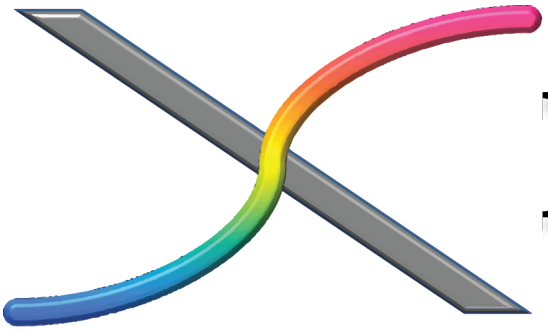
SEVENTH FRAMEWORK PROGRAMME
THEME ICT-2013.3.4
Advanced Computing, Embedded and Control Systems



EXCESS

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

D2.2
White-box methodologies, programming abstractions and libraries

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

Date of preparation (latest version): 27.02.2015

DOCUMENT INFORMATION

Deliverable Number	D2.2
Deliverable Name	White-box methodologies, programming abstractions and libraries
Authors	Phuong Ha Vi Tran Ibrahim Umar Aras Atalar Anders Gidenstam Paul Renaud-Goud Philippas Tsigas
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 2.2
Nature	R
Dissemination Level	PU
Planned Date	28.02.2015
Final Version Date	27.02.2015
Reviewed by	Christoph Kessler (LIU), Michael Gienger (HLRS)
MGT Board Approval	YES

DOCUMENT HISTORY

Partner	**Date**	**Comment**	**Version**
UiT (P. Ha, V. Tran)	02.12.2014	Deliverable skeleton	0.1
UiT (P. Ha, V. Tran, I. Umar)	23.01.2015	First version sent to WP2 partners	0.2
CTH (P. Renaud-Goud)	28.01.2015	Chalmers' part included	0.3
UiT (P. Ha, V. Tran, I. Umar)	06.02.2015	Consolidated version sent to internal reviewers	0.4
UiT (P. Ha, V. Tran)	27.02.2015	Final version	1.0

Abstract

This deliverable reports the results of white-box methodologies and early results of the first prototype of libraries and programming abstractions as available by project month 18 by Work Package 2 (WP2). It reports i) the latest results of Task 2.2 on white-box methodologies, 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 Myriad1 platform.

- Regarding white-box methodologies, we have devised new relaxed cache-oblivious models and proposed a new power model for Myriad1 platform and an energy model for lock-free queues on CPU platforms. For Myriad1 platform, the improved model now considers both computation and data movement cost as well as architecture and application properties. The model has been evaluated with a set of micro-benchmarks and application benchmarks. For Intel platforms, we have generalized the model for concurrent queues on CPU platforms to offer more flexibility according to the workers calling the data structure (parallel section sizes of enqueuers and dequeuers are decoupled).
- Regarding programming abstractions and libraries, we have continued investigating the trade-offs between energy consumption and performance of data structures such as concurrent queues and concurrent search trees based on the early results of Task 2.1. Based on the investigation, we have implemented a set of libraries and programming abstractions including concurrent queues and concurrent search trees that are energy-aware. The preliminary results show that our concurrent trees are faster and more energy efficient than the state-of-the-art on commodity HPC and embedded platforms.

Executive Summary

Computing technology is currently at the beginning of the disruptive transition from petascale to exascale computing (2010 - 2020), posing a great challenge on energy efficiency. High performance computing (HPC) in 2020 will be characterized by data-centric workloads that, unlike those in traditional sequential/parallel computing, are comprised of big, divergent, fast and complex data. In order to address energy challenges in HPC, the new data must be organized and accessed in an energy-efficient manner through novel fundamental data structures and algorithms that strive for the energy limit. Moreover, the general application- and technology-trend indicates finer-grained execution (i.e. smaller chunks of work per compute core) and more frequent communication and synchronization between cores and uncore components (e.g. memory) in HPC applications. Therefore, not only concurrent data structures and memory access algorithms but also synchronization is essential to optimize the energy consumption of HPC applications. However, previous concurrent data structures, memory access algorithms and synchronization algorithms were designed without energy consumption in mind. The design of energy-efficient fundamental concurrent data structures and algorithms for inter-process communication in HPC remains a largely unexplored area and requires significant efforts to be successful.

Work package 2 (WP2) aims to develop interfaces and libraries for energy-efficient inter-process communication and data sharing on the new EXCESS platforms integrating Movidius embedded processors. 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 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.

The latest results of Task 2.2 (PM7 - PM36) on white-box methodologies, programming abstractions and libraries available by project month 18 as well as the improved results of Task 2.1 on investigating and modeling the trade-off between energy and performance [41], are summarized in this report.

White-box methodologies

The white-box methodologies presented in this report include a new cache-oblivious methodology and new energy models that help develop energy-efficient data structures and algorithms:

- We have devised a new *relaxed* cache oblivious methodology that is appropriate for developing energy-efficient concurrent data structures and algorithms.
- We have proposed a new power model for Movidius embedded platform (Myriad1) which is able to predict the power consumed by a program running on a specific number of cores. We have validated the model with a set of micro-benchmarks and

real applications such as sparse/dense linear algebra kernels and the graph application kernels (e.g., the Graph 500 kernels [1]).

- We have proposed a way to model the energy behavior of lock-free queue implementations and parallel applications that use them on CPU platforms. Focusing on steady state behavior, we have decomposed energy behavior into throughput and power dissipation which can be modeled separately and later recombined into several useful metrics, such as energy per operation.

Programming abstractions and libraries

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

- We have developed a concurrent search tree library that contains several state-of-the-art concurrent search trees such as the non-blocking binary search tree, the Software Transactional Memory (STM) based red-black tree, AVL tree, and speculation-friendly tree, the fast concurrent B-tree, and the static cache-oblivious binary search tree. A family of novel locality-aware and energy efficient concurrent search trees, namely the DeltaTree, the Balanced DeltaTree, and Heterogeneous DeltaTree, are also enclosed in the concurrent search tree library. The DeltaTrees are platform-independent and up to 140% faster and 220% more energy efficient than the state-of-the-art on commodity HPC and embedded platforms.

- On lock-free queues on CPU platforms, we have automatized the process of estimating the performance and the power dissipation of any queue implementation, and integrated it in the EXCESS software.

This report is organized as follows. Section 1 provides the background and motivations of the work presented in this deliverable. Section 2 describes two EXCESS platforms and their setting to measure power consumption. The white-box methodologies such as relaxed cache-oblivious methodology and power models are presented in Section 3. Section 4 describes the first prototype of EXCESS libraries and programming abstractions including numerous concurrent search tree and queue implementations as well as their performance and energy analyses. Section 5 concludes the report with future works.

[1] http://www.graph500.org/

Contents

1 Introduction

1.1 Purpose

In order to address energy challenges in HPC and embedded computing, data must be organized and accessed in an energy-efficient manner through novel fundamental data structures and algorithms that strive for the energy limit. Due to more frequent communication and synchronization between cores and memory components in HPC and embedded computing, not only efficient design of concurrent data structures and memory access algorithms but also synchronization is essential to optimize the energy consumption. However, previous concurrent data structures, memory access algorithms and synchronization algorithms were designed without considering energy consumption. Although there are existing studies on the energy utilization of concurrent data structures demonstrating non-optimal results on energy consumption, the design of energy-efficient fundamental concurrent data structures and algorithms for inter-process communication in HPC and embedded computing is not yet widely explored and becomes a challenging and interesting research direction.

EXCESS aims to investigate the trade-offs between energy consumption and performance of concurrent data structures and algorithms as well as inter-process communication in HPC and embedded computing. By analyzing the non-intuitive results, EXCESS will devise a comprehensive model for energy consumption of concurrent data structures and algorithms for inter-process communication, especially in the presence of component composition. The new energy-efficient technology will be delivered through novel execution models for the energy-efficient computing paradigm, which consist of complete energy-aware software stacks (including energy-aware component models, programming models, libraries/algorithms and runtimes) and configurable energy-aware simulation systems for future energy-efficient architectures.

The goal of Work package 2 (WP2) is to develop interfaces and libraries for inter-process communication and data sharing on EXCESS platforms integrating Movidius embedded processors, 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.

In addition to investigating the trade-offs and devising comprehensive models for energy consumption of concurrent data structures and algorithms (Task 2.1), WP2 also identifies essential concurrent data structures and algorithms for inter-process communication in HPC with the focus on how to customize them (Task 2.2). We exploit common data-flow patterns to create generalized communication abstractions with which application designers can easily create and exploit the customization for the data-flow patterns. This task constitutes the interfaces and libraries for inter-process communication and data sharing on EXCESS platforms. The results also constitute a white-box methodology for tuning energy efficiency

and performance of concurrent data structures and algorithms.

This report summarizes i) the early results of Task 2.2 on white-box methodologies, programming abstractions and libraries 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 improved results of Task 2.1 constitute the theoretical basis for the whole work package.

1.2 White-box Methodologies

White-box methodology is a general study or a theoretical analysis of the principles and methods applied into a field of study or research to outline how the study or research should be taken. The term "white-box" means that the principles and methods in the methodology have prior knowledge of the inner workings and structures of the objects involved in the study. In the scope of EXCESS project, the energy efficiency, architecture and inner workings of a system must be well understood, and likewise for algorithms and data-sets. The "white-box" methodologies studied in this report include cache-oblivious methodology, a power model for Myriad1 platform and an energy model for lock-free concurrent queues.

1.2.1 Cache-oblivious Methodology

Energy efficiency is one of the most important factors in designing high performance systems. As a result, data must be organized and accessed in an energy-efficient manner through novel fundamental data structures and algorithms that strive for the energy limit. Unlike conventional locality-aware algorithms that only concern about whether the data is on-chip (e.g., cache) or not (e.g., DRAM), new energy-efficient data structures and algorithms must consider data locality in finer-granularity: *where on chip the data is*. Dally [25] predicted that for chips using the 10nm technology, the energy required between accessing data in nearby on-chip memory and accessing data across the chip will differ as much as 75x (2pJ versus 150pJ), whereas the energy required between accessing the on-chip data and accessing the off-chip data will only differ by 2x (150pJ versus 300pJ). 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 [25].

In order to devise locality-aware algorithms, we need theoretical execution models that promote data locality. One example of such models is the the cache-oblivious (CO) models [34], which enable the analysis of data transfer between two levels of the memory hierarchy. CO models are using the same analysis as the widely known I/O models [3] except in CO models an optimal replacement is assumed. Lower data transfer complexity implies better data locality and higher energy efficiency as energy consumption caused by data transfer dominates the total energy consumption [25]. These models require the knowledge of the algorithm and some parameters of the architecture to be known beforehand, hence they are white-box methods.

The cache-oblivious (CO) models (cf. Section 3.1.1.1) support not only fine-grained data locality but also portability. A CO algorithm that is optimized for 2-level memory, is

asymptotically optimized for unknown multilevel memory (e.g., register, L1C, L2C, ..., LLC, memory), enabling fine-grained data locality (e.g., minimizing data movement between L1C and L2C). As cache sizes and block sizes in the CO models are unknown, CO algorithms are expected to be portable across different systems. For example, the memory transfer cost of an algorithm (e.g., how many data blocks need to be transferred between two level of memory), which is analyzed using the CO model, will be applicable on both HPC machines and embedded platforms (e.g., Myriad1/2 platforms), irrespective of the variations in the hardware parameters such as memory hierarchy, specifications and sizes. The performance portability is useful for analyzing the data movement and energy consumption of an algorithm in a platform-independent manner.

The memory transfer cost of an algorithm obtained using the CO model can be regarded as a first piece of information that can enable software designers to rapidly analyze the performance and energy consumption of their algorithms. After all, memory transfer is one of the parameters that dominate the total energy consumption. As for the next step, the transfer cost can be fed directly into the energy model of a specific platform to get a good approximation on the energy consumption of the algorithm on the platform.

Algorithms and data structures analyzed using the cache-oblivious models [34] are found to be cache-efficient and disk-efficient [14, 28], making them suitable for improving energy efficiency in modern high performance systems. Nowadays, multilevel memory hierarchies in commodity systems are becoming more prominent as modern CPUs tend to have at least 3 level of caches and disks start to incorporate hybrid-SSD cache memories. With minimal effort, cache-oblivious algorithms are expected to be always locality-optimized irrespective of variations in memory hierarchies, enabling less data transfers between memory levels that directly translate into runtime energy savings.

Since their inception, cache-oblivious models have been extensively used for designing locality-aware fundamental algorithms and data structures [14, 28, 33]. Among those algorithms are scanning algorithms (e.g., traversals, aggregates, and array reversals), divide and conquer algorithms (e.g., median and selection, and matrix multiplication), and sorting algorithms (e.g., mergesort and funnel-sort [34]). Several static data structures (e.g., static search trees, and funnels) and dynamic data structures (e.g., ordered files, b-trees, priority queues, and linked-list) have been also analyzed using the cache-oblivious models. Performance of the said cache-oblivious algorithms and data structures have been reported similar to or sometimes better than the performance of their traditional cache-aware counterparts.

1.2.2 Power and Energy Models

The energy consumed by worldwide computing systems increases 7% annually and becomes a major concern in information technology society. In order to tackle this issue, the research community and industry have proposed several research approaches to reduce the energy consumption of IT systems [69].

One of the key research directions to improve energy-efficiency is to understand how much energy a computing system consumes and characterize the energy consumed by an individual component. By knowing the energy consumption of an algorithm on a specific computing

architecture, researchers and practitioners can design and implement new approaches to reduce the energy consumed by a certain algorithm on a specific platform.

The energy and power consumption of computing systems can be either measured by integrated sensors and external multimeters or estimated by models. Energy and power measurement equipment and sensors are not always available and can be costly to deploy and set up. Therefore, energy and power models are an alternative and convenient method to estimate the energy consumption of a computing component or a whole computing system. The models, however, need to be simple to use and should not interfere with the energy estimated results [69].

We have conducted a study on a power model of Myriad1 platform (cf. Section 3.2)and an energy model for lock-free queues on CPU platform (cf. Section 3.4).

1.3 Libraries and Programming Abstractions

Concurrent data structures (e.g., search trees and queues) and algorithms used as the building blocks of energy-efficient software systems must support high parallelism and fine-grained data locality. In this work we focus on two libraries of the most widely used concurrent data structures, namely the search trees and queues.

Concurrent search trees are fundamental data structures used widely in high performance file systems (e.g., TokuFS and XFS) and database systems (e.g., InnoDB, MyISAM, PostgreSQL and TokuDB). Concurrent search trees are usually used as the back end of dictionaries supporting search, insertion and deletion of records.

Concurrent FIFO queues are fundamental data structures that are key components in applications, algorithms, run-time and operating systems. The producer/consumer pattern, *e.g.*, is a common approach to parallelizing applications where threads act as either producers or consumers and synchronize and stream data items between them using a shared collection. A concurrent queue, *a.k.a.* shared "first-in, first-out" or FIFO buffer, is a shared collection of elements which supports at least the basic operations `Enqueue` (adds an element) and `Dequeue` (removes the oldest element). `Dequeue` returns the element removed or, if the queue is empty, NULL. A large number of lock-free (and wait-free) queue implementations have appeared in the literature, *e.g.*, [82, 65, 79, 66, 52, 37] being some of the most influential or most efficient results. Each implementation of a lock-free queue has obviously its strong and weak points so the impact on performance and energy when choosing one particular implementation for any given situation may not be obvious. As the number of known implementations of lock-free concurrent queues is growing, it is of great interest to describe a framework within which the different implementations can be ranked, according to the parameters that characterize the situation.

In this deliverable, we report our results on concurrent data structures such as concurrent search trees and lock-free queues (cf. Section 4).

1.4 Contributions

The main achievements in this report are summarized in two main parts as below.

White-box methodologies include new cache-oblivious methodology and energy models.

- We have devised a new *relaxed* cache oblivious model that are appropriate for developing energy-efficient concurrent data structures and algorithms.
- We have proposed an power model which is able to predict the power consumed by a program running on a specific number of cores. Given a certain platform and the computation intensity, the model can predict the power consumed by an algorithm, answering the question how many cores are required to run a program to achieve the optimized energy consumption. The model considers both platform and algorithm properties, giving more insights into how to design the algorithm to achieve better energy efficiency. The model has been validated by a set of micro-benchmarks and application kernels such as sparse/dense linear algebra kernels and graph kernels on Movidius embedded platform (Myriad1).
- We have continued the work done in D2.1 [44] on the modeling of queue implementations. We have moved from a grey-box model, where the performance and the power consumption of enqueue and dequeue operations were hidden, to a white-box model, where the impact of those operations are studied separately, and combined at the end. Additionally, we have generalized the model to offer more flexibility according to the workers calling the data structure (parallel section sizes of enqueuers and dequeuers are decoupled).

Programming abstractions and libraries provide a set of implemented concurrent search trees and their energy and performance analyses, as well as a set of implemented lock-free queues, together with a comparison between the predicted and measured throughput and power.

- We have developed a concurrent search tree library that contains several state-of-the-art concurrent search trees such as the non-blocking binary search tree, Software Transactional Memory (STM) based red-black tree, AVL tree, and speculation-friendly tree, fast concurrent B-tree, and static cache-oblivious binary search tree. A family of novel locality-aware and energy efficient concurrent search trees, namely DeltaTree, Balanced DeltaTree, and Heterogeneous DeltaTree are also enclosed in the concurrent search tree library. All the components in this library support the stand-alone benchmark program mode for the purpose of micro-benchmarking and library mode that is pluggable into any C/C++ based programs.
- The DeltaTrees are platform-independent and up to 140% faster and 220% more energy efficient than the state-of-the-art on commodity HPC and embedded platforms. In single thread evaluation, Heterogeneous DeltaTree is 38% faster than std::Set of GCC standard library in the theoretical worst-case scenario of inserting a sorted sequence of keys; and is 50% faster than std::Set for inserting a random sequence of keys the theoretical average-case scenario.

- On lock-free queues on CPU platforms, we have automatized the process of estimating the performance and the power dissipation of any queue implementation, and integrated it in the EXCESS software.

2 EXCESS Platforms and Energy Measurement Settings

In this section, we introduce briefly two EXCESS platforms that we work with. The system descriptions, which have been mentioned in Deliverable D2.1, are presented here to make this deliverable self-contained. As compared to Deliverable D2.1, this section is added with more updated and detailed information on measurement set-up for Myriad1 platform.

2.1 System A: CPU-based Platform

2.1.1 System Description

- CPU: Intel(R) Xeon(R) CPU E5-2687W v2
  - 2 sockets, 8 cores each
  - Max frequency: 3.4GHz, Min frequency: 1.2GHz, frequency speedstep by DVFS: 0.1-0.2GHz. Turbo mode: 4.0GHz.
  - Hyperthreading (disabled)
  - L3 cache: 25M, internal write-back unified, L2 cache: 256K, internal write-back unified. L1 cache (data): 32K internal write-back
- DRAM: 16GB in 4 4GB DDR3 REG ECC PC3-12800 modules run at 1600MTransfers/s. Each socket has 4 DDR3 channels, each supporting 2 modules. In this case 1 channel per socket is used.
- Motherboard: Intel Workstation W2600CR, BIOS version: 2.000.1201 08/22/2013
- Hard drive: Seagate ST10000DM003-9YN162 1TB SATA

2.1.2 Measurement Methodology for Energy Consumption

The energy measurement equipment for System A at CTH, described in Section 2.1.1, is shown in Figure 1 and outlined below. It has previously been described in detail in EXCESS D1.1 [56] and D5.1 [78].

The system is equipped with external hardware sensors for two levels of energy monitoring as well as built in energy sensors:

- At the system level using an external Watts Up .Net [29] power meter, which is connected between the wall socket and the system.
- At the component level using shunt resistors inserted between the power supply unit and the various components, such as CPU, DRAM and motherboard. The signals from the shunt resistors are captured with an Adlink USB-1901 [1] data acquisition unit (DAQ) using a custom utility.

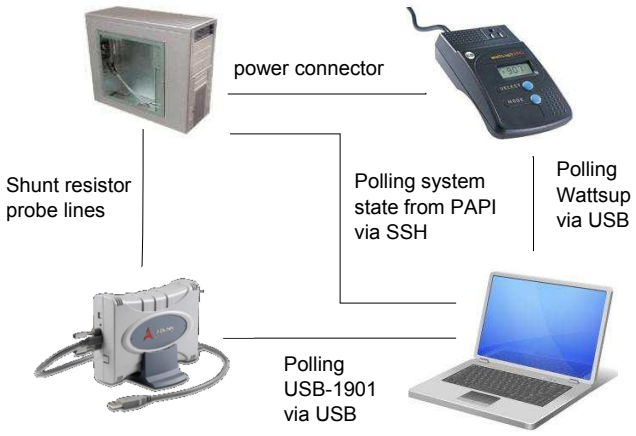


Figure 1: Deployment of energy measurement devices for System A.

- Intel's RAPL energy counters are also available for the CPU and DRAM components. A custom utility based on the PAPI library [17, 85] is used to record these counters and other system state parameters of interest.

For the work presented in this report the component level hardware sensors and the RAPL energy counters have mainly been used.

2.2 System B: Movidius Myriad1 Embedded Platform

2.2.1 Movidius Myriad1 Architecture

The Myriad1 platform developed by Movidius contains eight separate SHAVE (Streaming Hybrid Architecture Vector Engine) processors and one RISC core namely LEON. Each SHAVE one resides on one solitary power island.

The SHAVE processor contains a set of register files and a set of arithmetic units as Figure 2. In this work, we consider the following registers and functional units as described below.

- Integer Register File (IRF) Register file for storing integers from either the IAU or the SAU.
- Scalar Register File (SRF) Register file for storing integers from either the IAU or the SAU.
- Vector Register File (VRF) Register file for storing integers from either the VAU.
- Integer Arithmetic Unit (IAU) Performs all arithmetic instructions that operate on integer numbers, accesses the IRF.
- Scalar Arithmetic Unit (SRF) Performs all Scalar integer/floating point arithmetic.

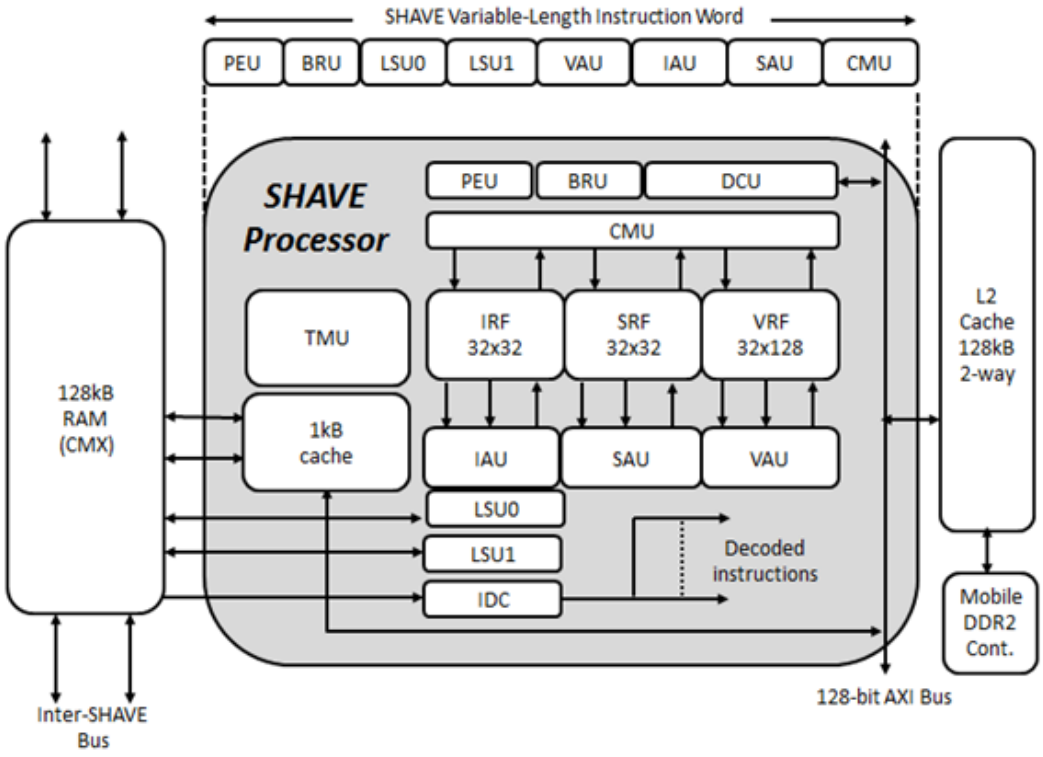


Figure 2: SHAVE Instruction Units

- Vector Arithmetic Unit (VAU) Performs all Vector integer/floating point arithmetic.
- Load Store Unit (LSU) There are two LSUs (LSU0 & LSU1) and they perform any memory access and IO instructions.
- Control Move Unit (CMU) This unit interacts with all register files, and allows for comparing and moving between the register files.

The memory architecture of Myriad1 obtained from deliverable D4.1 is shown in Figure 3. Eight SHAVE cores can access Double Data Rate Random Access Memory (DDR RAM) via L2 cache or bypass L2 cache.

Except from DDR RAM, Movidius introduces a new memory component on-chip RAM containing one megabyte of internal memory with high bandwidth local storage of data and instruction code for the SHAVE processors. This memory component is named CMX. The CMX is constructed to allow eight SHAVEs parallel access to data and program code memory without stalling. Each SHAVE can access data on its own slice of CMX. It can also access other CMX slices of other SHAVE cores with slower time as the trade-off. In the later sections, the energy model is validated with both memory components CMX and DDR.

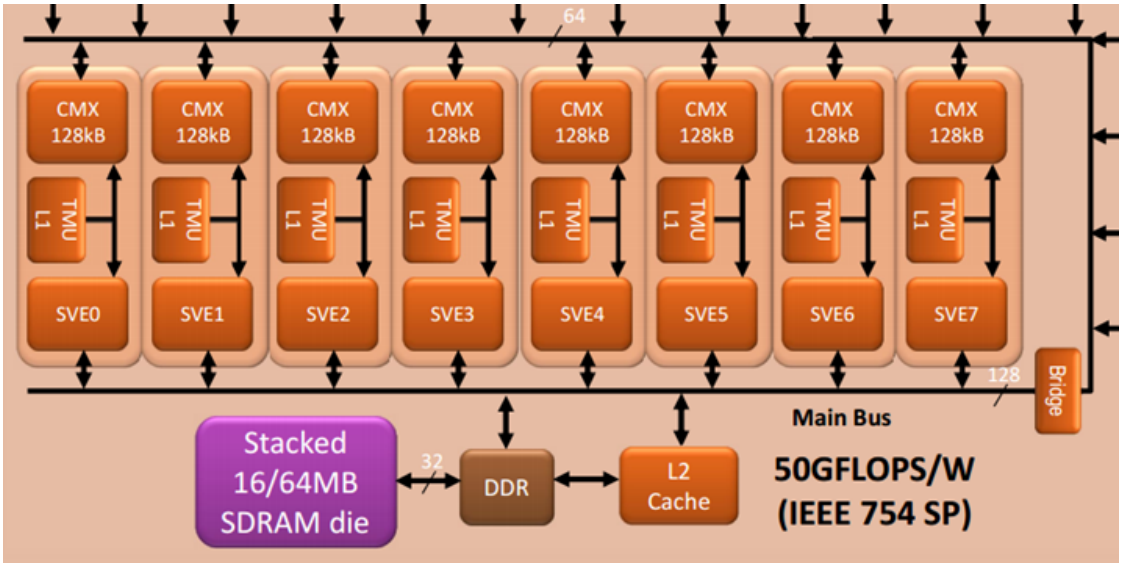


Figure 3: Myriad1 Memory Hierarchy

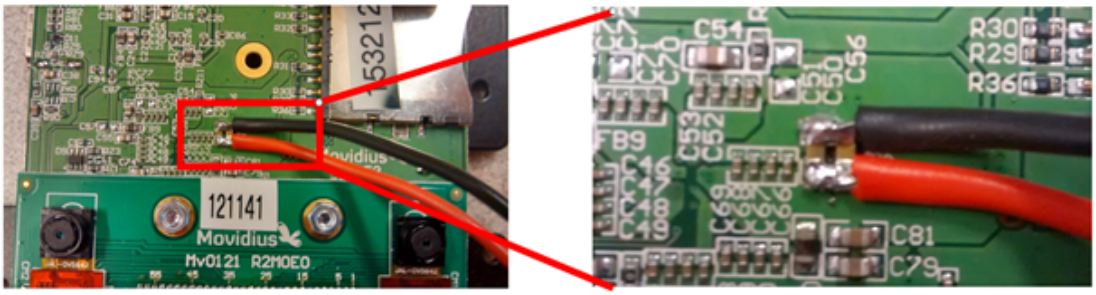


Figure 4: Myriad1 Power Supply Modification

2.2.2 Myriad1 Measurement Set-up

The platform supports the measurement of the power consumed only by the Myriad1 chip. We use a bench setup consisting of Myriad1 MV153 board, a DC step down converter down-regulating the 5V wall PSU to the 1.2V core voltage and one HAMEG multimeter measuring all the voltage, current and consumed power values.

The modifications were made to the MV153 board to bypass the on-board voltage regulator which down-regulates the 5V wall PSU to the 1.2V core voltage required by Myriad1. That allows an external bench power-supply to be used in its place as shown in Figure 4. The MV153 board is modified in order to measure the voltage, current and the consumed power of only Myriad1 chip instead of the whole board. The HAMEG multimeter provides the measured data at a rate of 50 times per second which is able to capture the measurements for benchmarks with execution time longer than 20 milliseconds.

3 White-box Methodologies

In this section, we present our studies on white-box methodologies including cache-oblivious methodology, a power model for Myriad1 platform and an energy model for lock-free concurrent queues.

3.1 Cache-oblivious Methodology

This section presents the cache-oblivious (CO) methodology. As pointed in Section 1.2.1, theoretical execution models that promote data locality are needed in order to devise energy-efficient concurrent algorithms and data structures. Better data locality will result in higher energy efficiency since energy consumption caused by data transfer is predicted to dominate the total energy consumption [25].

We first present two memory models: 1) the I/O model [3] and 2) cache-oblivious model [34], which enable the analysis of data transfer between two levels of the memory hierarchy. We then present some examples of CO algorithms and data structures that are analyzed using the CO models in Section 3.1.2 and 3.1.3, respectively. Section 3.1.4 presents a new relaxed cache-oblivious model on which a new concurrency-aware van Emde Boas (vEB) layout is devised (cf. Section 3.1.5).

3.1.1 Preliminaries

3.1.1.1 I/O model.

The I/O[2] model was introduced by Aggarwal and Vitter [3]. In their seminal paper, Aggarwal and Vitter postulated that the memory hierarchy consists of two levels, an internal memory with size M (e.g., DRAM) and an external storage of infinite size (e.g., disks). Data is transferred in B-sized blocks between those two levels of memory and the CPU can only access data that are available in the internal memory. In the I/O model, an algorithm's time complexity is assumed to be dominated by how many block transfers are required, as loading data from disk to memory takes much more time than processing the data.

For this I/O model, B-tree [7] is an optimal search tree [23]. B-trees and its concurrent variants [12, 21, 38, 39] are optimized for a known memory block size B (e.g., page size) to minimize the number of memory blocks accessed by the CPU during a search, thereby improving data locality. The I/O transfer complexity of B-tree is $O(\log_B N)$, the optimal.

However, the I/O model has its drawbacks. Firstly, to use this model, an algorithm has to know the B and M (memory size) parameters in advance. The problem is that these parameters are sometimes unknown (e.g., when memory is shared with other applications) and most importantly not portable between different platforms. Secondly, in reality there are different block sizes at different levels of the memory hierarchy that can be used in the design of locality-aware data layout for search trees. For example in [57, 72], Intel engineers

[2]The term "I/O" is from now on used a shorthand for block I/O operations

have come out with very fast search trees by crafting a platform-dependent data layout based on the register size, SIMD width, cache line size, and page size.

Existing B-trees limit spatial locality optimization to the memory level with block size B, leaving access to other memory levels with different block size unoptimized. For example a traditional B-tree that is optimized for searching data in disks (i.e., B is page size), where each node is an array of sorted keys, is optimal for transfers between a disk and RAM (cf. Figure 5c). However, data transfers between RAM and last level cache (LLC) are no longer optimal. For searching a key inside each B-sized block in RAM, the transfer complexity is $\Theta(\log(B/L))$ transfers between RAM and LLC, where L is the cache line size. Note that a search with optimal cache line transfers of $O(\log_L B)$ is achievable by using the van Emde Boas layout [13]. This layout has been proved to be optimal for search using the cache-oblivious model [34].

Cache-oblivious model

The cache-oblivious model was introduced by Frigo et al. in [34], which is similar to the I/O model except that the block size B and memory size M are unknown. Using the same analysis of the Aggarwal and Vitter's two-level I/O model, an algorithm is categorized as *cache-oblivious* if it has no variables that need to be tuned with respect to hardware parameters, such as cache size and cache-line length in order to achieve optimality, assuming that I/Os are performed by an optimal off-line cache replacement strategy.

If a cache-oblivious algorithm is optimal for arbitrary two-level memory, the algorithm is also optimal for any adjacent pair of available levels of the memory hierarchy. Therefore without knowing anything about memory level hierarchy and the size of each level, a cache-oblivious algorithm can automatically adapt to multiple levels of the memory hierarchy. In [14], cache-oblivious algorithms were reported performing better on multiple levels of memory hierarchy and more robust despite changes in memory size parameters compared to the cache-aware algorithms.

One simple example is that in the cache-oblivious model, B-tree is no longer optimal because of the unknown B. Instead, the van Emde Boas (vEB) layout-based trees that are described by Bender [9, 10, 11] and Brodal, [13], are optimal. We would like to refer the readers to [14, 34] for a more comprehensive overview of the I/O model and cache-oblivious model.

We provide some of the examples of cache-oblivious algorithms and cache oblivious data structures in the following texts.

3.1.2 Cache-oblivious Algorithms

3.1.2.1 Scanning algorithms and their derivatives

One example of a naive cache-oblivious (CO) algorithm is the *linear scanning* of an N element array that requires $\Theta(N/B)$ I/Os or transfers. Bentley's *array reversal algorithm* and Blum's *linear time selection algorithm* are primarily based on the scanning algorithm, therefore they also perform in $\Theta(N/B)$ I/Os [14, 28].

3.1.2.2 Divide and conquer algorithms.

Another example of CO algorithms in divide and conquer algorithms is the matrix operation algorithms. Frigo et al. proved that *transposition* of an $n \times m$ matrix was optimally solved in $\mathcal{O}(mn/B)$ I/Os and the *multiplication* of an $m \times n$-matrix and an $n \times p$-matrix was solved using $\mathcal{O}((mn+np+mp)/B+mnp/(B\sqrt{M}))$ I/Os, where M is the memory size [34]. As for square matrices (e.g., $N \times N$), using the Strassen's algorithm, the required I/O bound has been proved to be $O(N^2/B + N^{\lg 7}/B\sqrt{M})$, the optimal.

3.1.2.3 Sorting algorithms.

Demaine gave two examples of cache-oblivious sorting algorithm in his brief survey paper [28], namely the *mergesort* and *funnelsort* [34]. In the same text he also wrote that both sorting algorithms achieved the optimal $\Theta(\frac{N}{B}\log_2 \frac{N}{B})$ I/Os, matching those in the original analysis of Aggarwal and Vitter [3].

3.1.3 Cache-oblivious Data Structures

3.1.3.1 Static data structures

One of the examples of cache-oblivious (CO) static data structures is the *CO search trees* that can be achieved using the van Emde Boas (vEB) layout [71, 84]. The vEB-based trees recursively arrange related data in contiguous memory locations, minimizing data transfer between any two adjacent levels of the memory hierarchy (cf. Figure 7).

Figure 5 illustrates the vEB layout, where the size B of memory blocks transferred between 2-level memory in the I/O model [3] is 3 (cf. Section 3.1.1.1). Traversing a complete binary tree with the Breadth First Search layout (or BFS tree for short) (cf. Figure 5a) with height 4 will need three memory transfers to locate the key at leaf-node 13. The first two levels with three nodes (1, 2, 3) fit within a single block transfer while the next two levels need to be loaded in two separate block transfers that contain nodes (6, 7, 8)[3] and nodes (13, 14, 15), respectively. Generally, the number of memory transfers for a BFS tree of size N is $(\log_2 N - \log_2 B) = \log_2 N/B \approx \log_2 N$ for $N \gg B$.

For a vEB tree with the same height, the required memory transfers is only two. As shown in Figure 5b, locating the key in leaf-node 12 requires only a transfer of nodes (1, 2, 3) followed by a transfer of nodes (10, 11, 12). Generally, the memory transfer complexity for searching for a key in a tree of size N is now reduced to $\frac{\log_2 N}{\log_2 B} = \log_B N$, simply by using an efficient tree layout so that nearby nodes are located in adjacent memory locations. If $B = 1024$, searching a BFS tree for a key at a leaf requires 10x (or $\log_2 B$) more I/Os than searching a vEB tree with the same size N where $N \gg B$.

On commodity machines with multi-level memory, the vEB layout is even more efficient. So far the vEB layout is shown to have $\log_2 B$ less I/Os for two-level memory. In a typical machine having three levels of cache (with cache line size of 64B), a RAM (with page size

[3]For simplicity, we assume that the memory controller transfers a memory block of 3 nodes starting at the address of the node requested.

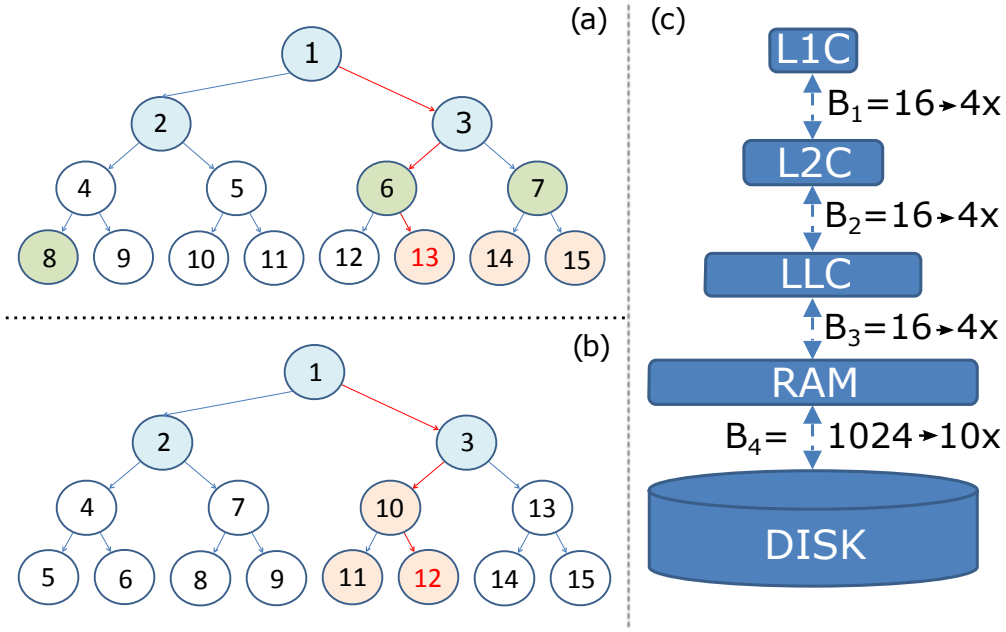


Figure 5: Illustration of required memory transfers in searching for (a) key 13 in BFS tree and (b) key 12 in vEB tree, where node's value is its memory location. An example of multi-level memory is shown in (c), where B_x is the block size B between levels of memory.

of 4KB) and a disk, searching a vEB tree can achieve up to 640x less I/Os than searching a BFS tree, assuming the node size is 4 bytes (Figure 5c).

3.1.3.2 Dynamic data structures.

In a standard *linked-list* structure supporting traversals, insertions and deletions, the best-known cache-oblivious solution was $\mathcal{O}((\lg^2 N)/B)$ I/Os for updates and $\mathcal{O}(K/B)$ for traversing K elements in the list [28].

The first cache-oblivious *priority queue* was due to Arge et al. [5] and it supports inserts and delete-min operations in $\mathcal{O}({}^{1}/_{B} \log_{M/B} {}^{N}/_{B})$ I/Os.

The vEB layout in static cache-oblivious search tree has inspired many cache-oblivious *dynamic search trees* such as cache-oblivious B-trees [9, 10, 11] and cache-oblivious binary trees [13]. All of these search tree implementations have been proved having the optimal bounds of $\mathcal{O}(\log_B N)$ in searches and require amortized $\mathcal{O}(\log_B N)$ I/Os for updates.

However, vEB-based trees poorly support *concurrent* update operations. Inserting or deleting a node may result in relocating a large part of the tree in order to maintain the vEB layout (cf. Section 3.1.5). Bender et al. [11] discussed the problem and provided important theoretical designs of concurrent vEB-based B-trees. Nevertheless, we have found that the theoretical designs are not very efficient in practice due to the actual overhead of maintaining necessary pointers as well as their large memory footprint.

3.1.4 New Relaxed Cache-oblivious Model

We observe that it is unnecessary to keep a vEB-based tree in a contiguous block of memory whose size is greater than some upper bound. In fact, allocating a contiguous block of memory for a vEB-based tree does not guarantee a contiguous block of *physical memory*. Modern OSes and systems utilize different sizes of continuous physical memory blocks, for example, in the form of pages and cache-lines. A contiguous block in virtual memory might be translated into several blocks with gaps in RAM; also, a page might be cached by several cache lines with gaps at any level of cache. This is one of the motivations for the new relaxed cache oblivious model proposed.

We define *relaxed cache oblivious* algorithms to be cache-oblivious (CO) algorithms with the restriction that an upper bound UB on the unknown memory block size B is known in advance. As long as an upper bound on all the block sizes of multilevel memory is known, the new relaxed CO model maintains the key feature of the original CO model [34]. First, temporal locality is exploited perfectly as there are no constraints on cache size M in the model. As a result, an optimal offline cache replacement policy can be assumed. In practice, the Least Recently Used (LRU) policy with memory of size $(1+\epsilon)M$, where $\epsilon > 0$, is nearly as good as the optimal replacement policy with memory of size M [74]. Second, analysis for a simple two-level memory are applicable for an unknown multilevel memory (e.g., registers, L1/L2/L3 caches and memory). Namely, an algorithm that is optimal in terms of data movement for a simple two-level memory is asymptotically optimal for an unknown multilevel memory. This feature enables algorithm designs that can utilize fine-grained data locality in the multilevel memory hierarchy of modern architectures.

The upper bound on the contiguous block size can be obtained easily from any system (e.g., page-size or any values greater than that), which is platform-independent. In fact, the search performance in the new relaxed cache oblivious model is resilient to different upper bound values (cf. Lemma 3.1 in Section 3.1.5).

3.1.5 New Concurrency-aware van Emde Boas Layout

We propose improvements to the conventional van Emde Boas (vEB) layout to support high performance and high concurrency, which results in new *concurrency-aware* dynamic vEB layout. We first define the following notations that will be used to elaborate on the improvements:

- b_i (unknown): block size in terms of the number of nodes at level i of the memory hierarchy (like B in the I/O model [3]), which is unknown as in the cache-oblivious model [34]. When the specific level i of the memory hierarchy is irrelevant, we use notation B instead of b_i in order to be consistent with the I/O model.

- UB (known): the upper bound (in terms of the number of nodes) on the block size b_i of all levels i of the memory hierarchy.

- $\Delta Node$: the largest recursive subtree of a van Emde Boas-based search tree that contains at most UB nodes (cf. dashed triangles of height 2^L in Figure 6b). ΔNode is a

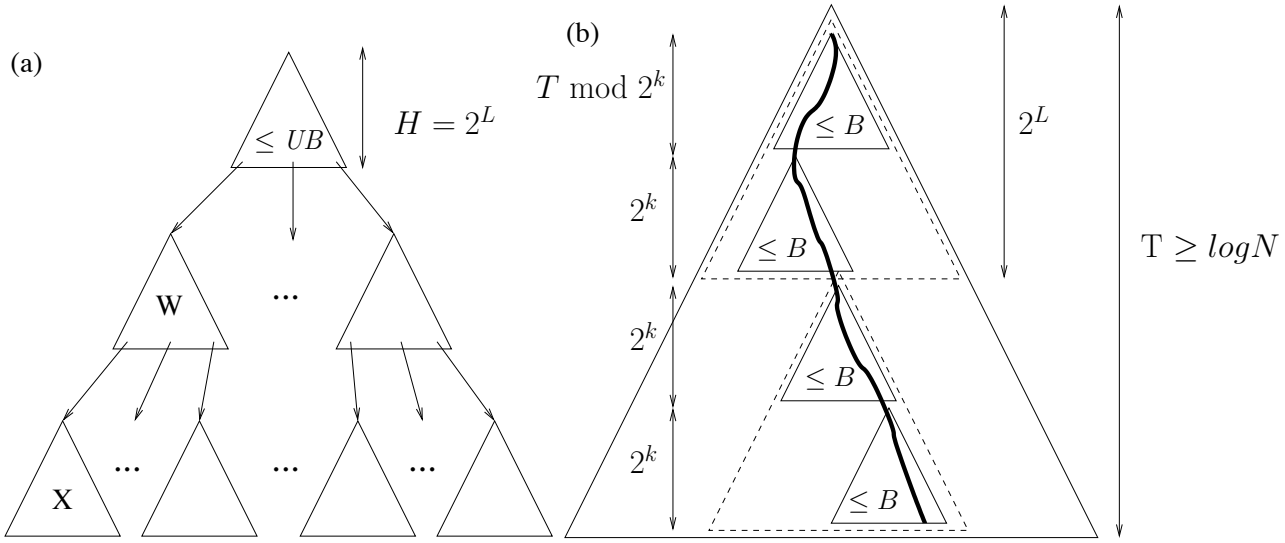


Figure 6: *(a)* New concurrency-aware vEB layout. *(b)* Search using concurrency-aware vEB layout.

fixed-size tree-container with the vEB layout.

- "level of detail" k is a partition of the tree into recursive subtrees of height at most 2^k.
- Let L be the level of detail of ΔNode. Let H be the height of a ΔNode, we have $H = 2^L$. For simplicity, we assume $H = \log_2(UB + 1)$.
- N, T: size and height of the whole tree in terms of basic nodes (not in terms of ΔNodes).

Conventional van Emde Boas (vEB) layout. The conventional van Emde Boas (vEB) layout has been introduced in cache-oblivious data structures [9, 10, 11, 13, 34]. Figure 7 illustrates the vEB layout. Suppose we have a complete binary tree with height h. For simplicity, we assume h is a power of 2, i.e., $h = 2^k, k \in \mathbb{N}$. The tree is recursively laid out in the memory as follows. The tree is conceptually split between nodes of height $h/2$ and $h/2 + 1$, resulting in a top subtree T and $m_1 = 2^{h/2}$ bottom subtrees $W_1, W_2, \cdots, W_{m_1}$ of height $h/2$. The $(m_1 + 1)$ top and bottom subtrees are then located in contiguous memory locations where T is located before $W_1, W_2, \cdots, W_{m_1}$. Each of the subtrees of height $h/2$ is then laid out similarly to $(m_2 + 1)$ subtrees of height $h/4$, where $m_2 = 2^{h/4}$. The process continues until each subtree contains only one node, i.e., the finest *level of detail*, 0.

The main feature of the vEB layout is that the cost of any search in this layout is $O(\log_B N)$ memory transfers, where N is the tree size and B is the *unknown* memory block size in the cache-oblivious model [34]. Namely, its search is cache-oblivious. The search cost is the optimal and matches the search bound of B-trees that requires the memory block size B to be *known in advance*. Moreover, at any level of detail, each subtree in the vEB layout is stored in a contiguous block of memory.

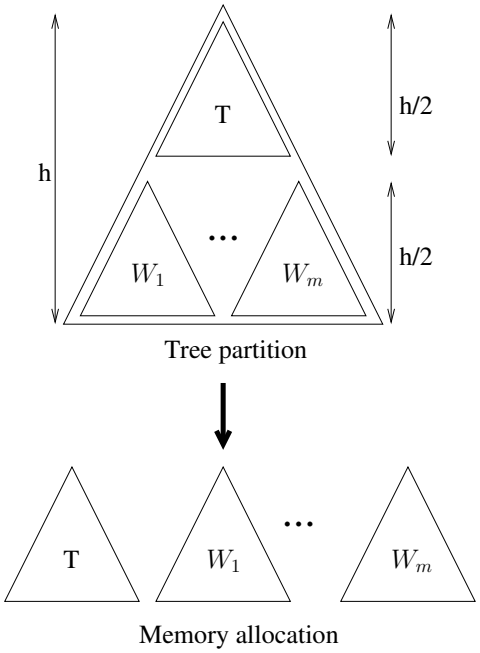


Figure 7: Static van Emde Boas (vEB) layout: a tree of height h is recursively split at height $h/2$. The top subtree T of height $h/2$ and $m = 2^{h/2}$ bottom subtrees $W_1; W_2; \ldots; W_m$ of height $h/2$ are located in contiguous memory locations where T is located before $W_1; W_2; \ldots; W_m$.

Although the conventional vEB layout is helpful for utilizing data locality, it poorly supports concurrent update operations. Inserting (or deleting) a node at position i in the contiguous block storing the tree may restructure a large part of the tree. For example, inserting new nodes in the full subtree W_1 (a leaf subtree) in Figure 7 will affect the other subtrees $W_2, W_3, \cdots, W_m$ by rebalancing existing nodes between W_1 and the subtrees in order to have space for new nodes. Even worse, we will need to allocate a new contiguous block of memory for the whole tree if the previously allocated block of memory for the tree runs out of space [13]. Note that we cannot use dynamic node allocation via pointers since at *any* level of detail, each subtree in the vEB layout must be stored in a *contiguous* block of memory.

Concurrency-aware vEB layout. In order to make the vEB layout suitable for highly concurrent data structures with update operations, we introduce a novel *concurrency-aware* dynamic vEB layout. Our key idea is that if we know an upper bound UB on the unknown memory block size B, we can support dynamic node allocation via pointers while maintaining the optimal search cost of $O(\log_B N)$ memory transfers without knowing B (cf. Lemma 3.1). The assumption on known upper bound UB is supported by the fact that in practice it is unnecessary to keep the vEB layout in a contiguous block of memory whose size is greater than some upper bound.

Figure 6a illustrates the new concurrency-aware vEB layout based on the relaxed cache oblivious model. Let L be the coarsest level of detail such that every recursive subtree contains at most UB nodes. Namely, let H and S be the height and size of such a subtree then $H = 2^L$ and $S = 2^H - 1 < UB$. The tree is recursively partitioned into level of detail L where each subtree represented by a triangle in Figure 6a, is stored in a contiguous memory block of size UB. Unlike the conventional vEB, the subtrees at level of detail L are linked to each other using pointers, namely each subtree at level of detail $k > L$ is not stored in a contiguous block of memory. Intuitively, since UB is an upper bound on the unknown memory block size B, storing a subtree at level of detail $k > L$ in a contiguous memory block of size greater than UB, does not reduce the number of memory transfers, provided there is perfect alignment. For example, in Figure 6a, traveling from a subtree W at level of detail L, which is stored in a contiguous memory block of size UB, to its child subtree X at the same level of detail will result in at least two memory transfers: one for W and one for X. Therefore, it is unnecessary to store both W and X in a contiguous memory block of size $2\,UB$. As a result, the memory transfer cost for search operations in the new concurrency-aware vEB layout is intuitively the same as that of the conventional vEB layout (cf. Lemma 3.1) while the concurrency-aware vEB supports high concurrency with update operations.

Lemma 3.1. *For any upper bound UB of the* unknown *memory block size B, a search in a complete binary tree with the new concurrency-aware vEB layout achieves the optimal memory transfer $O(\log_B N)$, where N and B are the tree size and the* unknown *memory block size in the cache-oblivious model [34], respectively.*

Proof. (Sketch) Figure 6b illustrates the proof. Let k be the coarsest level of detail such that every recursive subtree contains at most B nodes. Since $B \leq UB$, $k \leq L$, where L is the coarsest level of detail at which every recursive subtree (ΔNodes) contains at most UB nodes. That means there are at most 2^{L-k} subtrees along the search path in a ΔNode and no subtree of depth 2^k is split due to the boundary of ΔNodes. Namely, triangles of height 2^k fit within a dashed triangle of height 2^L in Figure 6b.

Because at any level of detail $i \leq L$ in the concurrency-aware vEB layout, a recursive subtree of depth 2^i is stored in a contiguous block of memory, each subtree of depth 2^k *within* a ΔNode is stored in at most 2 memory blocks of size B (depending on the starting location of the subtree in memory). Since every subtree of depth 2^k fits in a ΔNode (i.e., no subtree is stored across two ΔNodes), every subtree of depth 2^k is stored in at most 2 memory blocks of size B.

Since the tree has height T, $\lceil T/2^k \rceil$ subtrees of depth 2^k are traversed in a search and thereby at most $2\lceil T/2^k \rceil$ memory blocks are transferred.

Since a subtree of height 2^{k+1} contains more than B nodes, $2^{k+1} \geq \log_2(B+1)$, or $2^k \geq \frac{1}{2}\log_2(B+1)$.

We have $2^{T-1} \leq N \leq 2^T$ since the tree is a *complete* binary tree. This implies $\log_2 N \leq T \leq \log_2 N + 1$.

Therefore, the number of memory blocks transferred in a search is $2\lceil T/2^k \rceil \leq 4\lceil \frac{\log_2 N+1}{\log_2(B+1)} \rceil = 4\lceil \log_{B+1} N + \log_{B+1} 2 \rceil = O(\log_B N)$, where $N \geq 2$. □

A library of novel locality-aware and energy efficient concurrent search trees based on the new concurrency-aware vEB layout is presented in Section 4.1

3.2 Power Model for Computational Algorithms on Movidius Platform

The objective of this study is to build a power model that can estimate the power consumption of an algorithm on Myriad platform. In order to do so, the model considers both algorithmic and platform properties. Our model is inspired by Amdahl law analysis and the Roofline model of energy [19, 20]. Although the Roofline model also connects algorithmic and platform properties, it does not consider the number of cores nor memory contention when the number of cores varies. By estimating the power consumed by a system running a specific number of cores, our model can predict the number of cores required to achieve the least energy consumption. In this report, the model has been evaluated by micro-benchmarks and application kernels such as sparse/dense linear algebra kernels and graph kernels on Movidius embedded platform (Myriad1).

3.2.1 Energy Model Description

In Deliverable D2.1, we presented our initial ideas on an energy model for Myriad1 platform. In this Deliverable, we have improved the model by considering both computational and data movement power. Another improvement is that we create the micro-benchmarks whose sizes are approximately 1 KB that fits to Myriad1 cache buffer. The measurements are, therefore, more accurate and do not need to include inter-operation cost like the power model in Deliverable D2.1. The details on how to build the model and its evaluation are described as follows.

3.2.1.1 Computational Power

The computational power of a system is the required power to perform its computation using data from closest memory components such as register files. The computational power includes static power, active power and dynamic power of involved functional units. At first we aim to find out the corresponding values of static, active and dynamic power of each SHAVE core and each SHAVE arithmetic unit. The experimental results have shown that the power consumption of Movidius Myriad1 platform is ruled by the following model:

$$P^{comp} = P^{sta} + \#\{\text{SHV}\} \times (P^{act} + P^{dyn}_{SHV}) \quad (1)$$

In that formula, the static power P^{sta} is the needed power when the Myriad1 processor is on. The P^{act} is the power consumed when a SHAVE core is on. Therefore, if benchmarks or programs use n SHAVE cores, the active power needs to be multiplied with the number of used SHAVE cores.

The dynamic power P^{dyn}_{SHV} of each SHAVE is the power consumed by all working operation units in one SHAVE. Operation units include arithmetic units (e.g., IAU, VAU, SAU and

CMU) and load store unit (e.g., LSU1, LSU2). The experimental results show that different arithmetic operation units have different P^{dyn} values.

When running a benchmark with one more SHAVE, we can identify the sum of SHAVE P^{act} and P^{dyn} which is the power difference of the two runs (the run with one SHAVE core and the run with two SHAVEs). Given the sum of P^{act} and P^{dyn}, P^{sta} is derived from the Equations 1 . The experimental results show the average value of P^{sta} from all micro-benchmarks:

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

Dynamic power P^{dyn}_{SHV} of SHAVE running multiple units is computed by the following formula.

$$P^{dyn}_{SHV} = \sum_i P^{dyn}_i(op) \tag{3}$$

That means P^{dyn}_{SHV} is the sum of P^{dyn} of all involved arithmetic units. E.g. $P^{dyn}_{(SauIau)}) = P^{dyn}_{(Sau)} + P^{dyn}_{(Iau)}$.

For each operation unit, we obtain the two parameters P^{dyn}_{op} and P^{act} by using the actual power consumption of the benchmark for individual units and multiple units.

For example, P^{dyn}_{Iau}, P^{dyn}_{Sau} and P^{act} are identified from Equations 4, 5, 6.

$$P^{dyn}_{(Iau)} = P^{sta} + \#\{\text{SHV}\} \times (P^{act} + P^{dyn}_{(Iau)}) \tag{4}$$

$$P^{dyn}_{(Sau)} = P^{sta} + \#\{\text{SHV}\} \times (P^{act} + P^{dyn}_{(Sau)}) \tag{5}$$

$$P^{dyn}_{(SauIau)} = P^{sta} + \#\{\text{SHV}\} \times (P^{act} + P^{dyn}_{(Sau)} + P^{dyn}_{(Iau)}) \tag{6}$$

Then, the average value of P^{act} from all micro-benchmarks is derived as below.

$$P^{act} = 29.33 \text{ mW} \tag{7}$$

Given the values of P^{sta}, P^{act} and P^{dyn}_{SHV} as Equations 2, 7 and 3 respectively, the computation power P^{comp} for Movidius Myriad1 can be estimated by applying the below formula:

$$P^{comp} = P^{sta} + \#\{\text{SHV}\} \times \left(P^{act} + \sum_i P^{dyn}_i(op) \right) \tag{8}$$

Table 1 lists the dynamic power of each operation unit.

3.2.1.2 Data Movement Power

Since Myriad1 has two memory components within the chip (CMX and DDR), we model the data movement for both of the memory components. The data is either moved from CMX to registers or from DDR to registers before being processed by arithmetic units.

Data movement is performed by two units of the SHAVE core namely Load Store Unit (LSU). LSU loads and stores data from the memory components such as DDR and CMX to the register memory of SHAVE processor. Therefore, the consumed power of data movement

Operation Unit	P_{op}^{dyn}
SauXor	14.68
SauMul	17.69
VauXor	34.34
VauMul	51.98
IauXor	15.91
IauMul	18.48
CmuCpss	12.62
CmuCpivr	18.84
LsuLoad	29.87
LsuStore	37.49

Table 1: P_{op}^{dyn} of SHAVE Operation Units

also includes the static power P^{sta}, the active power P^{act}, LSU unit power P_{LSU}^{dyn} and the contention power P^{ctn} when there are more cores accessing memory than the number of available memory ports. Below is the formula to estimate the power consumption of data movement.

$$P^{data} = P^{sta} + min(m, n) \times (P^{act} + P_{LSU}^{dyn}) + max(n - m, 0) \times P^{ctn} \tag{9}$$

In the formula, n is the number of active cores running the program; m is the number representing memory ports or bandwidth available in the platform; the contention power P^{ctn} is the power overhead occurring when SHAVE cores actively wait for accessing data because of the limited memory ports (or bandwidth) in the platform architecture.

Myriad1 has two separate memory components: CMX and DDR. As described in section 2.2.1, each SHAVE core has its own CMX memory slice meaning that the number of data accessing ports m equals to the number of active cores n when transferring data between CMX memory and registers. Therefore, data movement for CMX does not cause contention power and the power consumption of CMX data movement P_{CMX}^{data} is:

$$P_{CMX}^{data} = P^{sta} + min(n, n) \times (P^{act} + P_{LSU}^{dyn}) + max(0, 0) \times P^{ctn} \tag{10}$$

$$P_{CMX}^{data} = P^{sta} + n \times (P^{act} + P_{LSU}^{dyn}) \tag{11}$$

The data movement between DDR and registers, however, has the contention power due to the limited DDR ports/bandwidth shared among all SHAVE cores.

$$P_{DDR}^{data} = P^{sta} + min(m, n) \times (P^{act} + P_{LSU}^{dyn}) + max(n - m, 0) \times P^{ctn} \tag{12}$$

3.2.1.3 A Power Model for Computational Algorithms

Typical applications require both computation and data movement. The power consumption then needs to consider the power of both the computation and data movement. In our model,

we use the concept of operational intensity proposed by Williams et.al. [87] to characterize algorithms. An algorithm can be characterized by the amount of computational work W and a number of data accesses Q. W is the number of operations performed by a program. Q is the number of transferred bytes required during a program execution. Both W and Q define the operation intensity I of the algorithm.

$$I = \frac{W}{Q} \tag{13}$$

For the power model, we assume that the core either perform computation or transfer data during its active state. The power consumption of a processor depends on the ratio of the computation to data movement during its execution. This portion can be calculated based on the time ratio of performing computation to transferring data. As the time needed to perform one operation is different from the time required to transfer one byte of data, we introduce a parameter α to the model. The parameter α is the property of the platform and its value depends on each platform architecture. The model considers both computation and data movement cost is derived below.

$$P = P^{comp} \times (\frac{W}{\alpha \times Q + W}) + P^{data} \times (\frac{\alpha \times Q}{\alpha \times Q + W}) \tag{14}$$

After converting W and Q to I by using the Equation 14, the final model is simplified as below:

$$P = P^{comp} \times (\frac{I}{\alpha + I}) + P^{data} \times (\frac{\alpha}{\alpha + I}) \tag{15}$$

The complete model with details to compute the power consumed by a given application/ algorithm is presented as below.

$$\begin{aligned} P = & P^{sta} \\ & + n \times (P^{act} + P^{dyn}_{SHV}) \times (\frac{I}{\alpha + I}) \\ & + (min(m,n) \times (P^{act} + P_{LSU}) + max(n-m,0) \times P^{ctn}) \times (\frac{\alpha}{\alpha + I}) \end{aligned} \tag{16}$$

Table 2 summarizes the parameter list of the proposed model.

3.2.2 Model Validation

3.2.2.1 Experimental validation with micro-benchmarks

Regarding the assembly code of micro-benchmarks, a fixed number of instructions is provided for all the experiments, meaning that each assembly code file contains five instructions in an iteration and the iteration is infinitely repeated. This convention keeps the continuity and consistency of experiments while giving insights into measuring the power consumption with different SHAVE units, enabling the comparisons among SHAVE units. As compared to micro-benchmarks used in Deliverable D2.1, the number of instructions in one iteration

Parameters	**Explanation**
P^{sta}	Static power of a SHAVE core
P^{act}	Active power of a SHAVE core
P^{dyn}_{SHV}	Dynamic power of a SHAVE core
P_{LSU}	Operation power of Load Store Unit
P^{ctn}	Contention power of a SHAVE core
m	Number of memory ports
n	Number of running SHAVE cores
I	Operation intensity of an algorithm
α	Time ratio of data transfer to computation

Table 2: Model Parameter List

for each mico-benchmark is re-calculated so that the whole iteration fits to L1 cache buffer size (1KB).

The assembly code files used in the experimental evaluation contain code that executes the instruction decode and instruction fetch. Majority of tests use pseudo-realistic data. By using pseudo-realistic data we have as many non-zero values as possible and avoiding data value repetition at different offsets. Analyses of experimental results are performed based on an identified set of micro-benchmarks. Each micro-benchmark is executed with different numbers of SHAVE such as 1, 2, 4, 6 and 8 SHAVE cores. A sample code of micro-benchmark performing SauXor operation is given in Figure 8.

Computation micro-benchmarks Applying this model to the computational micro-benchmarks for each of operation units, the model relative error is plotted in the Figure 9. Having all parameter data from Equation 2, 7 and Table 1, the model-predicted values are computed. Relative errors are the percentage difference between the actual power consumption measured by device and the predicted values from the model.

Under this model, the relative error which is the percentage difference between measured and estimated values varies from -5% to 6% which is an 11% margin. This proves that the model can be applied for micro-benchmarks running a single unit or any combination of two or three units in parallel. This model shows the compositionality of power consumption not only for multiple SHAVE cores but also for multiple operation units within a SHAVE.

Data movement micro-benchmarks Data movement micro-benchmarks are the micro-benchmarks running LSU units to transfer data from CMX memory to core registers. The micro-benchmarks are designed to execute load/store operation only or with other functional units. When executed with other computation units, the instructions are in different pipelines to make both LSU and computation units executed in parallel manner.

Since each of the eight SHAVE cores in Myriad1 can access CMX data with its own port, this matches the Equation 11 as presented in Section 3.2.1.2.

```
.version 00.50.00

.code .text.testILP

testILP:

lsu0.ldil i0 ___loop || lsu1.ldih i0 ___loop
lsu0.ldil i1 0xFFFF  || lsu1.ldih i1 0x7FFF //loads fp value in reg
lsu0.ldil i2 0x0D45  || lsu1.ldih i2 0x3F9E //loads fp value in reg
lsu0.ldil i3 0xB924  || lsu1.ldih i3 0x3DFC //loads fp value in reg
lsu0.ldil i4 0x9CA2  || lsu1.ldih i4 0x4008 //loads fp value in reg

    ___loop:
                      sau.xor i10 i2 i1
                      sau.xor i11 i3 i2
                      sau.xor i12 i1 i3
                      sau.xor i13 i1 i2
                      sau.xor i14 i3 i0
                      sau.xor i15 i2 i1
                      sau.xor i10 i2 i1
                      sau.xor i11 i3 i2
                      sau.xor i12 i1 i3
                      sau.xor i13 i1 i2
                      sau.xor i14 i3 i0
                      sau.xor i15 i2 i1
                      sau.xor i10 i2 i1
                      sau.xor i11 i3 i2
                      sau.xor i12 i1 i3
                      sau.xor i13 i1 i2
                      sau.xor i14 i3 i0
                      sau.xor i15 i2 i1
                      sau.xor i10 i2 i1
                      sau.xor i11 i3 i2
                      sau.xor i12 i1 i3
                      sau.xor i13 i1 i2
                      sau.xor i14 i3 i0
                      sau.xor i15 i2 i1
                      sau.xor i10 i2 i1
                      sau.xor i11 i3 i2
                      sau.xor i12 i1 i3
                      sau.xor i13 i1 i2
                      sau.xor i14 i3 i0
                      sau.xor i15 i2 i1
                      sau.xor i10 i2 i1
                      sau.xor i11 i3 i2
                      sau.xor i12 i1 i3
                      sau.xor i13 i1 i2
                      sau.xor i14 i3 i0
                      sau.xor i15 i2 i1
    bru.jmp i0  ||    sau.xor i10 i2 i1
                      sau.xor i11 i3 i2
                      sau.xor i12 i1 i3
                      sau.xor i13 i1 i2
                      sau.xor i14 i3 i0
                      sau.xor i15 i2 i1

.end
```

Figure 8: An example of micro-benchmark is written in asm code to perform SauXor operations

The experimental results of LSU also follow the power model shown in Figure 10. The relative error of data movement micro-benchmarks varies from -4% to 6% which is a 10%

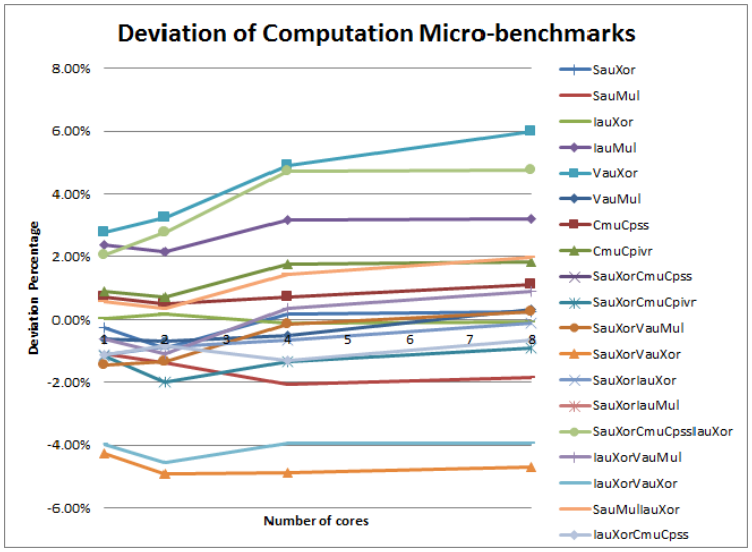


Figure 9: Relative error of the model for computational micro-benchmarks

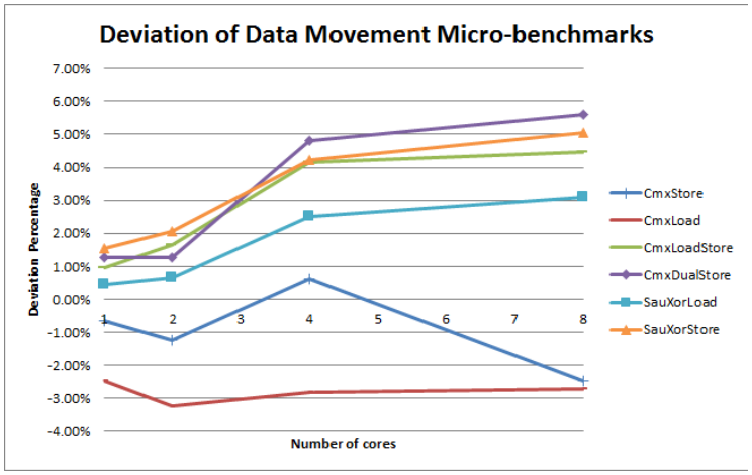


Figure 10: Relative error of the model for data movement micro-benchmarks

margin. This proves that the model can be applied to Myriad1 platform accessing CMX data.

Intensity-based micro-benchmarks: the combination of computation and data movement Since any application requires both computation and data movement, we design micro-benchmarks which execute both computation and data movement units. Unlike the combination of multiple units in different instruction pipelines as computational micro-benchmarks, this set of micro-benchmarks contains sequential instructions to trigger functional units in sequential order. With this way, the assembly code does not use the pipeline optimization feature of Myriad1 platform and therefore, provide raw performance for a more

precise analysis.

In order to validate the power model for any algorithms, the micro-benchmarks used in experiments at this phase also indicate different operation intensities. Operation intensity I is retrieved from the assembly code by counting the number of arithmetic instructions such as Xor, Mul and the number of LSU instructions. The number of arithmetic instructions indicate the amount of work W and LSU instructions multiplied by four (each LSU instruction load 32 bits which equals to four bytes) indicate the number of accessed data bytes Q. By changing the ratio of arithmetic instructions to LSU instructions, we created micro-benchmarks with different intensity values. A sample code of micro-benchmark with intensity $I = 0.25$ is given in Figure 11.

Intensity-based micro-benchmarks using CMX memory to store data has experimental results shown in Figure 12. Since eight SHAVE cores have eight accessing ports to CMX memory, cores do not wait for memory fetching and there is no contention power. The power model of Myriad1 using CMX memory is the Equation 17.

$$\begin{aligned} P = P^{sta} + n \times (P^{act} + P^{dyn}_{SHV}) \times (\frac{I}{\alpha + I}) \\ + n \times (P^{act} + P_{LSU}) \times (\frac{\alpha}{\alpha + I}) \end{aligned} \tag{17}$$

In the CMX power model in Equation 17, all parameters are known except α. By using non-regression techniques in Matlab function *lsqcurvefit* to find unknown parameters, α is identified as equal to one. This is reasonable since accessing four bytes of data in CMX requires average four cycles [67] which means one cycle per byte and an operation used in micro-benchmarks (e.g. SauXor) also requires one cycle to be executed.

We plot the model deviation error of Myriad using CMX in Figure 12. The relative error of intensity micro-benchmarks varies from -9% to 4% which is a 13% margin. This proves that the model can be applied to estimate the power consumption of an algorithm on Myriad1 platform accessing CMX data.

Intensity-based micro-benchmarks using DDR memory to store data has the experimental results as shown in Figure 13. Since eight SHAVE cores share the same data bus to DDR memory, contention power might happen when cores wait for memory fetching. The power model of Myriad1 using DDR memory is the Equation 18.

$$\begin{aligned} P = P^{sta} + n \times (P^{act} + P^{dyn}_{SHV}) \times (\frac{I}{\alpha + I}) \\ + (max(m, n) \times (P^{act} + P_{LSU}) + max(n - m, 0) \times P^{ctn}) \times (\frac{\alpha}{\alpha + I}) \end{aligned} \tag{18}$$

In the DDR power model in Equation 18, there are unknown parameters such as α, m and P^{ctn}. By using non-linear regression techniques, α and P^{ctn} are identified for each intensity value. Figure 14 shows how well the predicted data fits to measured data with the found parameter values. We plot the model deviation error of the model for intensity

```
.version 00.50.00

.include testdata.asminc

.code .text.testILP

testILP:
    lsu0.ldil i18 testmatrix || lsu1.ldih i18 testmatrix
    lsu0.ldil i0 ___loop || lsu1.ldih i0 ___loop
    lsu0.ldil i1 0xFFFF  || lsu1.ldih i1 0x7FFF //loads fp value in reg
    lsu0.ldil i2 0x0D45  || lsu1.ldih i2 0x3F9E //loads fp value in reg
    lsu0.ldil i3 0xB924  || lsu1.ldih i3 0x3DFC //loads fp value in reg
    lsu0.ldil i4 0x9CA2  || lsu1.ldih i4 0x4008 //loads fp value in reg

    ___loop:
                         sau.xor i12 i1 i2
                         LSU0.LDO64.L v23 i18 0x0
                         sau.xor i13 i1 i3
                         LSU0.LDO64.l v22 i18 0x8
                         sau.xor i14 i2 i3
                         LSU0.LDO64.l v21 i18 0x10
                         sau.xor i15 i2 i4
                         LSU0.LDO64.l v20 i18 0x28
                         sau.xor i16 i3 i4
                         LSU0.LDO64.l v19 i18 0x30
                         sau.xor i15 i4 i1
                         LSU0.LDO64.l v18 i18 0x38
                         sau.xor i12 i1 i2
                         LSU0.LDO64.L v23 i18 0x0
                         sau.xor i13 i1 i3
                         LSU0.LDO64.l v22 i18 0x8
                         sau.xor i14 i2 i3
                         LSU0.LDO64.l v21 i18 0x10
                         sau.xor i15 i2 i4
                         LSU0.LDO64.l v20 i18 0x28
                         sau.xor i16 i3 i4
                         LSU0.LDO64.l v19 i18 0x30
                         sau.xor i15 i4 i1
                         LSU0.LDO64.l v18 i18 0x38
                         sau.xor i12 i1 i2
                         LSU0.LDO64.L v23 i18 0x0
                         sau.xor i12 i1 i2
                         LSU0.LDO64.L v23 i18 0x0
                         sau.xor i13 i1 i3
                         LSU0.LDO64.l v22 i18 0x8
                         sau.xor i14 i2 i3
                         LSU0.LDO64.l v21 i18 0x10
        bru.jmp i0 || sau.xor i15 i2 i4
                         LSU0.LDO64.l v20 i18 0x28
                         sau.xor i16 i3 i4
                         LSU0.LDO64.l v19 i18 0x30
                         sau.xor i15 i4 i1
                         LSU0.LDO64.l v18 i18 0x38
.end
```

Figure 11: An example of micro-benchmark is written in asm code with intensity $I = 0.25$

micro-benchmarks using DDR memory in Figure 13. The relative error of intensity micro-benchmarks varies from -16% to 14%. The model has high accuracy at intensity lower than 1 and higher deviation with micro-benchmarks running two cores. This requires more investigation. However, for each intensity, the deviation is less than 24% margin.

The found tunable parameters such as α, m and P^{ctn} are summarized in the Table 3. The parameter values are derived from experimental results on micro-benchmarks using Matlab function $lsqcurvefit$.

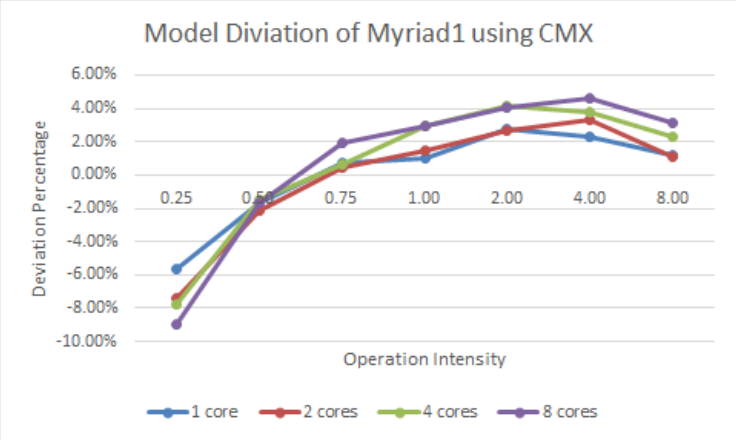


Figure 12: Relative error of the model for intensity micro-benchmarks using CMX

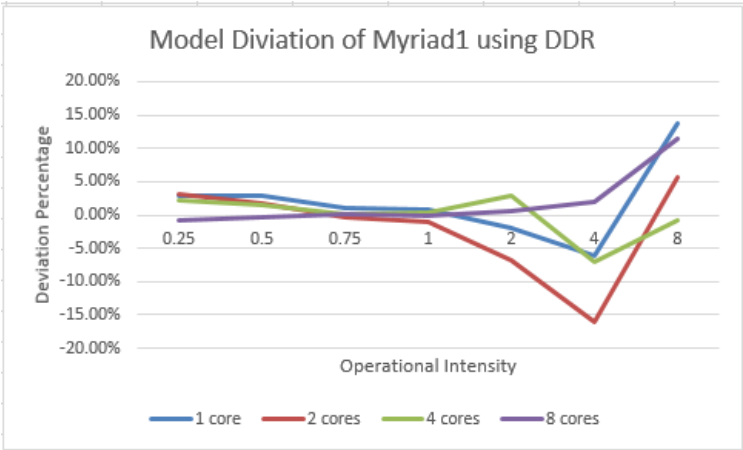


Figure 13: Relative error of intensity micro-benchmarks using DDR

3.2.2.2 Experiment with application benchmarks

We want to analyze application benchmarks that represent for typical applications. Therefore, we follow two metrics to choose the applications for our investigation. Since the energy consumption of an application depends on its operation intensity I, it is one main factor to choose application benchmarks. The energy consumption not only depends on the power but also program execution time. Therefore, the performance speed-up of an application is another main factor we consider to choose application benchmarks. The three benchmarks we choose are Matrix Multiplication, Sparse Matrix Vector Multiplication and Breadth First Search included in Berkeley Dwarfs [6] as shown in Figure 15. Matrix Multiplication is proved to have high operation intensity and high performance speed-up [68] while Sparse Matrix Vector Multiplication has low intensity and high speed-up due to its parallel scalability [86]. Breadth First Search, on the other hand, has low intensity and saturated low scalability [22]. Since applications with high intensity and low speed-up are typical sequen-

Intensity	P^{ctn}	α	m
0.25	0.1	0.91	1
0.5	0.1	1.72	1
0.75	0.1	2.49	1
1	1.97	3.87	1
2	5.25	10	1
4	0.1	10	1
8	0.1	10	2

Table 3: Values of Model Parameters derived from micro-benchmarks

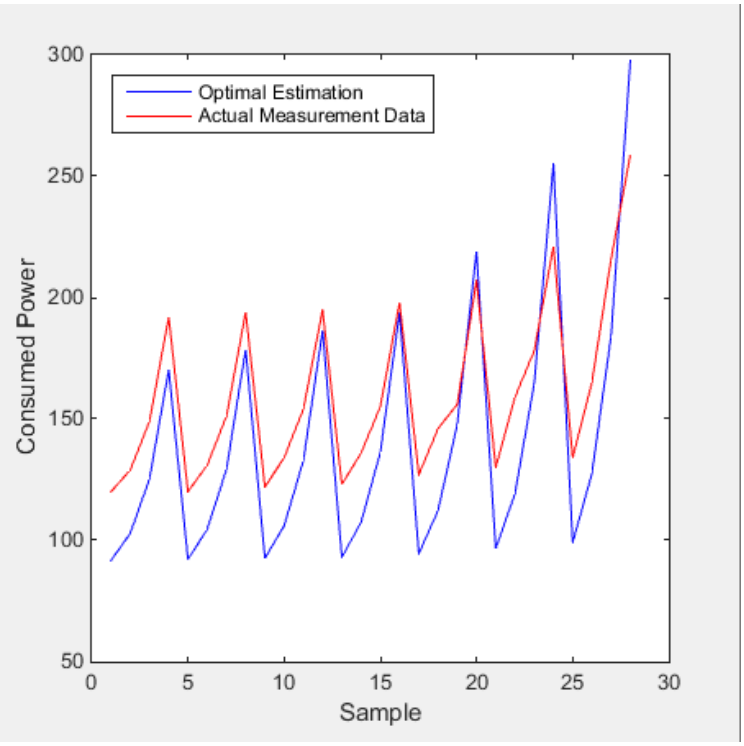


Figure 14: Parameter fitting over the sample of intensity micro-benchmarks on DDR

tial applications, which are not our target applications, so we do not include them in our validation.

Matrix multiplication Matrix multiplication (matmul) has been implemented based on the optimized assembly code for matmul using small blocks of data. Each block is the small submatrix of size 4×4. The matmul algorithm computes matrix C based on two input matrices A and B. All three matrices in this benchmarks are stored in DDR RAM. The operation intensity I of the matmul algorithm increase linearly with the matrix size. We store each matrix element with float type which means four bytes. The number of

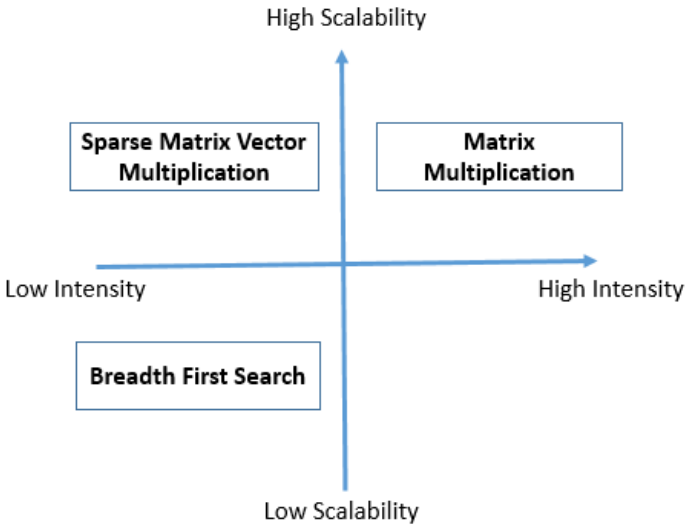


Figure 15: Application Categories

operations and accessed data are calculated based on matrix size n as: $W = 2 \times n^3$ and $Q = 16 \times n^2$. Intensity of matmul is also varied with matrix size as: $I = \frac{W}{Q} = \frac{n}{8}$ [68]. In our experiments, we measured power consumption of matmul with varied matrix size. Figure 16 is the predicted power consumption of matmul over intensity values using our power model. As shown in the diagram, power consumed by computation is the main contribution to the total power. With increased intensity, total power and computation power increase but power from accessing data decreases.

Apply the model by using the model parameters derived from micro-benchmarks, the deviation error of *matmul* estimated power compared to measured data on 8 cores is from 24% to 59% as shown in Figure 17. The highest deviation occurs at size 16×16 because it is performed in short execution time and the multimeter device can not capture the real consumed power. This also happens for *matmul* with matrix size less than 16×16. The deviation occurs because parameters have not considered the data-access patterns to access the same data or different data. Therefore, the model accuracy can be improved when we consider the data-access patterns and find tuning parameters such as α, P_{ctn} and m based on actual measurements of the application.

Sparse matrix vector multiplication Sparse matrix vector multiplication (SpMV) is another kernel benchmark that we implement on Myriad1. All input matrix and vector of this benchmarks reside in DDR RAM. The operation intensity I of SpMV algorithm does not depend on matrix size. The data layout of matrix in this implementation is compressed sparse row (CSR) format. Each element of matrix and vector is also stored with float type. From our implementation, the number of operations and accessed data are calculated based on the size of one matrix dimension n as: $W = 10 \times n$ and $Q = 13 \times 4 \times n$. The intensity of SpMV does not depend on matrix size and is a fixed value: $I = \frac{W}{Q} = \frac{5}{26} = 0.19$. As

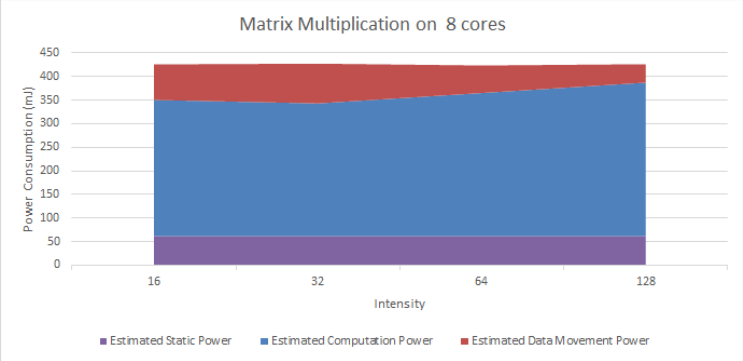


Figure 16: Power Analysis of Matrix Multiplication with 8 cores. Since the intensity of *matmul* is also varied with matrix size as: $I = \frac{W}{Q} = \frac{n}{8}$, its consumed power is also varied by the intensity. The red and blue stacked lines are the estimated power consumed by data movement and computation which contribute to the total power respectively.

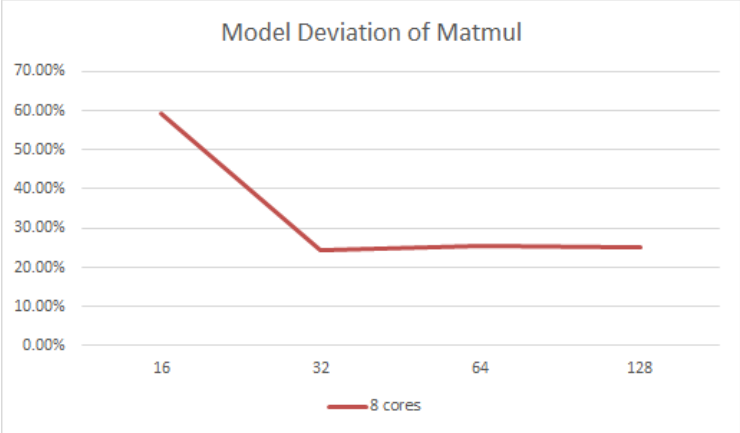


Figure 17: Deviation of estimated power from measured power for Matrix Multiplication

the result, predicted power consumption of SpMV with varied matrix sizes is the same over matrix size as shown in Figure 18.

Each element of matrix and vector is also stored with float type. From our implementation, the number of operations and accessed data is calculated based on the matrix size $n \times n$ as: $W = 10 \times n$ and $Q = 13 \times 4 \times n$. Intensity of $SpMV$ is not dependent on matrix size and is a fixed value: $I = \frac{W}{Q} = \frac{5}{26} = 0.19$. Since the intensity value is small, power consumed by accessing data contributes significantly to the total power. The deviation error of $SpMV$ estimated power compared to measured power is from 15% to 23% which is an 8% margin shown in Figure 19. Since we want to predict the trend of power consumption, an 8% margin is acceptable for us to identify when to use the race-to-hold (RTH) condition [54].

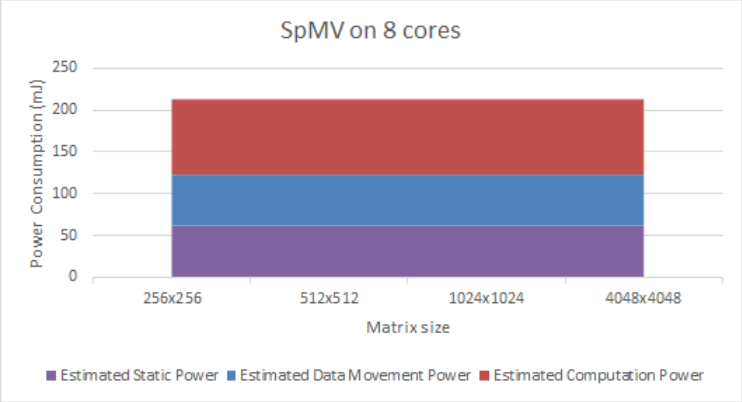


Figure 18: Power analysis of Sparse Matrix Vector Multiplication

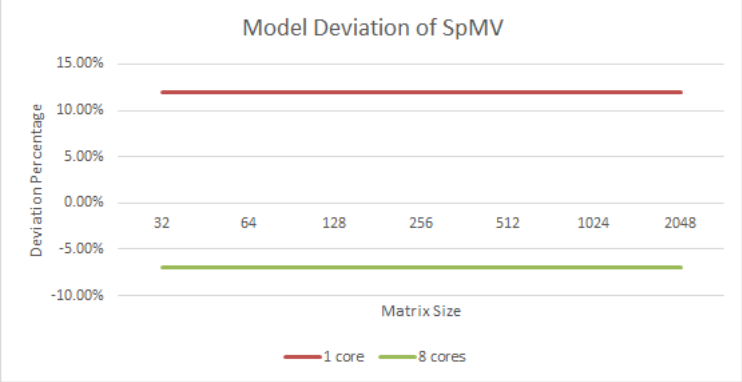


Figure 19: Deviation of estimated power from measured power of Sparse Matrix Vector Multiplication

Breadth First Search. We also implement the Graph500 kernel, namely Breadth First Search (BFS), on Myriad1. BFS is the graph kernel to explore the vertices and edges of a directed graph from a starting vertex. We use the algorithm in the current Graph500 benchmark and port it to Myriad1. The graph is stored in DDR RAM. There are two properties that define the size of a graph: degree and scale. Scale identifies the number of nodes in the graph while degree is the average number of edges from a node. In our experiments, we mostly use the default scale of 16 from the Graph500 [8], which means the graph has 2^{16} vertices. The degree is varied from 14 to 17. The operation intensity I of BFS algorithm does not depend on degree or scale. Each vertex data is stored with float type. From our implementation, the number of operations and accessed data are calculated based on the number of vertices m and the number of edges n: $W = 2 \times m + 5 \times n$ and $Q = 8 \times m + 16 \times n$. Intensity of BFS is a fixed value: $I = \frac{W}{Q} = 0.257$. As the result, predicted power consumption of BFS with varied degrees is the same over matrix size as shown in Figure 20. Power consumed by accessing data contributes significantly to the total power. Figure 21 is the deviation error of the power model for BFS compared to

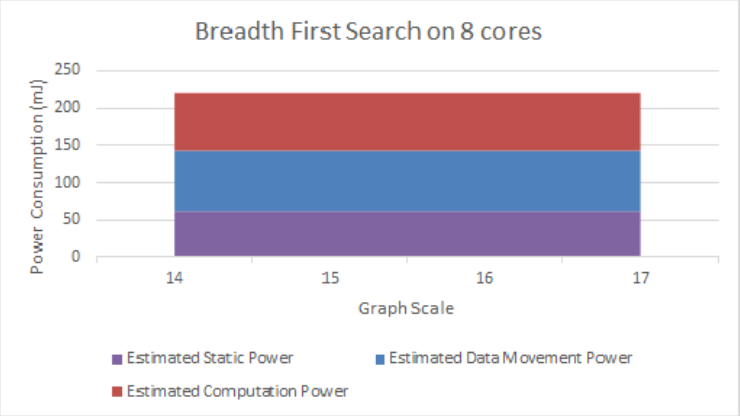


Figure 20: Power analysis of Breadth First Search

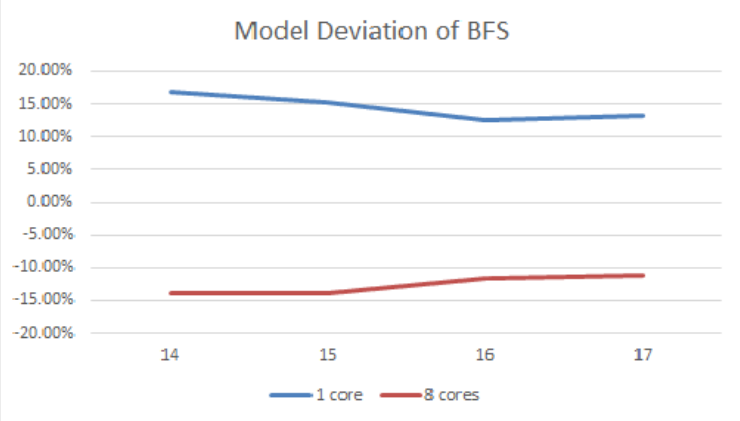


Figure 21: Deviation of estimated power from measured power of Breadth First Search

measured data, which ranges from -14% to 17%.

3.2.2.3 Remarks and future works

We have presented a power model for Myriad1 platform. In details, we have described the process to build the model and evaluate it. We have first validated the model with three sets of micro-benchmarks. We have also implemented three application kernels on Myriad1 platform, along with analyzing the amount of work W, the number of data accesses Q and the intensity I for each algorithm. Then we have applied our model to the implemented algorithms and evaluated its accuracy.

In the future, we plan to use the model with EXCESS execution framework described in Deliverable D3.1 and D3.2. In that context, the model can use the feedback data of consumed power measured by the framework to find the accurate value of α and P^{ctn} and m for a given platform. We also plan to improve the accuracy of the model by considering memory-access patterns of the implementation and instruction-pipeline parallelism.

3.3 Energy Model for Lock-Free Queues on CPU Platform

We consider the problem of modeling the energy behavior of lock-free concurrent queue data structures, more especially, of lock-free queue implementations and parallel applications that use them. Focusing on steady state behavior we decompose energy behavior into throughput and power dissipation which can be modeled separately and later recombined into several useful metrics, such as energy per operation. Based on our models, instantiated from synthetic benchmark data, and using only a small amount of additional application specific information, energy and throughput predictions can be made for parallel applications that use the respective data structure implementation.

To model throughput we propose a generic model for lock-free queue throughput behavior, based on a combination of the dequeuers' throughput and enqueuers' throughput. To model power dissipation we commonly split the contributions from the various computer components into static, activation and dynamic parts, where only the dynamic part depends on the actual instructions being executed. To instantiate the models a synthetic benchmark explores each queue implementation over the dimensions of processor frequency and number of threads.

Finally, we show how to make predictions of application throughput and power dissipation for a parallel application using a lock-free queue requiring only a limited amount of information about the application work done between queue operations. Our case study on a Mandelbrot application shows convincing prediction results.

In this Section 3.3, we present a so-called white-box model for performance and power of lock-free queue implementations. This model is white-box, in the sense that the phenomena that drive the evolution of the throughput and the power dissipation are clearly identified, and their contribution to both metrics is transparently stated. The Section 3.4 will then explain how to set the parameters of the model with only a few runs of the synthetic benchmark.

3.3.1 Motivation and Preliminaries

Lock-free implementations of data structures are a scalable approach for designing concurrent data structures. Lock-free data structures offer high concurrency and immunity to deadlocks and convoying, in contrast to their blocking counterparts. Concurrent FIFO queue data structures are fundamental data structures that are key components in applications, algorithms, run-time and operating systems. The producer/consumer pattern, *e.g.*, is a common approach to parallelizing applications where threads act as either producers or consumers and synchronize and stream data items between them using a shared collection. A concurrent queue, *a.k.a.* shared "first-in, first-out" or FIFO buffer, is a shared collection of elements which supports at least the basic operations `Enqueue` (adds an element) and `Dequeue` (removes the oldest element). `Dequeue` returns the element removed or, if the queue is empty, NULL. A large number of lock-free (and wait-free) queue implementations have appeared in the literature, *e.g.* [82, 65, 79, 66, 52, 37] being some of the most influential or most efficient results. Each implementation of a lock-free queue has obviously its strong

and weak points so the impact on performance and energy when choosing one particular implementation for any given situation may not be obvious.

As the number of known implementations of lock-free concurrent queues is growing, it is of great interest to describe a framework within which the different implementations can be ranked, according to the parameters that characterize the situation. A brute force approach could achieve this by running the implementations on hand on the whole domain of study, gathering and comparing measurements. This would yield high accuracy, but at a tremendous cost, since the domain is likely to be large. Additionally, it would only bring a limited understanding on the phenomena that drive the behavior of the queue implementations. Therefore, we propose generic models for predicting the behavior of lock-free queues under steady state usage. The models are instantiated for the queue implementations and machine on hand using empirical data from a limited number of points in the domain.

The implementations can be ranked according to a plethora of metrics. Traditionally, performance in terms of throughput has been the main metric. Furthermore, the notion of energy efficiency has now extended into every nook and cranny of Information Technology, at any scale, from the Exascale machines that need huge improvements in terms of power dissipation to be feasible [31], to the small electronic devices where the battery lifetime is a critical issue.

We decompose the energy behavior of queues, and subsequently applications, into two components: (i) throughput and (ii) power dissipation. We model these components separately. The predicted throughput and power dissipation can be recombined into the energy-efficiency metric energy per queue operation, which is the ratio between power dissipation and queue throughput. When modeling an application, this metric can be extended to energy per unit of application work. Further, plotting energy per operation or unit of work according to throughput allows exploration of the Pareto-optimal frontier of the energy−performance bi-criteria optimization problem for the queues or the application.

Lock-free queue data structures generally offer disjoint-access parallelism: enqueuers and dequeuers modify only their respective ends of the queue, and compete mostly with operations of the same kind. Nonetheless, when the queue is close to empty, both ends point to the same part of the queue, then enqueue and dequeue operations have to be synchronized, and every operation impacts the behavior of any other.

Concerning the queue as a whole, a successful event can be seen as the dequeue of a non-Null item, since this event implies that the item has been enqueued and dequeued. Also, the throughput of the queue is naturally defined as the number of such events per unit of time, which is a meaningful performance criterion for queues.

In this work, we focus on queues that are in a steady state, *i.e.* such that the rate of each operation attempt is constant. Then, the throughput $\mathcal{T}$ of the queue is the minimum between the throughput of all dequeues $\mathcal{T}_d$, even those returning Null, and throughput of enqueues $\mathcal{T}_e$. Indeed, if $\mathcal{T}_e > \mathcal{T}_d$, then the queue grows and the throughput is determined by the dequeuers, which cannot obtain any Null items; and if $\mathcal{T}_e \leq \mathcal{T}_d$, then the queue is mostly empty and Null items are dequeued, but the throughput is determined by the enqueuers.

Despite this decomposition, enqueuers' and dequeuers' throughput are still correlated when the queue is mostly empty. In addition, the interactions between them are rather asymmetric, as in broad terms, an enqueue can be delayed by any concurrent dequeue, while for a dequeue, concurrent enqueues will cease to disturb it as they move away from the dequeue end.

Based on these facts, we decorrelate the throughput into several uncorrelated and basic throughputs, and reconstitute the main throughput by combining them. Among the advantages of this process, we earn a better understanding of the performance (as the basic throughputs are meaningful), and we reduce the number of measurements needed to instantiate the model on the whole domain of study.

The domain of study that we envision here can be viewed as the Cartesian product of four sets: (i) number of threads accessing the queue, (ii) CPU frequencies, (iii) a range of dequeue access rates, (iv) a range of enqueue access rates. The cardinality of the first two sets is at most a few tens, while the last two are continuous sets that are not even bounded. Thanks to the removal of the dependencies between throughputs, we are able to instantiate the model with only a few data points, while the model covers the whole intervals.

Finally, this decomposition also eases the study of power dissipation, where we reuse the same ideas as in the throughput estimation part.

3.3.2 Framework

3.3.2.1 Synthetic Benchmark

```
procedure ENQUEUER
    while !done do
        Parallel_Work()
        Enqueue()
```

```
procedure DEQUEUER
    while !done do
        res ← Dequeue()
        if res ≠ NULL then
            Parallel_Work()
```

Figure 22: Queue benchmark

Skeleton We run the synthetic benchmark composed of the two functions described in Figure 22, starting with an empty queue. Half of the threads are assigned to be enqueuers while the remaining ones are dequeuers. We disable logical cores (hyper-threading) and map different threads into different cores, also the number of threads never exceeds the number of cores. In addition, the mapping is done in the following way: when adding an enqueuer/dequeuer pair, they are both mapped on the most filled but non-full socket.

The parallel sections (`Parallel_Work`) shall be seen as a processing activity, pre-processing for the enqueuers before they enqueue an item, and post-processing on an item from the queue for the dequeuers. We assume that memory accesses in the parallel sections are negligible, and represent the parallel sections as sequences of bunches of *pause* instructions in the benchmark; we note pw_e (resp. pw_d) the number of bunches of 90 *pause*s (which corresponds to 1000 cycles) that compose the parallel work in the enqueuer (resp. dequeuer).

From a high-level perspective, `Enqueue` and `Dequeue` operations follow a retry loop pattern: a thread reads an access point to the data structure, works locally with this view of the data structure, possibly performs memory management actions and 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* primitive. If it succeeds, *i.e.* if the access point has not been changed by another thread between the first read and the *Compare-and-Swap*, then it goes to the next parallel section, otherwise it repeats the process.

Queue Implementations We study some of the most well-known and studied lock-free and linearizable queues in the literature, as implemented in NOBLE [77].These queue algorithms are described in some detail in Section 4.2.1. The aim of this work is still to predict the behavior of any lock-free queue algorithm and not only the ones mentioned above. These algorithms are used to validate the model that we present in the following sections.

When we speak about implementations of the queues, we actually refer to the different implementations of enqueuing and dequeuing operations, along with their memory management schemes.

3.3.2.2 General Power Model

The power is split into three elements: the *static* part is the cost of turning the machine on, the *activation* part incorporates a fixed cost for each socket and each core in use, and the *dynamic* part is a supplementary cost that depends on the running application.

In accordance with the RAPL energy counters [26, 17, 85], we further decompose each part per-component, for memory, CPU, and *uncore* (denoted by a superscript M, C and U, respectively):

$$P = \sum_{X \in \{M,C,U\}} \left(P^{(stat,X)} + P^{(active,X)} + P^{(dyn,X)} \right).$$

We assume that we already know the platform characteristics, *i.e.* all static and active powers (they can be obtained as explained for instance in D1.2 [55]), and we try to find the application-specific dynamic powers. In order to keep the formulas readable, in the following, we denote by $P^{(X)}$ the dynamic power $P^{(dyn,X)}$.

3.3.2.3 Notations and Setting

We denote by n the number of running threads that call the same operation, and by f the clock frequency of the cores (we only consider the case where all cores share the same clock frequency).

We recall that pw_e (resp. pw_d) is the amount of work in the parallel section of an enqueuer (resp. dequeuer), as the number of bunches of 90 *pause*s. For a given queue implementation, we denote by cw_e (resp. cw_d) the amount of work in one try of the retry loop of the `Enqueue` (resp. `Dequeue`) operation. Associated with these amounts of work, we define, for $o \in \{d, e\}$,

the average execution time of the parallel section (resp. the retry loop and a single try of the retry loop) related to operation o as $\mathrm{t}(PS_o)$ (resp. $\mathrm{t}(RL_o)$ and $\mathrm{t}(SL_o)$).

In the same way, for $o \in \{d, e\}$, we denote by $P_o^{(C)}$ (resp. $P_{o,PS}^{(C)}$ and $P_{o,RL}^{(C)}$) the dynamic CPU power dissipated by component X in (resp. the parallel section related to and the retry loop related to) operation o.

Finally, for $o \in \{d, e\}$, we denote by r_o the ratio of the time that a thread spends in the retry loop, while it is associated with operation o.

In Sections 3.3.3 and 3.3.4, in order to keep expressions as simple as possible, we define one unit of time as λ sec, where λ is the execution time of $90 \times f$ *pause*s (as the *pause* instructions are perfectly scalable with clock frequency, λ is constant). Throughput is expressed in number of operations per unit of time, *i.e.* per λ secs. Finally, we derive the power in Watts.

All experiments and their underlying predictions are done on *S*ystem A (see Section 2.1), *i.e.* a platform composed of a dual-socket Intel® Xeon® processor, with eight cores per socket. The sizes of L3, L2 and L1 caches are 25 MB, 256 kB and 32 kB, respectively.

We run the implementations at the two extreme frequencies (excluding Turbo mode) 1.2 GHz and 3.4 GHz, for all possible even total numbers of threads, from 2 to 16, *i.e.* for $n \in \{1, \ldots, 8\}$.

3.3.3 Throughput Estimation

3.3.3.1 Throughput Decomposition Principles

We recall that the throughput of the queue is defined as:

$$\mathcal{T} = \min\left(\mathcal{T}_e, \mathcal{T}_d\right),$$

where $\mathcal{T}_e$ and $\mathcal{T}_d$ are the enqueuers' and dequeuers' throughput, respectively.

As we are in steady state, one operation o is performed every $\mathrm{t}(PS_o) + \mathrm{t}(RL_o)$ unit of time by each thread, and n threads attempt to concurrently execute o, hence the general expression of the throughput $\mathcal{T}_o$:

$$\mathcal{T}_o = \frac{n}{\mathrm{t}(PS_o) + \mathrm{t}(RL_o)}.$$

We have seen that the parallel sections of the benchmark are full of *pause*s, thus the time $\mathrm{t}(PS_o)$ spent in a given parallel section is straightforwardly given by $\mathrm{t}(PS_o) = pw_o/f$. The execution time of dequeue and enqueue operations is more problematic, for two main reasons. *Primo*, because of the lock-free nature of the implementations. As the number of retries is unknown, the time spent in the function call is not trivially computable. *Secundo*, when the activity on the queue is high, the threads compete for accessing a shared data, and they stall before actually being able to access the data. We name this as the *expansion*, as it leads to an increase in the execution time of a single try of the retry loop.

The contention on the queue is twofold. At any time, and even if it could be negligible, threads that perform the same operation disturb each other, since they try to access the same shared data. In addition, when the queue is mostly empty, enqueuers and dequeuers

try to access the same data, then interference occurs; enqueuers make dequeuers stall and *vice versa*. We call the former case *intra-contention*, and the latter one *inter-contention*.

As expected, we have noticed a marked difference between the execution time of a dequeue operation returning NULL and one that returns a queue item, *i.e.* whether the queue was empty or contained at least one item. That is why we decompose $\mathcal{T}_d$ into throughput of dequeue on empty queue $\mathcal{T}_d^{(+)}$ (that returns a NULL item), and dequeue on non-empty queue $\mathcal{T}_d^{(-)}$ (that does not return NULL).

Further, the impact of inter-contention on dequeue operations is negligible compared to the impact of the queue being empty; therefore we ignore inter-contention for dequeues.

In contrast, the queue being empty does not notably change the execution time of the enqueue operation, while dequeue operations can impact the behavior of concurrent enqueue operations greatly when the queue is close to empty. Hence, we split $\mathcal{T}_e$ into the enqueue throughput $\mathcal{T}_e^{(+)}$ when the queue is not inter-contended, and the enqueue throughput $\mathcal{T}_e^{(-)}$ when the queue experiences the maximum possible inter-contention.

These basic throughputs fulfill the two following inequalities: $\mathcal{T}_d^{(+)} \geq \mathcal{T}_d^{(-)}$ and $\mathcal{T}_e^{(+)} \geq \mathcal{T}_e^{(-)}$.

Thanks to this separation into the four basic throughput cases $\mathcal{T}_d^{(+)}$, $\mathcal{T}_d^{(-)}$, $\mathcal{T}_e^{(+)}$ and $\mathcal{T}_e^{(-)}$, we earn a better understanding of the factors that influence the general throughput, and we deinterlace their dependencies, which dramatically decreases the number of points in the parallel section sizes set where we need to take measurements for our modeling. More precisely, by construction, $\mathcal{T}_d^{(+)}$ and $\mathcal{T}_d^{(-)}$ do not indeed depend on pw_e, while $\mathcal{T}_e^{(+)}$ and $\mathcal{T}_e^{(-)}$ do not depend on pw_d. Nonetheless $\mathcal{T}_d$ (resp. $\mathcal{T}_e$) is defined as a barycenter between $\mathcal{T}_d^{(+)}$ and $\mathcal{T}_d^{(-)}$ (resp. $\mathcal{T}_e^{(-)}$ and $\mathcal{T}_e^{(+)}$), whose weights depend on both pw_d and pw_e.

In Section 3.3.3.2, we describe the basic throughputs, we combine them in Section 3.3.3.3, then we explain how to instantiate the parameters of the model in Section 3.4.1, and finally exhibit results in Section 4.2.2.1.

3.3.3.2 Basic Throughputs

We aim in this section at estimating the throughput $\mathcal{T}_o^{(b)}$ of one of the basic operations described in the previous subsection, where $o \in \{e, d\}$ and $b \in \{+, -\}$. We assume that $\mathcal{T}_o^{(b)}$ depends only on pw_o, in addition to the tacit dependencies on the clock frequency, number of threads and queue implementation. We denote by $cw_o^{(b)}$ the amount of work in a single try of the retry loop related to operation o in case b when the queue is not intra-contended.

Low Intra-Contention We study in this section the low intra-contention case, *i.e.* when (i) the threads do not suffer from expansion due to threads that perform the same operation, and (ii) a success is obtained with a single try of the retry loop. As it appears in Figure 23, we have a cyclic execution, and the length of the shortest cycle is $\mathrm{t}\left(PS_o\right) + \mathrm{t}\left(SL_o^{(b)}\right)$. Within each cycle, every thread performs exactly one successful operation, thus

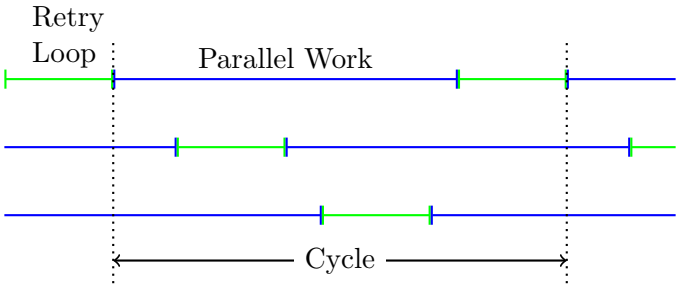


Figure 23: Cyclic execution under low intra-contention

the throughput is straightforward:

$$\mathcal{T}_o^{(b)} = \frac{n}{\mathrm{t}\left(PS_{\,o}\right) + \mathrm{t}\left(SL_o^{(b)}\right)} = \frac{nf}{pw_{\,o} + cw_o^{(b)}}. \tag{19}$$

High Intra-Contention As explained in Section 3.3.3.1, in this case, the direct evaluation of the execution time of a retry loop is more complex, but we have experimentally observed that the throughput is approximately linear with the expected number of threads that are in the retry loop at a given time. In addition, this expected number is almost proportional to the amount of work in the parallel section. As a result, a good approximation of the throughput, in high intra-contention cases, is a function that is linear with the amount of work in the $pw_{\,o}$.

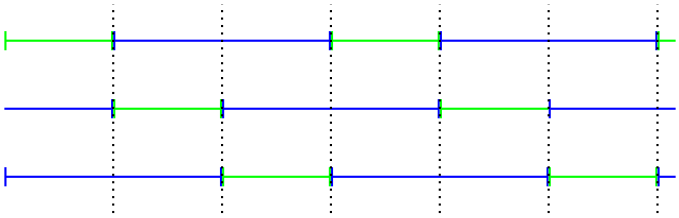


Figure 24: Intra-contention frontier

Frontier We now have to estimate whether the queue is highly intra-contended. We recall that, generally speaking, a long parallel section leads to a low intra-contended queue since threads are most of the time processing some computations and are not trying to access the shared data. Reversely, when the parallel section is short, the ratio of time that threads spend in the retry loop is higher, and gets even higher because of both expansion and retries.

That being said, there exists a simple lower bound of the amount of work in the parallel section, such that there exists an execution where the threads are never failing in their retry

loop. We plot in Figure 24 an ideal execution with $n = 3$ threads and $\mathrm{t}\left(PS_o\right) = (n-1) \times \mathrm{t}\left(SL_o^{(b)}\right)$. In this execution, all threads always succeed at their first try in the retry loop. Nevertheless, if we shorten the parallel section, then there is not enough parallel potential any more, and the threads will start to fail: the queue leaves the low intra-contention state.

In practice, this lower bound ($\mathrm{t}\left(PS_o\right) = (n-1) \times \mathrm{t}\left(SL_o^{(b)}\right)$) is actually a good approximation for the critical point where the queue switches its state.

3.3.3.3 Combining Basic Throughputs

We are given parallel sections sizes, and show how to link the throughput of the four basic operations, with the dequeuers' and enqueuers' throughput. There are two possible states for the queue: either it is mostly empty (*i.e.* some NULL items are dequeued), or it gets larger and larger.

In the first case, some of the dequeues will occur on an empty queue. In 1 unit of time, $\mathcal{T}_e$ items are enqueued. These items are dequeued in $\mathcal{T}_e/\mathcal{T}_d^{(-)}$ units of time (the queue is non-empty while they are dequeued), which leads to a slack of $1 - \mathcal{T}_e/\mathcal{T}_d^{(-)}$, where dequeues of NULL items can take place at a rate $\mathcal{T}_d^{(+)}$, hence the following throughput formula:

$$\mathcal{T}_d = \frac{\mathcal{T}_e}{\mathcal{T}_d^{(-)}} \times \mathcal{T}_d^{(-)} + \left(1 - \frac{\mathcal{T}_e}{\mathcal{T}_d^{(-)}}\right) \times \mathcal{T}_d^{(+)}. \tag{20}$$

Concerning the enqueuers, we use the same assumption on inter-contention as used on intra-contention in Section 3.3.3.2, saying that the throughput is linear with the expected number of threads inside the retry loop. Here, the expected number of threads inside the dequeue operation is proportional to the ratio r_d of the time spent by one dequeuer in its dequeue operation. We do not know $\mathrm{t}\left(RL_d\right)$, but we know that in average, to complete a successful operation, a thread needs $\mathrm{t}\left(PS_d\right) + \mathrm{t}\left(RL_d\right)$ units of time, and among this time it will spend $\mathrm{t}\left(PS_d\right)$ in the parallel section. Therefore

$$r_d = 1 - \mathrm{t}\left(PS_d\right) / (\mathrm{t}\left(PS_d\right) + \mathrm{t}\left(RL_d\right)) = 1 - \frac{\mathcal{T}_d \times pw_d}{n \times f}.$$

The minimum intra-contention is reached when this ratio is 0, while the maximum is obtained when it is 1, thus:

$$\mathcal{T}_e = \frac{\mathcal{T}_d \times pw_d}{n \times f} \times \mathcal{T}_e^{(+)} + \left(1 - \frac{\mathcal{T}_d \times pw_d}{n \times f}\right) \times \mathcal{T}_e^{(-)}. \tag{21}$$

In the second case, enqueuers and dequeuers do not access to the same part of the queue, thus inter-contention does not take place, then $\mathcal{T}_e = \mathcal{T}_e^{(+)}$, and all dequeues return a non-NULL item, hence $\mathcal{T}_d = \mathcal{T}_d^{(-)}$.

The discrimination of these two cases is trivial when enqueuers' and dequeuers' throughput are given: the queue is in the first state (mostly empty) if and only if $\mathcal{T}_e \leq \mathcal{T}_d$.

Reversely, if we know the four basic throughputs $\left(\mathcal{T}_e^{(+)}, \mathcal{T}_e^{(-)}, \mathcal{T}_d^{(+)}, \mathcal{T}_d^{(-)}\right)$, and aim at reconstituting the dequeuers' and enqueuers' throughput $(\mathcal{T}_d, \mathcal{T}_e)$, several solutions could be consistent, *i.e.* either a growing queue, such that $\mathcal{T}_e = \mathcal{T}_e^{(+)}$ and $\mathcal{T}_d = \mathcal{T}_d^{(-)}$ and fulfilling the inequality $\mathcal{T}_e > \mathcal{T}_d$, or a mostly empty queue, fulfilling Equations 20 and 21.

Theorem 3.2. *Given $\left(\mathcal{T}_e^{(+)}, \mathcal{T}_e^{(-)}, \mathcal{T}_d^{(+)}, \mathcal{T}_d^{(-)}\right)$, there exists a consistent solution $(\mathcal{T}_d, \mathcal{T}_e)$ with a growing queue if and only if $\mathcal{T}_e^{(+)} > \mathcal{T}_d^{(-)}$. In addition, this solution is unique and is such that $\mathcal{T}_e = \mathcal{T}_e^{(+)}$ and $\mathcal{T}_d = \mathcal{T}_d^{(-)}$.*

Proof. ($\Rightarrow$) If the queue is growing, then $\mathcal{T}_e > \mathcal{T}_d$. Moreover, dequeues never occur on an empty queue, hence $\mathcal{T}_d = \mathcal{T}_d^{(-)}$, and there is no inter-contention, thus $\mathcal{T}_e = \mathcal{T}_e^{(+)}$.
($\Leftarrow$) Let us assume now that $\mathcal{T}_e^{(+)} > \mathcal{T}_d^{(-)}$. $\mathcal{T}_e = \mathcal{T}_e^{(+)}$ and $\mathcal{T}_d = \mathcal{T}_d^{(-)}$ is a valid solution, such that the queue is growing, since then $\mathcal{T}_e > \mathcal{T}_d$.

By construction, $\mathcal{T}_e \leq \mathcal{T}_e^{(+)}$; if we had another solution such that the queue grows and $\mathcal{T}_e < \mathcal{T}_e^{(+)}$, it would mean that enqueues are inter-contended, which is possible only when the queue is mostly empty. This is absurd, hence the uniqueness. □

Theorem 3.3. *Given $\left(\mathcal{T}_e^{(+)}, \mathcal{T}_e^{(-)}, \mathcal{T}_d^{(+)}, \mathcal{T}_d^{(-)}\right)$, there exists a consistent solution $(\mathcal{T}_d, \mathcal{T}_e)$ with a mostly empty queue if and only if*

$$\frac{\mathcal{T}_e^{(-)}}{\mathcal{T}_d^{(-)}} \leq 1 - \frac{pw_d}{n \times f}\left(\mathcal{T}_e^{(+)} - \mathcal{T}_e^{(-)}\right). \tag{22}$$

In addition, this solution is unique and is given by Equations 21 and 20.

Proof. ($\Rightarrow$) Let a solution $(\mathcal{T}_d, \mathcal{T}_e)$ with a mostly empty queue. By construction, the throughputs follow Equations 21 and 20. As $\mathcal{T}_e$ is an increasing function according to $\mathcal{T}_d$ (because $\mathcal{T}_e^{(+)} \geq \mathcal{T}_e^{(-)}$), we derive

$$\mathcal{T}_e \geq \frac{\mathcal{T}_d^{(-)} \times pw_d}{n \times f} \times \mathcal{T}_e^{(+)} + \left(1 - \frac{\mathcal{T}_d^{(-)} \times pw_d}{n \times f}\right) \times \mathcal{T}_e^{(-)}.$$

The queue is mostly empty, thus the dequeues of non-NULL items have to be faster than the enqueues, which translates into $\mathcal{T}_d^{(-)} \geq \mathcal{T}_e$. The two inequalities combined show the implication.
($\Leftarrow$) Let us assume now that Inequality 22 is fulfilled. Equation 20 can be rewritten into

$$\mathcal{T}_e = \frac{\mathcal{T}_d - \mathcal{T}_d^{(+)}}{1 - \frac{\mathcal{T}_d^{(+)}}{\mathcal{T}_d^{(-)}}}.$$

Let us consider now $\mathcal{T}_e{}'$ and $\mathcal{T}_e{}''$ two functions of $\mathcal{T}_d{}'$ that fulfill the following system of equations:

$$\begin{cases} \mathcal{T}_e{}'\left(\mathcal{T}_d{}'\right) = \frac{\mathcal{T}_d{}' - \mathcal{T}_d^{(+)}}{1 - \frac{\mathcal{T}_d^{(+)}}{\mathcal{T}_d^{(-)}}} \\ \mathcal{T}_e{}''\left(\mathcal{T}_d{}'\right) = \frac{\mathcal{T}_d{}' \times pw_d}{n \times f} \times \mathcal{T}_e^{(+)} + \left(1 - \frac{\mathcal{T}_d{}' \times pw_d}{n \times f}\right) \times \mathcal{T}_e^{(-)}. \end{cases}$$

We have $\mathcal{T}_e{}'\left(\mathcal{T}_d^{(+)}\right) = 0$ and $\mathcal{T}_e{}'\left(\mathcal{T}_d^{(-)}\right) = \mathcal{T}_d^{(-)}$. According to Inequality 22, we know also that $\mathcal{T}_e{}''\left(\mathcal{T}_d^{(+)}\right) \leq \mathcal{T}_d^{(+)}$. In addition, $\mathcal{T}_e{}''$ is a linearly increasing function of $\mathcal{T}_d{}'$ and $\mathcal{T}_e{}'$ a linearly decreasing function of $\mathcal{T}_d{}'$. This shows that there exists a unique $\mathcal{T}_d$ such that $\mathcal{T}_e{}'\left(\mathcal{T}_d\right) = \mathcal{T}_e{}''\left(\mathcal{T}_d\right)$, and if we define $\mathcal{T}_e$ as $\mathcal{T}_e = \mathcal{T}_e{}'\left(\mathcal{T}_d\right) = \mathcal{T}_e{}''\left(\mathcal{T}_d\right)$, the pair $(\mathcal{T}_d, \mathcal{T}_e)$ is such that

$$\begin{cases} \mathcal{T}_d^{(-)} \leq \mathcal{T}_d \leq \mathcal{T}_d^{(+)} \\ \mathcal{T}_e^{(-)} \leq \mathcal{T}_e \leq \mathcal{T}_e^{(+)} \\ \mathcal{T}_e \leq \mathcal{T}_d \end{cases}.$$

This implies that $(\mathcal{T}_d, \mathcal{T}_e)$ is a solution with an empty queue, and we have shown that this solution is unique. □

Corollary 3.3.1. *Given* $\left(\mathcal{T}_e^{(+)}, \mathcal{T}_e^{(-)}, \mathcal{T}_d^{(+)}, \mathcal{T}_d^{(-)}\right)$*, there exists at least one solution* $(\mathcal{T}_d, \mathcal{T}_e)$*.*

Proof. We show that if the inequality of Theorem 3.2 is not fulfilled, *i.e.* if $\mathcal{T}_e^{(+)} \leq \mathcal{T}_d^{(-)}$, then the inequality of Theorem 3.3 is true. We have indeed

$$\begin{aligned} \left(1 - \frac{pw_d}{n \times f}\left(\mathcal{T}_e^{(+)} - \mathcal{T}_e^{(-)}\right)\right)\mathcal{T}_d^{(-)} - \mathcal{T}_e^{(-)} &= \left(1 - \frac{pw_d \times \mathcal{T}_e^{(+)}}{n \times f}\right)\mathcal{T}_d^{(-)} - \left(1 - \frac{pw_d \times \mathcal{T}_d^{(-)}}{n \times f}\right)\mathcal{T}_e^{(-)} \\ &\geq \left(1 - \frac{pw_d \times \mathcal{T}_e^{(+)}}{n \times f}\right)\mathcal{T}_d^{(-)} - \left(1 - \frac{pw_d \times \mathcal{T}_d^{(-)}}{n \times f}\right)\mathcal{T}_e^{(+)} \\ &\geq \mathcal{T}_d^{(-)} - \mathcal{T}_e^{(+)} \\ \left(1 - \frac{pw_d}{n \times f}\left(\mathcal{T}_e^{(+)} - \mathcal{T}_e^{(-)}\right)\right)\mathcal{T}_d^{(-)} - \mathcal{T}_e^{(-)} &\geq 0, \end{aligned}$$

which proves the Corollary. □

One can notice that if $\mathcal{T}_e^{(+)} > \mathcal{T}_d^{(-)}$ and Inequality 22 are fulfilled and the queue could be either mostly empty or growing. In this case, we choose, for each operation, the mean of the two solutions, in order to minimize the discontinuities.

3.3.4 Power Estimation

We recall that we are interested only in the dynamic powers as we assume that static and activation powers are known.

3.3.4.1 CPU Power

Firstly, as we map each thread on a dedicated core, there is no interference between the CPU power of different cores, so we can compute the dynamic power as

$$P^{(C)} = n \times P_e^{(C)} + n \times P_d^{(C)}. \tag{23}$$

Secondly, we assume that we can segment time and consider that, given a thread performing operation $o \in \{e, d\}$, the power dissipated in the retry loop and the power dissipated in the parallel section are independent. There only remains to weight the previous powers by the time spent in each of these regions:

$$P_o^{(C)} = r_o \times P_{o,RL}^{(C)} + (1 - r_o) \times P_{o,PS}^{(C)}. \tag{24}$$

As shown in Section 3.3.3.3, the ratio can be obtained through

$$r_o = 1 - \frac{\mathcal{T}_o \times pw_o}{n \times f}. \tag{25}$$

Altogether, we obtain the final formula for dynamic CPU power

$$P^{(C)} = n \left(\sum_{o \in \{e,d\}} P_{o,RL}^{(C)} + \frac{\mathcal{T}_o \times pw_o \times \left(P_{o,PS}^{(C)} - P_{o,RL}^{(C)} \right)}{n \times f} \right) \tag{26}$$

3.3.4.2 Memory and Uncore Power

We have noticed in [55] that the dynamic memory power is proportional to the intensity (number of units of memory accessed per unit of time) of main memory accesses and remote accesses, when the threads read separate places of the memory.

Here, the data structure does not directly involve the main memory since we keep its size reasonably bounded (if the queue reaches the maximum size, we suspend the measurements, empty the queue, and resume), hence the power dissipation in memory is only due to remote accesses, which only appears as the threads are spread across sockets (*i.e.* when $n > 4$).

Moreover, as the parallel sections are full of *pause*s, communications can only take place in the retry loop, and there is no dynamic memory power dissipated in the parallel sections. Concerning the retry loops, we make the following assumption: the amount of data accessed per second in a retry loop depends on the implementation, but given an implementation, once a thread is in the retry loop, it will always try to access the same amount of data per second. When the queue is highly intra-contended, if a thread fails then it will retry and will access the data in the same way as in the previous try; and if there is expansion, then the thread will still try to access the data for the whole time it is in the retry loop.

In addition, the dequeuers (and the same line of reasoning holds for the enqueuers) tries here to access the same data. Therefore either memory requests are batched together when sent outside the socket, or the Home Agent (cache coherency mechanism/memory controller)

keeps track of the previous requests. This implies that the number of threads attempting to access the data does not impact the dynamic memory power greatly when the rate of requests is high.

All things considered, as a thread working on operation o spends a fraction r_o of its time inside its retry loop, we obtain that the dynamic memory power dissipated in the retry loop is proportional to r_o (times the amount of data accessed per unit of time in the retry loop, which is a constant). Hence

$$P^{(M)} = r_e \times \rho_e^{(M)} + r_d \times \rho_d^{(M)}, \tag{27}$$

where $\rho_e^{(M)}$ and $\rho_d^{(M)}$ are constants.

The dynamic uncore power is computed exactly in the same way as the dynamic memory power.

3.4 White-box Methodology for Instantiating the Energy Model of Queues on CPU Platform

In this section, we show how to obtain the parameters of the model, so that predictions can further be made. Those parameters depend on the architecture where the application is running, thus we need some measurements on a few runs of the synthetic benchmark to discover them. This methodology is white-box, since we know which runs to use, so that the calibration of the model is achieved with a minimum possible set of runs.

3.4.1 Instantiating the Throughput Model

We recall that, for all $o \in \{e, d\}$ and $b \in \{+, -\}$, $\mathcal{T}_o^{(b)}$ depends only on pw_o, while $\mathcal{T}_e$ and $\mathcal{T}_d$ depend on both pw_d and pw_e. We denote now by $\mathcal{T}_d(pw_d, pw_e)$ (resp. $\mathcal{T}_e(pw_d, pw_e)$) the dequeuers' (resp. enqueuers') throughput as the amount of work in the parallel section of the dequeuers is pw_d and enqueuers' one is pw_e. The estimate of a value is denoted by a hat on top, while the measured value does not wear the hat.

Let $p_{\text{sma}} = 1$, $p_{\text{mid}} = 20$ and $p_{\text{big}} = 1000$ be three distinctive amounts of work, that corresponds to different states of the execution. If $pw_o = p_{\text{big}}$, we can neglect the impact of operation o on the queue, $pw_o = p_{\text{mid}}$ is a low intra-contention case since the non-expanded critical sections are experimentally less than 2 units of time, and $pw_o = p_{\text{sma}}$ corresponds to a highly inter- or intra-contention case. We note the we cannot use a 0 size as amount of work since it leads to undesirable results due to the back-to-back effect (a thread does not allow other threads to access the queue for several consecutive iterations).

3.4.1.1 Low Intra-Contention

The basic throughputs that are not intra-contended can be spawned from $cw_o^{(b)}$ (critical section size of operation o in case b), where $o \in \{e, d\}$ and $b \in \{+, -\}$, which we try to estimate here. We pick four points where the basic throughputs are easy to approximate.

We have $\mathcal{T}_d(p_{\text{mid}}, p_{\text{sma}}) < \mathcal{T}_e(p_{\text{mid}}, p_{\text{sma}})$, as the amounts of work in the retry loops are in practice less than 10. For the same reason, at this point, we are in low intra-contention from the dequeuers' point of view. Altogether,

$$\mathcal{T}_d(p_{\text{mid}}, p_{\text{sma}}) = \mathcal{T}_d^{(-)}(p_{\text{mid}}) = \frac{n \times f}{p_{\text{mid}} + cw_d^{(-)}}, \text{ hence}$$

$$\widehat{cw_d^{(-)}} = \frac{n \times f}{\mathcal{T}_d(p_{\text{mid}}, p_{\text{sma}})} - p_{\text{mid}}.$$

Then, according to Equation 20, we have

$$\frac{nf}{p_{\text{mid}} + \widehat{cw_d^{(+)}}} = \mathcal{T}_d^{(+)}(p_{\text{mid}})$$

$$\frac{nf}{p_{\text{mid}} + \widehat{cw_d^{(+)}}} = \frac{\mathcal{T}_d(p_{\text{mid}}, p_{\text{big}}) - \mathcal{T}_e(p_{\text{mid}}, p_{\text{big}})}{1 - \frac{\left(p_{\text{mid}} + \widehat{cw_d^{(-)}}\right) \times \mathcal{T}_e(p_{\text{mid}}, p_{\text{big}})}{n \times f}},$$

from which we can extract $\widehat{cw_d^{(+)}}$ since we know already $\widehat{cw_d^{(-)}}$.

In the same way, we can compute $\widehat{cw_e^{(+)}}$ then $\widehat{cw_e^{(-)}}$, by using $(p_{\text{big}}, p_{\text{mid}})$ and $(p_{\text{sma}}, p_{\text{mid}})$.

3.4.1.2 High Intra-Contention

We aim here at estimating $\mathcal{T}_o^{(b)}$ on a high intra-contention point. $p_{\text{sma}} = 1$ and $p_{\text{mid}} = 20$ are such that $\mathcal{T}_d(p_{\text{sma}}, p_{\text{mid}}) \geq \mathcal{T}_e(p_{\text{sma}}, p_{\text{mid}})$. According to Equation 20, we have

$$\mathcal{T}_d(p_{\text{sma}}, p_{\text{mid}}) = \mathcal{T}_e(p_{\text{sma}}, p_{\text{mid}}) + \left(1 - \frac{\mathcal{T}_e(p_{\text{sma}}, p_{\text{mid}})}{\widehat{\mathcal{T}_d^{(-)}}(p_{\text{sma}})}\right) \times \widehat{\mathcal{T}_d^{(+)}}(p_{\text{sma}}).$$

In addition, if $\mathcal{T}_d(p_{\text{sma}}, p_{\text{sma}}) \geq \mathcal{T}_e(p_{\text{sma}}, p_{\text{sma}})$, then

$$\mathcal{T}_d(p_{\text{sma}}, p_{\text{sma}}) = \mathcal{T}_e(p_{\text{sma}}, p_{\text{sma}}) + \left(1 - \frac{\mathcal{T}_e(p_{\text{sma}}, p_{\text{sma}})}{\widehat{\mathcal{T}_d^{(-)}}(p_{\text{sma}})}\right) \times \widehat{\mathcal{T}_d^{(+)}}(p_{\text{sma}}),$$

otherwise, $\mathcal{T}_d(p_{\text{sma}}, p_{\text{sma}}) = \widehat{\mathcal{T}_d^{(-)}}(p_{\text{sma}})$. In both cases, we can find the two unknowns $\widehat{\mathcal{T}_d^{(-)}}(p_{\text{sma}})$ and $\widehat{\mathcal{T}_d^{(+)}}(p_{\text{sma}})$ thanks to the two equations.

This last point is also used in the same way for enqueuers: if $\mathcal{T}_d(p_{\text{sma}}, p_{\text{sma}}) \geq \mathcal{T}_e(p_{\text{sma}}, p_{\text{sma}})$, then

$$\mathcal{T}_e(p_{\text{sma}}, p_{\text{sma}}) = \frac{\mathcal{T}_d(p_{\text{sma}}, p_{\text{sma}}) \times p_{\text{sma}}}{n \times f} \times \widehat{\mathcal{T}_e^{(+)}}(p_{\text{sma}}) + \left(1 - \frac{\mathcal{T}_d(p_{\text{sma}}, p_{\text{sma}}) \times p_{\text{sma}}}{n \times f}\right) \times \widehat{\mathcal{T}_e^{(-)}}(p_{\text{sma}}),$$

otherwise, $\mathcal{T}_e(p_{\text{sma}}, p_{\text{sma}}) = \widehat{\mathcal{T}_e^{(+)}}(p_{\text{sma}})$.

Like previously, we have $\mathcal{T}_d(p_{\text{mid}}, p_{\text{sma}}) < \mathcal{T}_e(p_{\text{mid}}, p_{\text{sma}})$, hence $\widehat{\mathcal{T}_e^{(+)}}(p_{\text{sma}}) = \mathcal{T}_e(p_{\text{mid}}, p_{\text{sma}})$. This implies that in any cases we can compute $\widehat{\mathcal{T}_e^{(+)}}(p_{\text{sma}})$, but we do not have access to $\widehat{\mathcal{T}_e^{(-)}}(p_{\text{sma}})$ if $\mathcal{T}_d(p_{\text{sma}}, p_{\text{sma}}) < \mathcal{T}_e(p_{\text{sma}}, p_{\text{sma}})$. In this case, the bottleneck of the queue is likely to be the dequeuers, hence we set the value $\widehat{\mathcal{T}_e^{(-)}}(p_{\text{sma}}) = \widehat{\mathcal{T}_e^{(+)}}(p_{\text{sma}})$ by default.

All $\widehat{\mathcal{T}_o^{(b)}}$ are then obtained by joining $\widehat{\mathcal{T}_o^{(b)}}(p_{\text{sma}})$ to the leftmost point of the low intra-contention part:

$$\widehat{\mathcal{T}_o^{(b)}}(pw_o) = \begin{cases} \frac{\frac{f}{\widehat{cw_o^{(b)}}} - \widehat{\mathcal{T}_o^{(b)}}(p_{\text{sma}})}{(n-1)\widehat{cw_o^{(b)}} - p_{\text{sma}}} \times (pw_o - p_{\text{sma}}) + \widehat{\mathcal{T}_o^{(b)}}(p_{\text{sma}}) & \text{if } pw_o \leq (n-1)\widehat{cw_o^{(b)}} \\ \frac{n \times f}{pw_o + \widehat{cw_o^{(b)}}} & \text{otherwise.} \end{cases}$$

Finally, dequeuers' and enqueuers' throughput are reconstituted as explained in Section 3.3.3.3: if Equation 22 is fullfilled, then they are computed through Equations 20 and 21 that can be rewritten as:

$$\begin{cases} \widehat{\mathcal{T}_d}(pw_d, pw_e) = \frac{\widehat{\mathcal{T}_d^{(+)}}(pw_d) + \widehat{\mathcal{T}_e^{(-)}}(pw_e)\left(1 - \frac{\widehat{\mathcal{T}_d^{(+)}}(pw_d)}{\widehat{\mathcal{T}_d^{(-)}}(pw_d)}\right)}{1 - \frac{pw_d}{nf}\left(\widehat{\mathcal{T}_e^{(+)}}(pw_e) - \widehat{\mathcal{T}_e^{(-)}}(pw_e)\right)\left(1 - \frac{\widehat{\mathcal{T}_d^{(+)}}(pw_d)}{\widehat{\mathcal{T}_d^{(-)}}(pw_d)}\right)} \\ \widehat{\mathcal{T}_e}(pw_d, pw_e) = \frac{\widehat{\mathcal{T}_d}(pw_d, pw_e) \times pw_d}{n \times f} \times \widehat{\mathcal{T}_e^{(+)}}(pw_e) + \left(1 - \frac{\widehat{\mathcal{T}_d}(pw_d, pw_e) \times pw_d}{n \times f}\right) \times \widehat{\mathcal{T}_e^{(-)}}(pw_e). \end{cases}$$

Otherwise, $\widehat{\mathcal{T}_d}(pw_d, pw_e) = \widehat{\mathcal{T}_d^{(-)}}(pw_d)$ and $\widehat{\mathcal{T}_e}(pw_d, pw_e) = \widehat{\mathcal{T}_e^{(+)}}(pw_e)$.

3.4.2 Instantiating the Power Model

We use once again $p_{\text{sma}} = 1$, $p_{\text{mid}} = 20$ and $p_{\text{big}} = 1000$ as three distinctive amounts of work, that allows easy approximations for the power dissipation expressions.

We have seen that if $X \in \{M, U\}$, then $P^{(X)} = r_d \times \rho_d^{(X)} + r_e \times \rho_e^{(X)}$, which can be approximated at $(pw_d, pw_e) = (p_{\text{big}}, p_{\text{sma}})$ by $P^{(X)}(p_{\text{big}}, p_{\text{sma}}) = r_e(p_{\text{sma}}) \times \rho_e^{(X)}$, since r_d is then nearly 0. It implies that

$$\widehat{\rho_e^{(X)}} = \frac{P^{(X)}(p_{\text{big}}, p_{\text{sma}})}{1 - \frac{\mathcal{T}_e(p_{\text{big}}, p_{\text{sma}}) \times p_{\text{sma}}}{n \times f}}.$$

We obtain $\widehat{\rho_d^{(X)}}$ similarly at $(pw_d, pw_e) = (p_{\text{sma}}, p_{\text{big}})$.

Concerning the dynamic CPU power, we firstly estimate the power dissipated in the parallel sections. According to the implementation, the CPU power dissipated by the parallel section of enqueuers and dequeuers is the same for both, and this power does not depend on the amount of work. These restrictions are not a loss of generality, since the aim here is to study the queue implementations. It can then be estimated by using $(p_{\text{big}}, p_{\text{big}})$, where the ratios r_o can be considered as 0, which leads to

$$\widehat{P^{(C)}_{o,PS}} = \frac{P^{(C)}(p_{\text{big}}, p_{\text{big}})}{2n}.$$

We reuse the point $(p_{\text{big}}, p_{\text{sma}})$, where r_d is very close to 0, to derive that

$$P^{(C)} = n\left(r_e(p_{\text{sma}}) \times \widehat{P^{(C)}_{e,RL}} + (1 - r_e(p_{\text{sma}}))\widehat{P^{(C)}_{e,PS}}\right) + n\widehat{P^{(C)}_{d,PS}},$$

which is equivalent to

$$\widehat{P^{(C)}_{e,RL}} = \frac{P^{(C)}(p_{\text{big}}, p_{\text{sma}})}{n\left(1 - \frac{\mathcal{T}_e(p_{\text{big}}, p_{\text{sma}})p_{\text{sma}}}{n \times f}\right)} - \left(\frac{2}{1 - \frac{\mathcal{T}_e(p_{\text{big}}, p_{\text{sma}})p_{\text{sma}}}{n \times f}} - 1\right)\widehat{P^{(C)}_{o,PS}}$$

Once again, we obtain $\widehat{P^{(C)}_{d,RL}}$ with the same line of reasoning at $(pw_d, pw_e) = (p_{\text{sma}}, p_{\text{big}})$.

Finally, $\widehat{P^{(M)}}$ and $\widehat{P^{(U)}}$ (resp. $\widehat{P^{(C)}}$) are computed by using Equation 27 (resp. Equations 23 and 24), and the estimates of the ratios that are issued from Section 3.3.3

$$\widehat{r_o} = 1 - \frac{\widehat{\mathcal{T}_o} \times pw_o}{n \times f}.$$

3.4.3 Summary

To summarize, we have built a model that needs to be calibrated (instantiated) before the execution of an application that would use the queue. The calibration phase starts with the run of synthetic benchmarks on the following set of points and the measurements of the dequeuers' and enqueuers' throughputs:

$$(pw_d, pw_e) \in \big\{(p_{\text{mid}}, p_{\text{sma}}), (p_{\text{mid}}, p_{\text{big}}), (p_{\text{sma}}, p_{\text{mid}}), (p_{\text{big}}, p_{\text{mid}}), \\ (p_{\text{sma}}, p_{\text{sma}}), (p_{\text{big}}, p_{\text{sma}}), (p_{\text{big}}, p_{\text{big}}), (p_{\text{sma}}, p_{\text{big}})\big\}.$$

The calibration phase continues with the extraction of the parameters that rule the four basic throughputs (hence the dequeuers' and enqueuers' throughputs) on the whole domain.

After this calibration phase, we are able, given any parallel section values (pw_d, pw_e), to estimate the throughput and the energy consumption of the application, through a few basic arithmetic operations, as explained in the previous subsections.

4 Programming Abstractions and Libraries

In this section, we describe our studies on programming abstractions and libraries such as concurrent search trees and concurrent lock-free queues.

4.1 Concurrent Search Trees

In this section, we present libraries of concurrent search trees and their performance and energy analysis.

4.1.1 Energy-efficient Concurrent Search Trees

4.1.1.1 DeltaTree (ΔTree)

Concurrent trees are fundamental data structures that are widely used in different contexts such as load-balancing [27, 46, 73] and searching [2, 15, 16, 24, 30, 32]. Most of the existing highly-concurrent search trees are not considering the fine-grained data locality. The non-blocking concurrent search trees [16, 32] and Software Transactional Memory (STM) search trees [2, 15, 24, 30] 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 [30] by Oracle Labs and the concurrent AVL tree developed by Stanford [15] 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 [57, 72] are not portable while there are big interests of concurrent data structures for unconventional platforms [47, 43]. Concurrency control techniques such as transactional memory [51, 48] and multi-word synchronization [49, 42, 59] do not take into account fine-grained locality while fine-grained locality-aware techniques such as van Emde Boas layout [71, 84] poorly support concurrency.

Based on the new concurrency-aware vEB (cf. Section 3.1.5), we implement ΔTree [80], a portable locality-aware unbalanced concurrent search tree. Figure 25 illustrates a ΔTree U which is composed by a group of subtrees (ΔNodes). A ΔNode's internal nodes are put together in cache-oblivious fashion using the concurrency-aware vEB layout (cf. Section 3.1.5). The search operation for ΔTree is wait-free. We are aware that the ΔTree has two major shortcomings, namely being an unbalanced tree and having a poor memory utilization because of the inter-node pointers usage.

In order to address the ΔTree shortcomings, we implement the Balanced ΔTree (bΔTree). bΔTree is devised by improving the structure and the algorithm of ΔTree to support the concurrent, Btree-like bottom-up insertions, which ensures balanced tree. Another major change is that in bΔTree, the fat-pointer ΔNodes are replaced with pointer-less ΔNodes. The removal of ΔNodes internal pointers made way for 200% more nodes to fit into the tree (in 64-bit x86 systems, a pointer costs 8 bytes of memory while an unsigned variable

requires only 4 bytes of memory). The concurrent search operations supported by bΔTree are still without locks and waits, but they are no longer wait-free. Based on our study, a more-compact tree results in less data transfers. And less data transfers leads to better performance and energy efficiency.

Finally, we implement the Heterogeneous ΔTree (hΔTree), which is a better performing, more-compact tree than bΔTree. hΔTree is devised by changing the leaf-oriented (external) tree layout of the leaf ΔNodes into an internal tree layout. This improvement allows 100% more nodes to fit inside the tree, which further improves the tree's overall operation performance.

Based on experimental insights, our ΔTrees are different from previous theoretical designs of concurrent cache-oblivious (CO) trees such as the concurrent packed-memory CO tree and concurrent exponential CO tree [11]. The concurrent packed-memory CO tree gives a good amortized memory transfer cost of $\Theta(\log_B N + (\log^2 N/B))$ for tree updates, assuming that operations occur *sequentially*. However, the proposed data representation requires each node to have the parent-child pointers. Besides the complication in re-arranging those pointers, we have found that eliminating pointers from the node to minimize memory footprint is significantly beneficial for cache-oblivious tree in practice (cf. improvement from ΔTree to bΔTree in Section 4.1.1.2). In the ΔTree experimental evaluation (cf. Section 4.1.3), bΔTree is 100% faster than ΔTree in searching, which is attained by simply removing pointers from the tree node.

In the concurrent exponential CO tree by Bender et al. [11], expected memory transfer cost for search and update operations is $\mathcal{O}(\log_B N + (\log_\alpha \lg N))$, assuming that all processors are *synchronous*. Cormen et al. [23, pp. 212], however, wrote that although the underlying exponential tree algorithm [4] is an important theoretical breakthrough, it is complicated and unlikely to compete with similar sorting algorithms. In fact, nodes in the exponential tree grow exponentially in size, which not only complicates maintaining inter-node pointers but also exponentially increases the tree's memory footprint in practice. In contrast, the memory footprint of ΔTree with the fixed size ΔNodes gradually expands on-demand when the tree grows. Thanks to the fixed size ΔNodes, ΔTree exploits further locality by utilizing a "map" and an efficient inter-node connection (cf. Section 4.1.1.2).

The ΔTree consists of $|U|$ ΔNodes of fixed size UB. Each of the ΔNode contains a *leaf-oriented* binary search tree (BST) $T_i, i = 1, \ldots, |U|$. The ΔTree U provides the following operations: INSERTNODE(v, U), which adds value v to the set U, DELETENODE(v, U) for removing a value v from the set, and SEARCHNODE(v, U), for determining whether value v exists in the set. We use the term *update* operation for either insert or delete operation. We assume that duplicate values are not allowed inside the set and a special value, for example 0, is reserved as an indicator of an EMPTY value.

Data structures. The implementation of ΔTree utilizes the data structure in Figure 26. The topmost level of ΔTree is represented by a struct UNIVERSE (line 19) that contains a pointer (**root**) to the root node of the first ΔNode ($T_1.n_1$). A ΔTree is formed by a group of ΔNodes.

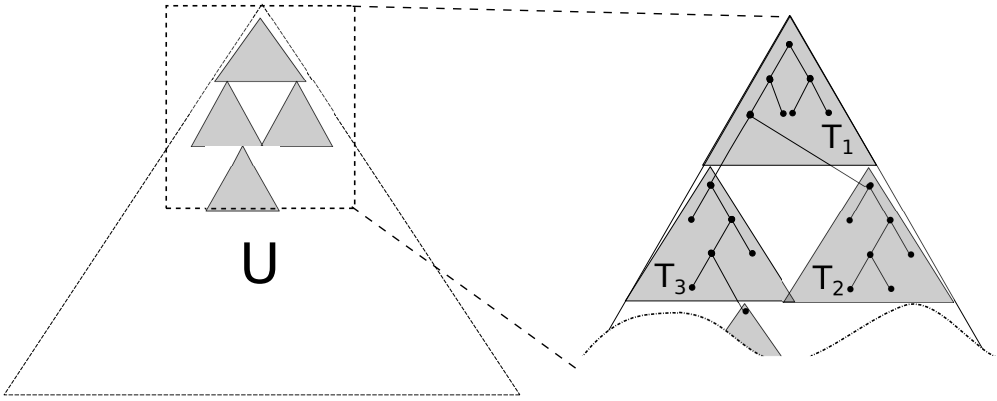


Figure 25: Depiction of a ΔTree U. Triangles T_x represent the ΔNodes.

Struct NODE n:
    member fields:
        $tid \in \mathbb{N}$, if > 0 indicates the node is *root* of a ΔNode with an id of tid (T_{tid})
        $value \in \mathbb{N}$, the node value, default is **empty**
        $mark \in \{true, false\}$, a value of **true** indicates a logically deleted node
        $left, right \in \mathbb{N}$, left and right child pointers
        $isleaf \in true, false$, indicates whether the node is a leaf of a ΔNode,
         default is **true**

Struct ΔNODE T:
    member fields:
        nodes, a group of pre-allocated NODE n $\{n_1, n_2, \ldots, n_{UB}\}$
        buffer, ΔNode's buffer (pre-allocated array of $b_1, b_2, \ldots, b_{\#threads}$)
        meta, ΔNode's metadata (single ΔNODEMETA)

Struct ΔNODEMETA M:
    member fields:
        locked, indicates whether a ΔNode is locked
        opcount, a counter for the active update operations
        root, pointer to the root node of the ΔNode ($T_x.n_1$)
        mirror, pointer to the root node of the ΔNode's mirror ($T_{x'}.n_1$)

Struct UNIVERSE U:
    member fields:
        root, pointer to the *root* of the topmost ΔNode ($T_1.root$)

Figure 26: ΔTree's data structures.

```
1:  function searchNode(v, U)
2:      p ← U.root
3:      while (TRUE) do
4:          lastnode ← p                                   ▷ atomic assignment
5:          if (p.value < v) then
6:              p ← p.left
7:          else
8:              p ← p.right
9:          if (!p or lastnode.isleaf = TRUE) then
10:             break
11:     if (lastnode.value = v) then
12:         if (lastnode.mark = FALSE) then                ▷ lastnode is not deleted
13:             return TRUE
14:         else
15:             return FALSE
16:     else
17:         {Search the last visited ΔNode's buffer for v}
18:         if ({v is found}) then
19:             return TRUE
20:         else
21:             return FALSE
```

Figure 27: ΔTree's *wait-free* search algorithm.

ΔTree's ΔNodes are represented by the struct ΔNode (line 8). A ΔNode consists of a collection of *UB* **nodes** and a **buffer** array. ΔNode's **buffer** array length is equal to the number of operating threads. Each ΔNode is accompanied by a metadata ΔNodeMeta that holds lock and counters variable.

Struct ΔNodeMeta (line 13) acts as metadata for every ΔNode. This structure consists of a field **opcount**, which is a counter that indicates the number of insert/delete threads that are currently operating within that ΔNode; and field **locked** that indicates whether a ΔNode is currently locked by a maintenance operation. When **locked** is set as *true*, no insert/delete threads are allowed to get into a ΔNode. Lastly, ΔNodeMeta contains a **root** pointer that points to the first node of ΔNodeMeta's accompanying ΔNode, and the pointer **mirror** that points to the *root* of the ΔNode's mirror (cf. Section 4.1.1.1).

Each node structure (line 1) contains field **value**, which holds a value for guiding the search, or a data value if it resides in a leaf-node. Field **mark** indicates a logically deleted value, if set to *true*. A *true* value of **isleaf** indicates the node is a leaf node, and *false* otherwise. Field **tid** is a unique identifier of a corresponding ΔNode and it is used to let a thread know whether itself has moved between ΔNodes. A **tid** is only defined in the root node of a ΔNode.

ΔTree functions. ΔTree provides basic functions such as search, insert and delete functions. We refer insert and delete operations as the *update* operations.

Function SEARCHNODE(v, U) (cf. Figure 27), is going to walk over the ΔTree to find whether the value v exists in U (Figure 27, lines 4–10). The SEARCHNODE(v, U) function returns **true** whenever v has been found, or **false** otherwise (Figure 27, line 12). This operation is guaranteed to be wait-free (cf. Lemma 4.1).

Lemma 4.1. Δ*Tree search operation is wait-free.*

Proof. (Sketch) The proof can be served based on these observations on Figure 27:

1. SEARCHNODE and invoked SEARCHBUFFER (line 17) do not wait for any locks.
2. The number of iterations in the *while* loop (line 3) is bounded by the *height* T of the tree, or $\mathcal{O}(T)$. The loop is always ended whenever a leaf node or an empty node is found in line 10.
3. SEARCHBUFFER time complexity is bounded by the buffer size, which is a constant.

Therefore the SEARCHNODE time is bounded by $\mathcal{O}(T)$, where T is the height of the tree. □

```
1: function INSERTNODE(v, p)
2:     p ← U.root
3:     BEGIN:
4:     if ({Entering new ΔNode T_x}) then
5:         Decrement previous ΔNode's opcount (T'_x.opcount)
6:         Wait if ΔNode is currently maintained
7:     if Not at the tree's last level node then
8:         if (v < p.value) then
9:             if Is at leaf (p.isleaf = TRUE) then
10:                if (CAS(p.left.value, empty, v) = empty) then
11:                    Do insert to the left
12:                    Decrement T_x.opcount
13:                else
14:                    Re-try insert from p
15:            else
16:                Go to left child (p ← p.left) ) and restart from BEGIN
17:        else if (v > p.value) then
18:            if (Is at leaf (p.isleaf = TRUE) then
19:                if (CAS(p.left.value, empty, p.value) = empty) then
20:                    Do insert to the right
21:                    Decrement T_x.opcount
22:                else
23:                    Re-try insert from p
24:            else
25:                Go to right child (p ← p.right) and restart from BEGIN
26:        else if (v = p.value) then
27:            if Is at leaf (p.isleaf = TRUE) then
28:                if v is not deleted (p.mark = FALSE) then
29:                    Decrement T_x.opcount                    ▷ v already exist
30:                else
31:                    Do insert right
32:            else
33:                Go to right child (p ← p.right) and restart from BEGIN
34:    else
35:        if v is already in T_x.buffer then
36:            Decrement T_x.opcount
37:        else
38:            Put v inside T_x.buffer                          ▷ buffered insert
39:            if (TAS(T_x.locked)) then                        ▷ Acquire maintenance lock
40:                Decrement T_x.opcount
41:                Wait for all updates to finish (T_x.opcount=0)
42:                do REBALANCE(T_x) or EXPAND(p)
```

```
43: function DELETENODE(v, p)
44:     p ← U.root
45:     BEGIN:
46:     if ({Entering new ΔNode T_x}) then
47:         Decrement previous ΔNode's opcount (T'_x.opcount)
48:         Wait if ΔNode is currently maintained
49:     if Is at leaf (p.isleaf = TRUE)) or at the tree's last level node then
50:         if (p.value = v) then
51:             if (CAS(p.mark, FALSE, TRUE) != FALSE)) then
52:                 Decrement T_x.opcount                          ▷ v is already deleted
53:             else
54:                 if p is still the leaf node then
55:                     if (TAS(T_x.locked)) then                  ▷ Acquire the maintenance lock
56:                         Decrement T_x.opcount
57:                         Wait for all updates to finish (T_x.opcount=0)
58:                         Do node merge if needed
59:                 else
60:                     Re-try delete from current node p
61:         else
62:             Search (T_x.buffer) for v
63:             if v is found in T_x.buffer.idx then
64:                 Remove v from T_x.buffer.idx                   ▷ buffered delete
65:             Decrement T_x.opcount
66:     else
67:         if (v < p.value) then
68:             Go to left child (p ← p.left)
69:         else
70:             Go to right child (p ← p.right)
71:         Restart from BEGIN
```

Figure 28: ΔTree's concurrent update (*insert* and *delete*) algorithms.

Function INSERTNODE(v, U) (cf. Figure 28, line 1) inserts value v at a leaf of ΔTree, provided v does not exist in the tree (Figure 28, line 29). Following the nature of a leaf-oriented tree, a successful insert operation replaces a leaf with a subtree of three nodes [32] (cf. Figure 29a and in Figure 28, lines 10 and 19). Because the ΔTree structure is pre-allocated, an insert operation needs only to do a "logical" replacement, or only replacing the value of a leaf and its children.

The function DELETENODE(v, U) (cf. Figure 28, line 43) *marks* the leaf that contains value v as deleted. DELETENODE(v, U) fails if v does not exist in the tree or the leaf containing v is already mark as deleted.

To avoid conflicting insert and delete operations, INSERTNODE and DELETENODE operations are using the single word CAS (Compare and Swap) and *leaf-checking* to coordinate between the operations (cf. Figure 28, lines 51 and 54 for delete, and 10 and 19 for insert).

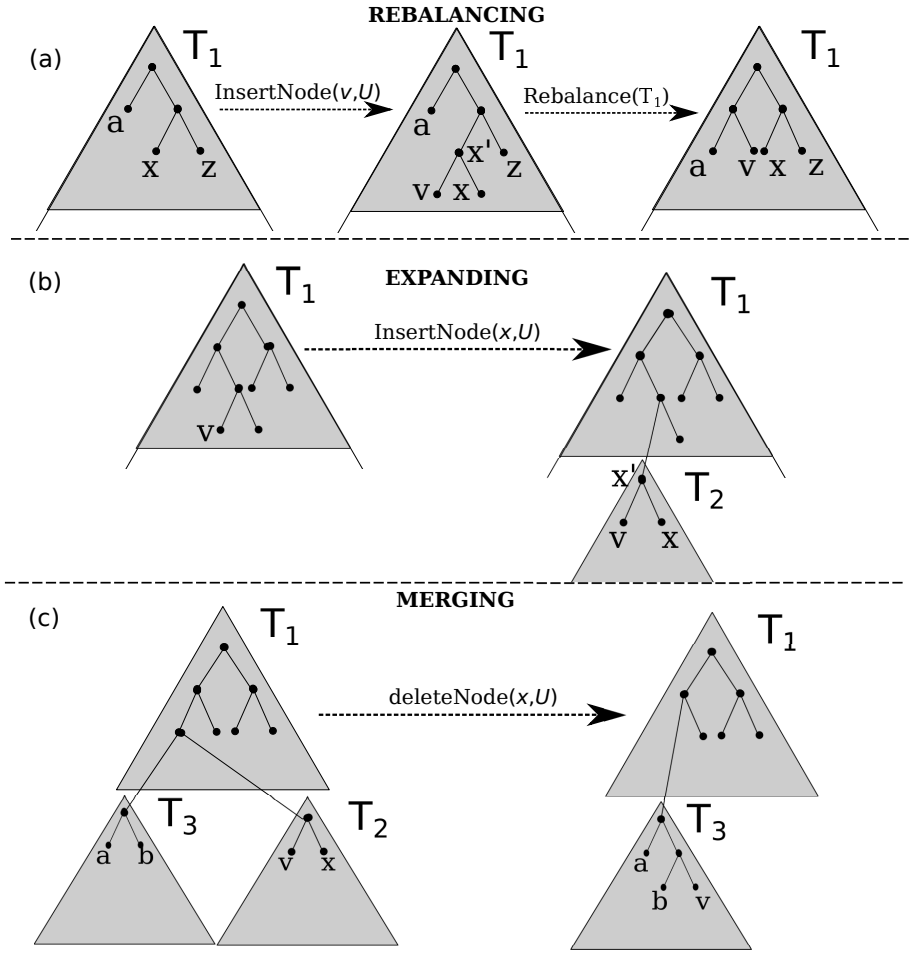


Figure 29: (a)*Rebalancing*, (b)*Expanding*, and (c)*Merging* operations on ΔTree.

Maintenance functions. Other than the basic functions, ΔTree has tree maintenance functions that are invoked in certain cases of inserting and deleting a node from the tree. These functions, namely rebalance, expand and merge, are the unique feature of ΔTree that acts as a garbage collectors for marked nodes and as a safeguard to maintain ΔTree's small height.

Function REBALANCE($T_v.root$) (cf. Figure 28 line 42) is responsible for rebalancing a ΔNode after an insertion. Figure 29a illustrates the rebalance operation. If a new node v is to be inserted at the last level H of ΔNode T_1, the ΔNode is rebalanced to a complete BST. All leaves' heights are then set to $\lfloor \log N \rfloor + 1$, where N is the number of leaves (e.g., a, v, x, z in Figure 29a). In the same Figure 29a, after rebalancing, tree T_1 height reduces from 4 to 3.

We also define the EXPAND(l) function (cf. Figure 28 line 42) that is responsible for creating a new ΔNode and connecting it to the parent of a leaf node l (cf. Figure 29b). Expanding is triggered only if 1) after INSERTNODE(v, U), leaf that contains v will be at the last level of a ΔNode; and 2) rebalancing will no longer reduce the current height of ΔNode

T_i. For example if an expand takes place at node l which is located at the lowest level of a ΔNode, or $depth(l) = H$, the parent of l swaps one of its child pointers that previously points to l into the root of the newly created ΔNode's *root* (cf. Figure 29b).

Function MERGE($T_x.root$) (cf. Figure 28 line 58) is defined to merge T_x with its sibling. For example in Figure 29c, T_2 is merged into T_3. Then the pointer of T_3's grandparent that previously points to the parent of both T_3 and T_2 is replaced by a pointer to T_3. Merge operation is invoked provided that a particular ΔNode, where a deletion has taken place, is filled less than $2^{t/2}$ of its capacity (where $t = \log(UB)$) and both nodes of that ΔNode and its siblings are going to fit into a single ΔNode.

Before any maintenance operation starts, the threads responsible are required to do Test and Set (TAS) on ΔNode's *locked* field (cf. Figure 28, lines 39 and 55). Advanced locking techniques [45, 53, 61] can also be used. If a ΔNode's *locked* has been set to TRUE, new incoming update operations are forced to wait at the tip of that ΔNode so that they will not interfere with the maintenance operation (cf. Figure 28, line 48 and 6) .

For efficiency, a ΔNode's maintenance operation starts after a batch of concurrent update operations within that ΔNode. The TAS also serves as a mechanism to make the concurrent update operations compete for a maintenance lock (*locked* variable). The losing threads will leave their maintenance burden to a thread who successfully acquired the ΔNode's *locked* variable.

After the maintenance operation finished, ΔNode's *locked* is released by the maintenance thread. Therefore, waiting update threads can continue with their respective operation within the newly-maintained ΔNode.

Note that although REBALANCE and MERGE execution are sequential within a ΔNode, invoking these functions involves at most $2 \times UB$ nodes, where a constant $UB \ll N$.

Mirroring. Whenever a ΔNode is undergoing a maintenance operation (balancing, expanding, or merging), a mirroring operation also takes place to facilitate the wait-free search. Mirroring works by maintaining the original ΔNode and writing the results into the mirror ΔNode.

After a maintenance operation finishes, the pointer to the root of the maintained ΔNode is switched to the root of the ΔNode's mirror. In result, the mirror, or maintained ΔNode, is now become part of ΔTree, replacing the "old" ΔNode. As the ΔNode's buffer is enclosed within a ΔNode, it is also switched at the same time ΔNode was switched. As a result, the original ΔNode and its helping buffer served as the latest snapshot, which enables wait-free search for ΔTree.

4.1.1.2 Balanced ΔTree

ΔTree served as a proof of concept of a concurrency-aware vEB-based search trees, although it has major weaknesses that can affect its empirical performance. First, the *left* and *right* pointers are occupying larger memory spaces compared to the data values. For example, in a ΔNode with 127 nodes, the set of pointers will occupy 2032 bytes (127×16 bytes) of memory in a 64-bit operating system. Therefore, the amount of space used by pointers alone

is four times the amount of space used by the data values (i.e., 127×4 bytes = 508 bytes, assuming node's *value* is a 4 bytes integer). This is inefficient, because every block transfer between any levels of memory carries a bigger portion of pointers instead of important data. Second, inserting a sequence of increasing or decreasing numbers into ΔTree will results a linked-list of ΔNode (i.e., ΔTree's height is equal to N, or the number of elements). This is a direct implication of the unbalanced form of ΔTree.

Balanced ΔTree (bΔTree) is implemented to address the ΔTree's weaknesses. bΔTree is developed by devising three major strategies, namely replacing internal ΔNode pointers with map, crafting the efficient inter-node connection, and ensures that the tree is always balanced.

Improving ΔTree to become a pointer-less concurrent cache-oblivious is difficult. For example, Brodal, et al. [13] have discussed about pointer-less CO BST and their implementation involves heavy calculation to determine the children positions of a node. In contrast to our ΔTree that consists of group of ΔNodes, they are treating their tree as N-sized vEB-layout tree. Thus, not only we need to address the node parent-child pointers, we must also consider the inter-ΔNode pointers.

As a matter of fact, having fixed-sized ΔNodes ($UB \ll N$) is actually advantageous. A single pre-calculated map of all UB nodes can be generated once and used by all operations throughout the whole tree. The inter-ΔNode connection still need pointers though, but instead of $2 \times UB$ pointers, we will show that a significant reduction of pointers ($^1/_2 \times UB$) is possible.

"Map" instead of pointers. The Balanced ΔTree is implemented by completely eliminating (*left* and *right*) pointers within a ΔNode. Instead, these pointers are replaced with the corresponding LEFT and RIGHT functions (Figure 30, lines 6 and 13). LEFT and RIGHT functions, given an arbitrary node v and the root memory address of its container ΔNode, will return the the addresses of the left and right child nodes of v. These functions will return 0 if v is the deepest node within a ΔNode. The LEFT and RIGHT functions requires a *map* array with UB length to determine a node's children address (Figure 30, line 1).

ΔNode's *map* is a read-only record of the nodes' memory address differences. The address differences of every ΔNode's nodes and its left and right children are pre-calculated and recorded when the Balanced ΔTree is initialized. Because ΔNodes are using the same fixed-size concurrency-aware vEB layout, only one map is needed for all of the available ΔNodes. Therefore, since only one *map* with size UB is used for all traversing, memory footprint for the Balanced ΔTree's ΔNode operations can be kept minimum (namely, the read-only map is always cached). Please note that the *map* only supports internal (within ΔNode) node traversal.

Inter-ΔNode connection. To facilitate the traversing from one ΔNode to another, the inter-ΔNode connection mechanism is used in addition to the *map*. To use the inter-ΔNode connection, first we logically assign a color to each ΔNode's node edge. Each node

```
1:  Struct Map:
2:      member fields:
3:          left ∈ ℕ, interval of the left child pointer address
4:          right ∈ ℕ, interval of the right child pointer address

5:  Map map[UB]

6:  function RIGHT(p, base)
7:      nodesize ← SIZEOF(single node)
8:      idx ← (p − base)/nodesize
9:      if (map[idx].right != 0) then
10:         return base + map[idx].right
11:     else
12:         return 0

13: function LEFT(p, base)
14:     nodesize ← SIZEOF(single node)
15:     idx ← (p − base)/nodesize
16:     if (map[idx].left != 0) then
17:         return base + map[idx].left
18:     else
19:         return 0
```

Figure 30: *Mapping* functions.

has only two outgoing edges, which are the left edge and the right edge. These left and right edges are assigned color 0 and 1, respectively. Now any paths traversed from the *root* of a ΔNode to reach any internal node will produce a bit-sequence of colors. This bit representation can be translated into an array index that contains a pointer to a child ΔNode. The maximum size of the bit representation is the height of ΔNode or $\log(UB)$ bits. We are using a leaf-oriented tree and allocate a pointer array with $^1/_2 \times UB$ length. Pseudocode in Figure 31 explains how the inter-ΔNode connection works in a pointer-less search function.

The removal of the internal nodes' pointers results in 200% more node counts in a ΔNode compared to the ΔTree with the same *UB* size. A ΔNode's struct is now consist of internal nodes, which is just an array of keys, plus an array of $^1/_2 \times UB$ pointers for the inter-ΔNode connection.

Concurrent and balanced tree. To enable both of the concurrency and balanced tree out of the ΔTree, new variables are added into the ΔNodes and changes are made to the ΔTree's update and maintenance operations. We improved ΔTree's update algorithms by adopting the algorithms from the concurrent lock-based B-link trees [60] by Lehman and

```
1:  function pointerLessSearch(key, ΔNode, maxDepth)
2:      while ΔNode is not leaf do
3:          bits ← 0
4:          depth ← 0
5:          p ← ΔNode.root
6:          base ← p
7:          link ← ΔNode.link
8:          while (p & p.value != EMPTY) do          ▷ continue until leaf node
9:              depth ← depth + 1                    ▷ increment depth
10:             bits ← bits << 1                     ▷ shift one bit to the left in each level
11:             if (key < p.value) then
12:                 p ← LEFT(p, base)
13:             else
14:                 p ← RIGHT(p, base)
15:                 bits ← bits + 1                  ▷ right child color is 1
    ▷ pad the bits to get the index of the child ΔNode's pointer:
16:         bits ← bits << (maxDepth − depth) − 1
    ▷ follow nextRight if highKey is less than searched value:
17:         if (ΔNode.highKey <= key) then
18:             ΔNode ← ΔNode.nextRight
19:         else
20:             ΔNode ← link[bits]                   ▷ jump to child ΔNode
21:     return ΔNode
```

Figure 31: Search within pointer-less ΔNode. This function will return the *leaf* ΔNode containing the searched key. From there, a simple binary search using LEFT and RIGHT functions is adequate to pinpoint the key location. The search operations are utilizing both the *nextRight* pointers and *highKey* variables to handle concurrent ΔNode splitting [60].

Yao.

The insert operations in bΔTrees are now done in a bottom-up fashion to ensure that the tree is always balanced. Meanwhile, the search operations are done in a top-down, left-to-right fashion.

The modification to how the insert and search operations work results in the omission of both MERGE($T_x.root$) and EXPAND(l) functions (cf. Section 4.1.1.1). Instead, the new SPLIT(T_v) functions is introduced to aid the bottom-up tree building.

SPLIT(T_v) functions splits the ΔNode T_v into T'_v and a right sibling T_x. The internal nodes that were previously occupying T_v are distributed evenly between T'_v and T_x. If T_v is the topmost ΔNode, a new parent ΔNode T_p is created as well, taking both T'_v and T_x as its children. The split operation concludes with an insertion of the lowest value from T_x nodes into T_p. The ΔNode split is triggered only when a ΔNode overflows after an insertion, which means it is at least filled with $(2^{t/2})$ internal nodes, where $t = \log(UB)$.

Beside the addition of split operation, two additional variables are added inside the struct ΔNodeMeta. These additional variables are *nextRight* pointer, which points to the right sibling ΔNode; and *highKey* variable that contains the upper-bound value of the ΔNode. Starting with NULL values, both of these new variables are populated when the ΔNode splits. For example, after T_v splits, the $T_v.nextRight$ is going to point to T_x, while $T_v.highKey$ is set to the minimum value of all the nodes in T_x. These new variables help the concurrent search with respect to the node splits [60].

With the addition of *nextRight* and *highKey* variables, Balanced ΔTree's search operations do not need locks and waits (cf. Figure 31). However, the search cannot be regarded as wait-free. According to Lehman and Yao [60], in the worst case, which is extremely unlikely, an infinite loop caused by continuous ΔNode splits might happen.

To address the memory waste, the same rebalancing procedure as in Figure 29a is employed by bΔTree. The rebalancing also helps to clean-up the nodes marked for deletion, keeping ΔNodes always in a good shape. A mirroring whenever the tree splits and rebalances is also retained to make the tree traversals not require any locks and waits.

4.1.1.3 Heterogeneous ΔTree

In Balanced ΔTree, the leaf-oriented (or external tree) layout is adopted for ΔNodes in order to facilitate the inter-ΔNode connection mechanism using $^1/_2 \times UB$ pointers. However, it is not necessary for the leaf ΔNodes to have leaf-oriented layout since they do not have any successor ΔNodes.

Based on the above observation, we devise a heterogeneous balanced ΔTree by changing the layout of the balanced ΔTree's leaf ΔNodes. This new layout uses internal tree instead of the external tree for the leaf ΔNodes. With this layout change, 100% more key nodes are now fit into any leaf ΔNode, if compared to the previously non-leaf ΔNodes with the same *UB* limit. To save more memory spaces, we also remove the array of pointers for intra-ΔNode connection in the leaf ΔNodes metadata.

With this improved version of ΔTree, or heterogeneous ΔTree, we find that the efficiency of searches is improved (cf. Section 4.1.3.2). Compared to ΔTree and balanced ΔTree, the heterogeneous ΔTree delivers lower number of cache misses and more efficient branching.

4.1.2 Libraries of Concurrent Search Trees

We have developed concurrent search tree libraries that contain the following components:

1. Non-blocking binary search tree (NBBST). The Non-blocking binary search tree (NBBST) library contains the non-blocking binary search tree of Ellen [32].

2. STM-based search trees. STM-based search trees library contains the Software Transactional Memory (STM)-based AVL tree (AVLtree), red-black tree (RBtree), and speculation friendly tree (SFtree) from the Synchrobench benchmark [40].

3. Concurrent B-tree (CBtree). Concurrent B-tree (CBTree) library contains an optimized Lehman and Yao B-link tree [60]. B-link tree is a highly-concurrent B-tree

variant and it is still being used as a backend in popular database systems such as PostgreSQL[4].

4. Static cache-oblivious tree using the static vEB layout (VTMtree). The Concurrent static vEB binary search tree (VTMTree) library contains a concurrent version of the static vEB binary search tree [13] developed using GNU C Compiler v4.9.1's STM.

5. DeltaTree (ΔTree). The tree families include Delta tree, Balanced Delta tree and Heterogeneous Delta tree are described in Section 4.1.1.

4.1.2.1 Obtaining and compilation.

The libraries are provided in a separate directory for easy access and maintenance. The repository address is http://gitlab.excess-project.eu/ibrahim/tree-libraries. A makefile for each of the libraries is also provided to aid compilations. The libraries have been tested on Linux and Mac OS X platforms.

4.1.2.2 Running and outputs.

By default, the provided makefile will build the standalone benchmark version of the libraries which will accept these following parameters:

```
-r <NUM> :  Allowable range for each element (0..NUM)
-u <0..100> :  Update ratio.  0 = Only search; 100 = Only updates
-i <NUM> :  Initial tree size (initial pre-filled element count)
-t <NUM> :  DeltaNode (UB) size (ONLY USED IN DELTATREE FAMILIES)
-n <NUM> :  Number of benchmark threads
-s <NUM> :  Random seed.  (0 = using time as seed, Default)
```

The benchmark outputs are formatted in this sequence:

```
 0:  range, insert ratio, delete ratio, #threads, #attempted insert,
#attempted delete, #attempted search, #effective insert, #effective
delete, #effective search, time (in msec.)
```

NOTE: `0:` characters are just unique token for easy tagging (e.g., for using `grep`).

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

```
$ ./DeltaTree -h
DeltaTree v0.1
===============
Use -h switch for help.

Accepted parameters
-r <NUM> :  Range size
-u <0..100> :  Update ratio.  0 = Only search; 100 = Only updates
-i <NUM> :  Initial tree size (inital pre-filled element count)
-t <NUM> :  DeltaNode size
-n <NUM> :  Number of threads
-s <NUM> :  Random seed.  0 = using time as seed
-d <0..1> :  Density (in float)
-v <0 or 1> :  Valgrind mode (less stats).  0 = False; 1 = True
-h :  This help

Benchmark output format:
"0:  range, insert ratio, delete ratio, #threads, attempted insert,
attempted delete, attempted search, effective insert, effective delete,
effective search, time (in msec)"
```

```
$ ./DeltaTree -r 5000000 -u 10 -i 1024000 -n 10 -s 0
DeltaTree v0.1
===============
Use -h switch for help.

Parameters:
- Range size r:  5000000
- DeltaNode size t:  127
- Update rate u:  10%
- Number of threads n:  10
- Initial tree size i:  1024000
- Random seed s:  0
- Density d:  0.500000
- Valgrind mode v:  0

Finished building initial DeltaTree
The node size is:  25 bytes
Now pre-filling 1024000 random elements...
...Done!

Finished init a DeltaTree using DeltaNode size 127, with initial 1024000
members
#TS: 1421050928, 511389
Starting benchmark...
Pinning to core 0...  Success
Pinning to core 3...  Success
Pinning to core 1...  Success
Pinning to core 8...  Success
Pinning to core 9...  Success
Pinning to core 10...  Success
Pinning to core 2...  Success
Pinning to core 11...  Success
Pinning to core 4...  Success
Pinning to core 12...  Success

0:  5000000, 5.00, 5.00, 10, 249410, 248857, 4501733, 195052, 53720,
1000568, 476

Active (alloc'd) triangle:258187(266398), Min Depth:12, Max Depth:30
Node Count:1165332, Node Count(MAX): 1217838, Rebalance (Insert) Done:
234, Rebalance (Delete) Done:  0, Merging Done:  1
Insert Count:195052, Delete Count:53720, Failed Insert:54358, Failed
Delete:195137
Entering top:  0, Waiting at the top:0
```

NOTE: `#TS:` is the benchmark start timestamp.

4.1.2.3 Pluggable library.

To use any component as a library, each library provides a (.h) header file and a simple, uniform API in C. These available and callable APIs are:

STRUCTURE:

`<libname>_t` : Structure variable declaration.

FUNCTIONS:

`<libname>_t* <libname>_alloc()` : Function to allocate the defined structure, returns the allocated (empty) structure.

`void* <libname>_free(<libname>_t* map)` : Function to release all memory used by the structure, returns NULL on success.

`int <libname>_insert(<libname>_t* map, void* key, void* data)` : Function to insert a key and a linked pointer (data), returns 1 on success and 0 otherwise.

`int <libname>_contains(<libname>_t* map, void* key)` : Function to check whether a key is available in the structure, returns 1 if yes and 0 otherwise.

`void *<libname>_get(<libname>_t* map, void* key)` : Function to get the linked data given its key, returns the pointer of the data of the corresponding key and 0 if the key is not found.

`int <libname>_delete(<libname>_t* map, void* key)` : Function to delete an element that matches the given key, returns 1 on success and 0 otherwise.

As an example, the concurrent B-tree library provides the `cbtree.h` file that can be linked into any C source code and provides the callable `cbtree_t* cbtree_alloc()` function. It is also possible to use the MAP selector header (`map_select.h`) plus defining which tree type to use so that MAP_<operator>functions are used instead as specific tree function as the below example:

```
#define MAP_USE_CBTREE
#include "map_select.h"

int main(void)
{
        long numData = 10;
        long i;
        char *str;
```

```
        puts("Starting...");
        MAP_T* cbtreePtr = MAP_ALLOC(void, void);
        assert(cbtreePtr);
        for (i = 0; i < numData; i++) {
                str = calloc(1, sizeof(char)); *str = a +(i%254);
                MAP_INSERT(cbtreePtr, i+1, str);
        }
        for (i = 0; i < numData; i++) {
                printf("%ld: %c\n", i+1,
                        *((char*)MAP_FIND(cbtreePtr, i+1)));
        }
        for (i = 0; i < numData; i++) {
                printf("%ld: %d\n", i+1,
                        MAP_CONTAINS(cbtreePtr, i+1));
        }
        for (i = 0; i < numData; i++) {
                MAP_REMOVE(cbtreePtr, i+1);
        }
        for (i = 0; i < numData; i++) {
                printf("%ld: %d\n", i+1,
                        MAP_CONTAINS(cbtreePtr, i+1));
        }
        MAP_FREE(cbtreePtr)
        puts("Done.");
        return 0;
}
```

4.1.2.4 Intel PCM integration.

All of the libraries provide support for Intel PCM measurement. To enable Intel PCM measurement metrics, the compiler must be invoked using `-DUSE_PCM` parameter during the libraries's compilation and all the Intel PCM compiled object files must be linked to the output executables.

4.1.3 Performance and Energy Analysis of Concurrent Search Trees

We have experimentally analyzed the performance and energy efficiency of ΔTree (Section 4.1.1.1), balanced ΔTree (bΔTree) (Section 4.1.1.2), and heterogeneous ΔTree (hΔTree) (Section 4.1.1.3) in comparison with that of the other prominent concurrent search trees in the libraries (cf. Section 4.1.3.1). The experimental evaluation was conducted on both Intel high performance computing (HPC) and ARM embedded platforms. We have also evaluated how ΔTrees would perform in the worst-case and average-case setups against the highly optimized B-tree and the widely used GCC's std::set (cf. Section 4.1.3.6). The

experimental evaluation was conducted on both Intel high performance computing (HPC) platforms and an ARM embedded platform.

4.1.3.1 Testbed choices

Pthreads were used for threading and all running threads were pinned to the available physical cores using pthread_setaffinity_np. GCC 4.9.1 was used with -O2 for all program compilations. All of the tests are repeated at least 10× to guarantee consistent results.

Several decisions have been made for choosing VTMtree's size, ΔNodes' *UB* and CBTree's block size. Since the VTMtree's size was fixed, we set it to 2^{23} so that VTMtree needed not to expand. For fair comparisons, ΔTrees' *UB* and CBTree's order were set to their respective values so that each ΔNode and CBTree's node fitted within a memory page (4KB). Therefore, CBTree's order was set to 336.

Performance benchmark setup Performance indicators (in operations/second) were calculated using the number of (rep = 5,000,000) operations divided by the maximum time for the threads to finish the whole operations. Combination of update rate $u = \{0, 20, 50\}$ and number of thread $nr = \{1, 2, \ldots, 16\}$ were used for each run. Update rate of 0 equals to 100% search, while 50 update rate equals to 50% insert and delete operations out of rep operations. All involved operations used random values of $v \in (0, init \times 2], v \in \mathbb{N}$. All the trees were initialized with a number $init$ of nodes to simulate trees that partially fit into the last level cache (LLC) of the evaluating platform. The $init$ values were 4,194,303 and 2,097,151 for the Intel HPC platform and ARM embedded platform, respectively.

The Intel HPC platform was equipped with dual Intel Xeon E5-2670 CPUs, with total of 16 cores (no hyperthreading). The platform had 32GB of RAM, 2MB (8×256KB) L2 cache, and shared 20MB L3 cache for each CPU. Linux with kernel version 2.6.32-358 was used in this platform.

The embedded ARM platform was a Samsung Exynos 5410 octa-core ARM board with ARM's big.LITTLE CPU. This board was equipped with four Cortex-A15 1.6Ghz cores and four Cortex-A7 1.2Ghz cores in a single CPU and 2GB LPDDR3 RAM. The A15 cores had a 2MB shared L2 cache while the A7 cores had a 512KB shared L2 cache. Although the CPU had a total of 8 cores, only 4 cores could be active at a time because of its design limitation. Linux with kernel version 3.4.98 was used in this board. As the GCC compiler available for this board did not support the transactional memory extension, we had to remove VTMtree from the benchmarks for this platform.

Energy benchmark setup To assess the energy consumption of the trees, energy indicators were subsequently collected during 100% search and 50% update benchmarks. The ARM platform was equipped with a built-in "off-chip" power measurement system that was able to measure the energy for the A15 cores, A7 cores, and memory in real-time. For the Intel HPC platform, a server with two Intel Xeon E5-2690 v2 of 20 cores in total was used. The Intel PCM library using built-in CPU counters was used to measure the energy for each CPU and DRAM.

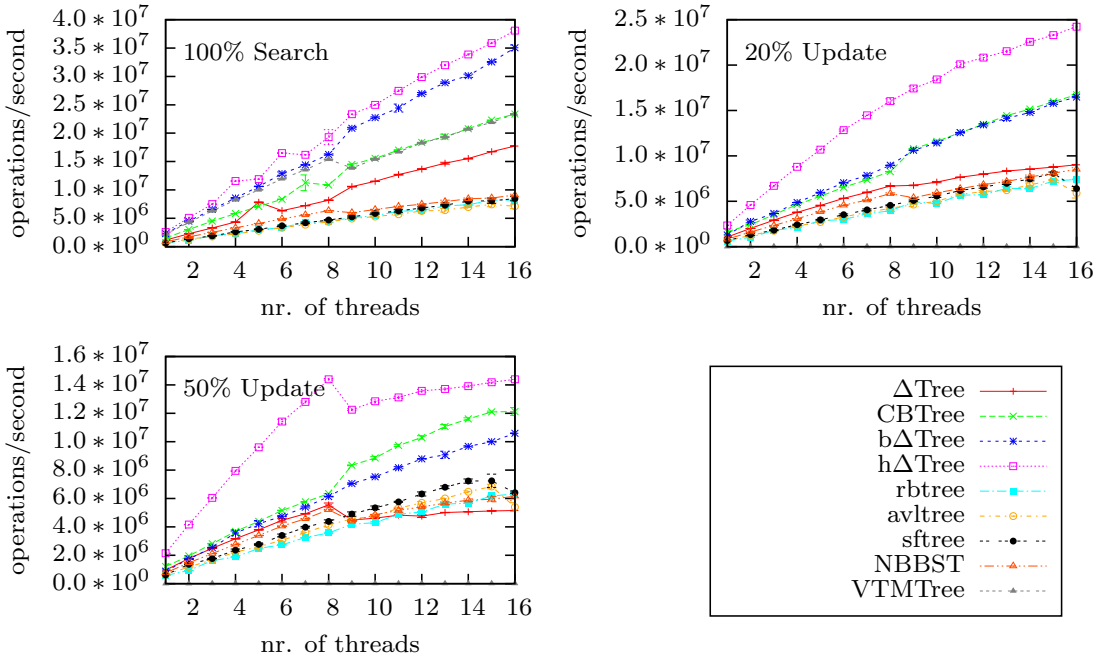


Figure 32: Performance comparison of the tested trees with 4,194,303 initial members on an Intel HPC platform with two 8-core chips. On a single chip, the heterogeneous ΔTree (hΔTree) is 140% faster than the concurrent B-tree (CBTree) in the 50%-update benchmark with 8 threads. The hΔTree performance decreases when the number of threads goes from 8 to 9 because of the cache-coherence issue between two chips (cf. Section 4.1.3.2).

The benchmarks run only with the minimum and maximum available physical cores. The total energy consumed by all CPUs and memory (in Joules) was collected and divided by the number of operations. The collected measurements did not include the initialization cost of trees.

4.1.3.2 Performance results

Among the new trees proposed in this paper, bΔTree was up to 100% faster than ΔTree for 100% search and was up to 20% faster in 50% updates. The hΔTree was up to 5% faster than bΔTree in 100% searching. However, in 20% and 50% updates, hΔTree was faster by up to 140% than the bΔTree (cf. Figure 32).

In comparison with the other trees, Figure 33 and 32 show that the heterogeneous ΔTree (hΔTree) was the fastest. The hΔTree was up to 140% faster than CBTree in the 50% update/8-thread setting on the Intel HPC platform, and was up to 100% faster on the embedded ARM platform in the 20% update. The bΔTree's and CBTree's performance was trailing behind hΔTree in all test-cases.

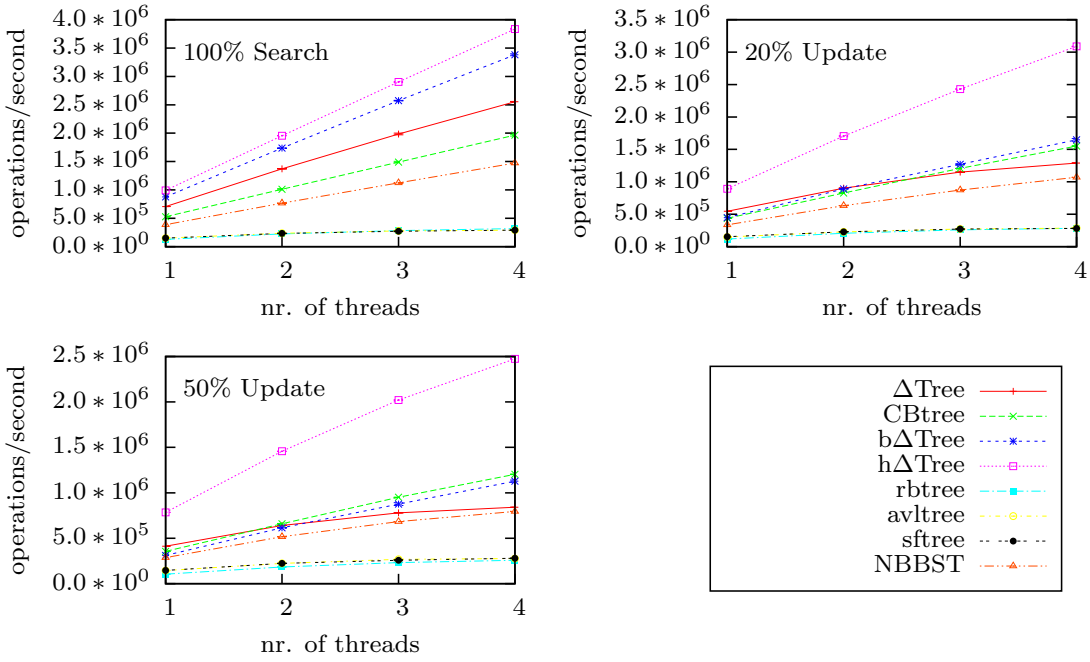


Figure 33: Performance comparison of the tested trees with 2,097,151 initial members on an embedded ARM platform. The heterogeneous ΔTree (hΔTree) is 100% faster than the concurrent B-tree (CBTree) in the 50%-update benchmark with 4 threads.

4.1.3.3 Energy consumption results

Among the new trees proposed, balanced ΔTree (bΔTree) was up to 85% more energy efficient than ΔTree (cf. Figure 34, 100% search/20-thread setting). On the other hand, heterogeneous ΔTree (hΔTree) was up to 125% more energy efficient than bΔTree (cf. Figure 34, 50% update/10-thread setting). The results indicated that the increased locality by pointer-less ΔNodes in bΔTree and heterogeneous ΔNodes with 100% more leaf nodes in hΔTree significantly lowered the energy consumption of the trees.

In comparison with the other trees (e.g., CBTree) on the HPC platform (Figure 34), hΔTree was up to 33% more energy efficient in the 100% search benchmark and 80% more energy efficient in the 50% update/10-thread setting.

Figure 35 demonstrated the energy efficiency of concurrency-aware vEB-based trees on the embedded ARM platform. The most energy efficient tree was hΔTree which was 220% more energy efficient than CBTree and NBBST in the 100% search/4-thread setting. In the 50% update/4-thread setting, hΔTree was the only one that managed to beat CBTree by 180%.

A detailed breakdown of the energy-affecting factors was conducted on the Intel HPC platform. Figure 36 shows the DRAM-only energy consumption whereas Figure 37 shows the amount of memory transfers between RAM and CPU for read-write operations.

Figure 37 shows that hΔTree was the most efficient in terms of memory transfers between

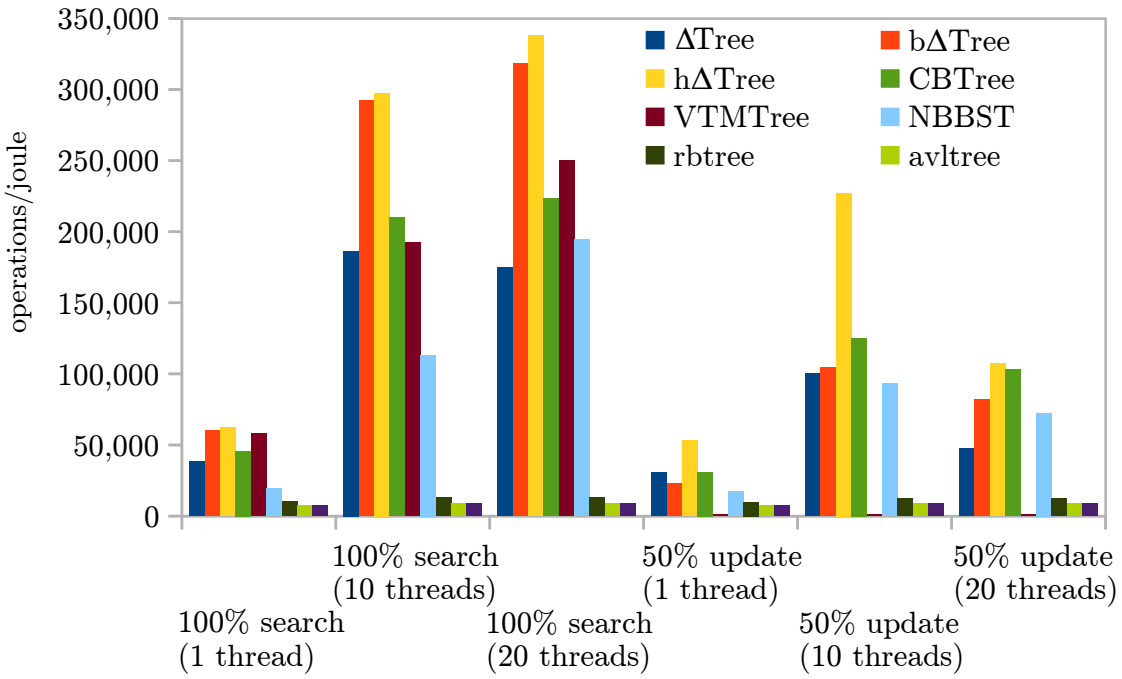


Figure 34: Energy efficiency comparison on an Intel HPC platform with two 10-core chips. On a single chip, the heterogeneous ΔTree (hΔTree) was 80% more energy efficient than the concurrent B-tree (CBTree) in the 50%-update benchmark with 10 threads. The energy efficiency decreases for some trees in the 50%-update benchmark with 20 threads (i.e., with 2 chips) because of the cache-coherence issue between two chips (cf. Section 4.1.3.2). (*Standard errors* $\leq 0.4\%$).

the RAM and CPU compared to the other trees. As a result, hΔTree had the lowest DRAM energy consumption (cf. Figure 36).

4.1.3.4 Energy and performance

The experimental results on energy efficiency (Figure 34 and 35) and on performance (Figure 32 and 33) showed that the usage of concurrency-aware vEB layouts was able to reduce the ΔTrees' energy consumption and at the same time increase their performance.

If we closely compare side-by-side all of the results on the Intel HPC platform, a strong relationship between energy efficiency, performance and data transfer can be concluded for this platform. As an example, in the 100% search benchmark where concurrency-aware vEB trees are expected to perform best, the heterogeneous ΔTree was 50% faster and 40% more energy efficient than CBTree. Figure 37 tells us that in this scenario, CBTree was transferring almost 2× more data compared to hΔTree.

However, there are other cases where the energy benefit exceeds the performance benefit, which we strongly suspect were attributed to having a locality-aware data structure. On the ARM platform, in the 100% search/4 threads benchmark, hΔTree was 220% more energy

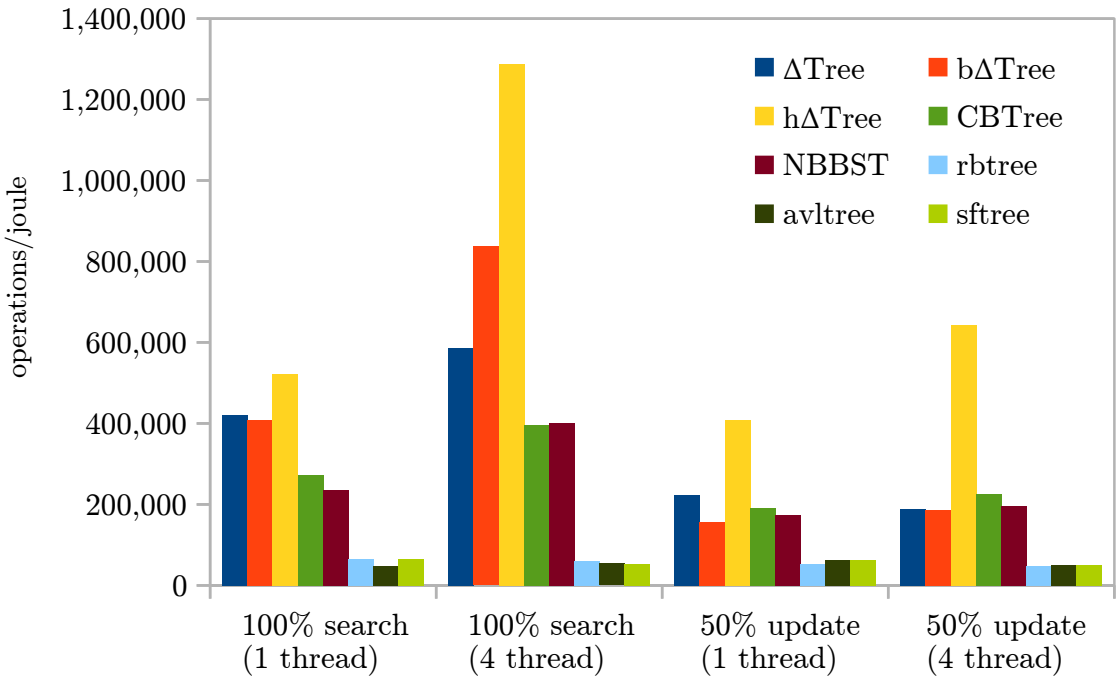


Figure 35: Energy efficiency comparison on an embedded ARM platform. The heterogeneous ΔTree (hΔTree) was 220% more energy efficient than the concurrent B-tree (CBTree) in the 100%-search benchmark with 4 threads. (*Standard errors* $\leq 0.27\%$)

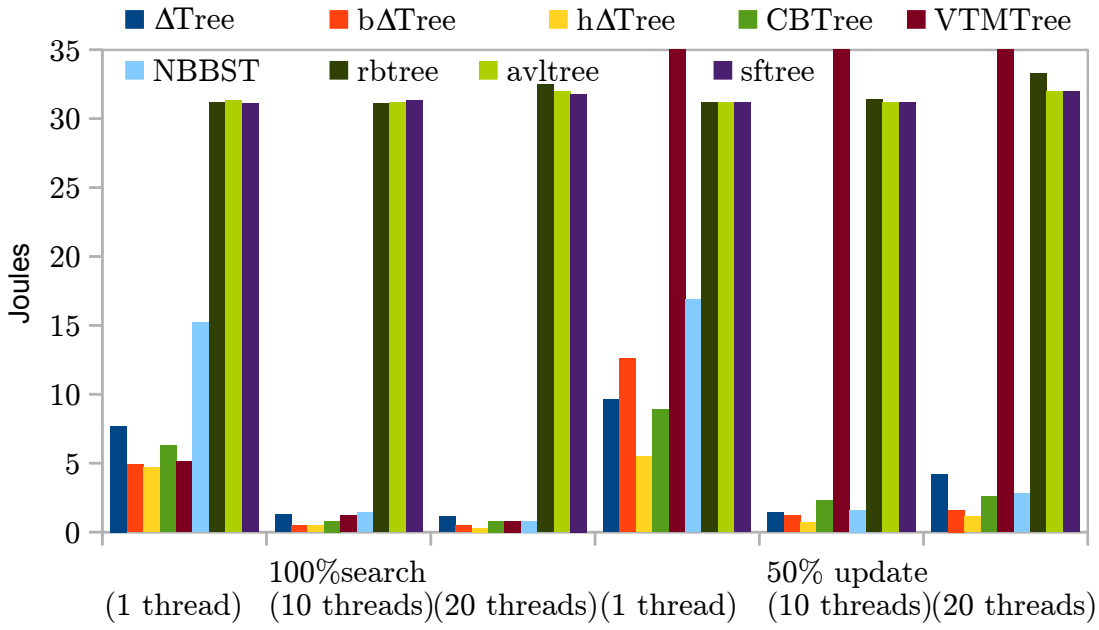


Figure 36: Memory (DRAM) energy consumption on Intel HPC platform. Measured using Intel PCM.

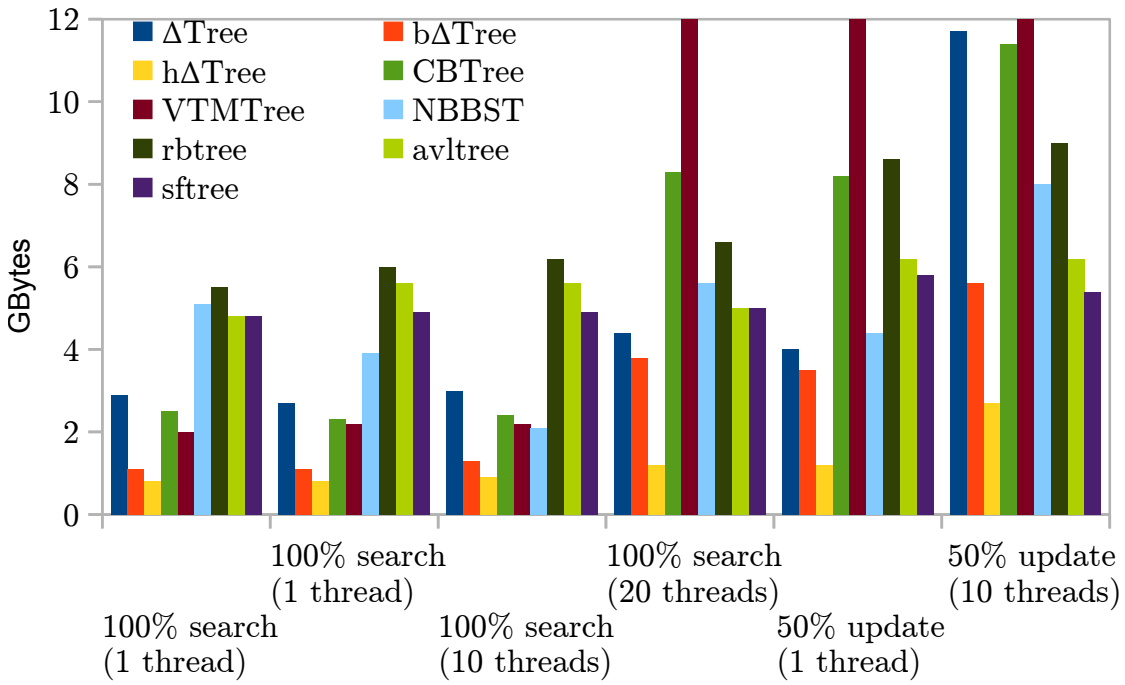


Figure 37: Amount of data transferred between RAM and CPU for Read + Write operations. Measured using Intel PCM. Data transfer goes up considerably for some trees in 50% update on 20 threads (w/ 2 CPUs) because of the cache-coherence mechanism.

efficient than CBTree while its performance was only 100% faster.

Figures 32 and 34 also show that a less than 2x performance benefit sometimes results in a 3x to 4x energy benefit for the STM-based trees on the HPC platform. The DRAM energy consumption of the STM-based trees (e.g., AVLtree, RBtree, SFtree and VTMTree) was up to 10x higher than that of the other trees (cf. Figure 36), which partly explains the phenomenon.

Finally, the above evaluation relies on the Intel CPU performance counters or the Intel PCM and ARM platform off-chip energy counters. Using more robust prediction models for performance [18] and energy [63] might enable us to obtain more insights that are useful in designing energy-efficient and highly concurrent data structures.

4.1.3.5 Multi-CPU coherency issue

Figure 32 shows some "dips" in performance for several trees after passing the 8-thread mark on the Intel HPC platform. The same behavior was reflected in energy efficiency results (cf. Figure 34). In this chart, the same "dips" were also shown in the 50% update/20 thread setting.

Our experimental analysis revealed that these "dips" were caused by the cache coherence protocol in a multi-CPU system. The Intel HPC platform consists of 2 CPUs with 8-cores each. Therefore, if a tree was exploiting data-locality in the cache of a single CPU, cache-

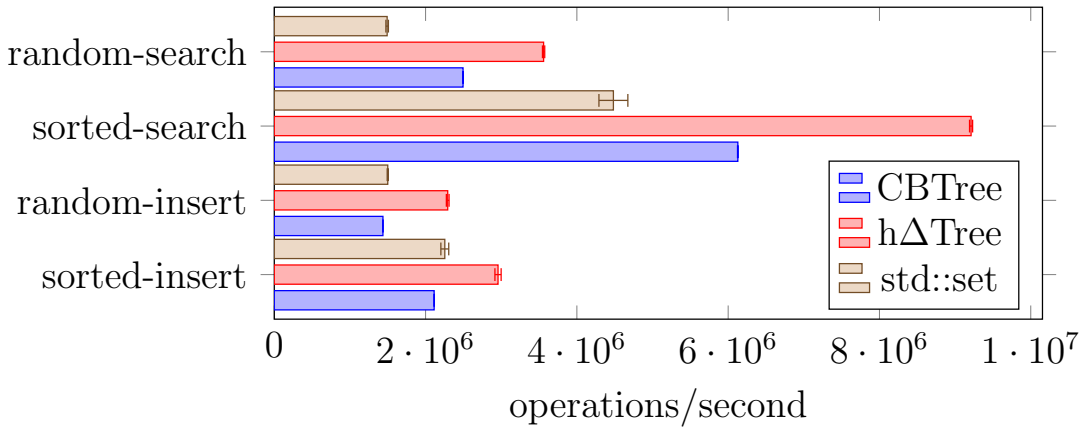


Figure 38: Random and sorted insertions/searches of 5,000,000 values on hΔTree, CBTree and std::set<int>.

coherence protocol did not involve the bus that connects CPUs and memory. Starting from 9 threads (or from 11 threads in the energy benchmarks), 2 CPUs were used and therefore the cache-coherence protocol must use the bus to transfer data. A closer look at Figure 37 explains that the dips were also related to the increase of RAM-CPU data transfers, which strengthens the cache-coherence theory. Moreover, there were no cache-coherence issues in the ARM single CPU platform. Therefore, the "dips" did not appear in any results on the ARM platform.

4.1.3.6 Comparison with GCC std::set

For ΔTree, inserting a sequence of increasing (sorted) numbers will result in a linked-list of ΔNode. The bottom-up insertions adopted by balanced ΔTree and subsequently heterogeneous ΔTree was supposed to prevent this (worst-case) from happening.

Therefore, we tested CBTree and hΔTree for random and sorted number insertions. GCC standard library std::set<int>was also included as a baseline. For the sorted inserts, starting with a blank tree, an increasing sequence of 5,000,000 numbers starting from 1 were inserted into the tree using a single thread. The same amount of random numbers was used instead for the random inserts. These tests were done on the same HPC platform described in Section 4.1.3.1. Since std::set is currently not supporting concurrent operations, a concurrent insert test was deliberately left out.

The result in Figure 38 showed that hΔTree was 38% faster than CBTree and 30% faster than std::set in the sorted inserts. For random inserts, the hΔTree was about 50% faster compared to CBTree and std::set. Further evaluation with *perf* revealed that in hΔTree, random inserts triggers more branch misses (6.82%) compared to sorted inserts (1.04%).

We also measured the time required to search all of the inserted values using the same way as they were inserted (random or sorted). In Figure 38, hΔTree was up to 105% faster in the random search compared to std::set. This lead was increased to 130% when searching the sorted sequence of inserted numbers.

The above results imply two things: 1) hΔTree performed better in both of the worst and average case of tree search/update compared to optimized B-tree and the widely used std::set; and 2) random value inserts and searches were not actually benefit the B-tree based tree. Therefore, the benchmark setups in Section 4.1.3.1 is assured to give a fair comparison of all the tested trees performance. The reader is referred to [41, 80] for more details about ΔTrees.

4.2 Concurrent Lock-Free Queues

This section exposes the quality of the prediction of the model on lock-free queues. We have used a library of well-known lock-free implementations, and have predicted their throughput and energy consumption thanks to the model explained in Section 3.3, instantiated with the process exhibited in Section 3.4. In Section 4.2.1, we first give a brief description of the queue implementations, then we show the experimental study in Section 4.2.2, that we apply both on synthetic benchmarks and a more realistic application.

4.2.1 Description of the Implementations

4.2.1.1 NOBLE

Most of the implementations that we use are part of the NOBLE library [76, 77]. The NOBLE library offers support for non-blocking multi-process synchronization in shared memory systems. NOBLE has been designed in order to: i) provide a collection of shared data objects in a form which allows them to be used by non-experts, ii) offer an orthogonal support for synchronization where the developer can change synchronization implementations with minimal changes, iii) be easy to port to different multi-processor systems, iv) be adaptable for different programming languages and v) contain efficient known implementations of its shared data objects. The library provides a collection of the most commonly used data types. The semantics of the components, which have been designed to be the very same for all implementations of a particular abstract data type, are based on the sequential semantics of common abstract data types and adopted for concurrent use. The set of operations has been limited to those which can be practically implemented using both non-blocking and lock-based techniques. Due to the concurrent nature, also new operations have been added, e.g. Update which cannot be replaced by Delete followed by Insert. Some operations also have stronger semantics than the corresponding sequential ones, e.g. traversal in a List is not invalidated due to concurrent deletes, compared to the iterator invalidation in STL. As the published algorithms for concurrent data structures often diverge from the chosen semantics, a large part of the implementation work in NOBLE, besides from adoption to the framework, also consists of considerable changes and extensions to meet the expected semantics.

The various lock-free concurrent queue algorithms that we include in this study are briefly described below.

4.2.1.2 Tsigas-Zhang (TZ)

Tsigas and Zhang [79] presented a lock-free extension of [58] for any number of threads where synchronization is done both on the array elements and the shared head and tail indices using `CAS`[5], and the ABA problem is avoided by exploiting two (or more) null values. We recall that the ABA problem is due to the inability of `CAS` to detect concurrent changes of a memory word from a value (A) to something else (B) and then again back to the first value (A). A *CAS* operation can not detect if a variable was read to be A and then later changed to B and then back to A by some concurrent processes. The *CAS* primitive will perform the update even though this might not be intended by the algorithm's designer. In [79] synchronization is done both directly on the array elements and the shared head and tail indices using `CAS`, thus supporting multiple producers and consumers. Moreover, for lowering the memory contention the algorithm alternates every other operation between scanning and updating the shared head and tail indices.

4.2.1.3 Valois (Val)

Valois [82, 83] makes use of linked list in his lock-free implementation which is based on the `CAS` primitive. He was the first to present a lock-free implementation of a linked-list. The list uses auxiliary memory cells between adjacent pairs of ordinary memory cells. The auxiliary memory cells were introduced to provide an extra level of indirection so that normal memory cells can be removed by joining the auxiliary ones that are adjacent to them. His design also provides explicit cursors to access memory cells in the list directly and insert or delete nodes on the places the the cursors point to.

4.2.1.4 Michael-Scott (MS)

Michael and Scott [65] presented a lock-free queue that is more efficient, synchronizing via the shared head and tail pointers as well as via the next pointer of the last node. Synchronization is done via shared pointers indicating the current head and tail node as well via the next pointer of the last node, all updated using `CAS`. The tail pointer is then moved to point to the new item, with the use of a `CAS` operation. This second step can be performed by the thread invoking the operation, or by another thread that needs to help the original thread to finish before it can continue. This helping behavior is an important part of what makes the queue lock-free, as a thread never has to wait for another thread to finish. The queue is fully dynamic as more nodes are allocated as needed when new items are added. The original presentation used unbounded version counters, and therefore required double-width `CAS` which is not supported on all contemporary platforms. The problem with the version counters can easily be avoided by using some memory management scheme as e.g. [64].

[5]The Compare-And-Swap (CAS) atomic primitive will update a given memory word, if and only if the word still matches a given value (e.g. the one previously read). CAS is generally available in contemporary systems with shared memory, supported mostly directly by hardware and in other cases in combination with system software.

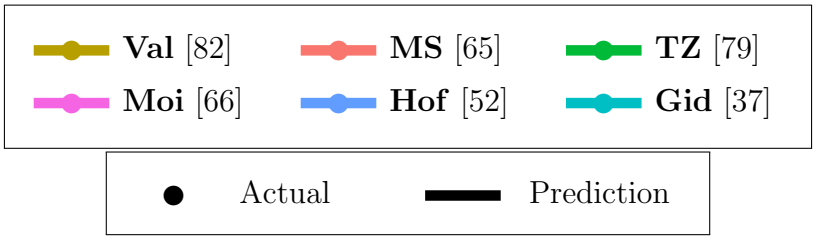


Figure 39: Key legend of the graphs

4.2.1.5 Moir *et al.* (Moi)

Moir *et al.* [66] presented an extension of the Michael and Scott [65] lock-free queue algorithm where elimination is used as a back-off strategy to increase scalability when contention on the queue's head or tail is noticed via failed `CAS` attempts. However, elimination is only possible when the queue is close to empty during the operation's invocation.

4.2.1.6 Hoffman-Shalev-Shavit (Hof)

Hoffman *et al.* [52] takes another approach in their design in order to increase scalability by allowing concurrent `Enqueue` operations to insert the new node at adjacent positions in the linked list if contention is noticed during the attempted insert at the very end of the linked list. To enable these "baskets" of concurrently inserted nodes, removed nodes are logically deleted before the actual removal from the linked list, and as the algorithm traverses through the linked list it requires stronger memory management than [64], such as [36] or [50] and a strategy to avoid long chains of logically deleted nodes.

4.2.1.7 Gidenstam-Sundell-Tsigas (Gid)

Gidenstam *et al.* [37] combines the efficiency of using arrays and the dynamic capacity of using linked lists, by providing a lock-free queue based on linked lists of arrays, all updated using `CAS` in a cache-aware manner. In resemblance to [58, 35, 79] this algorithm uses arrays to store (pointers to) the items, and in resemblance to [79] it uses `CAS` and two null values. Moreover, shared indices [35] are avoided and scanning [79] is preferred as much as possible. In contrast to [58, 35, 79] the array is not static or cyclic, but instead more arrays are dynamically allocated as needed when new items are added, making the queue fully dynamic.

4.2.2 Experiments: Predictions and Measurements

The legend depicted in Figure 39 lists the implementations that are compared, and will be used throughout this section.

4.2.2.1 Synthetic Benchmark

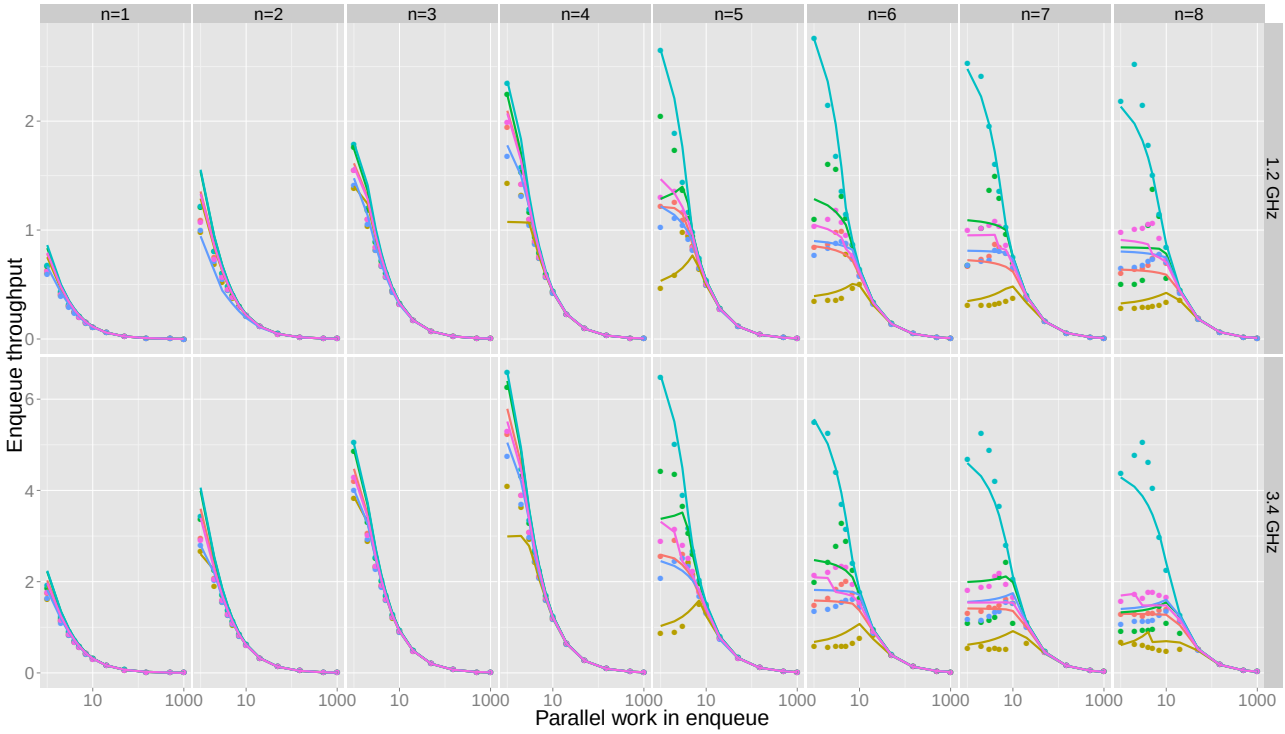
Throughput The throughput predictions are plotted in Figure 40 for the enqueuers, and in Figure 41 for the dequeuers. Points are measurements, while lines are predictions. We will follow this rule for all comparisons between prediction and measurement. In the actual execution, the queue goes through a transient state when the amount of work in the parallel section is near the critical point, but the prediction is not so far from the actual measurements, as illustrated in Figure 40. Under intra-contention, some of the curves get flat, since only one thread can be succeeding at the same time, according to the definition of the retry loop. Some curves even decrease because the successful one is stalled by other failing ones due to serialization of the atomic primitives, namely expansion. The slope presumably indicates the density of atomic primitives in retry loops which depends on the algorithm.

The comparison of Figures 40a and 40b illustrates the impact of inter-contention. A decrease of the highest point of $\mathcal{T}_e$, due to an increase of cw_e, can be observed for the more inter-contended case. When cw_e increases, some critical points shift slightly towards the right as the intra-contention starts with a larger pw_e. In Figure 41, decomposition of $\mathcal{T}_d$ is apparent. When enqueue rate is low, *i.e.* when pw_e is high, $\mathcal{T}_d$ is ruled by $\mathcal{T}_d^{(+)}$ due to majority of NULL dequeues, and it tends towards $\mathcal{T}_d^{(-)}$ when the enqueue rate increases.

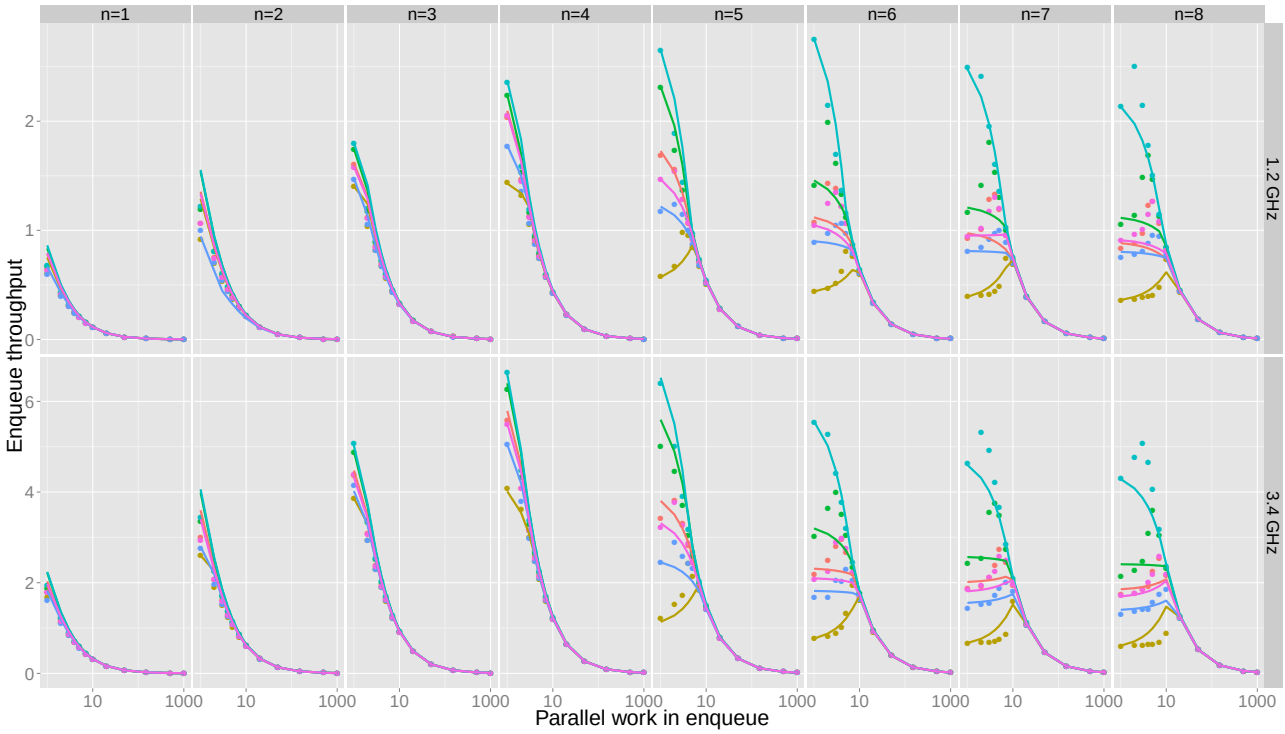
Power The prediction and measurements, regarding power, are plotted in Figures 42, 43a, 43b and 44, where we observe that the most significant differences lie in the dynamic memory power. The differences in CPU power are almost invisible, since the dynamic power of the parallel sections (composed of *pause*s instructions) is very close to the dynamic power of the retry loops. We remark some steps in the measured memory power, but we prefer to keep a continuous estimate.

As the retry loop, which is particular to each implementation, is mainly composed of memory operations, the main difference between the various implementations in terms of power occurs in the dynamic memory power, which we represent in Figure 42a (legend is in Figure 39). Overall, the prediction reacts correctly to the variations of parallel section sizes, and some specifics of the algorithms are caught, *e.g.* **Hof** detached from the others when $pw_e = 50$ or **Gid** mostly well-predicted both absolutely and relatively as the less power-dissipating implementation. One can observe once again the asymmetry between enqueue and dequeue operations by comparing the power values at $(pw_d, pw_e) = (2, 1000)$ and $(1000, 2)$; this asymmetry is predicted by the model, with a lower impact though.

Energy per Operation In Figure 45 is represented the energy per operation. Overall we observe that the successful operations (dequeue of a non-NULL item) are cheaper and cheaper when the number of threads is increasing on the same socket: the cost of turning the machine on is made profitable by an increase in performance. However, under high-contention, the lack of performance improvement while increasing the number of cores makes the use of supplementary cores useless. The inefficiency of adding cores is even more apparent when cores are spread across the sockets. In this case, under high-contention, performance



(a) $pw_d = 7$



(b) $pw_d = 50$

Figure 40: Enqueue throughput

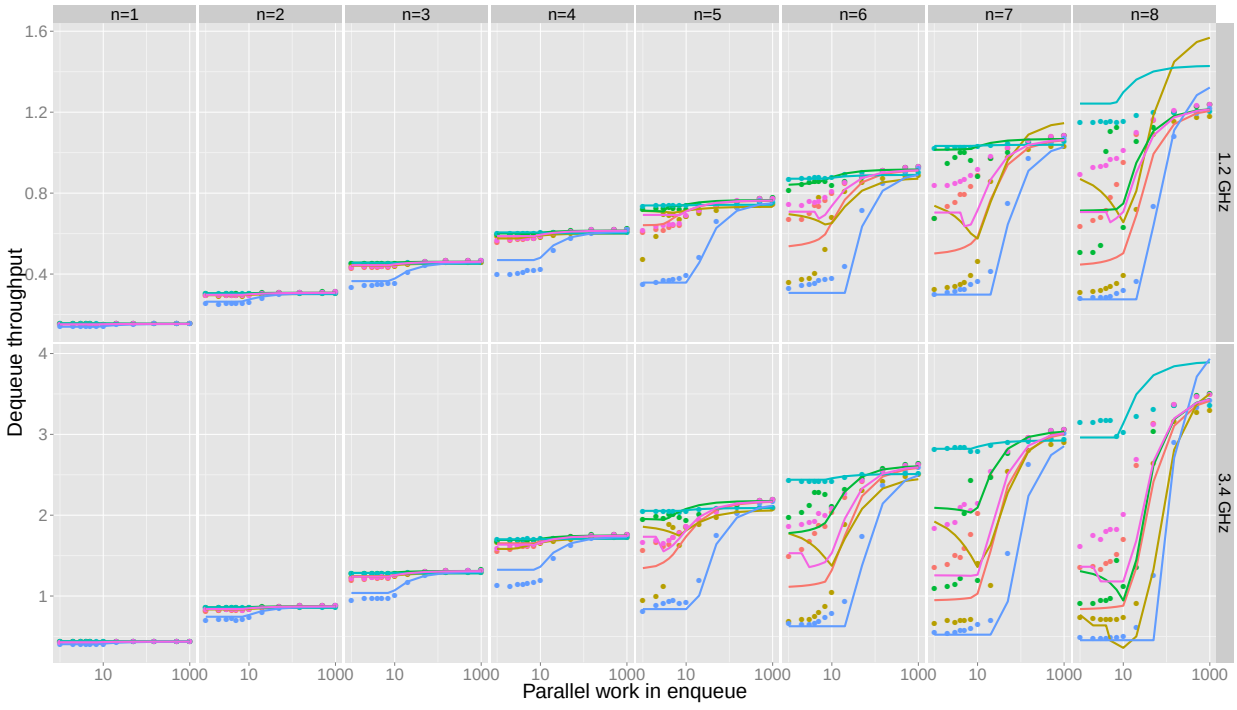


Figure 41: Dequeue throughput with $pw_d = 7$

could even be degraded by the implication of new cores, then, as performance decreases and power increases, the energy per operation dramatically increases.

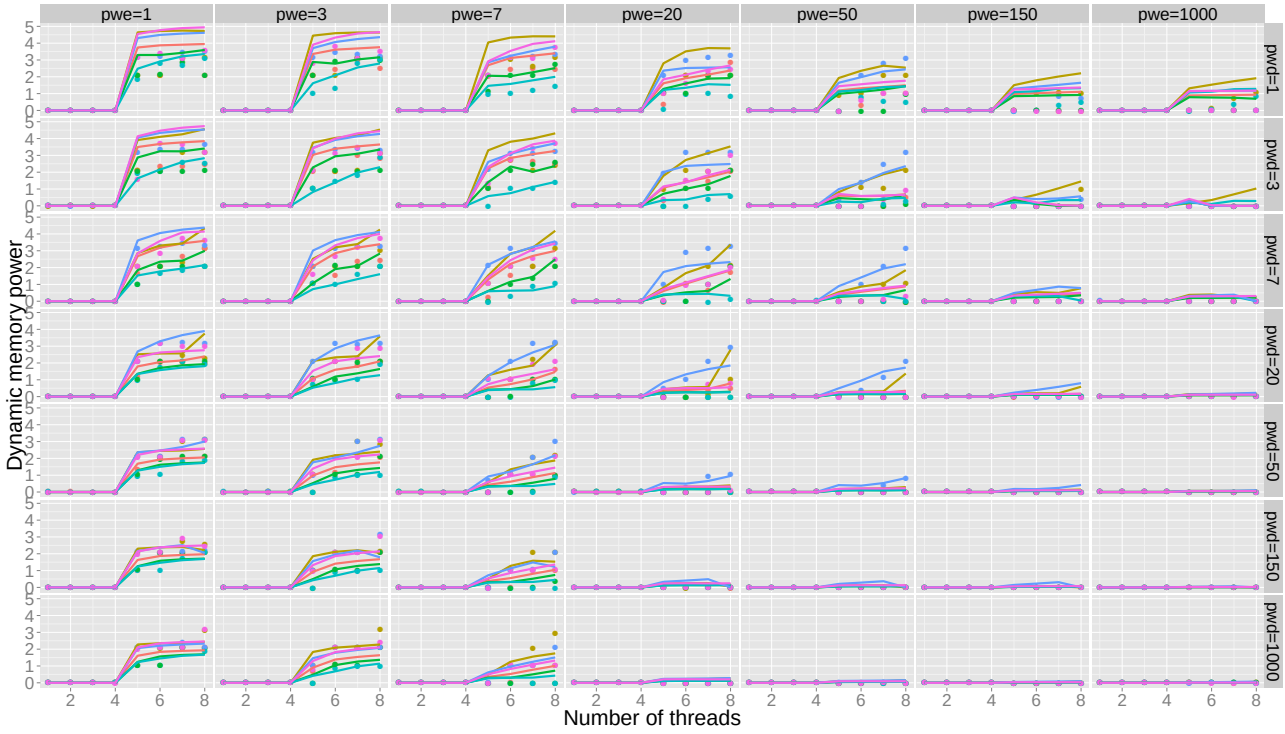
4.2.2.2 Towards Realistic Applications: Mandelbrot Set Computation

The performance and energy behavior of an application using a lock-free queue depends on both the application specific code and the implementation of the data structure. For applications where the queue is used in a steady state manner, predictions can be made using the model instantiated with the synthetic benchmark, combined with information about the behavior of the application specific code. What is needed is:

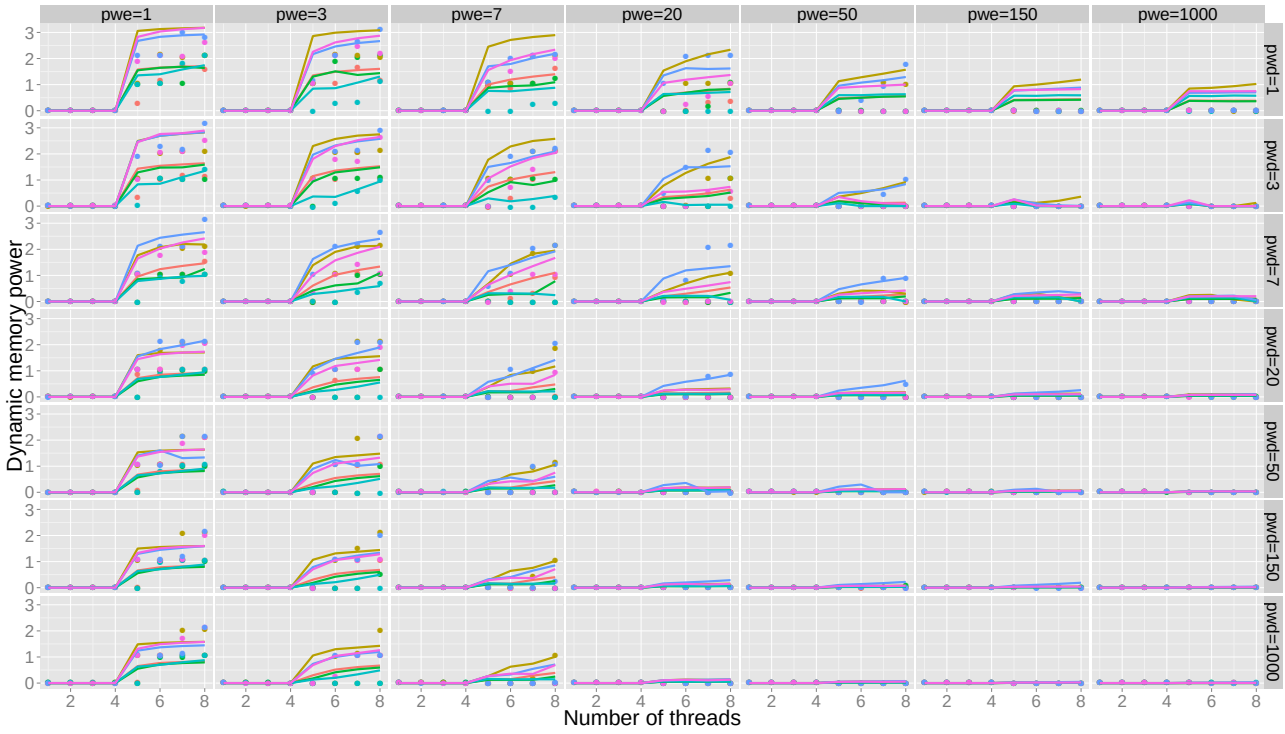
- The size of the parallel work part of the application, both for enqueuers and dequeuers. These may be distributions rather than single values.
- The dynamic power for these parts (as it may differ from that of the parallel work in the synthetic benchmark).

Description of Mandelbrot Set Application As a case-study we have used an existing application[6] that computes and renders an 8192×8192 pixel image of the Mandelbrot

[6]Previously used for evaluation in [75].

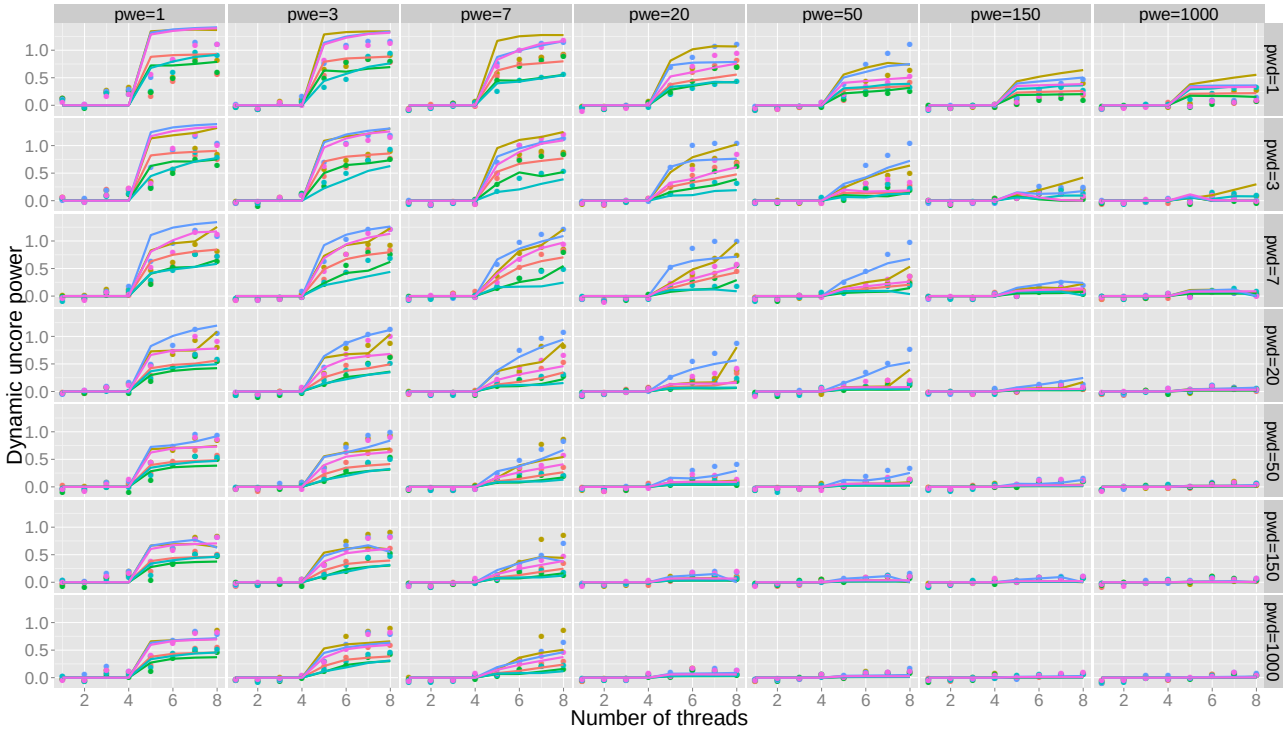


(a) $f = 3.4\,\mathrm{GHz}$

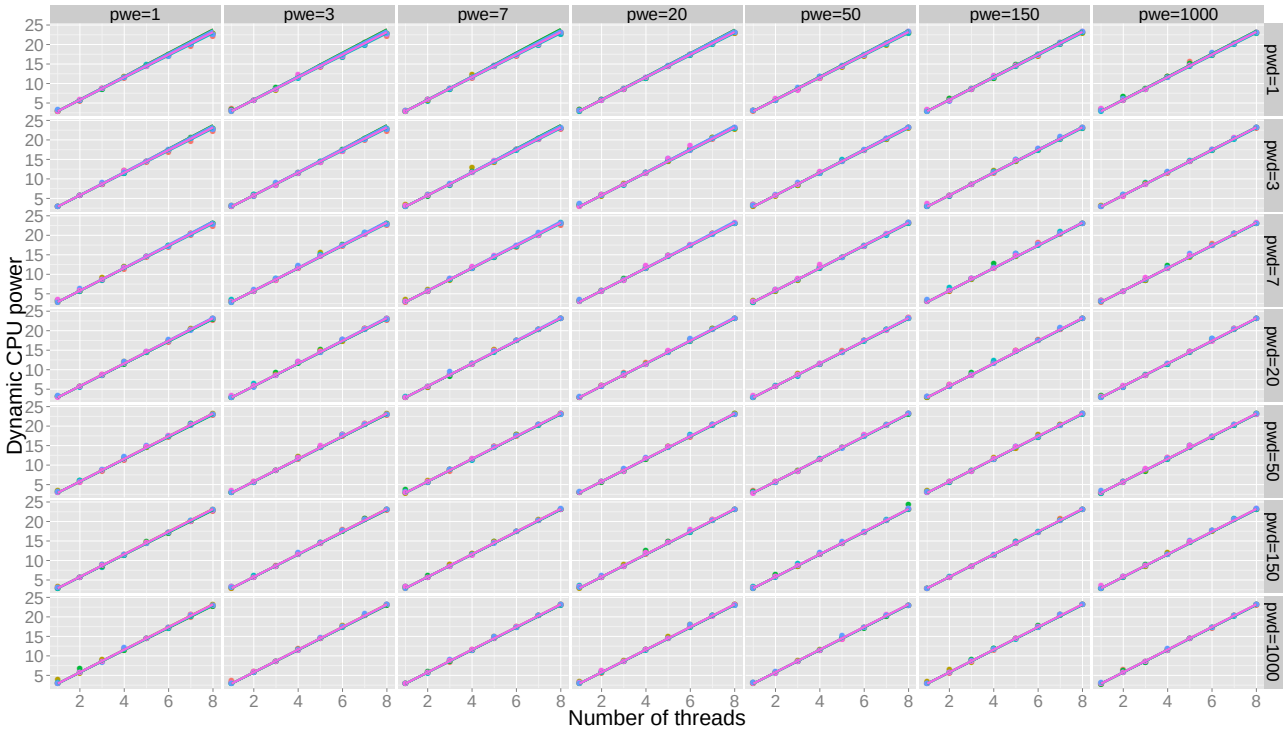


(b) $f = 1.2\,\mathrm{GHz}$

Figure 42: Dynamic memory power



(a) Dynamic uncore power at $f = 3.4\,\mathrm{GHz}$



(b) Dynamic CPU power at $f = 1.2\,\mathrm{GHz}$

Figure 43: Dynamic uncore and CPU power

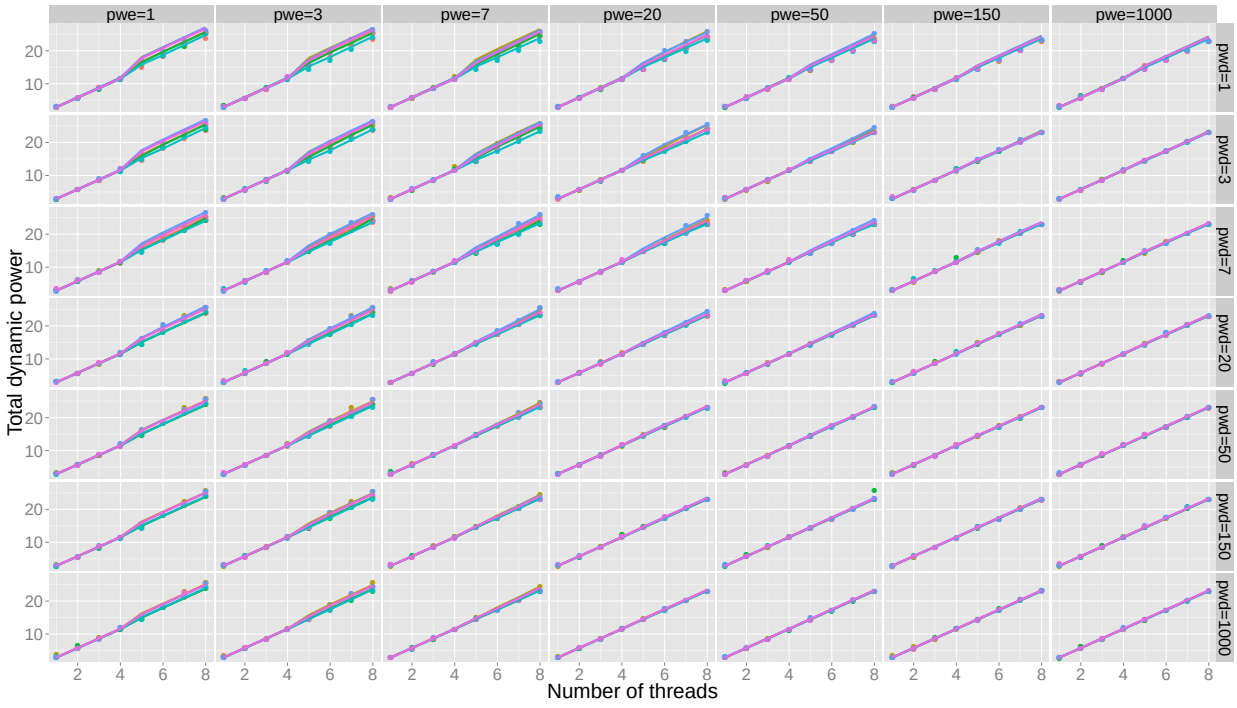


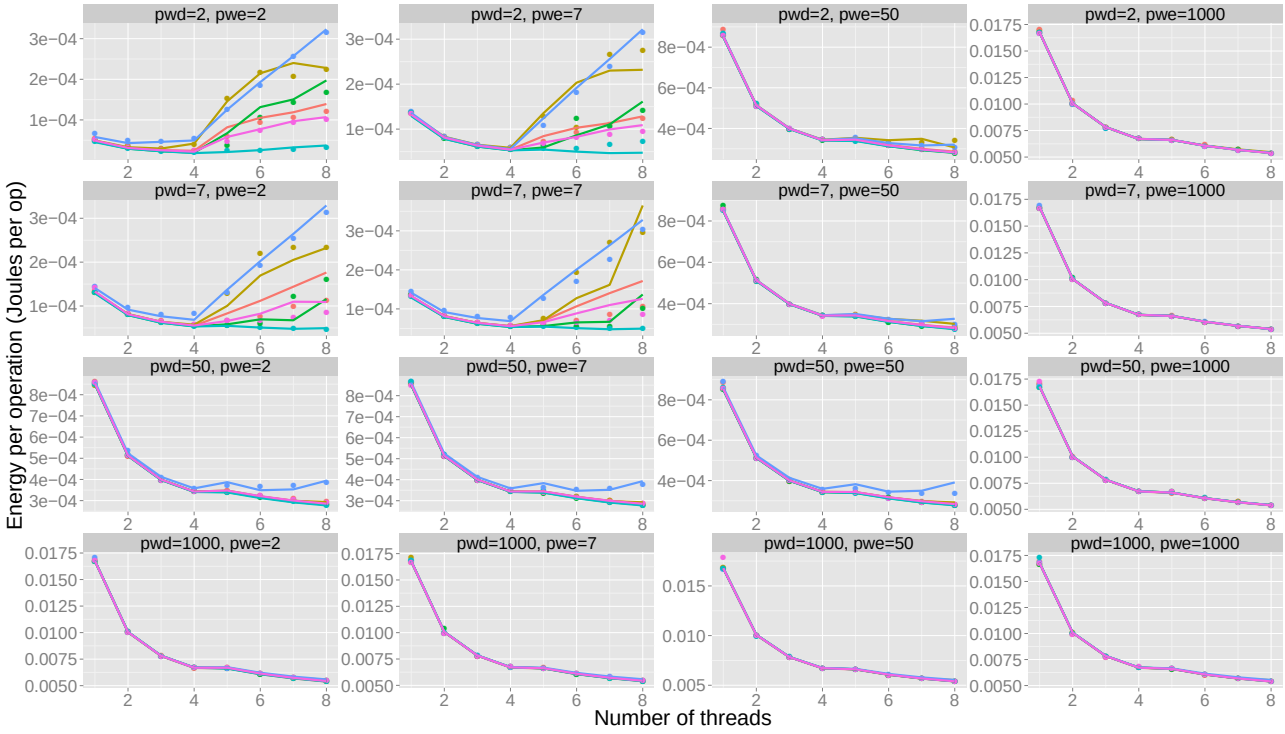
Figure 44: Sum of Dynamic Powers at $f = 1.2\,\mathrm{GHz}$

set [62] in parallel using the producer/consumer pattern. The program uses a concurrent queue to communicate between two major phases:

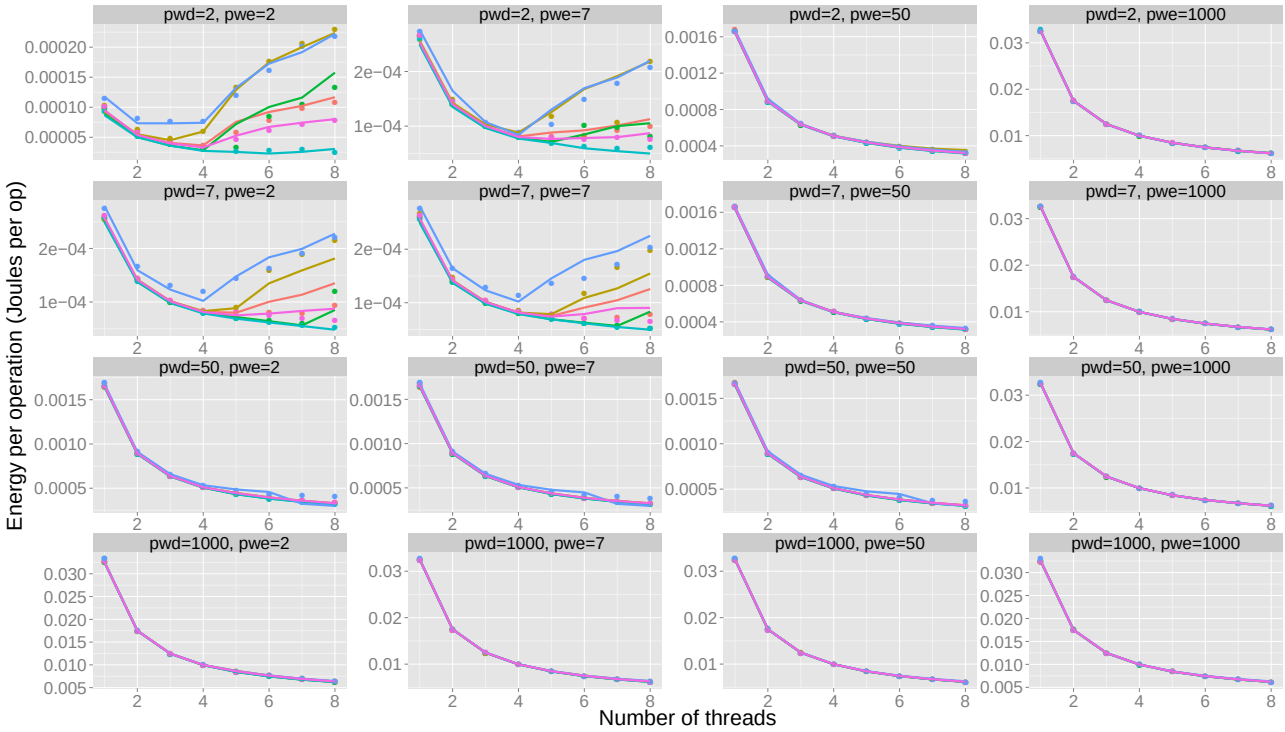
- Phase 1 consists of computing the number (with a maximum of 255) of iterations for a given set of points within a chosen region of the image. The results for each region together with its coordinates are then enqueued.
- Phase 2 consists of, for each region dequeued from the queue, computing the RGB values for each contained point and draw these pixels to the resulting image. The colors for the corresponding number of iterations are chosen according to a rainbow scheme, where low numbers are rendered within the red and high numbers are rendered within the violet spectrum.

Half of the threads perform phase 1 and the rest perform phase 2. The size of each square region is chosen to be one of 16×16, 8×8, 4×4, or 2×2 pixels which also determines the amount of work to perform per queue operation and, hence, the level of contention. Similarly to the synthetic benchmark, the application uses a dense pinning strategy, pinning producer/consumer pairs to consecutive pairs of cores.

This is just one of many possible ways to divide the work and pin threads, it remains as future work to explore other ways.



(a) $f = 3.4\,\mathrm{GHz}$



(b) $f = 1.2\,\mathrm{GHz}$

Figure 45: Energy per operation

Mandelbrot Prediction There are two main differences between the Mandelbrot application and the synthetic benchmark: (i) the instructions in the parallel section differ; and (ii) the size of the parallel section for producers varies in Mandelbrot.

Firstly, we need to measure the CPU power dissipation for Mandelbrot; we cannot expect to be able to predict the power dissipation of any application that uses a queue without having any knowledge about the power characteristics of the application. In contrast, memory power dissipation for the computation intensive Mandelbrot parallel section is negligible in comparison to queue operations; hence, the dynamic memory power that we have measured and extrapolated in the synthetic benchmark is unchanged.

Secondly, Mandelbrot provides a variety of producer parallel works. To deal with this, the pixel region is decomposed row-wise in an interleaved manner among producer threads. This decomposition leads to long enough execution intervals in which the parallel sections of the producer threads are similar and constant. This is due to the computationally expensive pixels belonging to the Mandelbrot set being concentrated together in the center of the domain and surrounded by cheaper pixels which diverge quickly. This characteristic is congruent with our model where the data structure is used in a steady state manner. Thus, predictions can be made using the instantiated model over a linear combination of execution intervals.

We measure the latency of the computation intensive producer and consumer parallel works for each frequency and contention level (2×2, 4×4, 16×16). For this process, we make use of CPUID, RDTSC and RDTSCP instructions as specified in [70]. The distribution of parallel works reveals that there are two main groups for producers, that corresponds to regions belonging to the Mandelbrot set or not. Concerning 2×2 contention, due to the wide distribution, we gather the parallel works into bins of width 10 pauses; the number of elements in the i^{th} bin is then denoted by $size^{(i)}$ and its average amount of work by $pw_e^{(i)}$. We scale the width of bins linearly with the area of the region for other contention levels. For the consumers, parallel works are similar for the whole execution.

To make predictions, we assume that all consumer/producer pair $(pw_d, pw_e^{(i)})$ is executed in a steady state during an interval of time. For each frequency, thread, algorithm and contention of interest, we obtain the throughput $\mathcal{T}^{(i)} = \mathcal{T}(pw_d, pw_e^{(i)})$ and the powers $P_i^{(X)} = P^{(X)}(pw_d, pw_e^{(i)})$ for this interval from the corresponding synthetic benchmark input. The only part of the model, instantiated with the synthetic benchmark that needs to be replaced by an application specific entry, is the dynamic CPU power parameter. Then, we combine intervals to obtain total execution time and average power dissipation. This accumulation strategy should be applied with care as the synthetic benchmark is based upon the steady state assumption. An interval which is assumed to take place with a mostly empty queue, could actually not be in this state due to leftover items from the previous interval. Although our model is capable of taking this initial state into consideration and provide metrics accordingly, we assume that each interval is independent. This approximation is reasonable since the consumer parallel work corresponds to the producer bin with one of smallest values, hence a mostly empty queue.

Note that we have implemented a constant back-off equivalent to the consumer parallel

work, after dequeuing a NULL item instead of retrying immediately, because of several advantages. It cannot decrease the performance, since either the queue is growing, and then the back-off never takes place, or the queue is mostly empty, and then the producers are the bottleneck of the queue. Conversely, it can increase the performance by diminishing the queue contention. Those motivations drove the design of the synthetic benchmark, that we can accordingly reuse here.

For each frequency, thread, algorithm and contention configuration, execution time and power estimates for Mandelbrot application are obtained with the following equations:

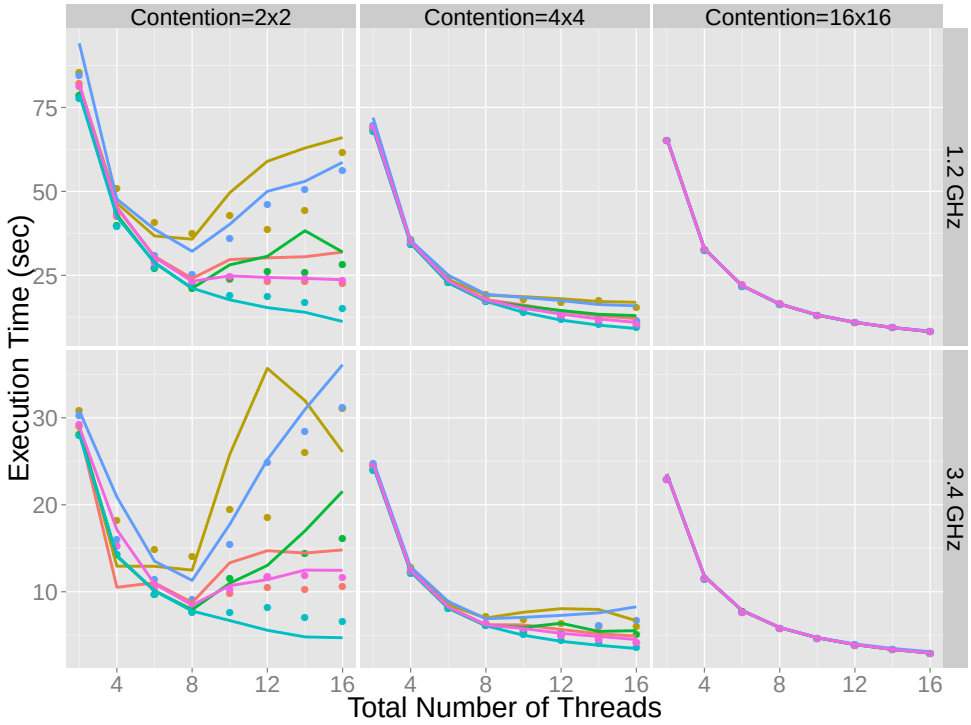
$$Time_{total} = \sum_{i=1}^{BinCount} size^{(i)} \times \frac{\lambda}{\mathcal{T}^{(i)}}$$

$$P^{(X)} = \frac{\sum_{i=1}^{BinCount} (size^{(i)} \times \frac{\lambda}{\mathcal{T}^{(i)}}) \times P_i^{(X)}}{Time_{total}}$$

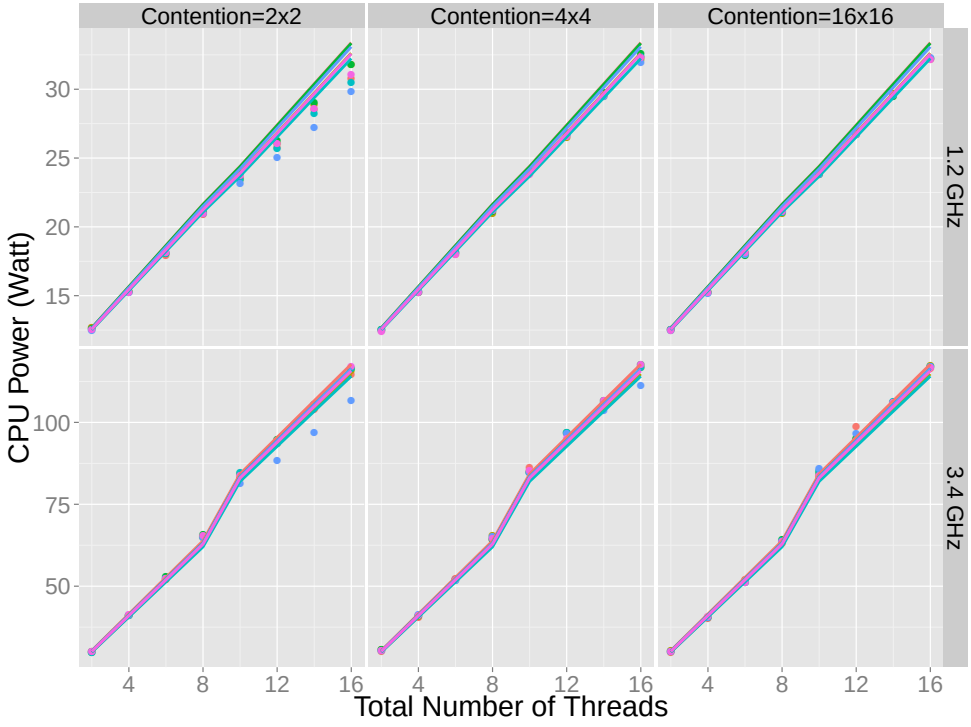
In Figure 47a, execution time estimates catch the queue algorithm specific trend for high contention cases, which exhibit a more complicated behavior than the low contention cases. Also, they reveal the impact of different queue implementations to overall application performance, which does not appear under low contention. For the highest contention level with region size 2×2, an increasing trend in execution time is observed after 8 threads for many algorithms. The reason is the increasing latency of atomic synchronization primitives originating from two main sources: (i) inter-socket communication, which starts after 8 threads due to our pinning strategy, and (ii) the increasing serialization (expansion) probability for atomic primitives due to increasing number of threads that interfere in the retry loop. The ratio of atomic primitives and the size of queue operations show variations between algorithms which in turn leads to different behaviors. For the 4×4 contention case, the difference between algorithms can still be observed but the parallel sections are large enough to avoid interference in the retry loop. Therefore, execution time decreases with the increasing number of threads. The difference between algorithms is due to different queue operation sizes which loses its significance gradually with the decreasing contention level, as observed in low contention cases.

Power estimates are quite satisfactory except algorithm **Hof** which is overestimated. In the power versus time plot which is not presented here, we observe a step like decrease in power at the end of the execution, implying that **Hof** is prone to unfairness among producers. Some producers finish their regions early and go to sleep which decreases the power dissipation.

As mentioned before, dynamic memory and uncore power are dominated by the queue implementations so we do not use any application specific memory/uncore power samples in our estimations, due to compute intensive character of the Mandelbrot parallel works. Even if this was not the case, memory/uncore power in the parallel sections could have been extracted. One can get the memory/uncore power measurement from the application and subtract the memory/uncore power that we have measured and extrapolated in the

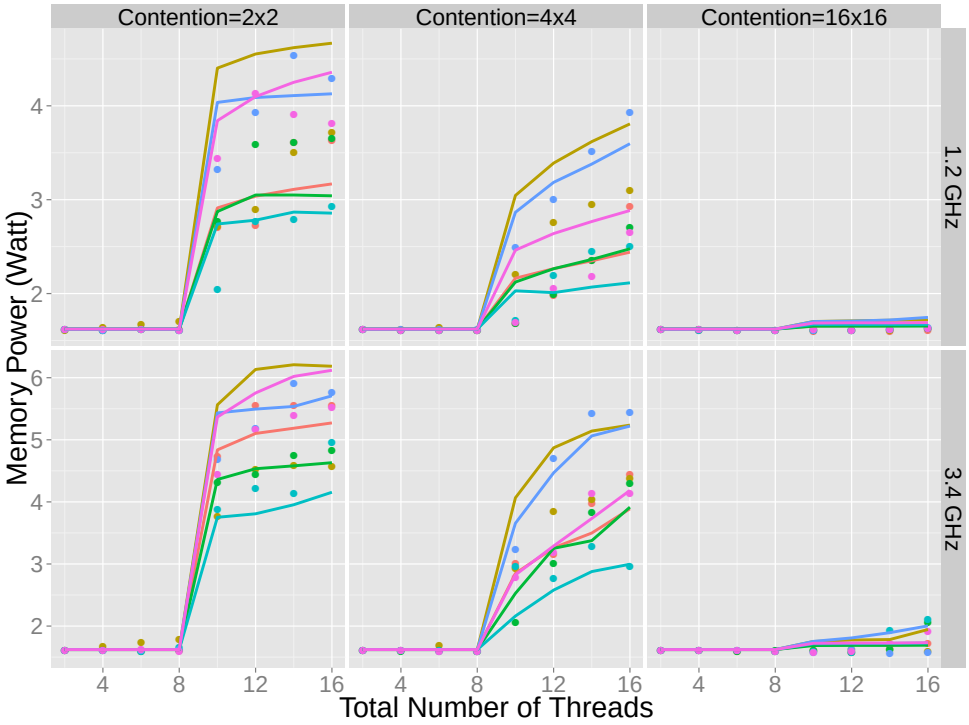


(a) Execution time

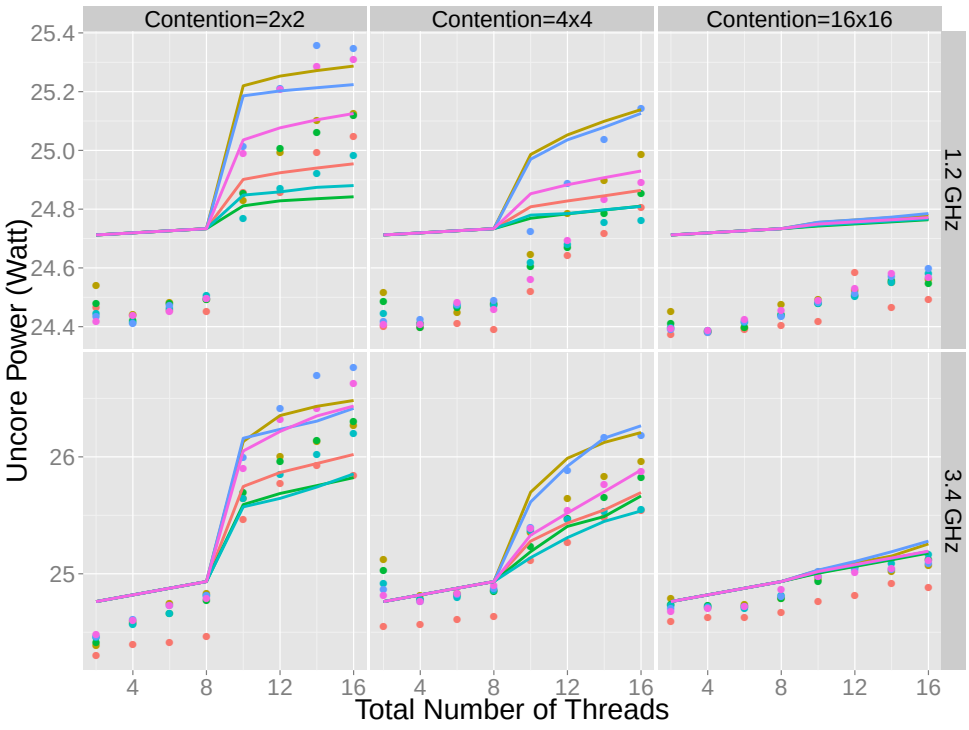


(b) CPU power

Figure 46: Mandelbrot results (1/2)



(a) Memory power



(b) Uncore power

Figure 47: Mandelbrot results (2/2)

synthetic benchmark. Then, using the ratio of retry loops and parallel sections thanks to our throughput model, the memory/uncore power can be estimated.

Similar to the synthetic benchmarks, Mandelbrot dynamic memory/uncore power becomes noticeable with the inter-socket communication, after 8 threads, and decreases gradually with the decreasing ratio of retry loops, with contention level.

5 Conclusions

In this work, we have presented our current results on the white-box methodology and the first prototype of libraries and programming abstractions as follows.

- We have devised a new *relaxed* cache oblivious model that are appropriate for developing energy-efficient concurrent data structures and algorithms.
- We have improved the power model for Myriad1 which is able to predict the power consumed by a program running on a specific number of cores. Given a certain platform and the computation intensity, the model can predict the power consumed by a computational algorithm, then helps to answers the question how many cores are required to run a program to achieve the optimized energy consumption. The model considers both platform and algorithm properties which give more insight on how to design the algorithm to achieve better energy optimization. The model is also validated the model with a set of micro-benchmarks and real applications such as sparse/dense linear algebra kernels and the graph computation algorithm.
- We have continued the work done in D2.1 [44] on the modeling of queue implementations. We have also generalized the model to offer more freedom according to the workers calling the data structure (parallel section sizes of enqueuers and dequeuers are decoupled).
- We have developed a libraries package of concurrent search trees that contains several state-of-the-art concurrent search trees such as the non-blocking binary search tree, the Software Transactional Memory (STM) based red-black tree, AVL tree, and speculation-friendly tree, the fast concurrent B-tree, the static cache-oblivious binary search tree and a family of novel locality-aware and energy efficient concurrent search trees, namely the DeltaTree, the Balanced DeltaTree, and Heterogeneous DeltaTree. The DeltaTrees are platform-independent and up to 140% faster and 220% more energy efficient than the state-of-the-art on commodity HPC and embedded platforms.
- We have implemented a set of queue implementations (Michael and Scott [65], Valois [82], Tsigas and Zhang [79], Gidenstam *et al.* [37], Hoffman *et al.* [52], Moir *et al.* [66]), and automatized the process of estimating the performance and the power consumption and integrated it in the EXCESS software.

These results are the starting point for our further research on providing energy-efficient libraries and algorithms.

In the next steps of this work, WP2 will continue the works of Task 2.2. The next tasks (Task 2.3 and Task 2.4) will continue to develop novel concurrent data structures and novel adaptive memory access algorithms that can control data movement (i.e. do not rely on general cache system), exploiting EXCESS platforms and anticipated hardware technology (e.g. exposed energy cost and configurable memory hierarchy). We will develop novel

concurrent data structures that can dynamically exploit and combine different data layouts (e.g. van Emde Boas layout, dynamic non-canonical layouts) and different kinds of memory (e.g., memory with different energy-efficiency and performance) to achieve optimal trade-offs between performance and energy consumption. Moreover, novel adaptive memory access algorithms that adapt to power-down mechanisms and dynamic speed scaling controlled by run-time systems will also be developed. The algorithms will have ability to adjust themselves to the monitoring information at run-time. The novel data structures and algorithms will constitute libraries for inter-process communication and data sharing on EXCESS platforms.

References

[1] Adlink Technology Inc. *USB-1900 Series 16-bit 250kS/s USB 2.0-based High-performance DAQ Module USB-1901/1902/1903 User's Manual*, 2011.

[2] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E. Tarjan. Cbtree: a practical concurrent self-adjusting search tree. In *Proc. 26th international Conf. Distributed Computing*, DISC'12, pages 1–15, 2012.

[3] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[4] A. Andersson. Faster deterministic sorting and searching in linear space. In *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on*, pages 135–141, Oct 1996.

[5] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 268–276, New York, NY, USA, 2002. ACM.

[6] 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.

[7] R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[8] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[9] Michael Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. *SIAM Journal on Computing*, 35:341, 2005.

[10] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In *Proc. 19th annual ACM Symp. Parallel algorithms and architectures*, SPAA '07, pages 81–92, 2007.

[11] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious b-trees. In *Proc. 17th annual ACM Symp. Parallelism in algorithms and architectures*, SPAA '05, pages 228–237, 2005.

[12] Anastasia Braginsky and Erez Petrank. A lock-free b+tree. In *Proc. 24th ACM Symp. Parallelism in algorithms and architectures*, SPAA '12, pages 58–67, 2012.

[13] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. 13th annual ACM-SIAM Symp. Discrete algorithms*, SODA '02, pages 39–48, 2002.

[14] GerthStølting Brodal. Cache-oblivious algorithms and data structures. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory - SWAT 2004*, volume 3111 of *Lecture Notes in Computer Science*, pages 3–13. Springer Berlin Heidelberg, 2004.

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

[16] Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In *Proc. 15th international Conf. Principles of Distributed Systems*, OPODIS'11, pages 207–221, 2011.

[17] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, August 2000.

[18] Lydia Y. Chen, Giuseppe Serazzi, Danilo Ansaloni, Evgenia Smirni, and Walter Binder. What to expect when you are consolidating: Effective prediction models of application performance on multicores. *Cluster Computing*, 17(1):19–37, March 2014.

[19] Jee Choi, Marat Dukhan, Xing Liu, and Richard Vuduc. Algorithmic time, energy, and power on candidate hpc compute building blocks. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th Int.*, pages 447–457, May 2014.

[20] 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. IEEE Computer Society.

[21] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[22] 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.

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

[24] Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, PPoPP '12, pages 161–170, 2012.

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

[26] H. David, E. Gorbatov, Ulf R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, pages 189–194, August 2010.

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

[28] Erik D. Demaine and Erik D." Demaine. Cache-oblivious algorithms and data structures. *IN LECTURE NOTES FROM THE EEF SUMMER SCHOOL ON MASSIVE DATA SETS*, 2002.

[29] Electronic Educational Devices. *Watts up Operators manual*, 2003.

[30] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proc. 20th international Conf. Distributed Computing*, DISC'06, pages 194–208, 2006.

[31] Jack Dongarra and Pete Beckman. The international exascale software roadmap. *International Journal of High Performance Computing Applications (IJHPCA)*, 25(1):3–60, 2011.

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

[33] Rolf Fagerberg. Cache-oblivious model. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*, pages 1–99. Springer US, 2008.

[34] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symp. Foundations of Computer Science*, FOCS '99, page 285, 1999.

[35] 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 (PPoPP '08)*, pages 43–52, New York, NY, USA, 2008. ACM.

[36] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1173–1187, 2009.

[37] Anders Gidenstam, Håkan Sundell, and Philippas Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In *Proceedings of the 14th International Conference on Principle of Distributed Systems (OPODIS 2010)*, volume 6490 of *Lecture Notes in Computer Science*, pages 302–317. Springer, 2010.

[38] Goetz Graefe. A survey of b-tree locking techniques. *ACM Trans. Database Syst.*, 35(3):16:1–16:26, July 2010.

[39] Goetz Graefe. Modern b-tree techniques. *Found. Trends databases*, 3(4):203–402, April 2011.

[40] Vincent Gramoli. Synchrobench: A benchmark to compare synchronization techniques for multicore programming. `https://github.com/gramoli/synchrobench`.

[41] 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 D2.1, EU FP7 project EXCESS, 2014. http://www.excess-project.eu.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

[50] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking Memory Management Support for Dynamic-sized Data Structures. *ACM Transactions on Computer Systems*, 23:146–196, May 2005.

[51] 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.

[52] Moshe Hoffman, Ori Shalev, and Nir Shavit. The baskets queue. In Eduardo Tovar, Philippas Tsigas, and Hacène Fouchal, editors, *Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS 2007)*, volume 4878 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2007.

[53] 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.

[54] Christoph Kessler, Lu Li, Usman Dastgeer, Rosandra Cuello, Oskar Sjstrm, Phuong Ha Hoai, and Vi Tran. Excess deliverable d1.3 - energy-tuneable domain-specific language/library for linear system solving. Technical report, EU FP7 project EXCESS, 2015.

[55] Christoph Kessler, Lu Li, Usman Dastgeer, Anders Gidenstam, and Aras Atalar. D1.2 Initial specification of energy, platform and component modelling framework. Technical Report FP7-611183 D1.2, EU FP7 Project EXCESS, August 2014.

[56] Christoph Kessler, Lu Li, Usman Dastgeer, Philippas Tsigas, Anders Gidenstam, Paul Renaud-Goud, Ivan Walulya, Aras Atalar, David Moloney, Phuong Ha Hoai, and Vi Tran. D1.1 Early validation of system-wide energy compositionality and affecting factors on the EXCESS platforms. Project Deliverable, EU FP7 project Execution Models for Energy-Efficient Computing Systems (EXCESS), www.excess-project.eu, April 2014.

[57] 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 *Proc. 2010 ACM SIGMOD Intl. Conf. Management of data*, SIGMOD '10, pages 339–350, 2010.

[58] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[59] 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.

[60] 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.

[61] 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.

[62] Benoit B. Mandelbrot. Fractal aspects of the iteration of $z \to \lambda z(1-z)$ for complex λ and z. *Annals of the New York Academy of Sciences*, 357:249–259, 1980.

[63] John C. McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta. Evaluating the effectiveness of model-based power characterization. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, 2011.

[64] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(8), August 2004.

[65] 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.

[66] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the 17th annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–262, 2005.

[67] David Moloney and Brendan Barry. Excess deliverable d4.1 - alpha release of energy aware platform simulator. Technical report, EU FP7 project EXCESS, 2014.

[68] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparrs Cabezas, Daniele G. Spampinato, and Markus Püschel. Applying the roofline model. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76 – 85, 2014.

[69] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Comput. Surv.*, 46(4):47:1–47:31, March 2014.

[70] Gabriele Paoloni. How to benchmark code execution times on Intel® ia-32 and ia-64 instruction set architectures. Technical Report 324264-001, Intel, September 2010.

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

[72] 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. *Proc. VLDB Endowment*, 4(11):795–806, 2011.

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

[74] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985.

[75] Håkan Sundell, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. A lock-free algorithm for concurrent bags. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2011.

[76] Håkan Sundell and Philippas Tsigas. NOBLE: A Non-Blocking Inter-Process Communication Library. In *Proceedings of the Workshop on Languages, Compilers and Run-time Systems for Scalable Computers (LCR)*, Lecture Notes in Computer Science, 2002.

[77] Håkan Sundell and Philippas Tsigas. NOBLE: Non-blocking programming support via lock-free shared abstract data types. *SIGARCH Computer Architecture News*, 36(5), 2008.

[78] Salam Traboulsi (ed.) and Yosandra Sandoval (ed.). D5.1 Report on specification of evaluation criteria. Technical Report FP7-611183 D5.1, EU FP7 Project EXCESS, February 2014.

[79] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proceedings of the 13th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 134–143, 2001.

[80] 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. `http://www.cs.uit.no/~ibrahim/DeltaTree-TR.pdf`.

[81] Ibrahim Umar, Otto Anshus, and Phuong Ha. Deltatree: A practical locality-aware concurrent search tree. techreport 2013-74, University of Tromso, 2013. arXiv:1312.2628.

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

[83] J. D. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, 1995.

[84] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proc. 16th Annual Symp. Foundations of Computer Science*, SFCS '75, pages 75–84, 1975.

[85] Vincent M. Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. Measuring energy and power with papi. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ICPPW '12, pages 262–268, Washington, DC, USA, 2012. IEEE Computer Society.

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

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

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)