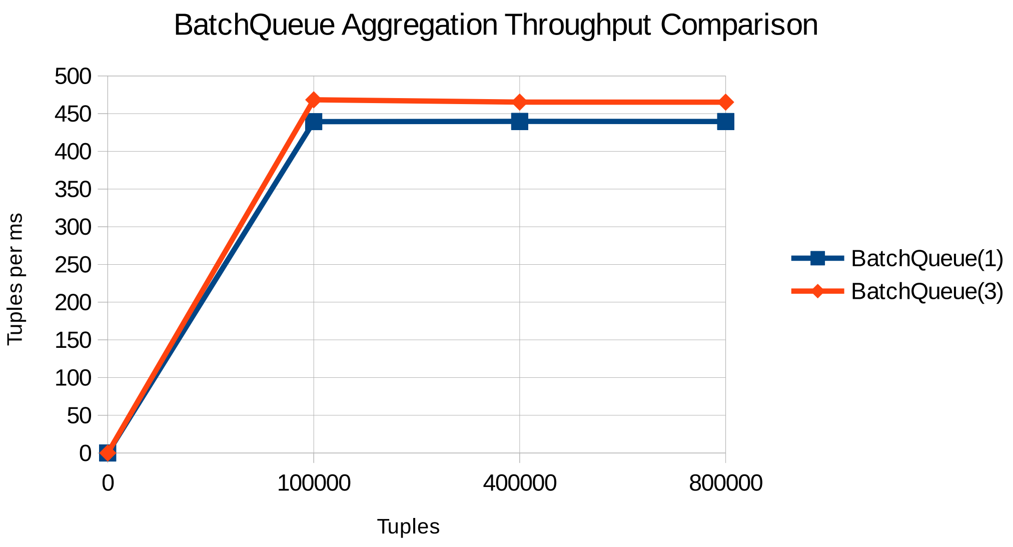


Figure 66: BQ aggregation - throughput comparison

aggregator stays the same as for the single producer version.

As a consequence aggregators may also wait longer for the last aggregator to consume everything from one batch until they stop busy waiting. Although initially there is an improvement in the stages up until the second stage aggregators, the application loses at the last stage of communication which also slows down the processing of incoming tuples by the

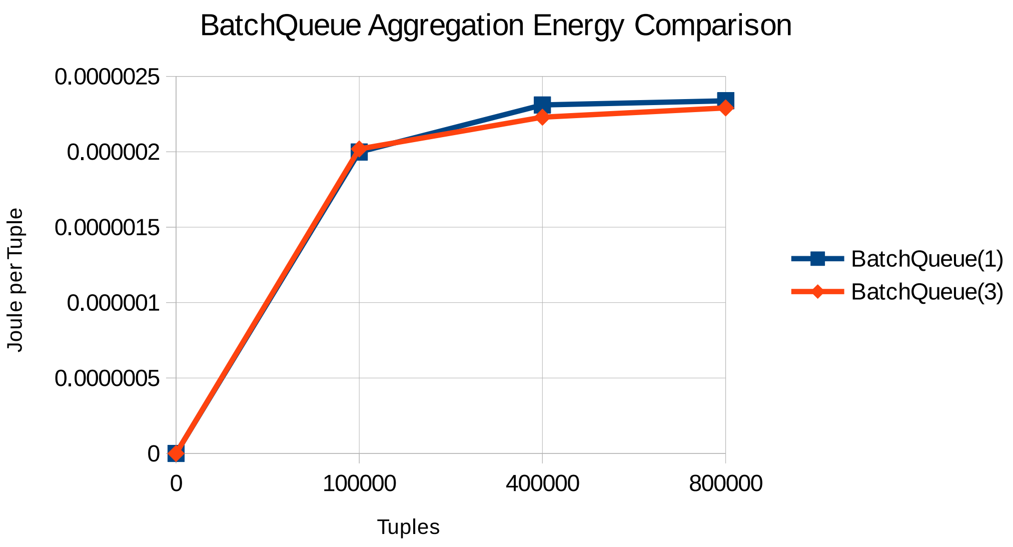


Figure 67: BQ aggregation - energy comparison

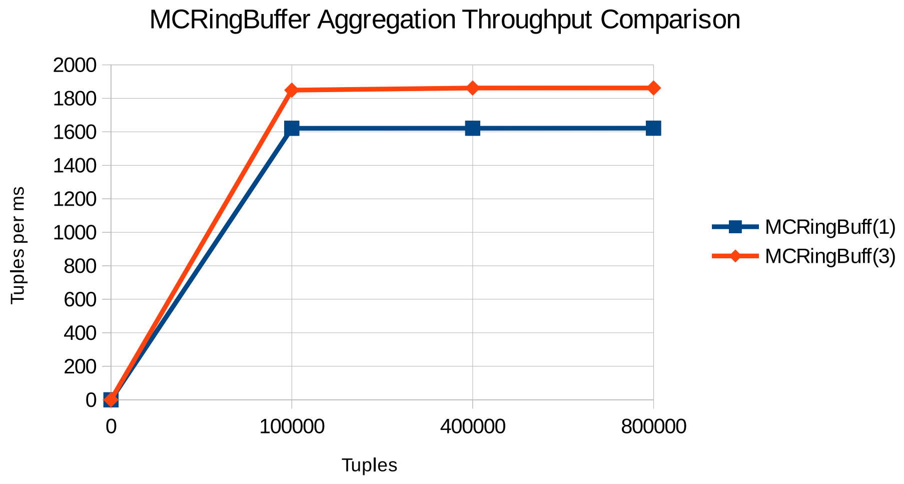


Figure 68: MCR aggregation - throughput comparison

aggregators, a fact which with its turn slows down the producers as well.

Table 14 provides the improvement gained by every algorithm from using three instead of a single producer for the aggregation.

A remark worth taking into consideration, is that apart from BQ's behaviour, data retrieved from the investigation on aggregation resemble the pattern seen in data element evaluation, especially for data element size larger than 64 bytes (see Fig. 48 and 49). This is because the evaluation on data element size, in a way simulates occasions where some

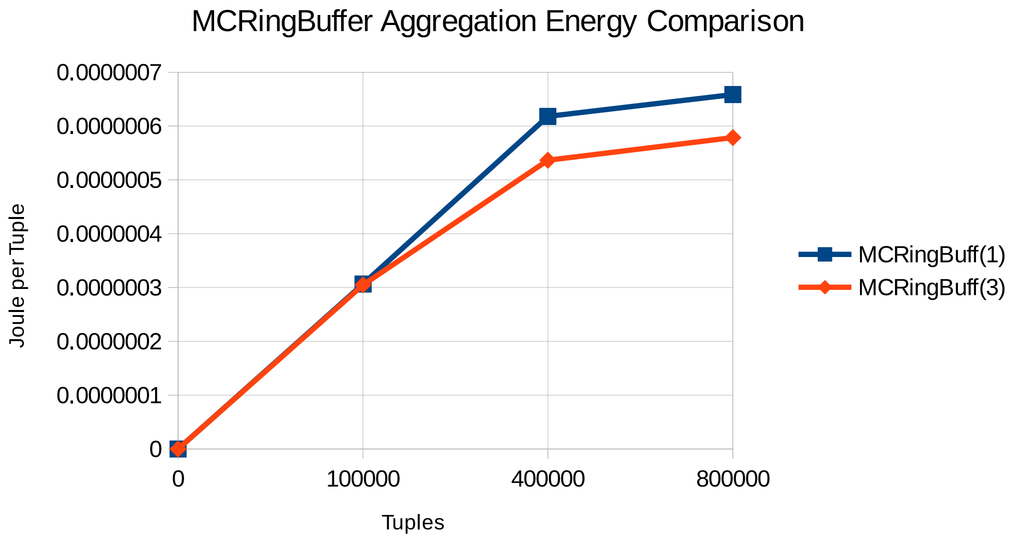


Figure 69: MCR aggregation - energy comparison

Table 14: Improvement from using three producers instead of one

	Lamport	FF	BQ	MCR
Throughput	12%	12%	6%	3%
Energy	26%	25%	2%	10%

work is performed between enqueues and dequeues. This is also what happens during the aggregation phase as well, where different aggregation processing algorithms are run from stage to stage corresponding to the work performed at the data element evaluation. This explains why MCR does not have the same advantage over the other algorithms as in other types of evaluation performed in Section 5.7.

5.9 Discussion and Conclusion

5.9.1 Research aim and main findings

The emergence of multicore platforms destined for embedded systems hosting computationally demanding applications, requires that limitations of these systems are taken into consideration in order to achieve optimal performance. Algorithmic design of concurrent data structures used in such applications play a major role on performance and energy consumption during the execution time, rendering their investigation a necessity for the research community.

Although extensive research has been conducted on concurrent data structures, their use and evaluation in data streaming applications and specifically in the aggregation phase has been neglected. For this reason four SPSC algorithms are used within the aggregation phase of a data streaming application evaluated on the embedded platform Myriad 2 in order to research their behaviour and applicability under the new circumstances.

Performance evaluation of the algorithms alone, corresponds to the behavior observed in a data streaming aggregation context as well, with the exception of BatchQueue, which shows that adopting data structures that employ batch processing of big batches is not the way to go. The smaller the batches the better the performance.

The winner of the investigation turns out to be MCRingBuffer as it manages to synchronize its processes more efficiently than other algorithms by making use of shared variables more sparsely. It needs to be noted that batches were not used for MCRingBuffer, as they did not seem to provide much of a difference on Myriad 2.

Furthermore the algorithms provide a much more efficient solution than lock-based approaches which, along with interrupt handling, provide the main ways of synchronization for embedded systems nowadays. The performance gains through the use of lock-free approaches over lock-based ones range from 83% to 97%, a fact that is quite encouraging, demonstrating the potential of these kinds of algorithms on embedded devices.

6 Libraries of Energy-efficient and Concurrent Data Structures

In this section, we describe our study on libraries of concurrent search trees.

6.1 Concurrent Search Trees

Recent research suggested that the energy consumption of future computing systems will be dominated by the cost of data movement [26]. As predicted by Dally, accessing data from nearby memory in a 10nm chip is 75× more energy efficient than accessing them from across the chip. Therefore, in order to construct energy efficient software systems, data structures and algorithms must support not only high parallelism but also fine-grained data locality. Moreover, the fine-grained data locality should be portable across platforms.

Concurrent trees are fundamental data structures that are widely used in different contexts such as load-balancing [29, 52, 93] and searching [5, 18, 19, 25, 31, 34]. Most of the existing highly-concurrent search trees are not considering the fine-grained data locality. The non-blocking concurrent search trees [19, 34] and Software Transactional Memory (STM) search trees [5, 18, 25, 31] have been regarded as the state-of-the-art concurrent search trees. They have been proven to be scalable and highly-concurrent. However these trees are not designed for fine-grained data locality. Prominent concurrent search trees which are often included in several benchmark distributions such as the concurrent red-black tree [31] by Oracle Labs and the concurrent AVL tree developed by Stanford [18] are not designed for data locality either. It is challenging to devise search trees that are portable, highly concurrent and fine-grained locality-aware. A platform-customized locality-aware search trees [65, 91] are not portable while there are big interests of concurrent data structures for unconventional platforms [53, 48]. Concurrency control techniques such as transactional memory [57, 54] and multi-word synchronization [55, 47, 71] do not take into account fine-grained locality while fine-grained locality-aware techniques such as van Emde Boas layout [88, 103] poorly support concurrency.

However, there are no studies discussing the effect of portable fine-grained locality on energy efficiency and performance (e.g., throughput) in concurrent search trees. This work aims to provide insights into whether it would be worthwhile to develop portable fine-grained locality-aware concurrent search trees, and what improvements and drawbacks can be expected from such solutions with respect to energy-efficiency and performance.

Evaluating the effect of portable fine-grained locality on energy efficiency and performance in concurrent search trees is challenging, mainly because practical fine-grained locality-aware concurrent search trees for a platform are usually customized for the particular platform and therefore not portable. To the best of our knowledge, the DeltaTree and GreenBST is the only practical *portable* fine-grained locality-aware concurrent search tree [100, 101].

Please note that the names of our trees have been changed according to comments we have received since 2013. DeltaTree that was described in the previous deliverables (i.e., D2.1 [46] and D2.2 [49]) and our technical report uploaded to Arxiv.org in 2013 [99] is a non-blocking and locality-aware concurrent search tree. Our SIGMETRICS'15 poster [101] presents DeltaTree, a *lock-based homogeneous B-link tree* with vEB-layout nodes, and it is the same tree as the Balanced ΔTree found in the previous deliverables. GreenBST is a *lock-based heterogeneous B-link tree* with vEB-layout nodes, which is a more improved tree than DeltaTree and is the same tree as the Heterogeneous ΔTree found in the previous deliverables.

We present the study on the effect of portable fine-grained locality on energy efficiency and performance in concurrent search trees. We found portable fine-grained locality-aware concurrent search tree can reduce the energy cost incurred by data movement within a system, and also achieve the best energy efficiency and performance over the evaluation platforms (cf. Table 15).

6.2 Benchmarks and Profiles of Concurrent Search Trees

To evaluate our conceptual idea of a locality-aware concurrent search tree, we compare GreenBST energy consumption and throughput with three prominent concurrent search trees (cf. Table 16). GreenBST is a concurrent B+tree based on the locality-aware concurrent search tree layout [101] and is an improved tree than DeltaTree. LFBST is the state-of-the-art fast concurrent non-blocking binary search tree. B-link (CBTree) tree is a concurrent B+tree that has been around for decades but still popular due to its effectiveness and practicality. Moreover, B-link tree also still used as a backend in popular database systems such as PostgreSQL[1] [87]. Citrus is an example of a concurrent search tree that uses read-copy-update (RCU), which is normally found in operating system internals, to handle concurrent operations within its data structure.

Besides energy efficiency and throughput, we also profile the cache and branching behavior of the tested trees to determine whether data-locality optimization can be translated into better energy efficiency and higher throughput. The experimental benchmarks were conducted on Intel high performance computing (HPC) platform, an ARM embedded platform, an accelerator platform based on the Intel Xeon Phi architecture (MIC platform) (cf. Table 15).

We measured the energy efficiency (in operations/Joule) and throughput (in operations/second) of all the trees in this evaluation. *Energy efficiency* indicators in *operations/Joule* were calculated using the number of operations (rep = 5,000,000) divided by the total CPU and DRAM energy consumption for the whole operations. The ARM and Myriad2 platforms were equipped with a built-in on-board power measurement system that was able to measure the energy of all CPU cores and memory (DRAM) continuously in real-time. For the Intel HPC platform, the Intel PCM library using built-in CPU counters was used to measure the CPU and DRAM energy. Energy indicators on MIC platform were collected by polling the `/sys/class/micras/power` interface every 50 milliseconds. *Throughput* indicators in *operations/second* were calculated using (rep = 5,000,000) operations divided by the maximum time the threads need to finish the whole operations.

[1]`https://github.com/postgres/postgres/blob/master/src/backend/access/nbtree/README`

Name	HPC	ARM	MIC	Myriad2
System	Intel Haswell-EP	Samsung Exynos5 Octa	Intel Knights Corner	Movidius Myriad2
Processors	2x Intel Xeon E5-2650L v3	1x Samsung Exynos 5410	1x Xeon Phi 31S1P	1x Myriad2 SoC
# cores	24 (without hyper-threading)	– 4x Cortex A15 cores – 4x Cortex A7 cores	57 (without hyper-threading)	– 1x LeonOS core – 1x LeonRT core – 12x Shave cores
Core clock	2.5 GHz	– 1.6 GHz (A15 cores) – 1.2 GHz (A7 cores)	1.1 GHz	600 MHz
L1 cache	32/32 KB I/D	32/32 KB I/D	32/32 KB I/D	– LeonOS (32/32 KB I/D) – LeonRT (4/4 KB I/D) – Shave (2/1 KB I/D)
L2 cache	256 KB	– 2 MB (shared, A15 cores) – 512 KB (shared, A7 cores)	512 KB	– 256 KB (LeonOS) – 32 KB (LeonRT) – 256 KB (shared, Shave)
L3 cache	30 MB (shared)	-	-	2MB ”CMX” (shared)
Interconnect	8 GT/s Quick Path Interconnect (QPI)	CoreLink Cache Coherent Interconnect (CCI) 400	5 GT/s Ring Bus Interconnect	400 GB/sec Interconnect
Memory	64 GB DDR3	2 GB LPDDR3	6 GB GDDR5	128 MB LPDDR II
OS	Centos 7.1 (3.10.0-229 kernel)	Ubuntu 14.04 (3.4.103 kernel)	Xeon Phi uOS (2.6.38.8+mpss3.5)	RTEMS (MDK 15.02.0)
Compiler	GNU GCC 4.8.3	GNU GCC 4.8.2	Intel C Compiler 15.0.2	Movidius MDK 15.02.0

Table 15: We use 4 different benchmark platforms to evaluate tree's energy efficiency and performance.

All trees were pre-filled with *init* values to simulate trees that partially fit into the last level cache. We used *init* = 8,388,607 for the benchmarks on the Intel HPC and MIC platforms and *init* = 4,194,303 for the benchmarks on the ARM platform.

Combination of update rate $u = \{0, 50\}$ and selected number of threads were used for each run. Update rate of 0 equals to 100% search, while 50 update rate equals to 50% search and 50% insert/delete operations out of *rep* operations. All involved operations were using randomly generated values $v \in (0, init \times 2], v \in \mathbb{N}$ as their parameter. The same range was also used to generate random numbers for the *init* values.

To ensure fair comparisons, all of the tree benchmark programs were using a unified benchmark source code that was linked directly during compilation. We also had set the *UB*

#	Algorithm	Ref	Description	Synchronization	Code authors	Data structure
1	GreenBST	-	Locality aware concurrent search tree	lock-based	this report	b+tree
2	LFBST	[82]	Improved non-blocking binary search tree	lock free	UT Dallas	binary tree
3	CBTree	[73]	Concurrent B+tree (B-link tree[2])	lock-based	this report	b+tree
4	Citrus	[11]	RCU-based search tree	lock-based	Technion	binary tree

Table 16: List of evaluated concurrent tree algorithms. These algorithms are sorted by synchronization type.

values of the GreenBST and the CBTree's B+tree order to their respective values so that each GNode and each CBTree's pages were within the system's page size of 4KB. We used POSIX thread library for concurrency on the HPC, ARM, and MIC platforms and running threads were pinned to the available physical cores.

6.2.1 Energy Evaluation

In the HPC platform (cf. Figure 70), GreenBST is up to 50% more energy efficient than CBTree as seen in the 100% searching case using 24 threads. For updates, GreenBST can be 30% more efficient than CBTree in the 50% update benchmark using 12 threads.

In the ARM platform (cf. Figure 71), GreenBST is 100% more energy efficient than CBTree in 50% update benchmark using 4 threads. Also, GreenBST is 75% more efficient than the other trees in the 100% search benchmark using 4 threads.

In the MIC platform (cf. Figure 72), GreenBST is up to 90% more energy efficient than CBTree in the 50% update benchmark using 57 threads. As for the search-only results, GreenBST is 20% more efficient than CBTree in the 100% search case using 28 threads.

Energy results from the HPC, ARM, and MIC platforms highlight the advantage in using the energy efficient and platform-independent locality-aware layout. GreenBST outperformed the other concurrent search trees on the HPC, ARM, and MIC platforms that have different memory hierarchies.

6.2.2 Throughput Evaluation

On the HPC platform (cf. Figure 73), GreenBST throughput is up to 50% faster than CBTree in the 100% search using 24 threads. The 50% update throughput of GreenBST is also 50% faster than CBTree when using 12 threads.

In the ARM platform (cf. Figure 74), GreenBST is 50% faster than CBTree in the 100% search case using 4 threads. GreenBST is also 50% faster than CBTree in the 50% update case using 4 threads.

In the MIC platform (cf. Figure 75), GreenBST is up to 100% faster than the other trees

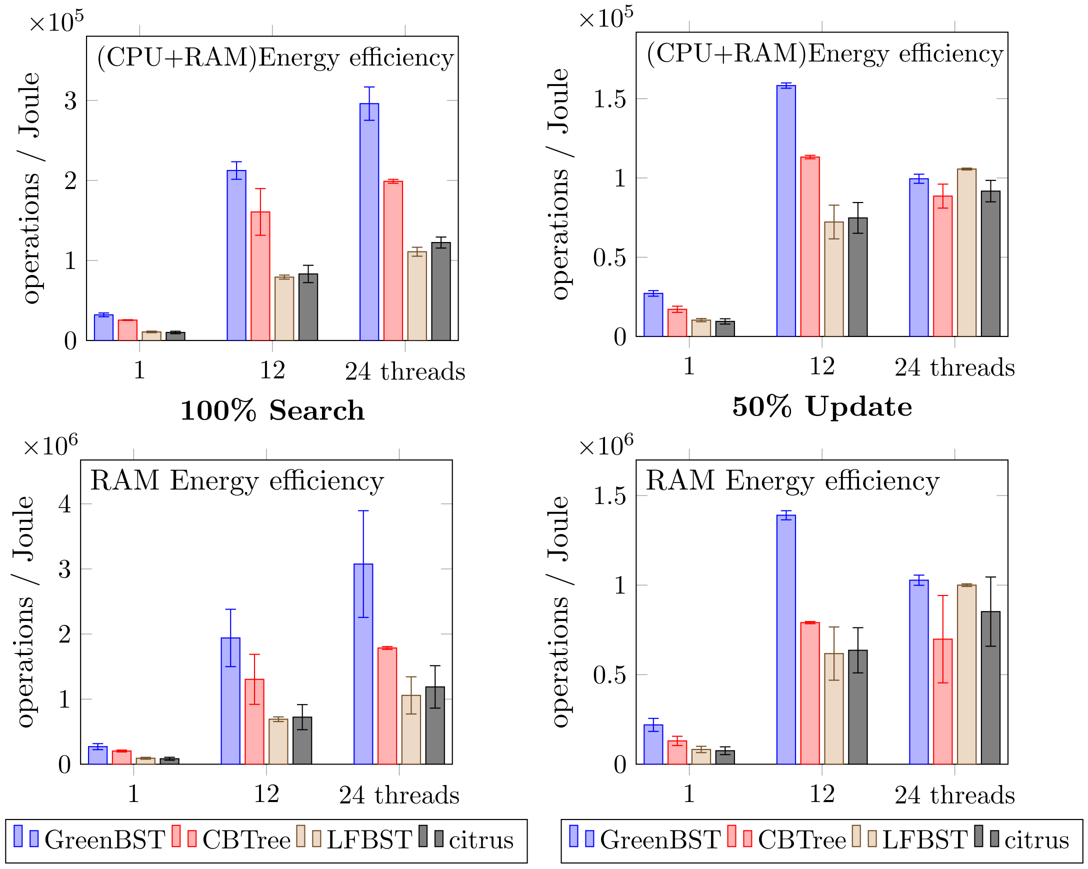


Figure 70: Energy comparison using 2^{23} initial values on the HPC platform. GreenBST is up to 50% more energy efficient than other trees in the 100% search benchmark using 12 threads (namely, on single CPU).

in 50% update case using 57 threads. GreenBST also outperforms CBTree by 20% in the 100% search case using 57 threads.

As in the previous energy evaluation sector, GreenBST has managed to outperform CB-Tree, LFBST, and citrus tree. These benchmark results have proved the advantages of the locality-aware layout on concurrent search tree. However, to provide insights into whether these energy efficiency and throughput advantages are really caused by the higher degree of locality that GreenBST has, we also collect and evaluate several key profiling results (i.e., cache miss and branch miss) of all the trees during the benchmarks.

6.2.3 Profiling Results

To gain more insight into which factors that have caused GreenBST's good energy efficiency and throughput, we extensively profiled all the trees in Table 16 when running the benchmarks on the HPM, ARM, and MIC platforms.

The profiling result on the HPC platform (cf. Figure 76) has revealed that GreenBST has

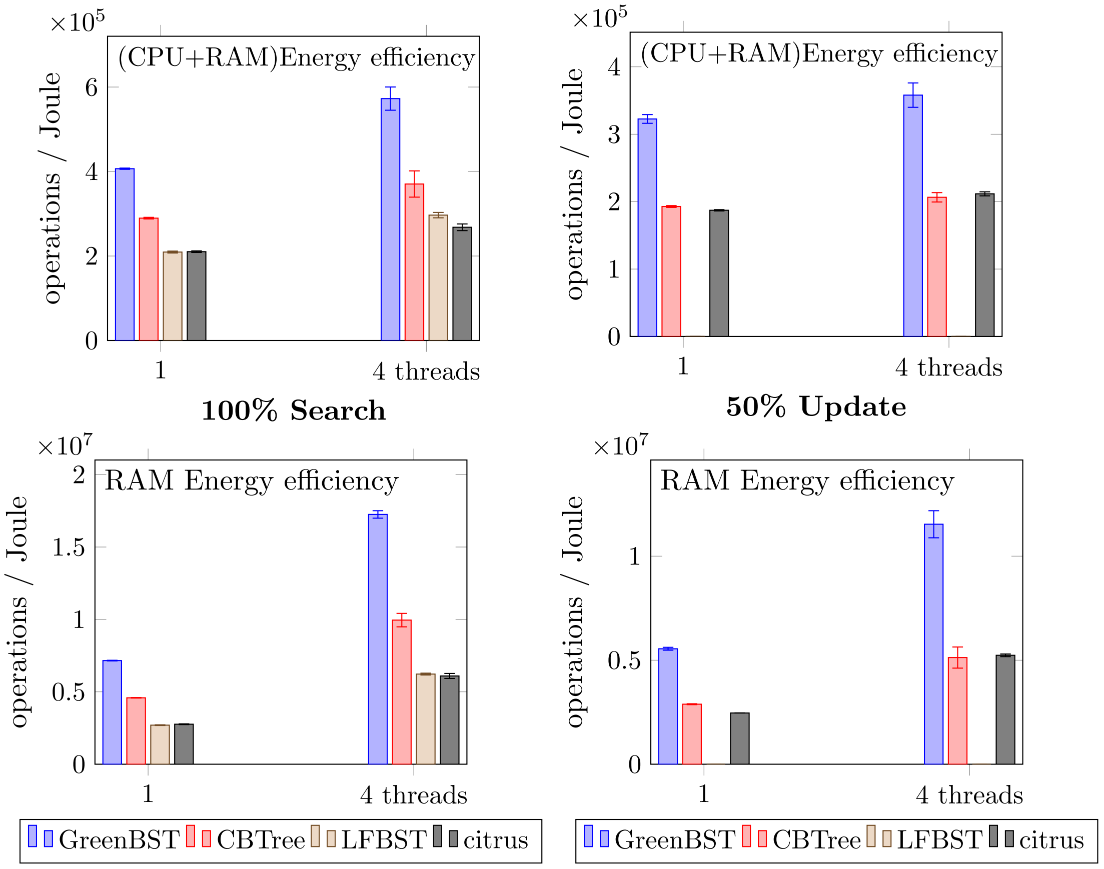


Figure 71: Energy comparison using 2^{22} initial values on the ARM platform. GreenBST is up to 100% more energy efficient than the other trees in the 50% update operation using 4 threads. Unfortunately, LFBST does not support the concurrent update operations on the ARM 32-bit platform.

the lowest L3 cache miss ratio. On the side note, GreenBST's branching performance is not as efficient as CBTree, which is mainly because of the recursive locality-aware layout [101]. However, it is obvious that GreenBST's efficient L3 cache performance eclipses its branching weakness, as GreenBST is the best performing tree compared to other trees.

On the ARM platform (cf. Figure 77), the GreenBST's L2 cache miss ratio is the lowest compared to the other trees. Again, the branching performance of GreenBST is considerably worse than the other trees, but it does not affect the GreenBST energy efficiency and throughput because of the energy and time savings improvement obtained from the lower data transfer.

Lastly, on the MIC platform (cf. Figure 78), the GreenBST's L2 cache miss ratio is worse than CBTree's, however after careful inspection, the total data transferred is actually less than CBTree's that indicates GreenBST has a more efficient data re-use. Also, similar to other platforms' results, CBTree branch miss ratio is slightly lower than GreenBST's because of the recursive tree layout adopted by GreenBST.

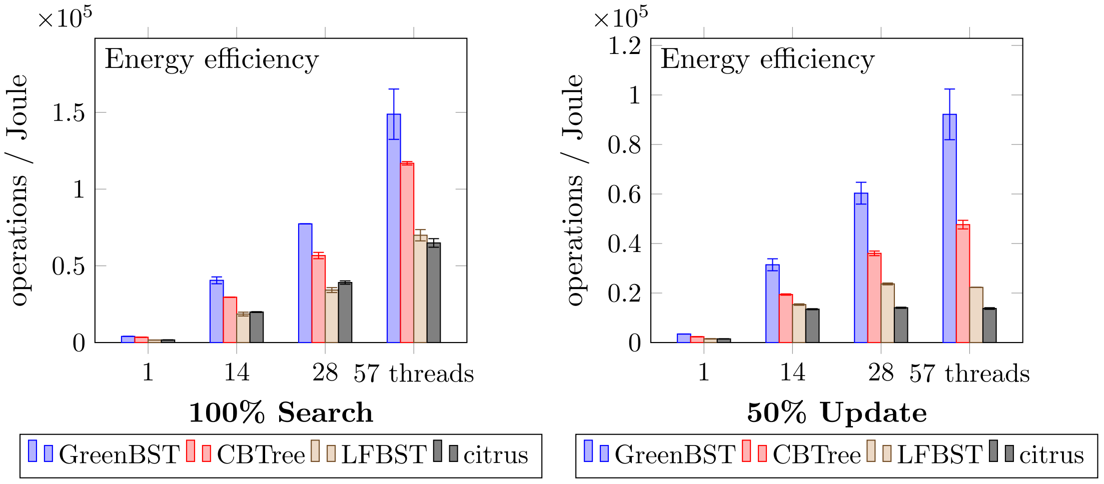


Figure 72: Energy comparison using 2^{23} initial values on the MIC platform. GreenBST is up to 90% more energy efficient than the other trees in the 50% update operation using 57 threads.

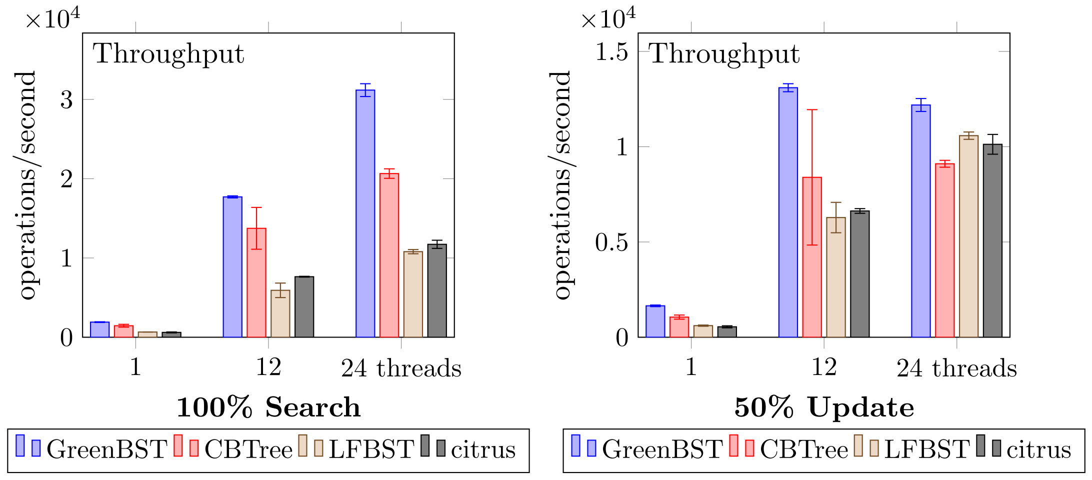


Figure 73: Throughput comparison using 2^{23} initial values on the HPC platform. GreenBST is up to 50% faster than the other trees in the 100% search operation benchmark using 24 threads.

Profiling Results on a Cycle-accurate Platform Simulator

To obtain even more insights into whether fine-grained data locality is able to lower data movements in a system, we tested the trees in Table 16 in the cycle-accurate computer system simulator platform GEM5. We collect the trees' load/write access to level 1, level 2, level 3 caches and DRAM when running the *100% search/1-thread* micro-benchmark using 4095 initial value.

The simulation results (cf. Table 17) shows GreenBST's L3 and DRAM load and store access are the lowest, despite having bigger L1 and L2 access compared to other trees.

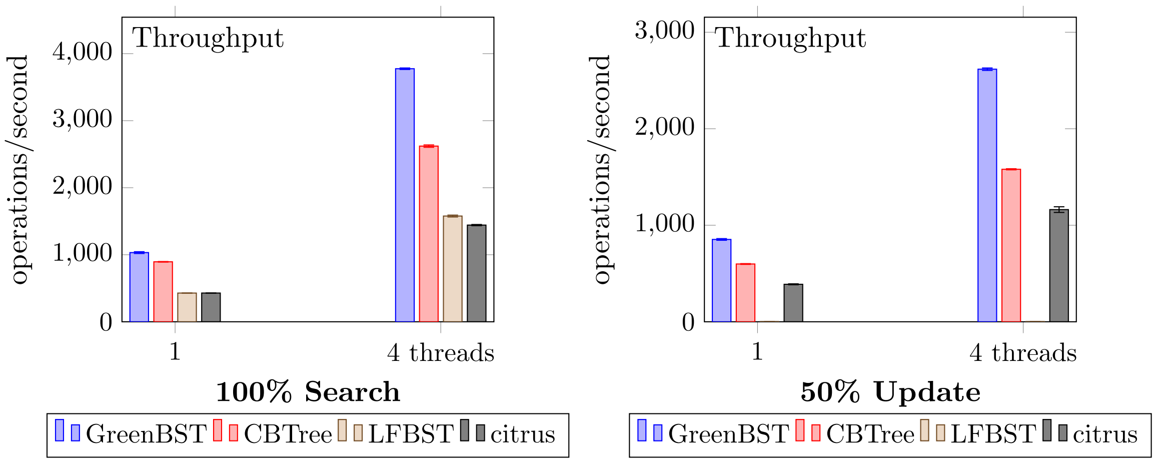


Figure 74: Throughput comparison using 2^{22} initial values on the ARM platform. GreenBST is up to 50% faster than other trees in 100% search operation with 4 threads. Unfortunately, LFBST does not support the concurrent update operations on the ARM 32-bit platform.

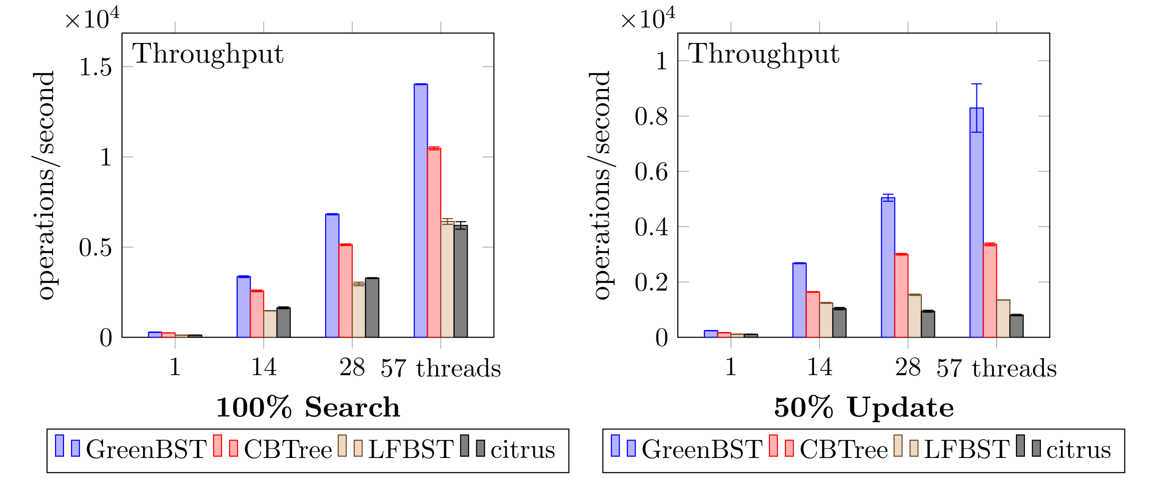


Figure 75: Throughput comparison using 2^{23} initial values on the MIC platform. GreenBST is up to 100% faster than the other trees in 50% update operation with 57 threads.

This indicates that most of GreenBST's data load/store instructions are served within the CPU, which implies that GreenBST can lower the number of data transfer between memory hierarchy.

6.3 Locality-aware Concurrent Search Tree on Myriad2 platform

We have implemented DeltaTree that works on the Myriad2 platform. The main idea of DeltaTree on Myriad2 platform is that we modify DeltaTree concurrency control while keeping the DeltaTree's original tree structure. The reason for the modification is because of the lack of support for atomic operations and limited number of usable hardware mutex

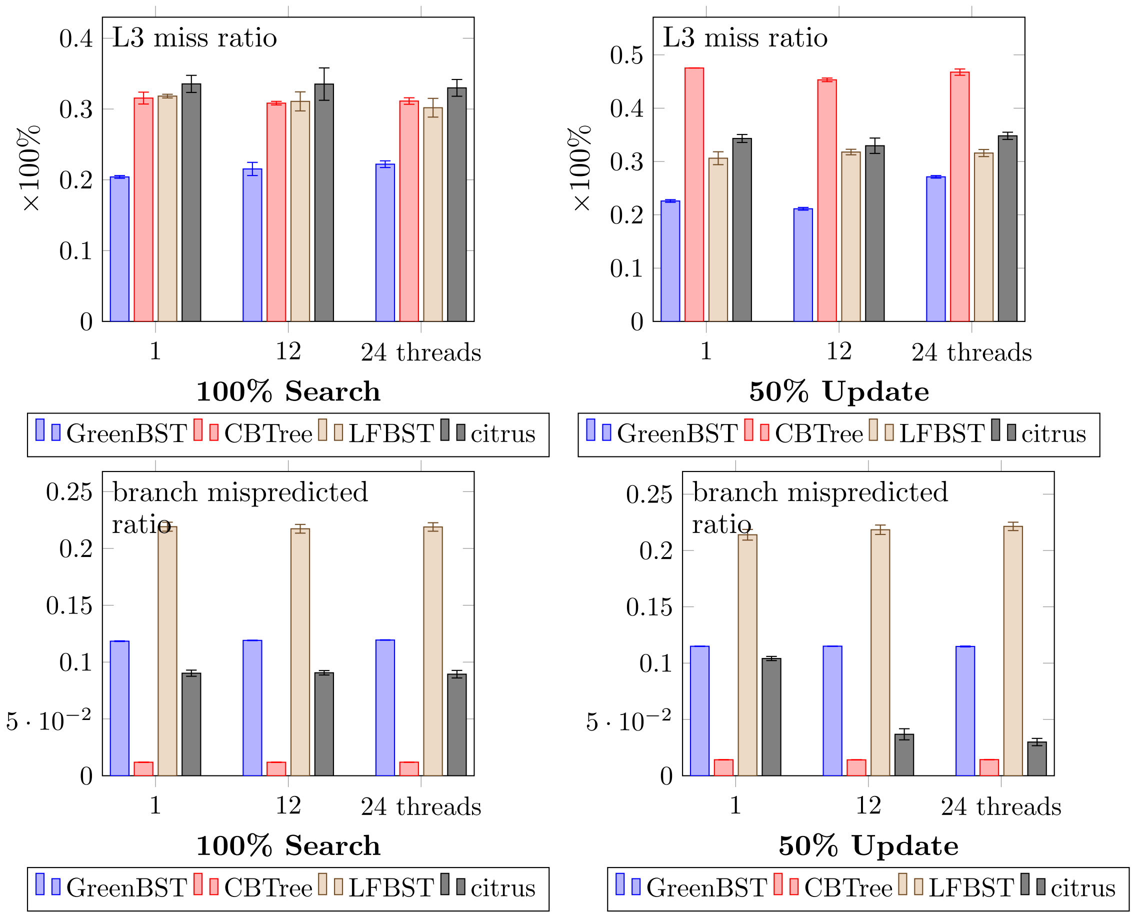


Figure 76: Profiling result on the HPC platform.

# of access	**GreenBST**	**CBTree**	**Citrus**	**LFBST**
L1	2451225	7227496	2408209	–
L2	49639	46907	81188	–
L3	1576	3002	13178	–
DRAM	1394	2142	3093	–

Table 17: Number of access as reported by the GEM5 system platform simulator. Unfortunately, LFBST is not able to run on the GEM5 simulator because it was designed only for 64-bit platforms.

on Myriad2. To circumvent these limitations, we utilize LeonRT as a lock manager for the shaves. With LeonRT acts as a lock manager, all shaves need to request a DeltaNode lock from LeonRT before it can lock the DeltaNode for update and maintenance operation. Our locking technique implementation uses only a shared array structure with $2 \times sv$ size, where sv is the number of active shaves. Advanced locking techniques [51, 64, 75] can also be used. For low latency lock operations, we put this lock structure in the Myriad2's CMX memory. All other DeltaTree structures (e.g., the tree itself) are placed in the DDR memory.

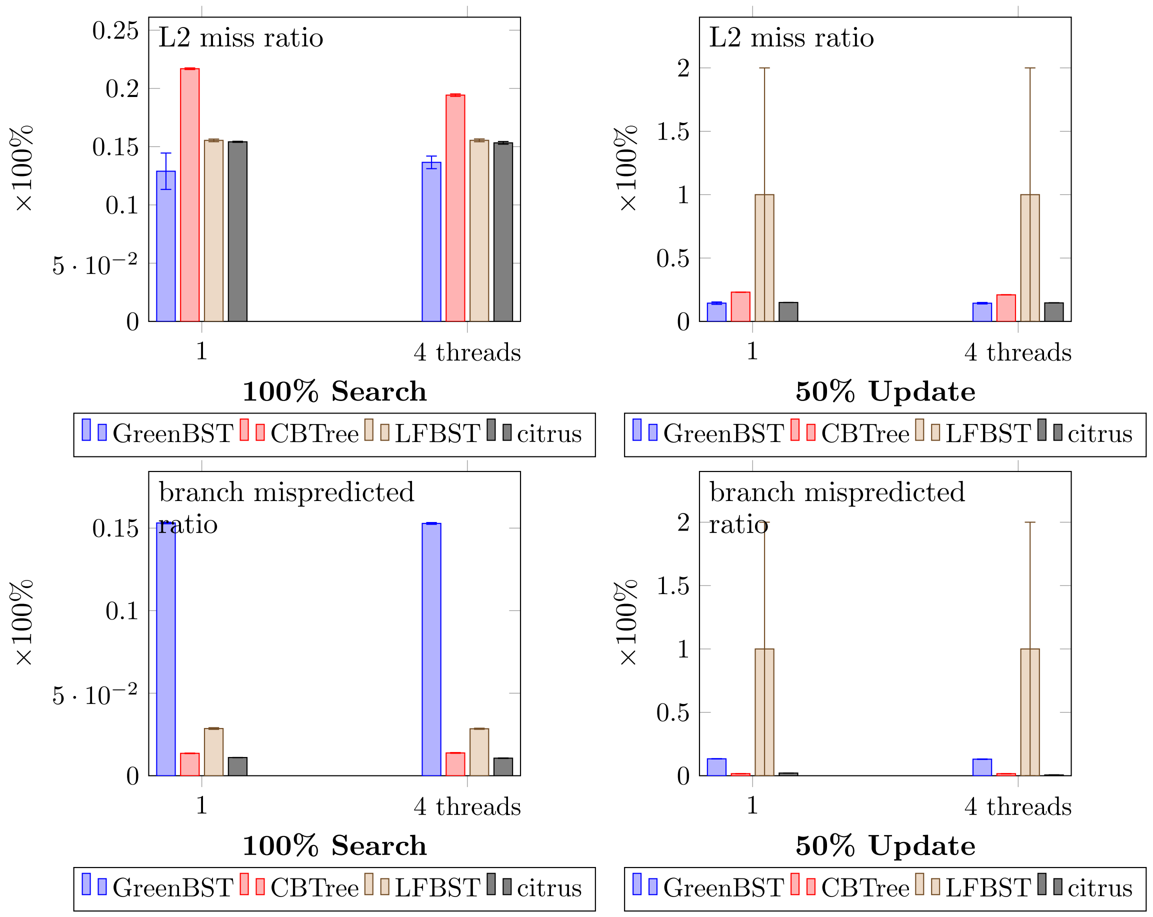


Figure 77: Profiling result on the ARM platform.

We tested our DeltaTree implementation on Myriad2 against the concurrent B+tree (Blink tree) [73]. The Blink tree implementation (CBTree) also utlized the same locking technique and memory placement strategy as DeltaTree. Figure 79 shows that the energy efficiency of DeltaTree is up to 4.7× better than CBTree in the 100% search using 12 shaves on the Myriad2 platform. For the 5% update case, DeltaTree is 150% more energy efficient than CBTree when using 1, 6, and 12 shaves. In terms of throughput on the Myriad2 platform, Figure 80 indicates that DeltaTree has up to 4× more throughput than CBTree in the 100% search case when using all available 12 shaves.

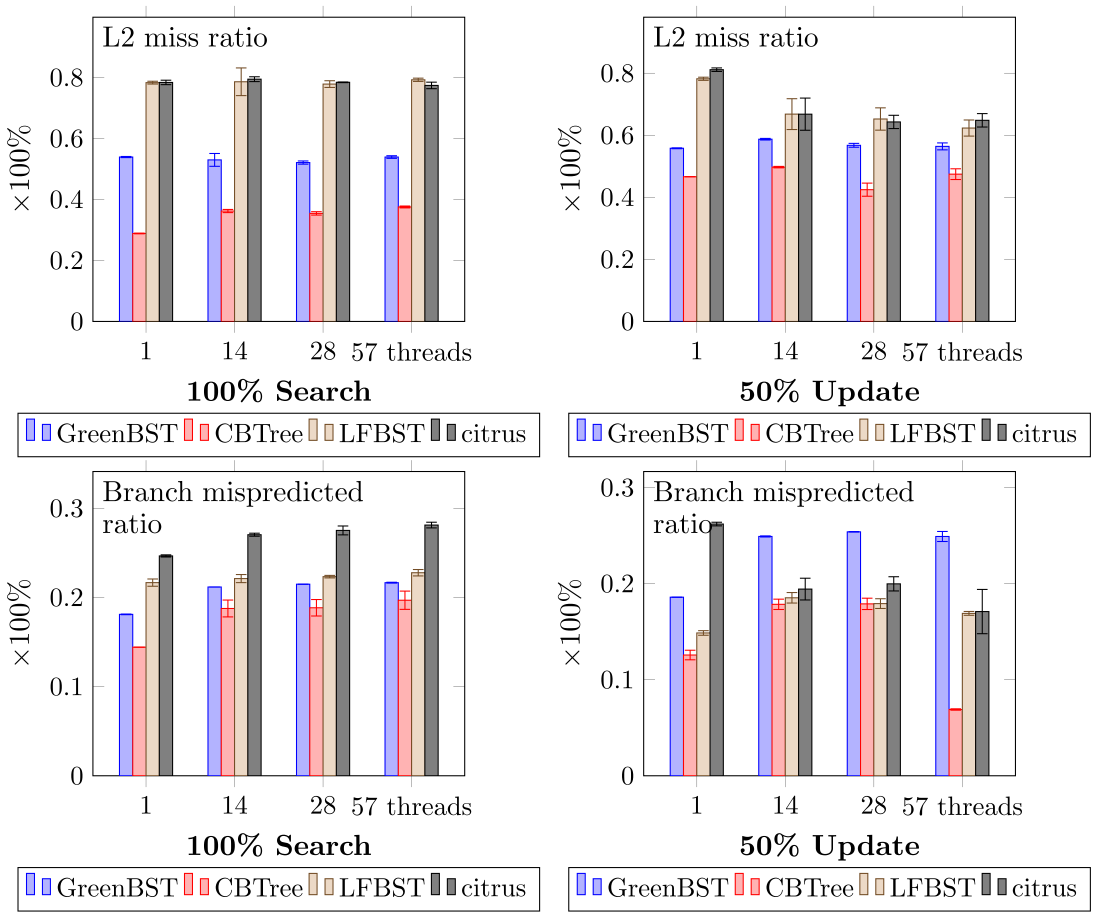


Figure 78: Profiling result on the MIC platform.

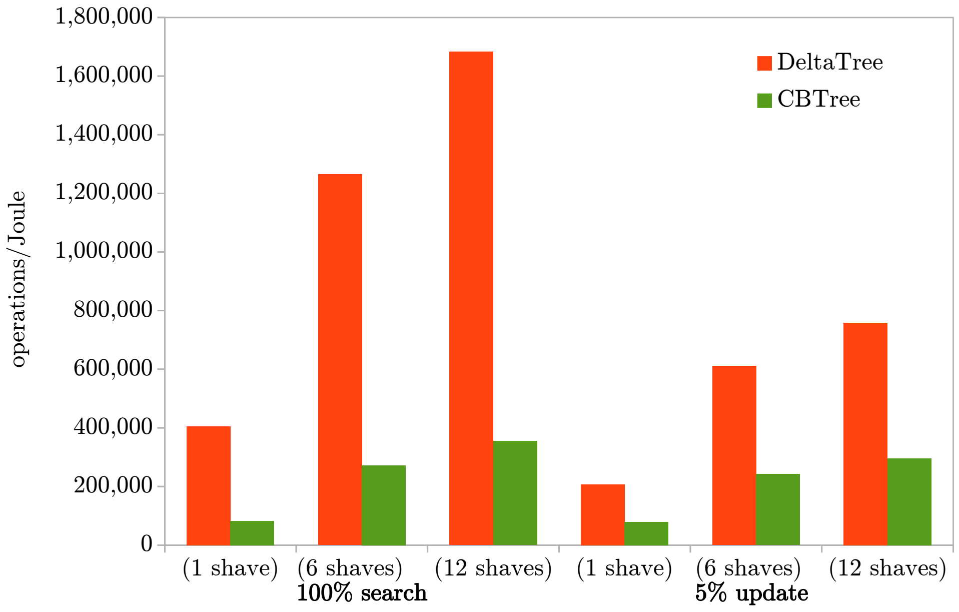


Figure 79: Energy comparison using 2^{20} initial values on an Myriad2 platform. DeltaTree is up to 4.7× more energy efficient than CBTree in 100% search operation with 12 shaves.

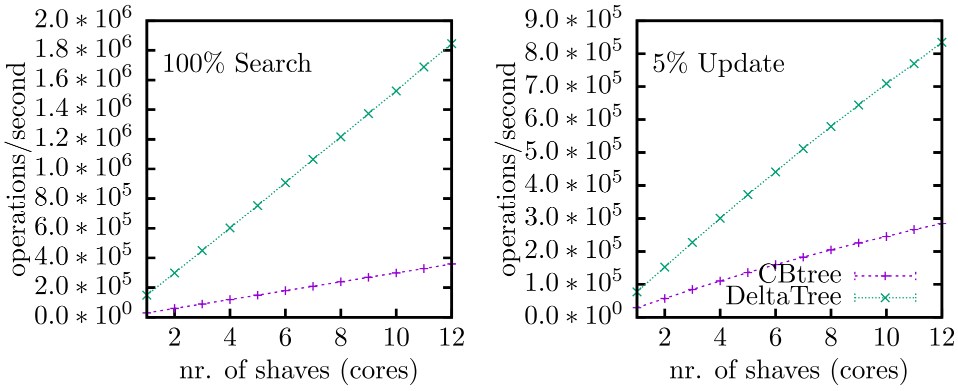


Figure 80: Throughput comparison using 2^{20} initial values on an Myriad2 platform. DeltaTree is up to 4× faster than CBTree in 100% search operation with 12 shaves

7 Conclusions

In this work, we have reported our current results on the new energy/power models modeling the trade-off of energy efficiency and performance of data structures and algorithms; as well as the latest prototype of libraries and programming abstractions.

- We have improved the power model for Myriad1 from the power models presented in Deliverable 2.2.
- We have proposed a new energy complexity model for multithreaded algorithms. This new general and validated energy complexity model for parallel (multithreaded) algorithms abstracts away possible multicore platforms by their static and dynamic energy of a computational operation and data access, and derives the energy complexity of a given algorithm from its *work*, *span* and *I/O* complexity.
- We have continued the modelling of the performance and the energy consumption of data structures on a CPU platform and need even less measurements points than previously.
- We have investigated on the optimization of streaming applications on Myriad2, from three points of view, which are performance, energy consumption, and space.
- We have implemented DeltaTree and a fast concurrent B-Tree on Myriad2 platform, and have shown that a specialized ultra low-power embedded platform such as Movidius Myriad2 can also benefit from the fine-grained locality data structures.

In the next steps of this work package, WP2 will continue the works of Task 2.1-2.4. The future works are to continue develop novel concurrent data structures and novel adaptive memory access algorithms. Moreover, identifying the best configuration (e.g., auto-tuning) to run the algorithms is also considered in the next steps.

References

[1] Java concurrency package. `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`. Accessed: 2016-01-20.

[2] Microsoft .net framework. `http://www.microsoft.com/net`. Accessed: 2016-01-20.

[3] Poski: Parallel optimized sparse kernel interface. http://bebop.cs.berkeley.edu/poski. Accessed: 2015-11-17.

[4] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[5] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E. Tarjan. Cbtree: a practical concurrent self-adjusting search tree. In *Proceedings of the 26th international conference on Distributed Computing*, DISC'12, pages 1–15, Berlin, Heidelberg, 2012. Springer-Verlag.

[6] Juan Alemany and Edward W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In Norman C. Hutchinson, editor, *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 125–134. ACM, 1992.

[7] M. Alioto. Ultra-low power vlsi circuit design demystified and explained: A tutorial. *IEEE Transactions on Circuits and Systems I: Regular Papers, vol.59, no.1, pp.3-29, Jan. 2012*, 59(1):3–29, Jan 2012.

[8] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? In David B. Shmoys, editor, *Proc. of ACM Symp. on Theory of Computing (STOC)*, pages 714–723. ACM, June 2014.

[9] P Alonso, M F Dolz, R Mayo, and E S Quintana-Orti. Modeling power and energy consumption of dense matrix factorizations on multicore processors. *Concurrency Computat.*, 2014.

[10] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems (TOCS)*, 15(2):134–165, 1997.

[11] Maya Arbel and Hagit Attiya. Concurrent updates with rcu: Search tree as an example. In *Proc. 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205. ACM, 2014.

[12] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the*

Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA '08, pages 197–206, New York, NY, USA, 2008. ACM.

[13] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. *Technical Report No. UCB/EECS-2006-183, University of California, Berkeley*, 2006.

[14] Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. Analyzing the performance of lock-free data structures: A conflict-based model. Technical Report 2014:15, Chalmers University of Technology, January 2015 `http://arxiv.org/abs/1508.03566`.

[15] MichaelA. Bender, GerthStoelting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. Optimal sparse matrix dense vector multiplication in the i/o model. *Theory of Computing Systems*, 47(4):934–962, 2010.

[16] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 355–366, 2011.

[17] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, 1999.

[18] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 257–268, 2010.

[19] Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In *Proceedings of the 15th international conference on Principles of Distributed Systems*, OPODIS'11, pages 207–221, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09. ACM, 2009.

[21] Jee Choi, Marat Dukhan, Xing Liu, and Richard Vuduc. Algorithmic time, energy, and power on candidate hpc compute building blocks. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 447–457, Washington, DC, USA, 2014.

[22] Jee Whan Choi, Daniel Bedard, Robert Fowler, and Richard Vuduc. A roofline model of energy. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 661–672, Washington, DC, USA, 2013.

[23] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 308–319, New York, NY, USA, 2013. ACM.

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[25] Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 161–170, New York, NY, USA, 2012. ACM.

[26] Bill Dally. Power and programmability: The challenges of exascale computing. In *DoE Arch-I presentation*, 2011.

[27] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In Michael Kaminsky and Mike Dahlin, editors, *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 33–48. ACM, November 2013.

[28] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[29] Giovanni Della-Libera and Nir Shavit. Reactive diffracting trees. *Journal of Parallel and Distributed Computing*, 60(7):853 – 890, 2000.

[30] Dave Dice, Yossi Lev, and Mark Moir. Scalable statistics counters. In Guy E. Blelloch and Berthold Vöcking, editors, *Proc. of the ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 43–52. ACM, July 2013.

[31] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 194–208, 2006.

[32] Kristijan Dragicevic and Daniel Bauer. A survey of concurrent priority queue algorithms. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–6, April 2008.

[33] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, 1997.

[34] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 131–140, 2010.

[35] Stijn Eyerman and Lieven Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 362–370, June 2010.

[36] Malte Forster and Jiri Kraus. Scalable parallel amg on ccnuma machines with openmp. *Computer Science - Research and Development*, 26(3-4):221–228, 2011.

[37] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS, 1999.

[38] Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 271–280, New York, NY, USA, 2006. ACM.

[39] Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 271–280, 2006.

[40] John Giacomoni, Tipp Moseley, and Manish Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 43–52. ACM, 2008.

[41] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, January 1992.

[42] James R. Goodman and Herbert Hing Jing Hum. Mesif: A two-hop cache coherency protocol for point-to-point interconnects. Technical report, University of Auckland, November 2009.

[43] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *Parallel and Distributed Systems, IEEE Transactions on*, 23(12):2351–2365, 2012.

[44] Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. Data-streaming and concurrent data-object co-design: Overview and algorithmic challenges. In *Algorithms, Probability, Networks, and Games - Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occasion of His 60th Birthday*, pages 242–260, 2015.

[45] Vincenzo Gulisano, Yiannis Nikolakopoulos, Ivan Walulya, Marina Papatriantafilou, and Philippas Tsigas. Deterministic real-time analytics of geospatial data streams through scalegate objects. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, pages 316–317, 2015.

[46] P. Ha, V. Tran, I. Umar, P. Tsigas, A. Gidenstam, P. Renaud-Goud, I. Walulya, and A. Atalar. Models for energy consumption of data structures and algorithms. Technical report, EU FP7 project EXCESS deliverable D2.1 (http://www.excess-project.eu), 2014.

[47] P. H. Ha and P. Tsigas. Reactive multiword synchronization for multiprocessors. In *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 184–193, 2003.

[48] P. H. Ha, P. Tsigas, and O. J. Anshus. The synchronization power of coalesced memory accesses. *IEEE Transactions on Parallel and Distributed Systems*, 21(7):939–953, 2010.

[49] Phuong Ha, Vi Tran, Ibrahim Umar, Aras Atalar, Anders Gidenstam, Paul Renaud-Goud, and Philippas Tsigas. White-box methodologies, programming abstractions and libraries. Technical Report D2.2, EU FP7 project EXCESS, 2015. http://www.excess-project.eu.

[50] Phuong Ha, Vi Tran, Ibrahim Umar, Philippas Tsigas, Anders Gidenstam, Paul Renaud-Goud, Ivan Walulya, and Aras Atalar. D2.1 Models for energy consumption of data structures and algorithms. Technical Report FP7-611183 D2.1, EU FP7 Project EXCESS, August 2014.

[51] Phuong Hoai Ha, Marina Papatriantafilou, and Philippas Tsigas. Efficient self-tuning spin-locks using competitive analysis. *Journal of Systems and Software*, 80(7):1077 – 1090, 2007.

[52] Phuong Hoai Ha, Marina Papatriantafilou, and Philippas Tsigas. Self-tuning reactive diffracting trees. *Journal of Parallel and Distributed Computing*, 67(6):674 – 694, 2007.

[53] Phuong Hoai Ha, P. Tsigas, and O. J. Anshus. Wait-free programming for general purpose computations on graphics processors. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2008.

[54] Phuong Hoai Ha, Philippas Tsigas, and Otto J. Anshus. Nb-feb: A universal scalable easy-to-use synchronization primitive for manycore architectures. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, pages 189–203, 2009.

[55] Phuong Hoai Ha, Philippas Tsigas, Mirjam Wattenhofer, and Rogert Wattenhofer. Efficient multi-word locking using randomization. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 249–257, 2005.

[56] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. *Journal of Parallel and Distributed Computing*, 70(1):1–12, 2010.

[57] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, 1993.

[58] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[59] Thomas Preud Homme, Julien Sopena, Gaël Thomas, and Bertil Folliot. Batchqueue: Fast and memory-thrifty core to core communication. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pages 215–222. IEEE, 2010.

[60] C. Imes, D.H.K. Kim, M. Maggio, and H. Hoffmann. Poet: a portable approach to minimizing energy under soft real-time constraints. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 75–86, April 2015.

[61] Intel. Lock scaling analysis on Intel® Xeon® processors. Technical Report 328878-001, Intel, April 2013.

[62] Intel Corporation. Intel Threading Building Blocks (Intel TBB). `https://www.threadingbuildingblocks.org/`, 2016. Accessed: 2016-01-20.

[63] H. Jacobson, A. Buyuktosunoglu, P. Bose, E. Acar, and R. Eickemeyer. Abstraction and microarchitecture scaling in early-stage power modeling. In *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA), 2011*, pages 394–405, Feb 2011.

[64] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitve spinning for a shared-memory multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 41–55, 1991.

[65] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD Intl. Conference on Management of data*, SIGMOD '10, pages 339–350, 2010.

[66] Alex Kogan and Maurice Herlihy. The future(s) of shared data structures. In Magnús M. Halldórsson and Shlomi Dolev, editors, *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 30–39. ACM, July 2014.

[67] V.A. Korthikanti and Gul Agha. Analysis of parallel algorithms for energy conservation in scalable multicore architectures. In *International Conference on Parallel Processing, 2009. ICPP '09.*, pages 212–219, Sept 2009.

[68] Vijay Anand Korthikanti and Gul Agha. Towards optimizing energy costs of algorithms for shared memory architectures. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 157–165, New York, NY, USA, 2010. ACM.

[69] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. Technical report, Ithaca, NY, USA, 1997.

[70] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

[71] Andreas Larsson, Anders Gidenstam, Phuong H. Ha, Marina Papatriantafilou, and Philippas Tsigas. Multiword atomic read/write registers on multiprocessor systems. *J. Exp. Algorithmics*, 13:7:1.7–7:1.30, 2009.

[72] Patrick PC Lee, Tian Bu, and Girish Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[73] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, December 1981.

[74] Lu Li and Christoph Kessler. Validating energy compositionality of GPU computations. In *HIPEAC Workshop on Energy Efficiency with Heterogeneous Computing (EEHCO)*, January 2015.

[75] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 25–35, 1994.

[76] Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In Roberto Baldoni, Nicolas Nisse, and Maarten van Steen, editors, *Proceedings of the International Conference on Principle of Distributed Systems (OPODIS)*, volume 8304 of *Lecture Notes in Computer Science*, pages 206–220. Springer, December 2013.

[77] Xu Liu and John Mellor-Crummey. A tool to analyze the performance of multithreaded programs on numa architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 259–272, New York, NY, USA, 2014. ACM.

[78] A. Cristiano I. Malossi, Yves Ineichen, Costas Bekas, Alessandro Curioni, and Enrique S. Quintana-Orti. Systematic derivation of time and power models for linear algebra kernels on multicore architectures. *Sustainable Computing: Informatics and Systems*, 7:24 – 40, 2015.

[79] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[80] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 267–275, May 1996.

[81] Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. A probabilistic graphical model-based approach for minimizing energy under performance constraints. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 267–281, New York, NY, USA, 2015. ACM.

[82] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proc. 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 317–328, 2014.

[83] G. Ofenbeck, R. Steinmann, V. Caparros, D.G. Spampinato, and M. Puschel. Applying the roofline model. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2014*, pages 76–85, March 2014.

[84] L. Papadopoulos, I. Walulya, P. Tsigas, D. Soudris, and B. Barry. Evaluation of message passing synchronization algorithms in embedded systems. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pages 282–289, July 2014.

[85] Lazaros Papadopoulos, Ivan Walulya, Paul Renaud-Goud, Philippas Tsigas, Dimitrios Soudris, and Brendan Barry. Performance and power consumption evaluation of concurrent queue implementations in embedded systems. *Computer Science - Research and Development*, pages 1–11, 2014.

[86] Witold Pedrycz. Why triangular membership functions? *Fuzzy Sets Syst.*, 64(1):21–30, May 1994.

[87] PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org`, 2008.

[88] Harald Prokop. Cache-oblivious algorithms. Master's thesis, MIT, 1999.

[89] Swapnoneel Roy, Atri Rudra, and Akshat Verma. An energy complexity model for algorithms. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science*, ITCS '13, pages 283–304, New York, NY, USA, 2013. ACM.

[90] Y. Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[91] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Rajagopalan Satish, and Pradeep Dubey. Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. *Proceedings of the VLDB Endowment*, 4(11):795–806, 2011. VLDB 2011.

[92] Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 263–268, May 2000.

[93] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, 1996.

[94] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: A platform for os-level power management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09. ACM, 2009.

[95] T. Suzumura, K. Ueno, H. Sato, K. Fujisawa, and S. Matsuoka. Performance characteristics of graph500 on large-scale distributed environment. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 149–158, Nov 2011.

[96] T. Takagi and M. Sugeno. Fuzzy identification of systems and its applications to modeling and control. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-15(1):116–132, Jan 1985.

[97] Vi Ngoc-Nha Tran, Brendan Barry, and Ha. Rthpower: Accurate fine-grained power models for predicting race-to-halt effect on ultra-low power embedded systems. In *Proceedings of the 17th IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS 16, 2016.

[98] R. Kent Treiber. *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[99] Ibrahim Umar, Otto Anshus, and Phuong Ha. Deltatree: A practical locality-aware concurrent search tree. Technical Report IFI-UIT 2013-74, UiT The Arctic University of Norway, 2013. arXiv:1312.2628.

[100] Ibrahim Umar, Otto J. Anshus, and Phuong H. Ha. Effect of portable fine-grained locality on energy efficiency and performance in concurrent search trees. In *Procs. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 36:1–36:2, 2016.

[101] Ibrahim Umar, Otto Johan Anshus, and Phuong Hoai Ha. Deltatree: A locality-aware concurrent search tree. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '15, pages 457–458, 2015. Poster paper.

[102] J. D. Valois. Implementing Lock-Free Queues. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems*, pages 64–69, 1994.

[103] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, SFCS '75, pages 75–84, Washington, DC, USA, 1975. IEEE Computer Society.

[104] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, 2007. SC '07.*, pages 1–12, Nov 2007.

[105] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[106] Xiao Yu, Zhengyu He, and Bo Hong. A queuing model-based approach for the analysis of transactional memory systems. 25(6):808–825, 2013.

Glossary

BRU	Branch Repeat Unit (on SHAVE processor)
CAS	Compare-and-Swap instruction
CMX	Connection MatriX on-chip (shared) memory unit, 128KB (Movidius Myriad)
CMU	Compare-Move Unit (on SHAVE processor)
Component	1. [hardware component] part of a chip's or motherboard's circuitry; 2. [software component] encapsulated and annotated reusable software entity with contractually specified interface and explicit context dependences only, subject to third-party (software) composition.
Composition	1. [software composition] Binding a call to a specific callee (e.g., implementation variant of a component) and allocating resources for its execution; 2. [task composition] Defining a macrotask and its use of execution resources by internally scheduling its constituent tasks in serial, in parallel or a combination thereof.
CPU	Central (general-purpose) Processing Unit
uncore	including the ring interconnect, shared cache, integrated memory controller, home agent, power control unit, integrated I/O module, config Agent, caching agent and Intel QPI link interface
CTH	Chalmers University of Technology
DAQ	Data Acquisition Unit
DCU	Debug Control Unit (on SHAVE processor)
DDR	Double Data Rate Random Access Memory
DMA	Direct (remote) Memory Access
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
DVFS	Dynamic Voltage and Frequency Scaling
ECC	Error-Correcting Coding
EXCESS	Execution Models for Energy-Efficient Computing Systems
GPU	Graphics Processing Unit
HPC	High Performance Computing
IAU	Integer Arithmetic Unit (on SHAVE processor)
IDC	Instruction Decoding Unit (on SHAVE processor)
IRF	Integer Register File (on SHAVE processor)
LEON	SPARCv8 RISC processor in the Myriad1 chip
LIU	Linköping University
LLC	Last-level cache
LSU	Load-Store Unit (on SHAVE processor)
Microbenchmark	Simple loop or kernel developed to measure one or few properties of the underlying architecture or system software
PAPI	Performance Application Programming Interface

PEPPHER	Performance Portability and Programmability for Heterogeneous Many-core Architectures. FP7 ICT project, 2010-2012, www.peppher.eu
PEU	Predicated Execution Unit (on SHAVE processor)
Pinning	[thread pinning] Restricting the operating system's CPU scheduler in order to map a thread to a fixed CPU core
QPI	Quick Path Interconnect
RAPL	Running Average Power Limit energy consumption counters (Intel)
RCL	Remote Core Locking (synchronization algorithm)
SAU	Scalar Arithmetic Unit (on SHAVE processor)
SHAVE	Streaming Hybrid Architecture Vector Engine (Movidius)
SoC	System on Chip
SRF	Scalar Register File (on SHAVE processor)
SRAM	Static Random Access Memory
TAS	Test-and-Set instruction
TMU	Texture Management Unit (on SHAVE processor)
USB	Universal Serial Bus
VAU	Vector Arithmetic Unit (on SHAVE processor)
Vdram	DRAM Supply Voltage
Vin	Input voltage level
Vio	Input/Output voltage level
VLIW	Very Long Instruction Word (processor)
VLLIW	Variable Length VLIW (processor)
VRF	Vector Register File (on SHAVE processor)
Wattsup	Watts Up .NET power meter
WP1	Work Package 1 (here: of EXCESS)
WP2	Work Package 2 (here: of EXCESS)