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

D2.4
Report on the final prototype of programming abstractions for energy-efficient inter-process communication

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

SEVENTH FRAMEWORK PROGRAMME

Date of preparation (latest version): 31.08.2016

The opinions of the authors expressed in this document do not necessarily reflect the official opinion of EXCESS partners or of the European Commission.

DOCUMENT INFORMATION

Deliverable Number	D2.4
Deliverable Name	Report on the final prototype of programming abstractions for energy-efficient inter-process communication
Authors	Phuong Ha Vi Tran Ibrahim Umar Aras Atalar Anders Gidenstam Paul Renaud-Goud Philippas Tsigas Ivan Walulya
Responsible Author	Phuong Ha e-mail: `phuong.hoai.ha@uit.no` Phone: +47 776 44032
Keywords	High Performance Computing; Energy Efficiency
WP/Task	WP2/Task 2.1, 2.2, 2.3, 2.4
Nature	R
Dissemination Level	PU
Planned Date	31.08.2016
Final Version Date	31.08.2016
Reviewed by	
MGT Board Approval	

DOCUMENT HISTORY

Partner	**Date**	**Comment**	**Version**
UiT (P.Ha, V.Tran)	01.07.2016	Deliverable skeleton	0.1
Chalmers (A. Atalar)	19.07.2016	Input - energy model and energy evaluation	0.2
Chalmers (I. Walulya)	20.07.2016	Input - implementation of streaming aggregation	0.2
UiT (V.Tran)	03.08.2016	Input - energy and power model	0.3
Chalmers (I. Walulya)	22.08.2016	Input revise	0.4
UiT (I.Umar, V.Tran)	29.08.2016	Input revise	0.5

Abstract

Work package 2 (WP2) aims to develop libraries for energy-efficient inter-process communication and data sharing on the EXCESS platforms. The Deliverable D2.4 reports on the final prototype of programming abstractions for energy-efficient inter-process communication. Section 1 is the updated overview of the prototype of programming abstraction and devised power/energy models. The Section 2-6 contain the latest results of the four studies:

- GreenBST, a energy-efficient and concurrent search tree (cf. Section 2)
- Customization methodology for implementation of streaming aggregation in embedded systems (cf. Section 3)
- Energy Model on CPU for Lock-free Data-structures in Dynamic Environments (cf. Section 4.10)
- A General and Validated Energy Complexity Model for Multithreaded Algorithms (cf. Section 5)

Executive Summary

Work package 2 (WP2) investigate and model the trade-offs between energy consumption and performance of data structures and algorithms for inter-process communication. WP2 also provides concurrent data structures and algorithms that support energy-efficient massive parallelism while minimizing inter-component communication.

The main achievements of Deliverable D2.4 are summarized as follows.

- We have described the cache-oblivious abstraction that is used in developing our energy-efficient and concurrent data structures. We also present in the same section a detailed description of GreenBST, an energy-efficient concurrent search tree that was briefly described in D2.3. Also in this deliverable, GreenBST is tested with new state-of-the-art concurrent search trees that are not included in D2.3. The latest experimental results showed that GreenBST is more energy efficient and has higher throughput for both the concurrent search- and update- intensive workloads than the state-of-the-art. We also have implemented GreenBST for Myriad2 platform and have conducted an experimental evaluation using the implementation.

- We present a methodology for the customization of streaming aggregation implemented in modern low power embedded devices. The methodology is based on design space exploration and provides a set of customized implementations that can be used by developers to perform trade-offs between throughput, latency, memory and energy consumption. We compare the proposed embedded system implementations of the streaming aggregation operator with the corresponding HPC and GPGPU implementations in terms of performance per watt. Our results show that the implementations based on low power embedded systems provide up to 54 and 14 times higher performance per watt than the corresponding Intel Xeon and Radeon HD 6450 implementations, respectively.

- We present an energy model on CPU for lock-free data-structures in dynamic environments. Lock-free data structures are based on retry loops and are called by application-specific routines. In D2.3, we illustrate the performance impacting factors and the model that we use to cover a subset of the lock-free structures that we consider here. In the former study, the analysis is built upon properties that arise only when the sizes of the retry loops and the application-specific work are constant. In this work, we introduce two new frameworks that can be used to the capture the performance of a wider set of lock-free data structures (*i.e.* the size of retry loops follow a probability distribution) in dynamic environments (*i.e.* the size of application specific follows a probability distribution). These analyses allow us to estimate the energy consumption of an extensive set of lock-free data structures that are used under various access patterns.

- We introduces a new general energy model ICE for analyzing the energy complexity of a wide range of multi-threaded algorithms. Compared to the EPEM model reported in

D2.3, this model proposed using Ideal Cache memory model to compute I/O complexity of the algorithms. Besides a case study of SpMV to demonstrate how to apply the ICE model to find energy complexity of parallel algorithms, Deliverable D2.4 also reports a case study to apply the ICE model to Dense Matrix Multiplication (matmul). The model is then validated with both data-intensive (i.e., SpMV) and computation-intensive (i.e., matmul) algorithms according to three aspects: different algorithms, different input types/sizes and different platforms. In order to make the reading flow easy to follow, we include in this report a complete study of ICE model along with latest results.

Contents

1 Introduction

D2.4 reports the final prototype of programming abstraction based on the results from Task 2.1 to 2.4, including: i) the latest results of Task 2.1 on investigating and modeling the trade-off between energy and performance of concurrent data structures and algorithms [69] ii) the improved results of Task 2.2 on providing essential concurrent data structures and algorithms for inter-process communication [72] and iii) the additional results of Task 2.3 on developing novel concurrent data structures and Task 2.4 on memory-access algorithms that are locality- and heterogeneity-aware [74]. The detailed studies (including their motivation, contributions and current results) of D2.4 are introduced in the followings subsections.

1.1 Energy-efficient and Concurrent Data Structures and Algorithms

Like other fundamental abstractions for energy-efficient computing, search trees need to support both high concurrency and fine-grained data locality. However, existing locality-aware search trees such as ones based on the van Emde Boas layout (vEB-based trees), poorly support *concurrent* (update) operations while existing highly-concurrent search trees such as the non-blocking binary search trees do not consider data locality.

We present GreenBST, a practical energy-efficient concurrent search tree that supports fine-grained data locality as vEB-based trees do, but unlike vEB-based trees, GreenBST supports high concurrency. GreenBST is a k-ary leaf-oriented tree of GNodes where each GNode is a fixed size tree-container with the van Emde Boas layout. As a result, GreenBST minimizes data transfer between memory levels while supporting highly concurrent (update) operations. Our experimental evaluation using the recent implementation of non-blocking binary search trees, highly concurrent B-trees, conventional vEB trees, as well as the portably scalable concurrent trees shows that GreenBST is efficient: its energy efficiency (in operations/Joule) and throughput (in operations/second) are up to 65% and 69% higher, respectively, than the other trees on a high performance computing (HPC) platform (Intel Xeon), an embedded platform (ARM), and an accelerator platform (Intel Xeon Phi). The results also provide insights into how to develop energy-efficient data structures in general.

1.2 Customization methodology for implementation of streaming aggregation in embedded systems

Streaming aggregation is a fundamental operation in the area of stream processing and its implementation provides various challenges. Data flow management is traditionally performed by high performance computing systems. However, nowadays there is a trend of implementing streaming operators in low power embedded devices, due to the fact that they often provide increased performance per watt in comparison with traditional high performance systems. In this work, we present a methodology for the customization of streaming aggregation implemented in modern low power embedded devices. The methodology is based on design space exploration and provides a set of customized implementations that can be used by developers to perform trade-offs between throughput, latency, memory and energy consumption. We compare the proposed embedded system implementations of the streaming aggregation operator with the corresponding HPC and GPGPU implementations in terms of performance per watt. Our results show that the implementations based on low power embedded systems provide up to 54 and 14 times higher performance per watt than the corresponding Intel Xeon and Radeon HD 6450 implementations, respectively.

1.3 Energy Model on CPU for Lock-free Data-structures in Dynamic Environments

In this section, we firstly consider the modeling and the analysis of the performance of lock-free data structures. Then, we combine the perfomance analysis with our power model that is introduced in D2.1 [75] and D2.3 [73] to estimate the energy efficiency of lock-free data structures that are used in various settings.

Lock-free data structures are based on retry loops and are called by application-specific routines. In contrast to the model and analysis provided in D2.3, we consider here the lock-free data structures in dynamic environments. The size of each of the retry loops, and the size of the application routines invoked in between, are not constant but may change dynamically.

We present two analytical frameworks for calculating the performance of lock-free data structures. The new frameworks follow two different approaches. The first framework, the simplest one, is based on queuing theory. It introduces an average-based approach that facilitates a more coarse-grained analysis, with the benefit of being ignorant of size distributions. Because of this independence from the distribution nature it covers a set of complicated designs. The second approach, instantiated with an exponential distribution for the size of the application routines, uses Markov chains, and is tighter because it constructs stochastically the execution, step by step.

Both frameworks provide a performance estimate which is close to what we observe in practice. We have validated our analysis on (i) several fundamental lock-free data structures such as stacks, queues, deques and counters, some of them employing dynamic helping mechanisms, and (ii) synthetic tests covering a wide range of possible lock-free designs. We show the applicability of our results by introducing new back-off mechanisms, tested in application contexts, and by designing an efficient memory management scheme that typical lock-free algorithms can utilize. Finally, we reveal how these results can be used to obtain the energy consumption of the lock-free data structures.

1.4 A General and Validated Energy Complexity Model for Multithreaded Algorithms

Like time complexity models that have significantly contributed to the analysis and development of fast algorithms, energy complexity models for parallel algorithms are desired as crucial means to develop energy efficient algorithms for ubiquitous multicore platforms. Ideal energy complexity models should be validated on real multicore platforms and applicable to a wide range of parallel algorithms. However, existing energy complexity models for parallel algorithms are either theoretical without model validation or algorithm-specific without ability to analyze energy complexity for a wide-range of parallel algorithms.

This paper presents a new general validated energy complexity model for parallel (multithreaded) algorithms. The new model abstracts away possible multicore platforms by their static and dynamic energy of computational operations and data access, and derives the energy complexity of a given algorithm from its *work*, *span* and *I/O* complexity. The new model is validated by different sparse matrix vector multiplication (SpMV) algorithms and dense matrix multiplication (matmul) algorithms running on high performance computing (HPC) platforms (e.g., Intel Xeon and Xeon Phi). The new energy complexity model is able to characterize and compare the energy consumption of SpMV and matmul kernels according to three aspects: different algorithms, different input matrix types and different platforms. The prediction of the new model regarding which algorithm consumes more energy with different inputs on different platforms, is confirmed by the experimental results. In order to improve the usability and accuracy of the new model for a wide range of platforms, the platform parameters of ICE model are provided for eleven platforms including HPC, accelerator and embedded platforms.

2 Libraries of Energy-efficient and Concurrent Data Structures

In this section, we describe the cache-oblivious abstraction that is used in developing our energy-efficient and concurrent data structures. The inclusion of the cache-oblivious abstraction that is previously described in the D2.2 is intended to help the readers to fully understand the methodology that is used for promoting energy-efficiency in data structures (cf. Section 2.1). The section continues with the detailed description of GreenBST, an energy-efficient concurrent search tree (cf. Section 2.2 and 2.3). In contrast to the D2.3, GreenBST in this deliverable is presented with more details, emphasizing on its complete structure and concurrency control. The section concludes with the experimental results of the developed libraries of concurrent data structure (cf. Section 2.4). We add several state-of-the-art trees that are not included in D2.3 in the energy efficiency and throughput comparison of the concurrent data structure libraries.

2.1 Cache-oblivious Abstraction

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 [48] 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 [48].

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 [58], 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 [15] 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 [48]. 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 2.1.2) 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 [58] are found to be cache-efficient and disk-efficient [35, 52], 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 [35, 52, 56]. Among those algorithms are scanning algorithms (e.g., traversals, aggregates, and array reversals), divide and conquer algorithms (e.g., median selection, and matrix multiplication), and sorting algorithms (e.g., mergesort and funnel-sort [58]). 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.

2.1.1 I/O model.

The I/O[1] model was introduced by Aggarwal and Vitter [15]. 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.

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

For this I/O model, B-tree [28] is an optimal search tree [46]. B-trees and its concurrent variants [33, 44, 65, 66] 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 [91, 118], Intel engineers 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. 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 [34]. This layout has been proved to be optimal for search using the cache-oblivious model [58].

2.1.2 Cache-oblivious model

The cache-oblivious model was introduced by Frigo et al. in [58], 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 [35], 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 [29, 30, 31] and Brodal, [34], are optimal. We would like to refer the readers to [35, 58] for a more comprehensive overview of the I/O model and cache-oblivious model.

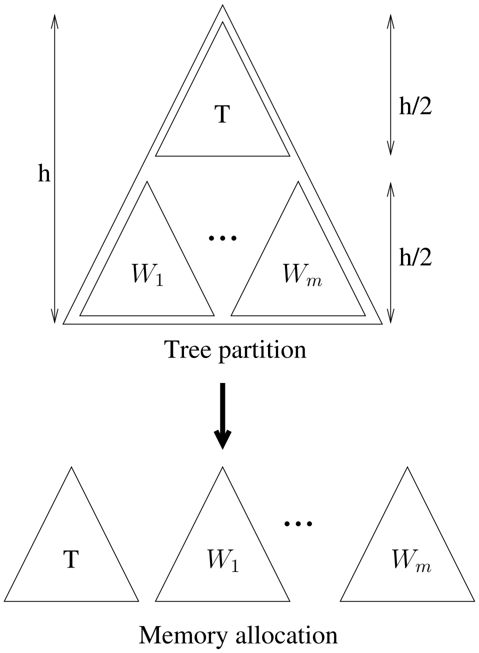


Figure 1: Static van Emde Boas (vEB) layout: a tree of height h is recursively split at height $h/2$. The top subtree T of height $h/2$ and $m = 2^{h/2}$ bottom subtrees $W_1; W_2; \ldots ; W_m$ of height $h/2$ are located in contiguous memory locations where T is located before $W_1; W_2; \ldots ; W_m$.

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

2.1.3 Cache-oblivious Algorithms

2.1.3.1 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 [35, 52].

2.1.3.2 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 [58]. As for square matrices (e.g., $N \times N$), using the Strassen's algorithm and the cache-oblivious model, the required I/O bound has been proved to be $O(N^2/B + N^{\lg 7}/B\sqrt{M})$.

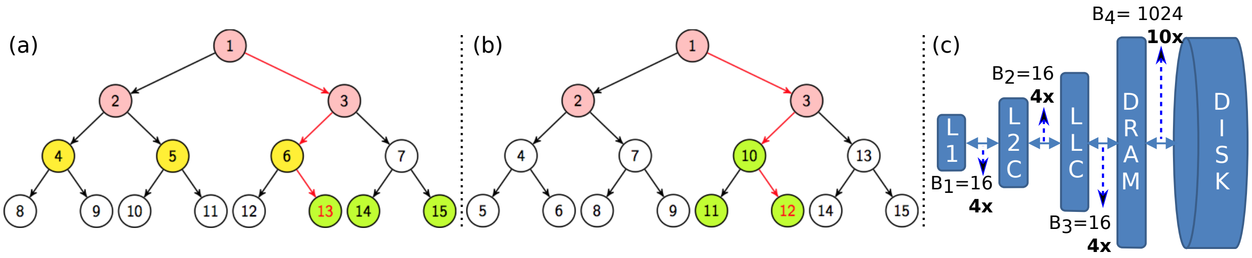


Figure 2: Illustration of the required data block transfer in searching for (a) key 13 in BFS tree and (b) key 12 in vEB tree, where a node's value is *its address in the physical memory*. Note that in (b), adjacent nodes are grouped together (e.g., (1,2,3) and (10,11,12)) because of the *recursive* tree building. The similarly colored nodes indicates a single block transfer B. An example of multi-level memory is shown in (c), where B_x is the *block transfer* size B between levels of memory.

2.1.3.3 Sorting algorithms.

Demaine gave two examples of cache-oblivious sorting algorithm in his brief survey paper [52], namely the *mergesort* and *funnelsort* [58]. 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 [15].

2.1.4 Cache-oblivious Data Structures

2.1.4.1 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 [113, 136]. 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 1).

Figure 2 illustrates the vEB layout, where the size B of memory blocks transferred between 2-level memory in the I/O model [15] is 3 (cf. Section 2.1.1). Traversing a complete binary tree with the Breadth First Search layout (or BFS tree for short) (cf. Figure 2a) 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 $(4, 5, 6)$ 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 2b, 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 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 2c).

2.1.4.2 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 [52].

The first cache-oblivious *priority queue* was due to Arge et al. [20] 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 [29, 30, 31] and cache-oblivious binary trees [34]. 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 2.1.6). Bender et al. [31] 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.

2.1.5 New Relaxed Cache-oblivious Model

We observe that 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 [58]. 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 [122]. Second,

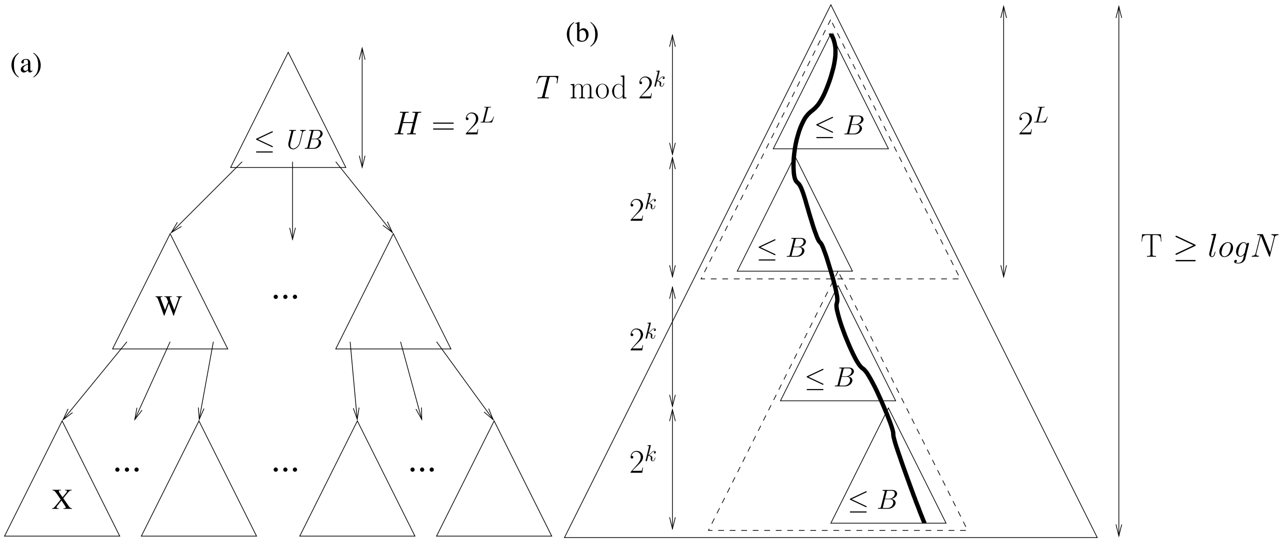


Figure 3: *(a)* New concurrency-aware vEB layout. *(b)* Search using concurrency-aware vEB layout.

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 1 in Section 2.1.6).

2.1.6 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 [15]), which is unknown as in the cache-oblivious model [58]. 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.

- Δ*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 3b). ΔNode is a 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 [29, 30, 31, 34, 58]. Figure 1 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 [58]. 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.

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 1 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 [34]. 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 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 3a 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 3a, 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 3a, 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 $2UB$. 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 1) while the concurrency-aware vEB supports high concurrency with update operations.

Lemma 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 [58], respectively.*

Proof. (Sketch) Figure 3b 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 3b.

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 $N \geq 2$. □

Unlike the conventional vEB layout, the new concurrency-aware vEB layout can solve the concurrency problems that might arise if the whole tree structure must be placed in a contiguous memory allocation. For example, when a conventional vEB layout tree is full, all of the tree structure must be re-allocated into a new bigger contiguous memory; and as a result, the whole tree must be locked to ensure correct concurrent search and update operations. The concurrency-aware vEB layout supports dynamic node allocation and new containers of size *UB* can be appended as needed to the existing tree structure whenever the tree is full. Therefore, in the concurrency-aware vEB layout, fine-grained locks can be use as the synchronization mechanism for concurrent tree operations.

A library of novel locality-aware and energy efficient concurrent search trees based on the new concurrency-aware vEB layout is presented in Section 2.2. The practical information on how to use the library is available in Appendix A.

2.2 GreenBST

Recent researches have suggested that the energy consumption of future computing systems will be dominated by the cost of data movement [48, 127, 128]. It is predicted that for 10nm technology chips, the energy required between accessing data in nearby on-chip memory and accessing data across the chip, will differ as much as 75× (2pJ versus 150pJ), whereas the energy required between accessing on-chip data and accessing off-chip data will only differ 2× (150pJ versus 300pJ) [48]. Therefore, in order to construct energy-efficient software systems, data structures and algorithms must not only be concerned with whether the data is on-chip (e.g., in cache) or not (e.g., in DRAM), but must consider also data locality in *finer-granularity*: where the data is located on the chip.

Concurrent trees are fundamental data structures that are widely used in different contexts such as load-balancing [51, 77, 119] and searching [13, 36, 37, 47, 54, 55]. Concurrent search trees are crucial data structures that are widely used as a backend in many important systems such as databases (e.g., SQLite [84]), filesystems (e.g., Btrfs [115]), and schedulers (e.g., Linux's Completely Fair Scheduler (CFS)), among others. These important systems can access and organize data in a more energy efficient manner by adopting the energy-efficient concurrent search trees as their backend structures.

Devising fine-grained data locality layout for concurrent search trees is challenging, mainly because of the trade-offs needed: (i) a platform-specific locality optimization might not be *portable* (i.e., not work on different platforms while there are big interests of concurrent data structures for unconventional platforms [78, 71]), (ii) the usage of transactional memory [83, 79] and multi-word synchronization [80, 70, 97] complicates locality because each core in a CPU needs to consistently track read and write operations that are performed by the other cores, and (iii) fine-grained locality-aware layouts (e.g., van Emde Boas layout) poorly support concurrent update operations. Some of the fine-grained locality-aware search trees such as Intel Fast [91] and Palm [118] are optimized for a specific platform. Concurrent B-trees (e.g., B-link tree [98]) only perform well if their B size is optimal. Highly concurrent search trees such as non-blocking concurrent search trees [55, 111] and Software Transactional Memory (STM)-based search trees [13, 47], however, do not take into account fine-grained data locality.

Fine-grained data locality for *sequential* search trees can be theoretically achieved using the van Emde Boas (vEB) layout [113, 136], which is analyzed using cache-oblivious (CO) models [58]. An algorithm is categorized as *cache-oblivious* for a two-level memory hierarchy if it has no variables that need to be tuned with respect to cache size and cache-line length, in order to optimize its data transfer complexity, assuming that the optimal off-line cache replacement strategy is used. If a *cache-oblivious* algorithm is optimal for an arbitrary two-level memory, the algorithm is also asymptotically optimal for any adjacent pair of available levels of the memory hierarchy [35]. Therefore, cache-oblivious algorithms are expected to be locality-optimized irrespective of variations in memory hierarchies, enabling less data transfer between memory levels and thereby saving energy.

However, the throughput of a vEB-based tree when doing *concurrent* updates is lower compared to when it is doing *sequential* updates. Inserting or deleting a node may result in relocating a large part of the tree in order to maintain the vEB layout. Solutions to this problem have been proposed [31]. The first proposed solution's structure requires each node to have parent-child pointers. Update operations may result in updating the pointers. Pointers will also increase the tree memory footprint. The second proposed solution uses the exponential tree algorithm [18]. Although the exponential tree is an important theoretical breakthrough, it is complex [46]. The exponential tree grows exponentially in size, which not only complicates maintaining its inter-node pointers, but also exponentially increases the tree's memory footprint. Recently, we have proposed a *concurrency-aware vEB layout* [133, 131], which has a higher throughput when doing *concurrent* updates compared to when it is doing *sequential* updates. In the same study, we have proposed DeltaTree, a B+tree that uses the concurrency-aware vEB layout. We have documented that the concurrency-aware vEB layout can improve DeltaTree's *concurrent* search and update throughput over a concurrent B+tree [133].

Nevertheless, we find DeltaTree's throughput and energy efficiency are lower than the state-of-the-art concurrent search trees (e.g., the portably scalable search tree [49]) for the update-intensive workloads (cf. Figure 4). Our investigation reveals that the cost of DeltaTree's runtime maintenance (i.e., rebalancing the nodes) dominates the execution time. However, reducing the frequency of the runtime maintenance lowers DeltaTree's energy efficiency and throughput for the search-intensive workloads, because DeltaTree nodes will then be

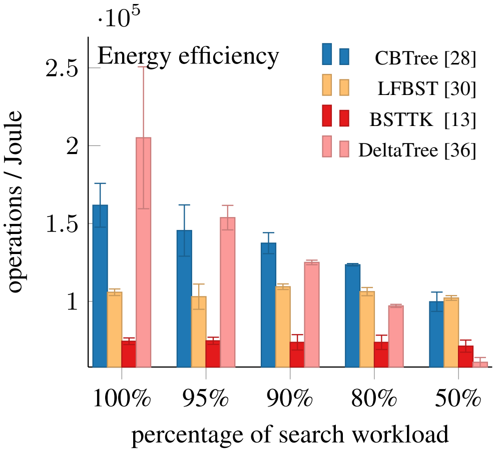


Figure 4: Result of 5 million tree operations of decreasing search percentage workloads using 12 cores (1 CPU). DeltaTree's energy efficiency and throughput are lower than the other concurrent search trees after 95% search workload on a dual Intel Xeon E5-2650Lv3 CPU system with 64GB RAM.

sparsely populated and frequently imbalanced. Note that DeltaTree energy efficiency and throughput are already optimized for the search intensive workloads [133, 134].

In this section, we present *GreenBST*, an energy-efficient concurrent search tree that is more energy efficient and has higher throughput for both the concurrent search- and update-intensive workloads than the other concurrent search trees (cf. Table 2.2). GreenBST applies two significant improvements on DeltaTree in order to lower the cost of the tree runtime maintenance and reduce the tree memory footprint. First, unlike DeltaTree, GreenBST rebalances incrementally (i.e., fine-grained node rebalancing). In DeltaTree, the rebalance procedure has to rebalance *all* the keys within a node and the frequency of rebalancing cannot be lowered as they are necessary to keep DeltaTree in good shape (i.e., keeping DeltaTree's height low and its nodes are densely populated). Incremental rebalance makes the overall cost of each rebalance in GreenBST lower than DeltaTree. Second, we reduce the tree memory footprint by using a different layout for GreenBST's leaf nodes (*heterogeneous* layout). Reduction in the memory footprint also reduces GreenBST's data transfer, which consequently increases the tree's energy efficiency and throughput in both update- and search- intensive workloads. We will show that with these improvements, GreenBST can become up to 195% more energy efficient than DeltaTree (cf. Section 2.4).

We evaluate GreenBST's energy efficiency (in operations/Joule) and throughput (in operations/second) against six prominent concurrent search trees (cf. Table 2.2) using a parallel micro-benchmarks *Synchrobench* [67] and STAMP database benchmark *Vacation* [108] (cf. Section 2.4). We present memory and cache profile data to provide insights into what make GreenBST energy efficient (cf. Section 2.4). We also provide insights into what are the key ingredients for developing energy-efficient data structures in general (cf. Section 2.5).

Table 1: List of the evaluated concurrent search tree algorithms.

#	**Algorithm**	**Ref**	**Description**	**Synchronization**	**Code authors**	**Data structure**
1	SVEB	[34]	*Conventional* vEB layout search tree	global mutex	U. Aarhus	binary-tree
2	CBTree	[98]	Concurrent B-tree (B-link tree)	lock-based	U. Tromsø	b+tree
3	Citrus	[19]	RCU-based search tree	lock-based	Technion	binary tree
4	LFBST	[111]	Non-blocking binary search tree	lock free	UT Dallas	binary tree
5	BSTTK	[49]	Portably scalable concurrent search tree	lock-based	EPFL	binary tree
6	DeltaTree	[133]	Locality aware concurrent search tree	lock-based	U. Tromsø	b+tree
7	**GreenBST**	-	Improved locality aware concurrent search tree	lock-based	this paper	b+tree

Our contributions.

Our contributions are threefold:

1. We have devised a new *portable fine-grained locality-aware* concurrent search trees, *GreenBST* (cf. Section 2.3). GreenBST are based on our proposed concurrency-aware vEB layout [133] with the two improvements, namely the incremental node rebalance and the heterogeneous node layouts.
2. We have evaluated GreenBST throughput (in operations/second) and energy efficiency (in operations/Joule) with six prominent concurrent search trees (cf. Table 2.2) on three different platforms (cf. Section 2.4). We show that compared to the state of the art concurrent search trees, GreenBST has the best energy efficiency and throughput across different platforms for most of the concurrent search- and update- intensive workloads.
   GreenBST code and evaluation benchmarks are available at: `https://github.com/uit-agc/GreenBST`.
3. We have provided insights into how to develop energy-efficient data structures in general (cf. Section 2.5).

2.3 GreenBST design overview

We devise GreenBST based on the concurrency-aware vEB layout [133] (cf. Section 2.1.6), based on the idea that the layout has the same data transfer efficiency between two memory levels as the *conventional* sequential vEB layout [113, 136]. Therefore, theoretically, we can use the concurrency-aware layout within a *concurrent* search tree to minimize data movements between memory levels, which can eventually be a basis of an energy-efficient concurrent search tree.

GreenBST and DeltaTree is designed by devising three major strategies, namely it uses a common GNode map instead of pointers or arithmetic-based implicit BST (i.e., a node's successor memory address is calculated *on the fly*) for node traversals, crafting an efficient inter-node connection, and using balanced layouts. In addition to the shared common traits

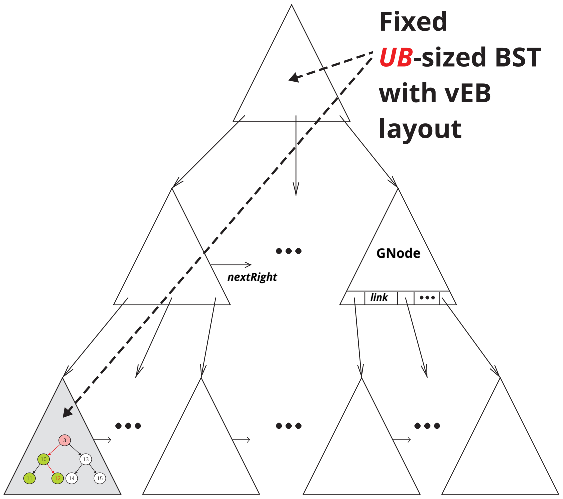


Figure 5: Illustration of the GreenBST layout.

with DeltaTree, GreenBST also employs two new major strategies: (i) GreenBST uses incremental GNode rebalance and (ii) GreenBST uses heterogeneous GNode layouts.

2.3.1 Data structures.

GreenBST is a collection of GNodes where each GNode consists of *UB* internal **nodes** that hold the tree keys and a $^1/_2 UB$ **link** array that links the GNode internal leaf nodes to another GNode's root node (cf. Figure 5). The chain of GNodes formed a B+tree (to avoid confusion, from this point onward, we refer to the "fat" nodes of GreenBST as GNode and the GNode's internal tree nodes as *internal nodes* or *nodes*). Each GNode also contains a lock (**locked**); a **rev** counter that is used for optimistic concurrency [95]; **nextRight** variable, which is a pointer that points to the GNode's right sibling; and **highKey** variable, which contains the lowest key member of the right sibling GNode. These last four variables are used for GreenBST concurrency control.

2.3.2 Cache-resident map instead of pointers or arithmetic implicit array.

GreenBST does not use pointers to link between its internal nodes, instead it uses a single map-based implicit BST array. This approach is unique to the concurrency-aware vEB layout as it benefits from the usage of the fixed-size GNodes. The usage of pointers and arithmetic-based implicit array in cache-oblivious (CO) trees has been previously studied [34] and both are found to have weaknesses. Pointer-based CO tree search operation is slow, mainly because of overheads in every data transfer between memory (although CO tree can minimize data transfers, the inclusion of pointers can lower the amount of meaningful data (e.g., keys) in each block transfer). The implicit array that uses arithmetic calculation for every node traversal may increase the cost of computation, especially if the tree is big.

The cache-resident-maps technique emulates BST's (left and right) child traversals inside

```
1: Struct Map:
2:     member fields:
3:         $left \in \mathbb{N}$, left child pointer address interval
4:         $right \in \mathbb{N}$, right child pointer address intvl.

5: Map $map[UB]$

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

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

Figure 6: Map structure and the *mapping* functions.

a GNode using a combination of a cache-resident GNode *map* structure and LEFT and RIGHT functions (cf. Figure 6). The LEFT and RIGHT functions, given an arbitrary node v and its GNode's root memory addresses, return the addresses of the left and right child nodes of v, or 0 if v has no children (i.e., v is an internal leaf node of a GNode). The LEFT and RIGHT operations throughout GreenBST share a common cache-resident *map* instance (cf. Figure 6, line 5). All GNodes use the same fixed-size vEB layout, so only one *map* instance with size *UB* is needed for all traversing operations. This makes GreenBST's memory footprint small and keeps the frequently used *map* instance in cache.

Note that the mapping approach does not induce memory fragmentation. This is because the mapping approach applies only for each GNode, and *map* is only used to point to internal nodes within a GNode. GNode layout uses a contiguous memory block of fixed size *UB* and *update* operations can only change the values of GNode internal nodes (e.g., from EMPTY to a key value in the case of insertion), but cannot change GNode's memory layout.

2.3.3 Inter-GNode connection.

To enable traversing from a GNode to its child GNodes, we develop a new inter-GNode connection mechanism. We logically assign binary values to GNode's internal edges so that each path from GNode root to an internal leaf node is represented by a unique bit-sequence. The bit-sequence is then used as an index in a **link** array containing pointers to child GNodes. As GNode's internal node has only left and right edges, we assign 0 and 1 to the left and right edges, respectively. The maximum size of the bit representation is GNode's height or $\log(UB)$ bits. We allocate a link pointer array whose size is half *UB* length. The algorithm in Figure 7 explains how the inter-GNode connection works in a pointer-less search function.

2.3.4 Balanced and concurrent tree.

GreenBST adopts the concurrent algorithms of B-link tree that provides lock-free search operations and adopts the B+tree structure for its high-level structure [98]. However, unlike B-link tree, GreenBST is an in-memory tree and uses optimistic concurrency to handle lock-free concurrent search operations even in the occurrences of the unique "in-place" GNodes maintenance operations.

```
 1: function SEARCH($key$, $GNode$, $maxDepth$)
 2:     while GNode is not leaf do
 3:         $rev \leftarrow GNode.rev$                    ▷ Get revision
 4:         $bits \leftarrow 0$
 5:         $depth \leftarrow 0$
 6:         $p \leftarrow GNode.nodes[0]$
 7:         $base \leftarrow p$
 8:         $link \leftarrow GNode.link$
          ▷ continue until leaf node:
 9:         while ($p$ & $p.key! = $ EMPTY ) do
          ▷ increment depth:
10:             $depth \leftarrow depth + 1$
          ▷ shift one bit to the left in each level
11:             $bits \leftarrow bits << 1$
12:             if ($key < p.key$) then
13:                 $p \leftarrow$ LEFT($p$, $base$)
14:             else
15:                 $p \leftarrow$ RIGHT($p$, $base$)
          ▷ right child color is 1:
16:                 $bits \leftarrow bits + 1$
          ▷ pad the $bits$:
17:         $bits \leftarrow bits << (maxDepth - depth) - 1$
18:         if ($GNode.rev$ != $rev$ or not even) then
19:             Goto 3                                ▷ Re-try GNode search
          ▷ follow $nextRight$ if $key \leq highKey$:
20:         if ($GNode.highKey \leq key$) then
21:             $GNode \leftarrow GNode.nextRight$
22:         else
23:             $GNode \leftarrow link[bits]$                ▷ child GNode
24:     return $GNode$
```

Figure 7: Search within pointer-less GNode. This function will return the *leaf* GNode containing the searched key. From there, an implicit array 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 search even during GNode split.

Similar to B-link tree, GreenBST *insert* operations build the tree from the bottom up, but unlike B-link tree, GreenBST insert operation can trigger *rebalance* operation, a unique GreenBST feature to maintain GNode's small height.

Function REBALANCE(T_i) is responsible for rebalancing a GNode T_i after an insertion. If a new node v is inserted at the *last level* node of a GNode, that GNode is rebalanced to a complete BST. A rebalance operation sets all GNode leaves node height to $\lfloor \log N \rfloor + 1$, where N is the count of the GNode's internal nodes and $N \leq UB$. Note that this is the default rebalance strategy used by DeltaTree, the incremental rebalance used by GreenBST is explained further in this section.

The *delete* operation in GreenBST simply marks the requested key (v) as deleted. This function fails if v does not exist in the tree or v is already marked. GreenBST does not employ merge operation between GNodes as node reclamation is done by the rebalance and split operations. The offline memory reclamation techniques used in the B-link tree [98] can be deployed to merge nearly empty GNodes in the case where delete operations are the majority. Our new search trees aim at workloads dominated by search operations.

GreenBST concurrency control uses locks and **nextRight** and **highKey** variables to coordinate between search and update operations [98] in addition to **rev** variable that is used for the search's optimistic concurrency. When a GNode needs to be maintained by either rebalance or split operations, the GNode's **rev** counter is incremented by one before the operation starts. The GNode counter is incremented by one again after the maintenance operation finishes. Note that all maintenance procedures happen when the lock is still held by the insert operation and therefore, only one operation may update **rev** counter and maintain a GNode at a time. The usage of **rev** counter is to prevent search from returning a wrong key because of the "in-place" GNode maintenance operation. Advanced locking techniques [76, 90, 102] can also be used.

The *search* operation in GreenBST uses a combination of function SEARCH (cf. Figure 7) and an implicit tree traversal using a map. Function SEARCH traverses the tree from the internal root node of the root GNode down to a leaf GNode, at which the search is handed over to the implicit tree traversal to find the searched key within the leaf GNode. GreenBST *search* operation does not wait nor use lock, even in the occurrence of the concurrent updates.

GreenBST *search* uses optimistic concurrency [95] to ensure the operation always returns the correct answer even if it arrives at a GNode that is undergoing the in-place maintenance operation (i.e., *rebalance* and *split*). First, before starting to traverse a GNode, a search operation records the GNode **rev** counter. Before following a link to a child GNode or returning a key, the search operation re-checks again the counter. If the current counter value is an odd number or if it is not equal to the recorded value, the search operation needs to retry search as this indicates that GNodes are being or have been maintained.

2.3.5 Incremental Rebalance.

As explained earlier, the rebalance in DeltaTree always involves UB keys, which eventually makes insertions require amortized $\mathcal{O}(UB)$ time. GreenBST borrows the incremental rebalance idea similar to the conventional vEB layout [34] that has the amortized $\mathcal{O}((\log^2 UB)/(1-\Gamma_1))$ time if used in GreenBST. However, unlike the conventional vEB layout that might have to rebalance the whole tree, we only apply the incremental rebalance to GNodes. To explain the idea, we denote *density(w)* as the ratio of number of keys inside a subtree rooted at w divided by the number of maximum keys that a subtree rooted at w can hold. For example, a subtree with root w that is located three levels away from an internal leaf of a GNode can hold at most $2^3 - 1$ keys. If the subtree only contains 3 keys, then *density(w)* $=^3 /_7 = 0.42$. We also denote a *density threshold* $0 < \Gamma_1 < \Gamma_2 < ... < \Gamma_H = 1$, where H is the GNode's height. The main idea is: after a new key is inserted at an internal leaf position v, we find the nearest ancestor w of v where *density(w)* $\leq \Gamma_{depth(w)}$ and *depth(w)* is the level where w resides, counted from the root of the GNode. If that w is found, we rebalance the subtree rooted at w.

2.3.6 Heterogeneous GNodes.

We aim to reduce the overhead of rebalancing and lower the GreenBST height with the usage of different layouts for the leaf GNodes. All DeltaTree's GNodes use the leaf-oriented BST layout, hence DeltaTree uses *homogeneous* GNodes. Unlike DeltaTree, leaf GNodes in GreenBST use the internal tree layout instead of the external (or leaf-oriented) tree layout. GreenBST uses *heterogeneous* GNodes as there are two difference GNode layouts used. In the internal tree layout, keys are located in all nodes of a tree, while in the external tree layout, keys are only located in the leaf nodes. The reasoning behind this choice is although leaf-oriented GNodes layout is required for inter-GNode connection (i.e., between parent- and child- GNodes), leaf GNodes do not have any children and therefore, do not need to adopt same structure as the other GNodes.

Table 2: We use 4 different benchmark platforms to evaluate the trees' energy efficiency and performance.

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

2.4 GreenBST experiments

We run several different benchmarks to evaluate GreenBST throughput and energy efficiency. We combine the benchmark results with the last level cache (LLC) and memory profiles of the trees to draw a conclusion of whether GreenBST improved fine-grained data locality layout (i.e., heterogeneous layout) and concurrency (i.e., lower overall cost of runtime maintenance) over DeltaTree are able to make GreenBST the most energy-efficient tree across different platforms. In addition, we would like to also conclude whether GreenBST improvements over DeltaTree are useful to increase GreenBST's energy efficiency when processing the update-intensive workloads. Note that we are not collecting the computation profiles (e.g., Mflops/second) because all the tree operations are data-intensive instead of compute-intensive.

We conduct an experiment on GreenBST and several prominent concurrent search trees (cf. Table 2.2) using parallel micro-benchmark that is based on Synchrobench [67] (cf. Figure 8). The trees' LLC and memory profiles during the micro-benchmarks are collected and presented in Figure 8d and 8e, respectively. To investigate GreenBST behavior in real-world applications, we implement GreenBST and CBTree as the backend structures in the STAMP database benchmark `Vacation` [108], alongside the `Vacation`'s original backend structure red-black tree (rbtree) (cf. Figure 9).

All the experimental benchmarks are conducted on an Intel high performance computing (**HPC**) platform with 24 core 2× Intel Xeon E5-2650Lv3 CPU and 64GB of RAM, an **ARM**

Table 3: The tree memory footprint after 2^{23} integer keys insertion on the HPC platform.

Tree name	SVEB	CBTree	citrus	LFBST	BSTTK	DeltaTree	**GreenBST**
Memory used (in **GB**)	0.1	0.4	0.8	0.7	1.0	0.6	**0.4**

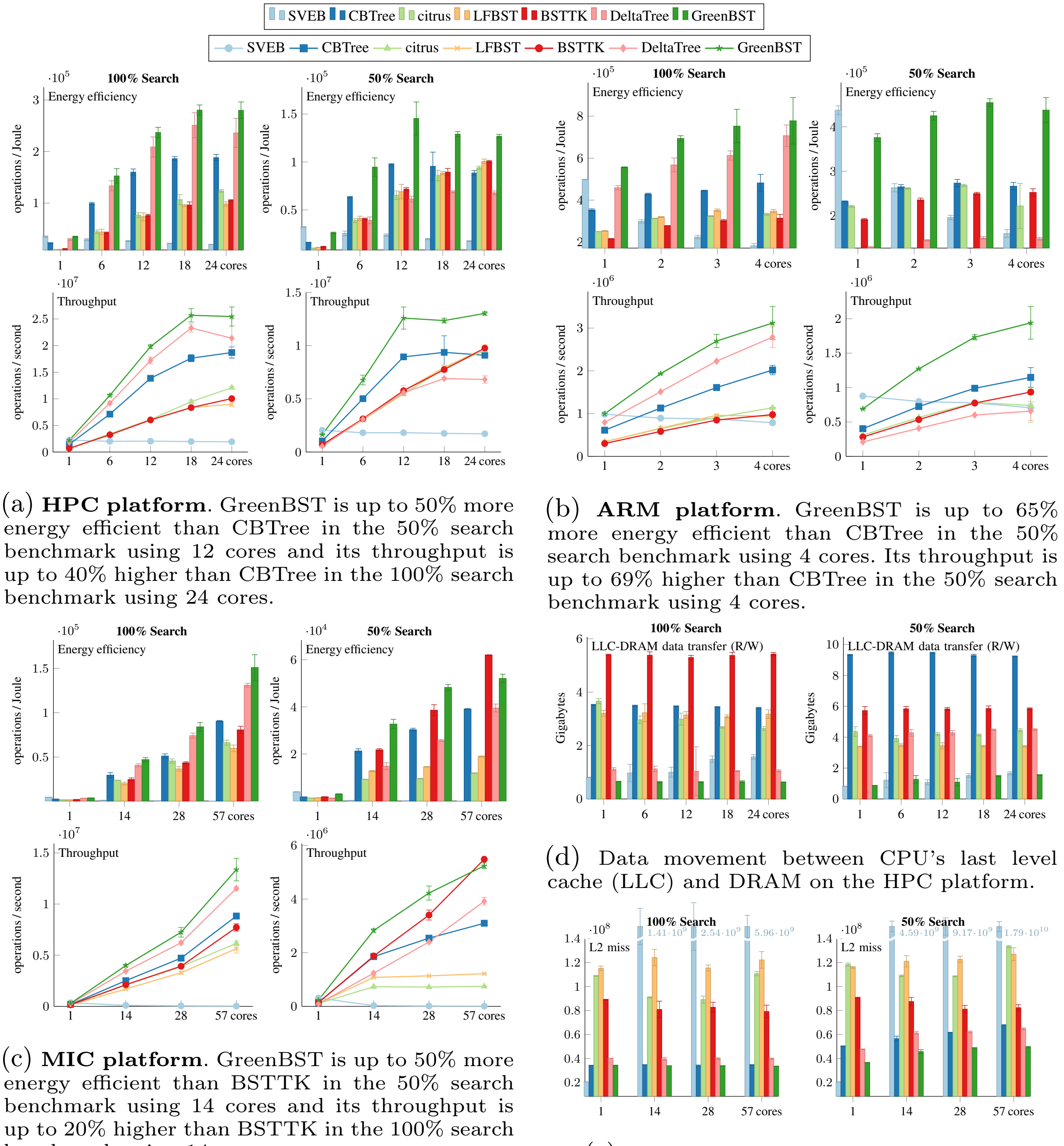
embedded platform with an 8 core Samsung Exynos 5410 CPU and 2GB of RAM (Odroid XU+E), an accelerator platform based on the Intel Xeon Phi 31S1P with 57 cores and 6GB of RAM (**MIC** platform), and a specialized computing platform (**Myriad2** platform). The detailed specifications for the testing platforms can be found in Table 2. For the parallel micro-benchmark, the trees are pre-initialized with several initial keys before running 5 million operations of 100% (search-intensive) and 50% searches (update-intensive), respectively. The initial keys given to both the ARM and MIC platforms are 2^{22} keys and to the HPC platform are 2^{23} keys. All experiments are repeated at least 5 times to guarantee consistent results.

Energy efficiency metrics (in operations/Joule) are the energy consumption divided by the number of operations and *throughput metrics* (in operations/second) are the number of operations divided by the maximum time for the threads to finish the whole operations. Energy metrics are collected from the on-board power measurement on the ARM platform, Intel RAPL interface on the HPC platform, and micras sysfs interface (i.e., `/sys/class/micras/power`) on the MIC platform.

2.4.1 Experimental results on HPC, ARM, and MIC platforms

Based on the results in Figure 8 and 9, GreenBST's energy efficiency and throughput are the highest compared to DeltaTree and the other trees. Because of its *incremental rebalance*, GreenBST outperforms DeltaTree (and the other trees) in the update-intensive workloads. With its *heterogeneous layout*, GreenBST is able to outperform DeltaTree in the search-intensive workloads. GreenBST energy efficiency and throughput are up to 195% higher than that of DeltaTree for the update intensive benchmark and up to 20% higher for the search intensive benchmark (cf. Figure 8b). Compared to the other trees, GreenBST energy efficiency and throughput are up to 65% and 69% higher, respectively. Note that CBTree (B-link tree) is a highly-concurrent B-tree variant that it's still used as a backend in popular database systems such as PostgreSQL.

The reason behind GreenBST good results is that GreenBST's data transfer (cf. Figure 8d) and LLC misses (cf. Figure 8e) are among the lowest of all the trees. We would like to emphasize that even GreenBST memory footprint is the same to that CBTree (cf. Table 3), GreenBST data transfer is significantly lower than CBTree's. These facts prove that the combination of locality-aware layout and the optimizations that GreenBST has over DeltaTree are beneficial to both fine-grained locality and concurrency, which are the key ingredients of an energy-efficient concurrent search tree.



(a) **HPC platform**. GreenBST is up to 50% more energy efficient than CBTree in the 50% search benchmark using 12 cores and its throughput is up to 40% higher than CBTree in the 100% search benchmark using 24 cores.

(b) **ARM platform**. GreenBST is up to 65% more energy efficient than CBTree in the 50% search benchmark using 4 cores. Its throughput is up to 69% higher than CBTree in the 50% search benchmark using 4 cores.

(c) **MIC platform**. GreenBST is up to 50% more energy efficient than BSTTK in the 50% search benchmark using 14 cores and its throughput is up to 20% higher than BSTTK in the 100% search benchmark using 14 cores.

(d) Data movement between CPU's last level cache (LLC) and DRAM on the HPC platform.

(e) L2 cache misses on the MIC platform.

Figure 8: (a,b,c) Energy efficiency and throughput comparison of the trees. On the HPC platform, DeltaTree and GreenBST energy efficiency and throughput decreases in the 50% search benchmark using 18 and 24 cores (i.e., with 2 chips) because of the coherence overheads between two CPUs (cf. Section 2.5). In the 50% search benchmark using 57 cores (MIC platform), BSTTK energy efficiency and throughput beats GreenBST by 20% because of the coherence overheads in the MIC platform (cf. Section 2.5). (d) LLC-DRAM data movements on the HPC platform, collected from the CPU counters using Intel PCM. (e) L2 cache miss counter on the MIC platform, collected using PAPI library.

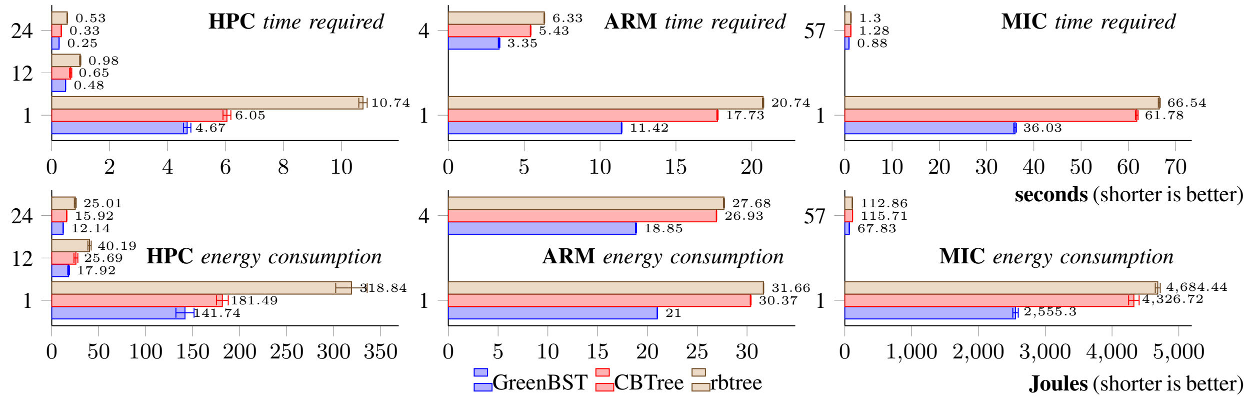


Figure 9: GreenBST energy efficiency and throughput against CBTree and STAMP's built-in red-black tree (rbtree) for the vacation benchmark. At best, GreenBST consumes 41% less energy and requires 42% less time than CBTree (in the 57 clients benchmark on the MIC platform).

2.4.2 Experimental results on Myriad2 platform

We have implemented DeltaTree and GreenBST that work on the Myriad2 platform by crafting a new concurrency control for the trees. A new concurrency is required because Myriad2 platform does not support atomic operations and has a limited number of usable hardware mutexes. Therefore, to circumvent these limitations, we create a new concurrency control scheme that works similarly to a ticket lock mechanism. In this scheme, we utilize LeonRT processor as a lock manager for the shaves. With LeonRT acting as a lock manager, all shaves need to request a DeltaNode or a GNode lock from LeonRT before it can lock the DeltaNode or GNode for update and maintenance operation. Our locking technique implementation uses only a shared array structure with $2 \times sv$ size, where sv is the number of active shaves. For low latency lock operations, we put this lock structure in the Myriad2's CMX memory. All other DeltaTree and GreenBST structures are unchanged (e.g., the tree itself) and placed in the DDR memory.

We tested our GreenBST and DeltaTree implementations on Myriad2 against the concurrent B+tree (B-link tree) [98]. The B-link tree implementation (CBTree) also utilized the same locking technique and memory placement strategy as GreenBST and DeltaTree.

Figure 10 shows that the energy efficiency of GreenBST is up to 4× better than that of CBTree in the 100% search using 12 shaves on the Myriad2 platform. In terms of throughput on the Myriad2 platform, Figure 11 indicates that GreenBST has up to 4× more throughput than CBTree in the 100% search case when using all available 12 shaves.

2.5 Discussions

Some of the benchmark results show that besides data movements, efficient concurrency control is also necessary in order to produce energy-efficient data structures. For example,

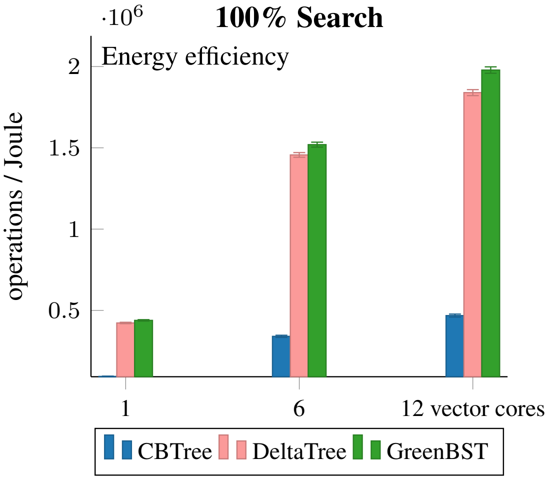


Figure 10: Energy comparison using 2^{20} initial values on an Myriad2 platform. DeltaTree is up to 4× more energy efficient than CBTree in 100% search operation with 12 shaves.

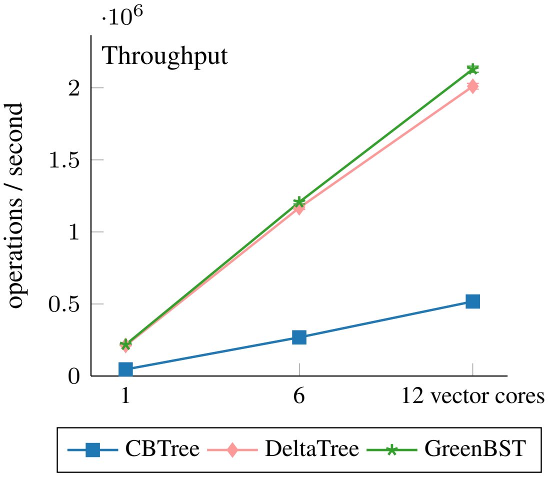


Figure 11: Throughput comparison using 2^{20} initial values on an Myriad2 platform. DeltaTree is up to 4× faster than CBTree in 100% search operation with 12 shaves.

the conventional vEB tree (SVEB) always transferred the smallest amount of data between memory and the CPU, but unfortunately, its energy efficiency and throughput failed to scale when using 2 or more cores. SVEB is not designed for concurrent operations and an inefficient concurrency control (a global mutex) has to be implemented in order to include the tree in this study (note that we are unable to use a more fine-grained concurrency because SVEB uses recursive layout in a contiguous memory block). Therefore, even if SVEB has the smallest amount of data transfer during the micro-benchmarks, the concurrent cores have to spend a lot of time waiting and competing for a lock. This is inefficient as a CPU core still consumes power (e.g., static power) even when it is waiting (idle).

Finally, an important lesson that we have learned is that minimizing overheads in locality-aware data structures can reduce the structure's energy consumption. One of the main differences between DeltaTree and GreenBST is that DeltaTree uses the homogeneous (leaf-oriented) layout, while GreenBST does not. Leaf-oriented GNodes increases DeltaTree's memory footprint by 50% as compared to GreenBST (cf. Figure 8e) and has caused higher data transfer between LLC and DRAM (cf. Figure 8d). Bigger leaf size also increases maintenance cost for each leaf GNode, because there are more data that need to be arranged in every rebalance or split operation, which leads to lower update concurrency. Therefore, DeltaTree energy efficiency and throughput are lower than GreenBST.

Inter-CPU and many-core coherence issue

Our experimental analysis has revealed that multi-CPU and many-core cache coherence, if triggered, can degrade concurrent update throughput and energy efficiency of the locality-aware trees. Figure 8a shows the "dips" in GreenBST's 50% update energy efficiency and throughput on the HPC platform (i.e., in the *50% update/18 cores* and *50% update/24 cores* cases). Figure 8c also shows that BSTTK beats GreenBST in the *50% update/57 cores* case on the MIC platform.

Using the CPU performance counters, we have found that the GreenBST concurrent updates frequently triggered the inter-CPU coherency mechanism. In the HPC platform, coherency mechanism causes heavy bandwidth saturation in the CPU interconnect. In the MIC platform, it causes most of the L2 data cache misses to be serviced from other cores and saturates the platform's bidirectional ring interconnect. These facts highlight the challenge faced by the locality-aware concurrent search tree: because of its locality awareness (i.e., related data are kept nearby and often re-used), the tree concurrent update operations might trigger heavy interconnect traffic on the multi-CPU platforms. The coherency mechanisms increase the total number of data transfer and the platform's energy consumption.

2.6 Conclusions

The results presented in this paper not only show that GreenBST is an energy-efficient concurrent search tree, but also provide an important insight into how to develop energy efficient data structures in general. On single core systems, having locality-aware data structures that can lower data movement has been demonstrated to be good enough to increase energy-efficiency. However, on multi-CPU and many cores systems, data-structures' locality-awareness alone is not enough and good concurrency and multi-CPU cache strategy are needed. Otherwise, the energy overhead of "waiting/idling" CPUs or multi-CPU coherency mechanism can exceed the energy saving obtained by fewer data movements.

3 Customization methodology for implementation of streaming aggregation in embedded systems

3.1 Introduction

Efficient real-time processing of data streams produced by modern interconnected systems is a critical challenge. In the past, low-latency streaming was mostly associated with network operators and financial institutions. Processing of millions of events such as phone calls, text messages, data traffic over a network and extracting useful information is important for guaranteeing high Quality of Service. Stream processing applications that handle traditional streams of data were mostly implemented by using Stream Processing Engines (SPEs) running on high performance computing systems.

However, nowadays digital data come from various sources, such as sensors from interconnected city infrastructures, mobile cameras and wearable devices. In the deviced-driven world of Internet of Things, there is a need in many cases for processing data on-the-fly, in order to detect events while they are occurring. These data-in-motion come in the form of live streams and should be gathered, processed and analyzed as quickly as possible, as they are being produced continuously. Low-power embedded devices or embedded micro-servers [120] are expected not only to monitor continuous streams of data, but also to detect patterns through advanced analytics and enable proactive actions. Applying analytics to these streams of data before the data is stored for post-event analysis (data-at-rest) enables new service capabilities and opportunities.

Streaming aggregation is a fundamental operator in the area of stream processing. It is used to extract information from data streams through data summarization. Aggregation is the task of summarizing attribute values of subsets of tuples from one or more streams. A number of tuples are grouped and aggregations are computed on their attributes in real-time fashion. High frequency trading in stock markets (e.g. continuously calculating the average number of each stock over a certain time window), real time network monitoring (e.g. computing the average network traffic over a time window) are examples of data stream processing, where streaming aggregation along with other operators is used to extract information from streams of tuples.

Streaming aggregation performance is affected a lot by the cost of data transfer. So far, streaming aggregation scenarios have been implemented and evaluated in various architectures, such as GPUs, Nehalem and Cell processors [117]. Indeed, there is a trend to utilize low power embedded platforms on running computational demanding applications in order to achieve high performance per watt [124][61][114][88].

Modern embedded systems provide different characteristics and features (such as memory hierarchy, data movement options, OS support, etc.) depending on the application domain that they target. The impact of each one of these features on performance and energy consumption of the whole system, when running a specific application, is often hard to predict at design time. Even if it is safe to assume in some cases that the utilization of a specific feature will improve or deteriorate the value of a specific metric in a particular context, it is hard to quantify the impact without testing. This problem becomes even harder when developers attempt to improve more than one metric simultaneously. A similar problem is the porting of an application running on a specific system to another with different specifications. The application usually need to be customized in the new platform differently, in order to provide improved performance and energy efficiency. The typical solution followed by developers is to try to optimize the implementation of the application on the embedded platform in an ad-hoc manner, which is a time consuming process that may yield suboptimal results. Therefore, there is a need for a systematic customization approach: Exploration can assist the effective tuning of the application and platform design options, in order to satisfy the design constraints and achieve the optimization goals.

Towards this end, in this work, we propose a semi-automatic step-by-step exploration methodology for the customization of streaming aggregation implemented in embedded systems. The methodology is based i) on the identification of the parameters of the streaming aggregation operator that affect the evaluation metrics and ii) on the identification of the embedded platform specification features that affect the evaluation metrics when executing streaming aggregation. These parameters compose a design space. The methodology provides a set of implementation solutions. For each solution, the application and the platform parameters have different values. In other words, each customized streaming aggregation implementation is tuned differently, so it provides different results for each evaluation metric. Developers can perform trade-offs between metrics, by selecting different customized implementations. Thus, instead of evaluating solutions in ad-hoc manner, the proposed approach provides a systematic way to explore the design space.

The main contributions of this work are summarized as follows:

i. We present a methodology for efficient customization of streaming aggregation implementation in embedded systems.

ii. We show that streaming aggregation implemented on embedded devices yields significantly higher performance per watt in comparison with corresponding HPC and general purpose GPU (GPGPU) implementations.

Finally, based on the experimental results of the demonstration of the methodology, we draw interesting conclusions on how each one of the application and platform parameters (i.e. design options) affects each one of the evaluation metrics. The methodology is demonstrated in two streaming aggregation scenarios implemented in four embedded platforms with different specifications: Myriad1, Myriad2, Freescale I.MX.6 Quad and Exynos 5 octa. The evaluation metrics are throughput, memory footprint, latency, energy consumption and scalability.

3.2 Related Work

Stream processing on various high performance architectures has been studied in the past extensively. Many works focus on the parallelization of stream processing [25], [68], [137]. They describe how the stream processing operators should be assigned to partitions to increase parallelism. The authors in [39] describe another way of improving the performance of streaming aggregation: They propose lock-free data structures for the implementation of streaming aggregation on multicore architectures. The evaluation has been conducted on a 6-core Xeon processor and the results show improved scalability.

With respect to stream processing engines (SPEs), Aurora and Borealis [12] are among the most well known ones. Several works that focus on the evaluation of stream processing operators on specific parallel architectures can be found in the literature. For example, an evaluation on heterogeneous architectures composed of CPU and a GPU accelerator is presented in [137]. The authors of [117] evaluate streaming aggregation implementations on Core 2 Quad, Nvidia GTX GPU and on Cell Broadband Engine architectures. The aggregation model used in this work is more complex, since it focuses on timestamp-based tuple processing.

There exists several works that describe the usage of low power embedded processors to run server workloads. More specifically, many works propose the integration of low-power ARM processors in servers [124] [61], or present energy-efficient clusters built with mobile processors [114].

In the area of embedded systems stream processing, several works focus on compilers that orchestrate parallelism, while they handle resource and timing constraints efficiently [42]. A programming language for stream processing in embedded systems has been proposed in [112]. These works are complementary to ours: The conclusions we drive from this work could assist the implementation of efficient compilers and development frameworks for stream programming.

Design space exploration in embedded systems is another area related with the present work. Exploration methodologies have been proposed for tuning at system architecture level [64], for customization of dynamic data structures [26] and of dynamic memory management optimization [139]. These customization approaches are complementary to the one proposed in the present work. Performance and energy consumption of streaming aggregation implementation could improve with effective customization of data structures or of the dynamic memory management of the system.

3.3 Streaming Aggregation

In this Section we provide a description of the streaming aggregation operator and we analyze the design challenges of implementing a streaming aggregation scenario on an embedded platform.

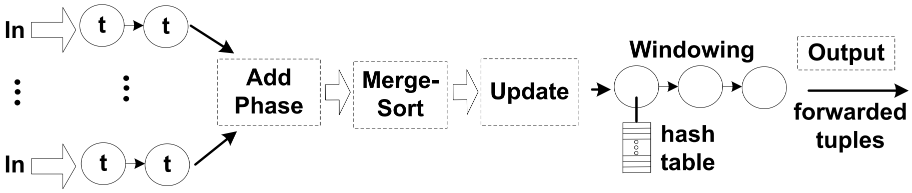


Figure 12: Time-based streaming aggregation scenario phases.

3.3.1 Streaming Aggregation description

Streaming aggregation is a very common operator in the area of stream processing. It is used to group a set of inbound tuples and compute aggregations on their attributes, similarly to the *group-by* SQL statement. In the context of this work, we discuss two aggregation scenarios: *multiway time-based with sliding windows* and *count-based with tumbling windows.*

3.3.1.1 Multiway time-based streaming aggregation

In multiway aggregation, multiple streams of incoming tuples, which are stored in queues, are combined into one stream, through a merge operator and their tuples are sorted given their timestamp attribute. It consists of 4 phases, as presented in Fig. 12:

1. *Add*: Incoming tuples are fetched from each input stream.

2. *Merge-Sort*: The tuples are merged and sorted, by the *merge* operator.

3. *Update*: Each tuple is assigned to the windows that it contributes to.

4. *Output*: Tuples with the computed aggregated value are forwarded.

During the *Add* phase tuples from each input stream are fetched and forwarded to the Merge-Sort phase. Since the incoming tuples are stored in a queue, they are forwarded in a FIFO manner.

Merge-Sort operation is used to combine streams that were sorted on a given attribute into a single stream, whose tuples are also ordered on the same attribute. In the context of this work, the tuples are sorted in timestamp order.

Merge and *Sort* are tightly coupled operations in streaming aggregation scenarios since they share the same resource (i.e. the incoming dequeued tuples) and they can be considered a single primitive operation. Merge-Sort phase ensures deterministic processing of the incoming tuples. A tuple is ready to be processed and forwarded to the next phase, if at least one tuple with an equal or higher timestamp has been received at each input stream.

In the *Update* phase the windowing operation is taking place and each single tuple is assigned to the window that it contributes to. In the context of this work, the aggregated values are computed over sliding windows, which have two attributes: *size* and *advance.* As an example, a window with *size* 5 time units and *advance* 2 time units, covers periods: [0,

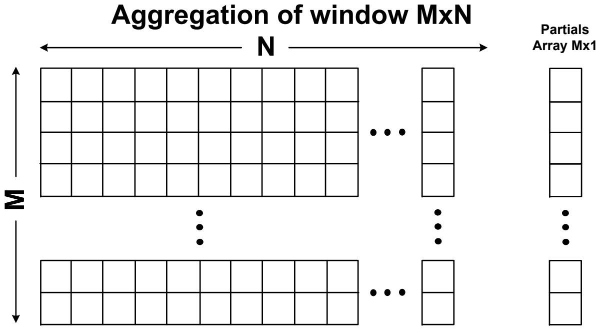


Figure 13: Window and partials array data structures used in the count-based streaming aggregation scenario.

5), [2, 7), [4, 9), etc. A tuple with timestamp 3, would contribute to windows [0, 5) and [2, 7).

In the *Output* phase, the aggregated value is calculated for all windows in which no more incoming tuples are expected to contribute (i.e. completed windows). The deterministic processing of tuples that took place in the earlier phases (more specifically during the *Add* and *Merge-Sort* phases), ensures that the aggregated value will be calculated only for completed windows. A new tuple is created for each aggregated value and it is forwarded, as a result of the aggregation operator.

Multiway time-based streaming aggregation provides pipeline parallelism, which can be exploited by assigning each phase on a different processing element (PE). However, performance relies not only on the exploitation of parallelism or on the computational power that the system provides, but also on the efficient data transfer between the phases. The sorted tuples of the Merge-Sort phase are used by the Update phase to be assigned to the windows that each one contributes to. The Update phase provides to the Output phase information on the windows in which the last tuples contributed to. Thus, the Output phase identifies the completed windows and calculates the aggregated value for each one. The utilization of efficient means of forwarding the information from one phase to another, affects both performance and energy consumption. The same applies to the way by which memory accesses on shared data are synchronized. Other important implementation issues that should be taken into account are the size of the queues in which the inbound tuples are stored (input queues) and the memory allocation of both the queues and the data structure in which the windows are stored.

3.3.1.2 Count-based streaming aggregation

In count-based aggregation, the window size is determined by the number of tuples buffered, instead of the time passed. Our case study considers fixed size windows and aggregation takes place periodically, i.e. when a specific number of tuples is received. Every time an aggregation is completed, all currently stored tuples are evicted and the next window is initially empty (tumbling window).

To implement the count-based aggregation scenario, we followed an approach based on [117]. The time intervals between aggregations are based on the number of tuples stored in the window and results of a specific window may depend on results of the previous one. Thus, an extra data structure is needed to store the partially aggregated results of the last window, which may be used in the following aggregation.

Figure 13, shows the data structures used in the count-based scenario: A MxN window and the partials array, with 1xM entries. M is the maximum number of input streams and N is the window width. When it is not possible to compute the aggregated value of N tuples for a specific input stream before the current window is forwarded, the partially aggregated result is stored in partials array. This result is used by the following window to compute the aggregated value of N tuples for the specific input stream. The output is a single tuple that it is produced by a query executed in the M aggregated values.

Apparently, count-based streaming aggregation provides data parallelism. Each window row can be assigned to a different processing element (PE) to compute the aggregated value of each input stream in parallel. Similarly to the time-based scenario, data transfer overhead, memory allocation issues and the window size affect the performance and the energy consumption of the operator. The embedded systems provide various solutions and each one has different impact on each evaluation metric. The design options for all the aforementioned implementation issues compose a design space that it is described in the following Section.

3.4 Customization Methodology

In this Section, we first present the design space for the streaming aggregation customization and then we describe the proposed methodology.

3.4.1 Design Space

The design space of the streaming aggregation implementation is presented as a set of decision trees, grouped into two categories (Fig. 14):

- *Category A* consists of decision trees that refer to memory configuration and allocation. Cache configuration options (private cache for each core or shared cache for all cores) are depicted in decision tree *A4*. *A5* is related with the dynamic memory allocation that can be based on freelists or in *malloc/free* system calls.

- In *category B* are assigned decision trees related to data movement and means by which accesses to shared resources are synchronized. The first three decision trees refer to different ways that data can be copied from global to local memories, or from one local memory to another (depending on the embedded system's memory hierarchy). Decision trees *B4* and *B5* are about synchronization between PEs, when accessing shared buffers. At low level, synchronization can be accomplished by spinning on shared variables (i.e. busy waiting) or by using other platform specific solutions. In platforms that run OS and support POSIX threads developers can utilized semaphores or monitors.

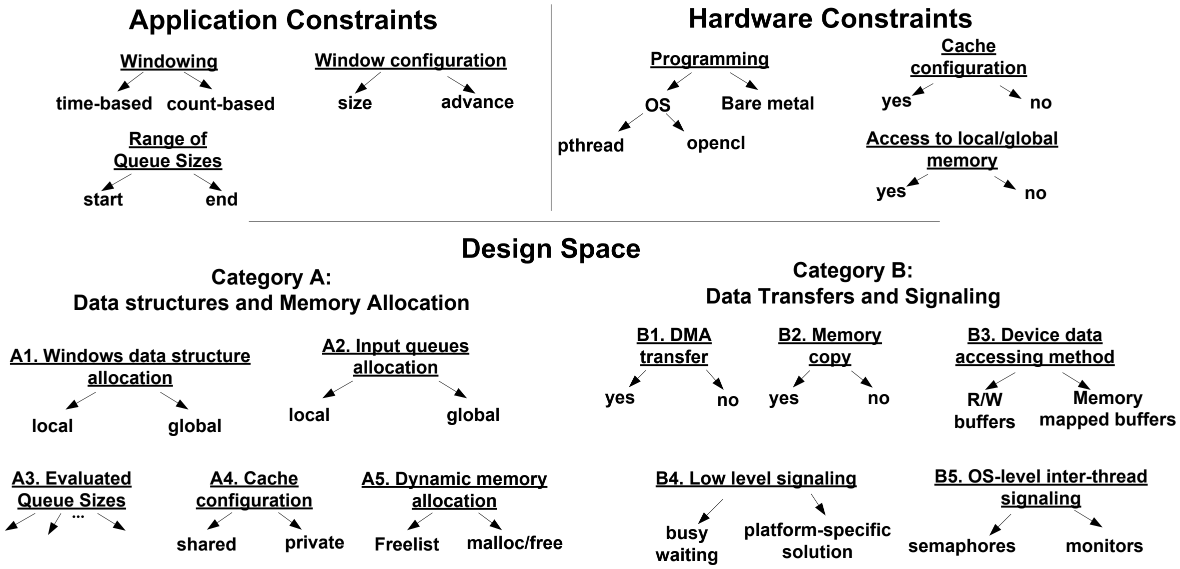


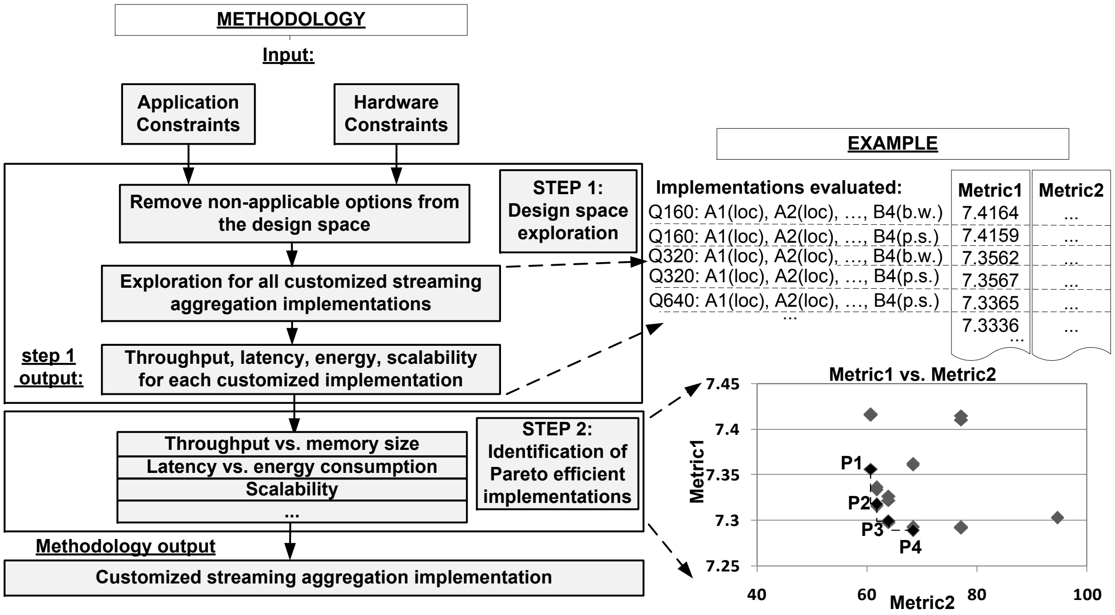
Figure 14: Constraints and Design space for streaming aggregation.

Table 4: Decision trees or leaves disabled for each application and hardware constraint.

App./Hw constraint	Decision tree/ leaf disabled
Windowing(tuple-based)	A2, A3, A5, B4
Window configuration	may disable A1(local)
Programming(bare metal)	B3 and B5
Programming(pthread)	B1, B3, B4
Programming(OpenCL)	B1, B2, B4, B5
Cache config.(no)	A4
Access to local/global(no)	A1, A2

Apparently, not all design options are applicable in any context. Fig. 14 shows the application and the hardware constraints that affect which decision trees or leaves are applicable in each specific context. The constraints are used to prune the decision trees and leaves that yield implementations which do not adhere to developer's requirements or they are not supported by the embedded platform.

Table 4 summarizes the design options that are disabled, due to application and hardware constraints. As an example, if the embedded platform runs an OS, access to DMA and to low-level signaling mechanisms are most likely handled by the OS directly, so these design options are not exposed to developers. *Window configuration* constraint may force the allocation of the data structures in a global memory. All constraints are provided manually. Constraints that prune non-compatible design space options "convert" the platform-independent design space into platform-dependent. Thus, they make the customization approach applicable in different contexts and in various embedded platforms.


METHODOLOGY
Input:
Application Constraints
Hardware Constraints
Remove non-applicable options from the design space
STEP 1: Design space exploration
Exploration for all customized streaming aggregation implementations
step 1 output:
Throughput, latency, energy, scalability for each customized implementation
Throughput vs. memory size
Latency vs. energy consumption
Scalability
...
STEP 2: Identification of Pareto efficient implementations
Methodology output
Customized streaming aggregation implementation
EXAMPLE
Metric1 vs. Metric2
7.45
7.4
7.35
7.3
7.25
Metric1
P1
P2
P3
P4
40
60
80
100
Metric2

Implementations evaluated:	Metric1	Metric2
Q160: A1(loc), A2(loc), …, B4(b.w.)	7.4164	...
Q160: A1(loc), A2(loc), …, B4(p.s.)	7.4159	...
Q320: A1(loc), A2(loc), …, B4(b.w.)	7.3562	...
Q320: A1(loc), A2(loc), …, B4(p.s.)	7.3567	...
Q640: A1(loc), A2(loc), …, B4(p.s.)	7.3365	...
...	7.3336 ...	...

Figure 15: Customization methodology.

After the pruning, valid customized streaming aggregation implementations are instantiated from the remaining decision tree leaves of the design space. In other words, the implementations that will finally be explored are the ones that are produced by combining the remaining leaves to create consistent implementations. Each one of these combinations is a valid customized solution that should be evaluated. All combinations of the remaining tree leaves are evaluated by brute-force exploration.

3.4.2 Methodology description

The exploration methodology consists of two steps and it is presented in Fig. 15. The inputs of the methodology are the application and hardware constraints. The output is a streaming aggregation implementation with customized software and hardware parameters.

The first step of the methodology aims at the pruning of the design space and the implementation of the design space exploration. First, the non-applicable options are removed from the design space due to the application and hardware constraints. Then, the streaming aggregation is executed once for each different combination of the decision tree leaves of the design space. For each customization, throughput, latency, memory size and energy consumption results are gathered. Scalability is another metric that can be evaluated, in case there is a relatively large number of PEs available. In the second step, the Pareto efficient implementations are identified. The trade-offs that can be performed by customization of the streaming aggregation on an embedded platform are presented in the form of Pareto curves. Developers can select the implementation that is most efficient according to the optimization target.

The tool flow that supports the methodology consists of a set of bash shell scripts that

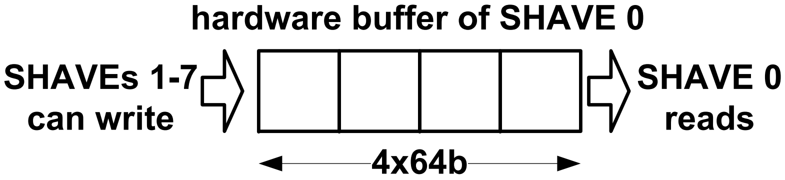


Figure 16: Myriad1 hardware buffers.

handle the first step of the methodology. For the second phase, the design space pruning and the exploration are performed automatically, provided that the hardware constraints are set manually. All performance results are collected automatically. However, power (which is used to calculate energy consumption) is measured manually, since it is usually based on platform-specific hardware instrumentation. Also, the tool flow integrates a script that calculates the Pareto curve for each requested pair of metrics.

Finally, it is important to state that most design options are normally provided as functions, macros, or compiler directives from either the platform SDK, or from the POSIX/OpenCL libraries. Therefore, it should not require significant programming effort by developers to switch between the design options presented in Fig. 14. Although the number of available implementations in some cases is increased, the systematic methodology we propose guarantees that all Pareto efficient implementations can be identified.

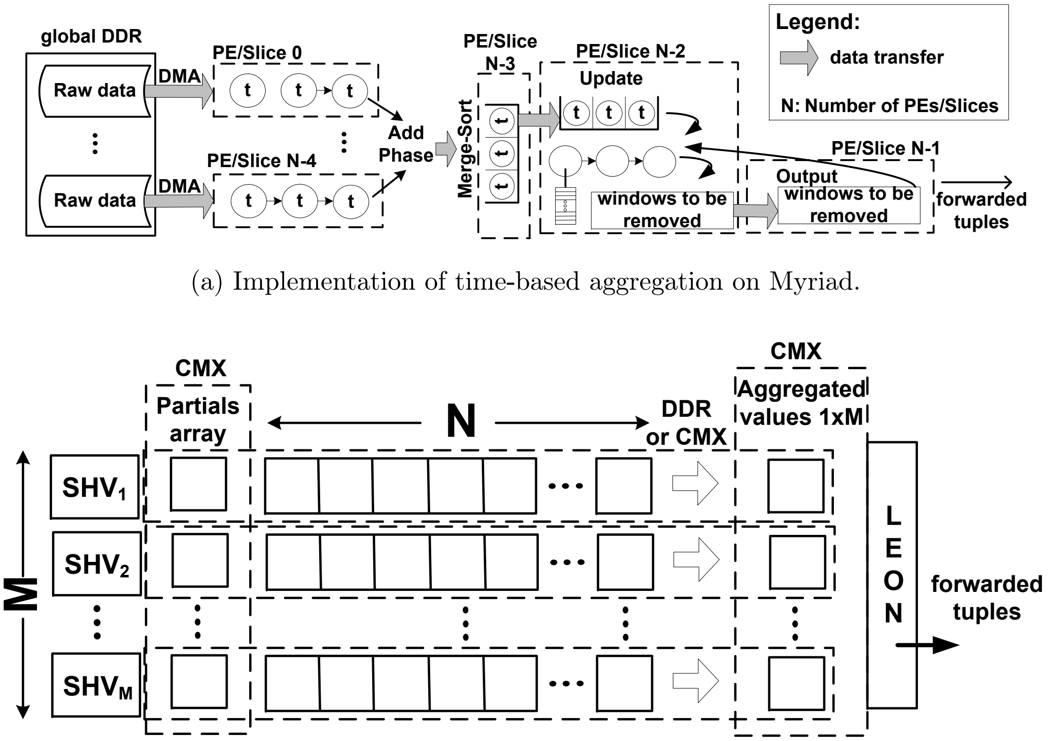
3.5 Demonstration of the Methodology

In this Section we first provide a short description of the embedded architectures that we used for demonstration of the methodology. Then, we present the experimental setup and the evaluation results, which are discussed in the last subsection.

3.5.1 Platforms description

Myriad embedded processors are designed by Movidius Ltd. [6]. They target computer vision and data streaming applications. Myriad architectures are utilized in the context of Project Tango, which aims at the design of mobile devices capable of creating a 3D model of the environment around them [8]. They belong to the family of low power mobile processors and provide increased performance per watt [88].

Myriad1 architecture is designed at 65nm. It integrates 8 VLIW processing cores named Streaming Hybrid Architecture Vector Engine (SHAVEs) operating at 180MHz and a LEON3 processor that controls the data flow, handles interrupts, etc.. More technical information about Myriad1 can be found in [110]. A local DMA engine is available for each SHAVE. Additionally, Myriad1 provides a set of hardware buffers for direct communication between the SHAVE cores. Each SHAVE has its own hardware buffer and they are accessed in FIFO manner. The size of each one is 4x64 bit words. As shown in Fig. 16, each SHAVE can push data into the buffer of any other SHAVE and it can read data only from its own buffer. A SHAVE writes to the tail of another buffer and the owner of the buffer can read from



(a) Implementation of time-based aggregation on Myriad.

(b) Implementation of count-based aggregation on Myriad.

Figure 17: Implementation of time-based and count-based streaming aggregation on Myriad.

the head. An interesting feature of the Myriad1 hardware buffers is the fact that when a SHAVE tries to write to a full FIFO or read from its own FIFO that happens to be empty, it stalls and enters a low energy mode. We take advantage of this, in order to propose energy efficient streaming aggregation implementations on Myriad1 platform.

Myriad2 is designed at 28nm [27]. In contrast with Myriad1, Myriad2 integrates 12 SHAVE cores operating at 504MHz, along with two independent LEON4 processors: LEON-RT targeting job management and LEON-OS suitable for running RTEMS/Linux, etc.. Myriad2 provides a single top-level DMA engine and the hardware buffers size is 16x64 words.

Regarding the memory specifications, Myriad1 provides 1MB local memory with unified address space that it is named Connection Matrix (CMX). Each 128KB are directly linked to each SHAVE processor providing local storage for data and instruction code. Therefore, the CMX memory can be seen as a group of 8 memory "slices", with each slice being connected to each one of the 8 SHAVEs. Each SHAVE accesses its own CMX slice more efficiently in comparison with the rest CMX slices. Myriad2 CMX memory is 2MB and each slice is 128KB. Also, Myriad2 provides 1KB L1 and 256KB L2 cache. Finally, both platforms provide a global DDR memory of 64MB.

Concerning the memory allocation of the time-based streaming aggregation data structures, the incoming streams of raw data (produced by sensors, cameras, etc.) are placed

Table 5: Hardware constraints for Myriad1, Myriad2, I.MX.6 Quad and Exynos for both scenarios.

	Time-based aggregation			Count-based aggregation		
	Myriad1	Myriad2	I.MX.6	Myriad1	Myriad2	Exynos
windowing	time	time	time	count	count	count
programming	bare metal	bare metal	pthread	bare metal	bare metal	OpenCL
cache config.	no	yes	no	no	yes	no
access local/global mem.	yes	yes	no	yes	yes	yes

in DDR memory. Each input queue is handled by a different SHAVE and it is placed in its local slice. Each SHAVE that handles an input queue fetches chunks of raw data in its own memory slice, by using DMA transfers. Then, it converts the raw data into tuples and stores them in its own input queue. The windows are stored in a linked list data structure, which is allocated in the CMX slice of the SHAVE core that handles the Update phase. Memory allocation and other implementation details are displayed in Fig. 17a. Regarding the count-based aggregation scenario that uses a MxN window, each one of the M SHAVEs continuously fetches raw data that correspond to N tuples from DDR to CMX. However, if N is very large and tuples cannot be stored and processed in CMX, they are placed and aggregated in DDR. Each SHAVE computes the aggregated value of N tuples and forwards the result to LEON, which produces the output tuple that corresponds to the specific window. The implementation diagram in Fig. 17b.

Freescale I.MX 6 Quad integrates four ARM Cortex A9 cores that operate at 1GHz [10]. It belongs to a family of multicore ARM-based platforms that target single board computers and run Linux-based OS. It provides 1GB RAM and two cache memory levels. On I.MX.6 the raw data are placed in data files. Chunks of raw data are fetched in RAM using *freed()* function. Then, tuples are created and placed in the input queues to be forwarded to the subsequent streaming aggregation phases.

Exynos 5 octa is an ARM-based platform that targets mobile computers. It is designed at 28nm by SAMSUNG and it is based on big.LITTLE architecture [43]. It integrates two ARM clusters: 4 Cortex-A15 and 4 Cortex-A7 cores. Exynos 5 integrates a PowerVR SGX544 GPU that supports OpenCL1.1. It includes 3 processing cores running at 533MHz. The evaluation board integrating Exynos is the Odroid-XU that provides 2GB DDR3 RAM [3]. In the context of this work, we used PowerVR GPU to perform aggregation in the count-based streaming scenario, implemented in OpenCL.

3.5.2 Experimental Setup

The dataset we used to demonstrate the proposed methodology has been collected from the online audio distribution platform SoundCloud [9]. It consists of a subset of approximately 40,000 users that exchanged comments between 2007 and 2013. The incoming tuples contain the following attributes: *timestamp*, *user_ id*, *song_ id* and *comment*. The aggregation function forwards the id of the user with the largest number of comments in each window. In the time-based aggregation scenario the window is sliding, while in the count-based, the window is tumbling, so the aggregated value is calculated over the last MxN tuples.

The aggregation operator is implemented entirely in C. Throughput is measured as tuples processed per second, while latency as the timestamp difference between an output tuple with the aggregated value and the latest input tuple that produced it. The energy consumption results on I.MX.6 were obtained based on hardware instrumentation using a Watts Up PRO meter device and following a setup similar to methods proposed in the literature [85][121]. In Myriad2 power was measured though the MV198 power measurement board integrated on Myriad2 evaluation board. In Myriad1 power was estimated, based on moviSim simulator provided by Movidius MDK. In Exynos it is measured based on power sensors that are provided by Odroid-XU-e evaluation board [3]. All the values presented are the average of 10 executions, by elimination of the outliers. Each single experiment is executed from 30 seconds up to one minute.

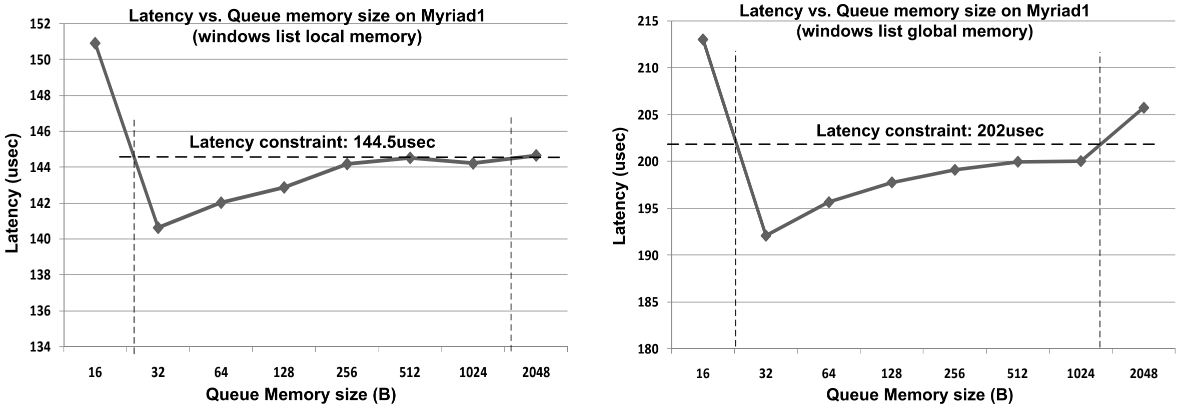
The time-based aggregation scenario, which is actually a pipeline, is demonstrated in Myriad and I.MX.6 Quad platforms. The count-based scenario, that provides increased data parallelism, is demonstrated in Myriad and in Exynos embedded GPU. As stated earlier, Myriad1 provides 8 PEs. In time-based aggregation, each one of the merge-sort, update and output phases is assigned to a single PE. Each one of the remaining 5 PEs handles a single input queue. In Myriad2, which integrates 12 PEs, the input queues are 9. In I.MX.6 Quad that provides 4 PEs, we assigned each phase on single PE and the remaining PE handles 5 input queues.

The hardware constraints of the evaluation boards are presented in Table 5. The experiments we performed are the following: In the time-based aggregation scenario, in I.MX.6 we implemented the methodology using a single window configuration. However, for Myriad1 and Myriad2, we present results for two different scenarios: in the first one the window configuration (i.e. the window *size* and *advance* values) are set, so that the maximum memory size of the windows data structure is small enough to fit in the local memory. In the second experiment, the windows data structure can only fit in the global memory. Thus, we study how the memory allocation of the windows data structure affects the evaluation metrics. In the count-based scenario, the aggregation is performed in parallel by the accelerator of each platform: The SHAVEs in Myriad and the GPU in Exynos.

The output of the methodology is a set of Pareto points for throughput vs. memory size and latency vs. energy consumption. In time-based scenario, we present results for scalability for Myriad1 and Myriad2. The implementations that are evaluated for scalability are the ones that were found to be Pareto efficient in latency vs. energy consumption evaluation.

3.5.3 Time-based aggregation results

In the time-based scenario, we evaluate each implementation for a number of queue sizes. The queue sizes we select are the ones that provide latency below a fixed threshold. Therefore, we first measure latency for a range queue sizes and select the size values which provide latency below the threshold. Then, we proceed to the implementation of the methodology.



(a) Windows list in local mem..

(b) Windows list in global mem..

Figure 18: Latency vs. Queue size on Myriad1.

Table 6: Myriad1 Pareto efficient points description. B4(p.s.) (i.e. platform specific) refers to Myriad hardware buffers.

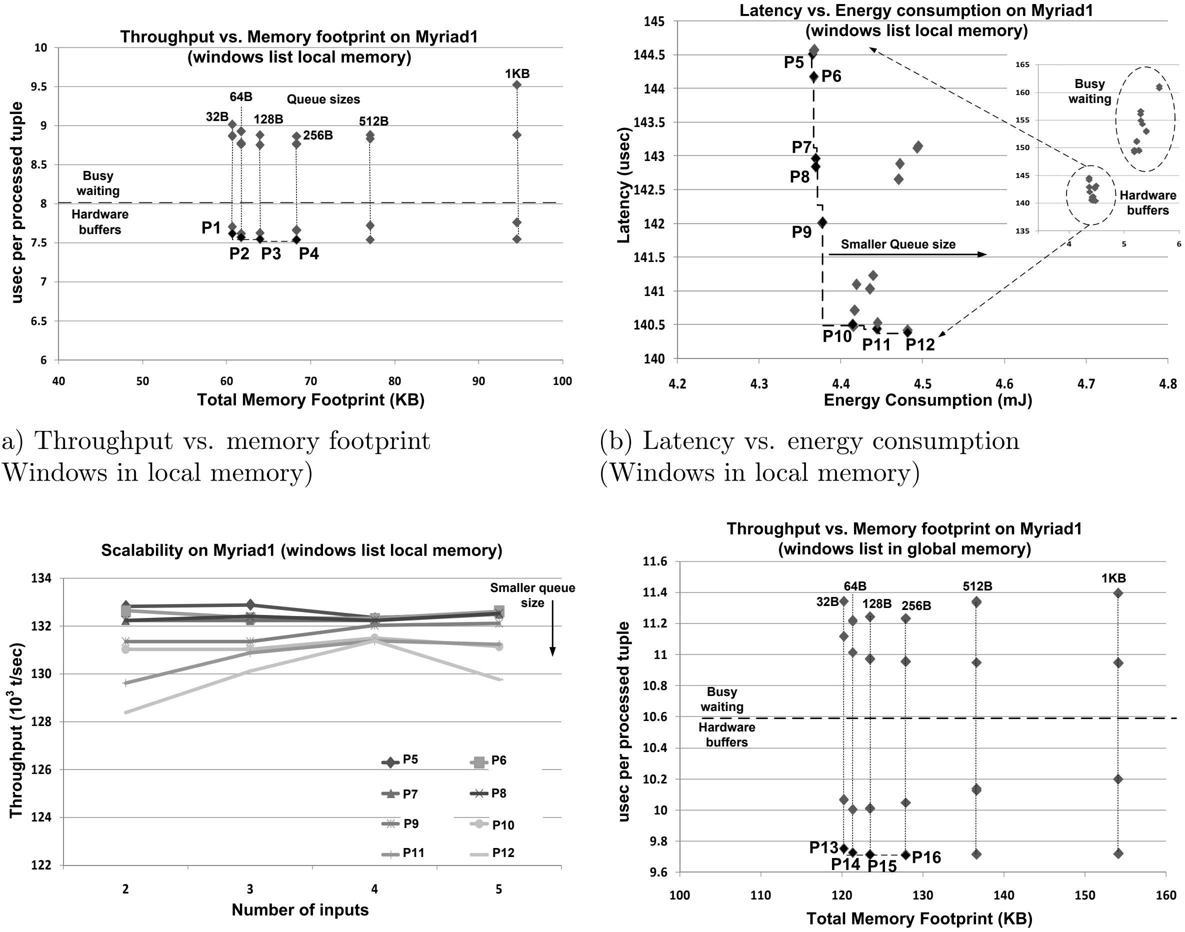
Pareto	Description	Pareto	Description	Pareto	Description
P1	A1(l), A2(l), A3(32B), A5(fl), B2(yes), B4(p.s.)	P8	A1(l), A2(l), A3(128B), A5(fl), B2(yes), B4(b.w.)	P15	A1(l), A2(l), A3(128B), A5(fl), B2(yes), B4(b.w.)
P2	A1(l), A2(l), A3(64B), A5(fl), B2(yes), B4(p.s.)	P9	A1(l), A2(l), A3(64B), A4(fl), B1(yes), B4(p.s.)	P16	A1(on), A2(on), A3(256B), A5(fl), B2(yes), B4(p.s.)
P3	A1(l), A2(l), A3(128B), A5(fl), B2(yes), B4(p.s.)	P10	A1(l), A2(l), A3(64B), A5(fl), B2(yes), B4(p.s.)	P17	A1(l), A2(l), A3(256B), A5(fl), B1(yes), B4(p.s.)
P4	A1(l), A2(l), A3(256B), A5(fl), B2(yes), B4(p.s.)	P11	A1(l), A2(l), A3(64B), A5(fl), B1(yes), B4(p.s.)	P18	A1(l), A2(l), A3(128B), A5(fl), B2(yes), B4(p.s.)
P5	A1(l), A2(l), A3(512B), A5(fl), B2(yes), B4(p.s.)	P12	A1(l), A2(l), A3(32B), A5(fl), B2(yes), B4(p.s.)	P19	A1(l), A2(l), A3(64B), A5(fl), B2(yes), B4(p.s.)
P6	A1(l), A2(l), A3(256B), A5(fl), B2(yes), B4(p.s.)	P13	A1(l), A2(l), A3(32B), A5(fl), B2(yes), B4(p.s.)	P20	A1(l), A2(l), A3(32B), A5(fl), B2(yes), B4(p.s.)
P7	A1(l), A2(l), A3(128B), A5(fl), B1(yes), B4(p.s.)	P14	A1(l), A2(l), A3(64B), A5(fl), B2(yes), B4(p.s.)	P21	A1(l), A2(l), A3(32B), A5(fl), B2(yes), B4(b.w.)

48 implementations are evaluated in Myriad and 4 in I.MX.6 Quad. The number of implementations that are evaluated can be reduced by selecting a smaller number of queue size values. (However, in this case fewer Pareto points may be identified).

3.5.3.1 Demonstration on Myriad1

In the first experiment in Myriad1 the window size and advance values are configured so that the windows data structure can fit in the local memory. Assuming latency constraint of 144.5usec, the range of queue sizes that we evaluate are from 32B to 1024B (Fig. 18a).

The results for throughput vs. memory evaluation are displayed in Fig. 19a. We notice that the Pareto points can be divided in two categories: The ones with performance lower than 8.0usec/tuple that correspond to implementations that utilize busy waiting and the rest ones that utilize the Myriad hardware buffers. (In both axes, the lower the values, the higher the efficiency). 4 Pareto efficient points are identified, which are described in Table 6.



(a) Throughput vs. memory footprint (Windows in local memory)

(b) Latency vs. energy consumption (Windows in local memory)

(c) Scalability (Windows in local memory)

(d) Throughput vs. memory footprint (Windows in global memory)

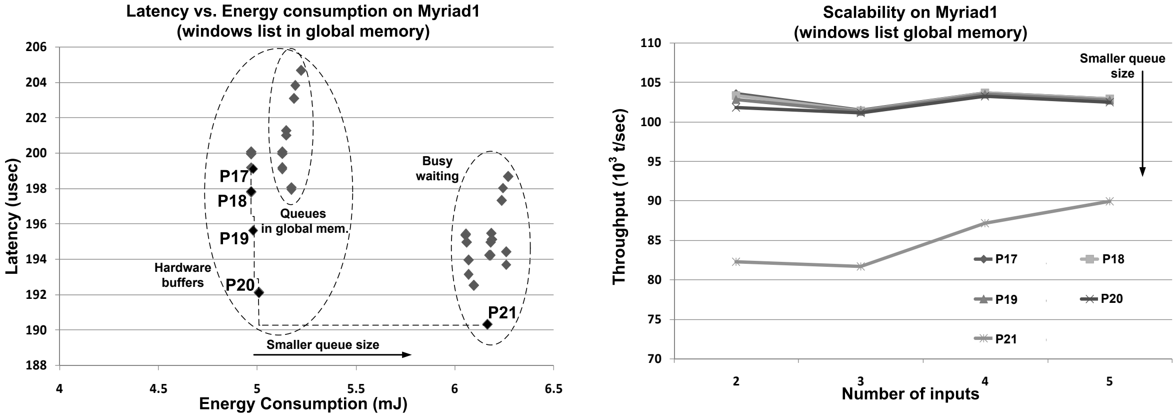
Figure 19: Evaluation of time-based streaming aggregation implementations on Myriad1.

All Pareto efficient customized implementations can be used to perform trade-offs between throughput and memory: throughput can increase up to 1.02% and maximum memory size can drop up to 11.2% by selecting P4 and P1 solutions respectively.

Pareto points of latency vs. energy can be grouped into the same categories: The ones that exploit busy waiting and the rest that utilize hardware buffers. The later are more efficient both in terms of latency and energy consumption. 8 Pareto points can be identified that can be used to perform trade-offs between the aforementioned metrics: up to 2.85% lower latency (P12) and up to 2.6% lower energy consumption (P5).

Finally, scalability evaluation of the Pareto points of latency vs. energy is shown in Fig. 19c. Throughput remains almost constant for all implementations or increases with the number of inputs. The only exception is P12, in which the queues have very small size (32B).

In the second experiment, we assume latency threshold to be 202usec (Fig. 18b). We



(a) Latency vs. energy consumption (Windows in global memory)

(b) Scalability (Windows in global memory)

Figure 20: Evaluation of time-based streaming aggregation implementations on Myriad1.

notice in both Fig. 19d and Fig. 20a that throughput is lower and latency higher in comparison with the previous experiment, since in this one the windows are placed in the global memory. The Pareto efficient points demonstrated in Fig. 19d can be used to perform trade-offs between throughput and memory size (up to 0.5% for throughput by selecting P16 and up to 5.9% in memory size by selecting P13). In Fig. 20a, we notice that Pareto point P21 is the most efficient in terms of latency (4.45% lower in comparison with P17), while P17 implementation is the most energy efficient (19.3% lower consumption than P21). In the scalability evaluation of Fig. 20b, it is shown that all implementations provide high throughput that it is affected by the number of inputs only slightly, apart from P21 that utilizes busy-waiting and yields much lower throughput in comparison with the rest of the implementations.

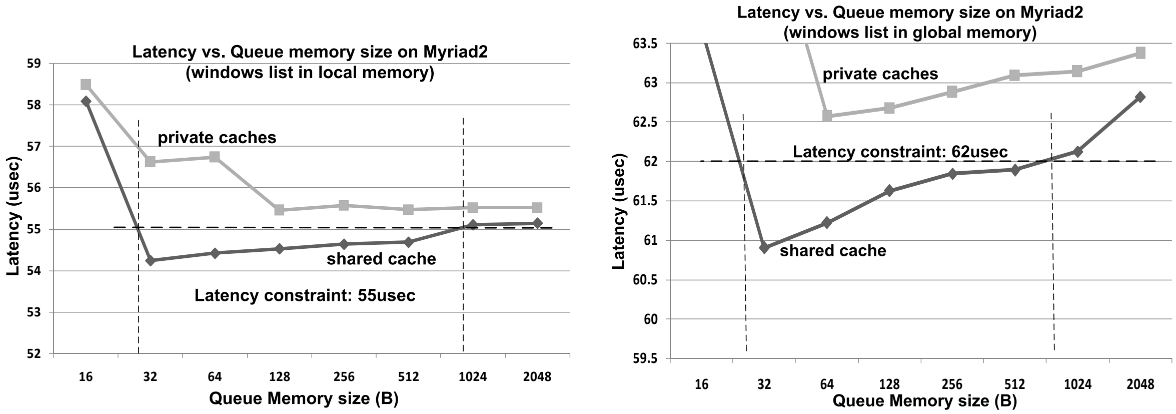
3.5.3.2 Demonstration on Myriad2

Fig. 21a and Fig. 21b show latency vs. queue sizes on Myriad2 for two different cache configurations, shared and private (decision tree *A4* in Fig. 14). We notice that shared cache provides lower latency than private in both cases, up to 4.2%. Therefore, all implementations that utilize private cache are pruned and they are not evaluated in step 1 of the methodology.

In the first experiment in Myriad2, the windows data structure is placed in the local memory. Latency constraint is assumed to be at 55usec and therefore queue sizes from 32B to 512B will be evaluated (Fig. 21a).

Throughput vs. memory footprint results of the methodology are shown in Fig. 22a. Implementations based on *memcpy* provide higher performance than the ones based on dma transfers between the CMX slices. The 5 Pareto efficient points that are identified provide trade-offs up to 3.7% for throughput (P5) and up to 22.5% for memory footprint (P1).

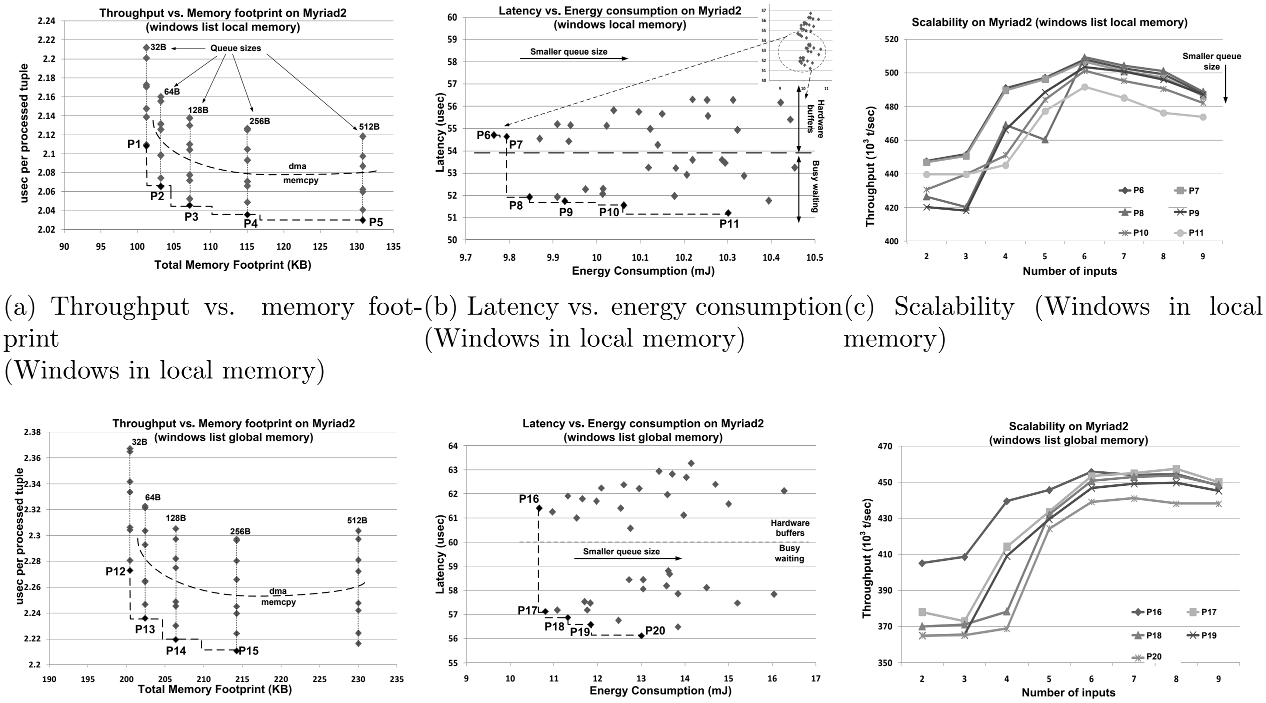
Latency vs. energy results are displayed in Fig. 22b. The Pareto points can be grouped into 2 categories: the ones that utilize busy waiting synchronization scheme and the rest ones



(a) Windows list in local mem..

(b) Windows list in global mem..

Figure 21: Latency vs. Queue size on Myriad2.



(a) Throughput vs. memory footprint (Windows in local memory)

(b) Latency vs. energy consumption (Windows in local memory)

(c) Scalability (Windows in local memory)

(d) Throughput vs. memory footprint (Windows in global memory)

(e) Latency vs. energy consumption (Windows in global memory)

(f) Scalability (Windows in global memory)

Figure 22: Evaluation of time-based streaming aggregation implementations on Myriad2.

that are based on hardware buffers. The 6 Pareto efficient points can be used to perform trade-offs between latency and energy (up to 6.37% for latency by selecting implementation P11 and 5.2% for energy consumption, by selecting P6).

Table 7: Myriad2 Pareto efficient points description. B4(p.s.) (i.e. platform specific) refers to Myriad hardware buffers

Par.	Description	Par.	Description	Par.	Description
P1	A1(l), A2(l), A3(32B), A4(s), A5(fl), B2(y), B4(b.w.)	P8	A1(l), A2(l), A3(512B), A4(s), A5(fl), B2(y), B4(b.w.)	P15	A1(l), A2(l), A3(256B), A4(s), A5(fl), B2(y), B4(p.s.)
P2	A1(l), A2(l), A3(64B), A4(s), A5(fl), B2(yes), B4(p.s.)	P9	A1(l), A2(l), A3(128B), A4(s), A5(fl), B1(y), B4(b.w.)	P16	A1(l), A2(l), A3(256B), A4(s), A5(fl), B2(y), B4(p.s.)
P3	A1(l), A2(l), A3(128B), A4(s), A5(fl), B2(y), B4(p.s.)	P10	A1(l), A2(l), A3(64B), A4(s), A5(fl), B2(y), B4(b.w.)	P17	A1(l), A2(l), A3(512B), A4(s), A5(fl), B1(y), B4(b.w.)
P4	A1(l), A2(l), A3(256B), A4(s), A5(fl), B2(y), B4(p.s.)	P11	A1(l), A2(l), A3(32B), A4(s), A5(fl), B1(y), B4(b.w.)	P18	A1(l), A2(l), A3(128B), A4(s), A5(fl), B2(y), B4(b.w.)
P5	A1(l), A2(l), A3(512B), A4(s), A5(fl), B2(y), B4(p.s.)	P12	A1(l), A2(l), A3(32B), A4(s), A5(fl), B2(y), B4(b.w.)	P19	A1(l), A2(l), A3(64B), A4(s), A5(fl), B2(y), B4(b.w.)
P6	A1(l), A2(l), A3(512B), A4(s), A5(fl), B2(y), B4(p.s.)	P13	A1(l), A2(l), A3(64B), A4(s), A5(fl), B2(y), B4(p.s.)	P20	A1(l), A2(l), A3(32B), A4(s), A5(fl), B2(y), B4(b.w.)
P7	A1(l), A2(l), A3(256B), A4(s), A5(fl), B1(y), B4(p.s.)	P14	A1(l), A2(l), A3(128B), A4(s), A5(fl), B2(y), B4(p.s.)		

With respect to scalability in Fig. 22c, we notice that throughput for all implementations increases up to 6 inputs and then it drops slightly. As in Myriad1 experiments, implementations with lower queue size tend to provide lower throughput.

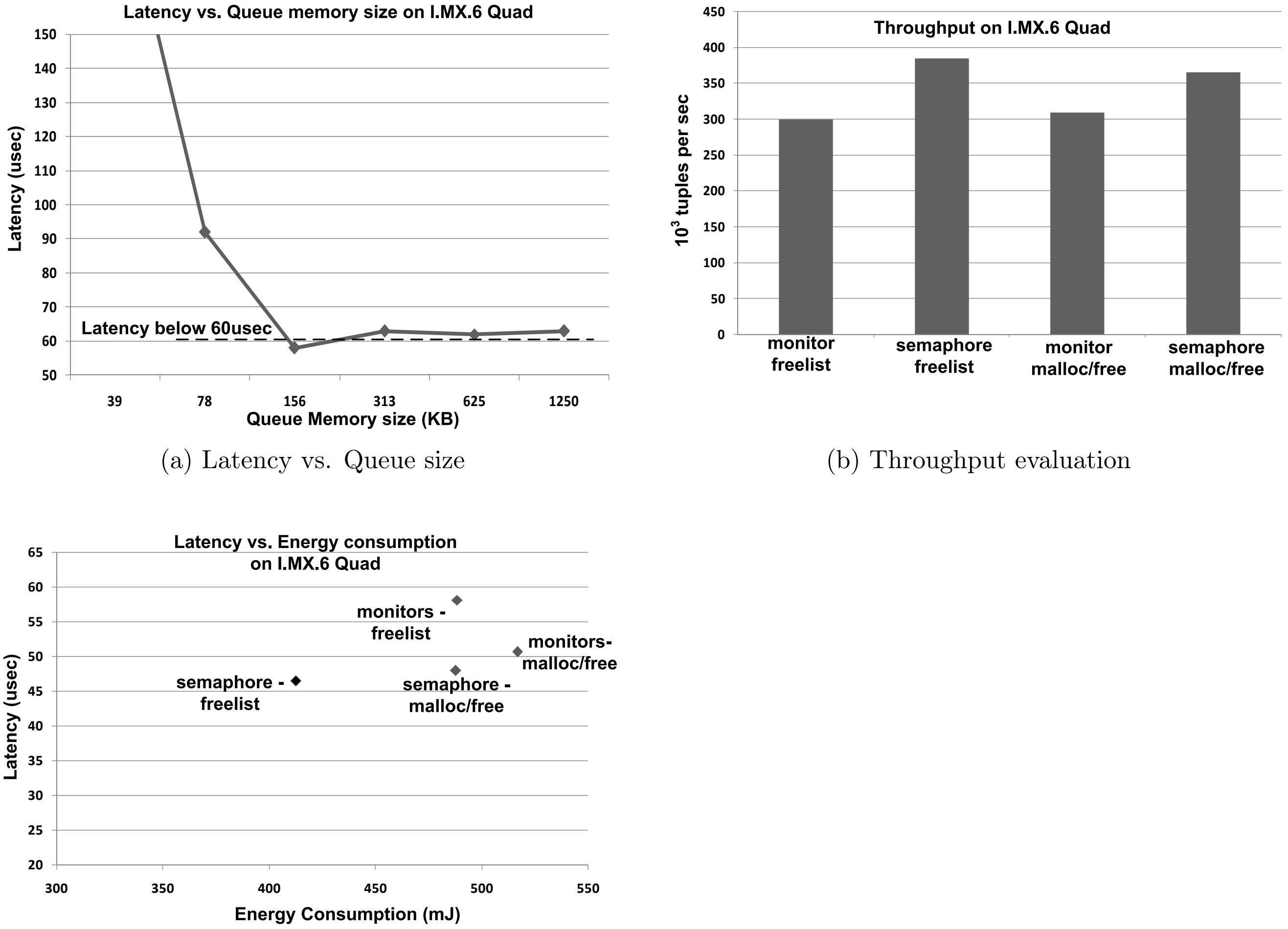
In the second experiment, in which the windows data structure is placed in global memory due to its increased memory size, latency constraint is set to 62usec (Fig. 21b) and throughput vs. memory footprint results are presented in Fig. 22d. 4 Pareto efficient points have been identified that provide throughput vs. memory size trade-offs (up to 6.4% for throughput and up to 3.07% for latency). Correspondingly, the 5 Pareto efficient points in latency vs. energy consumption evaluation displayed in Fig. 22e can be used for performing trade-offs, up to 8.59% for latency (P20) and 18% for energy (P16). Scalability results in Fig. 22f are slightly different from the ones in the previous experiment. Implementations scale up to 8 inputs and most of them tend to provide slightly lower throughput when 9 inputs are used.

3.5.3.3 Demonstration on I.MX.6 Quad

Few customized implementations exist for I.MX.6, since the operating system handles many design options. In the I.MX.6 Quad experiment latency threshold has been set to 60usec and a single effective queue size has been found: 156KB (Fig. 23a). 4 customized implementations have been evaluated and throughput results are shown in Fig. 23b, while latency vs. energy results are displayed in Fig. 23c. We notice that the most efficient implementation in terms of both throughput, latency and energy is the one that utilizes semaphores for synchronization, along with freelist-based memory management.

3.5.4 Count-based aggregation results

In the count-based scenario, we evaluate each implementation for different window sizes. The selected values are provided to the first step of the methodology. 24 different implementations are evaluated in each platform.



(a) Latency vs. Queue size

(b) Throughput evaluation

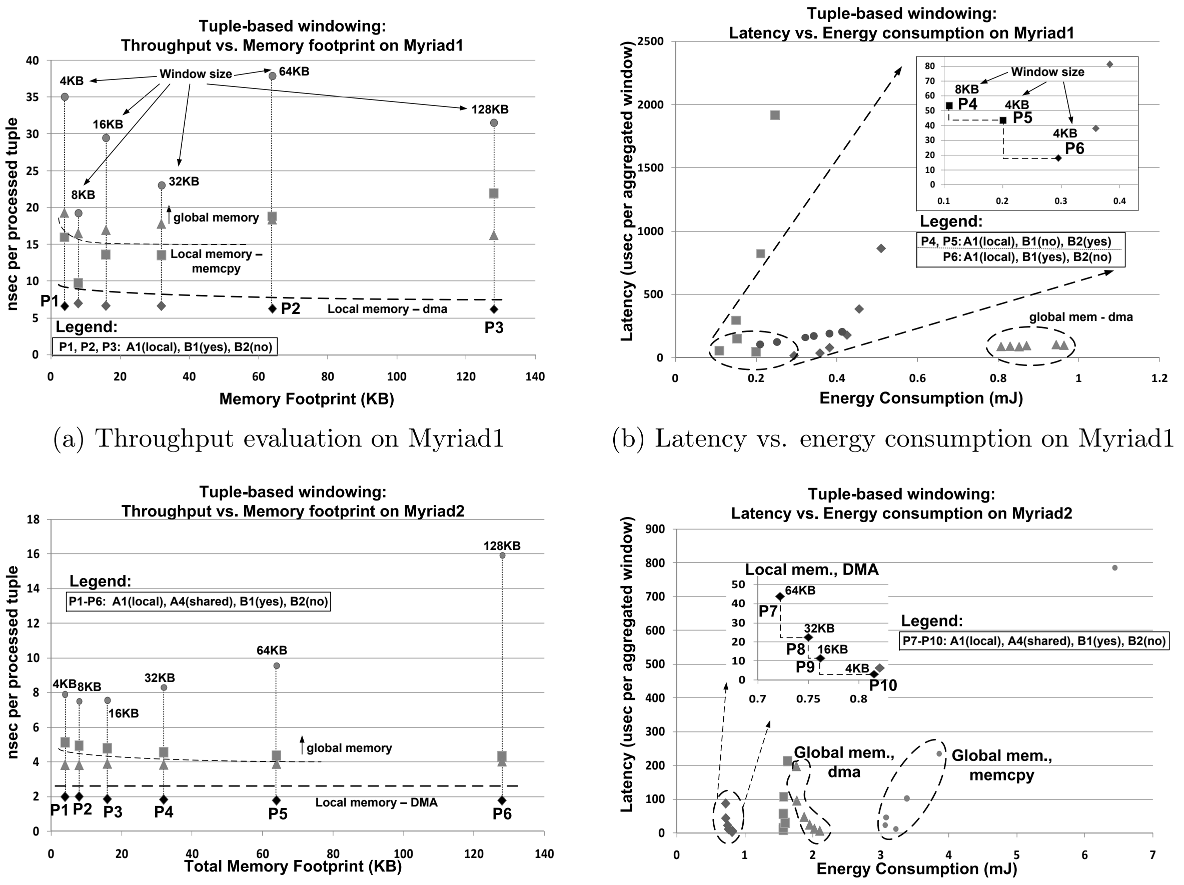
(c) Latency vs. energy consumption

Figure 23: Evaluation results of time-based streaming aggregation implementations on I.MX.6 Quad.

3.5.4.1 Demonstration on Myriad1

Fig. 24a shows throughput vs. memory footprint on Myriad1. Implementations that process tuples in local memory and transfer data from global to local memory through DMA provide higher throughput. For instance, at 4KB window size, P1 provides 58% higher throughput than the implementation that uses *memcpy* for data transfer.

Latency vs. energy consumption results are presented in Fig. 24b. We notice that smaller windows provide lower latency. Also, transferring tuples in local memories provides lower latency than processing windows in Myriad1 global memory. 3 Pareto points are identified that provide trade-offs between latency and energy consumption.



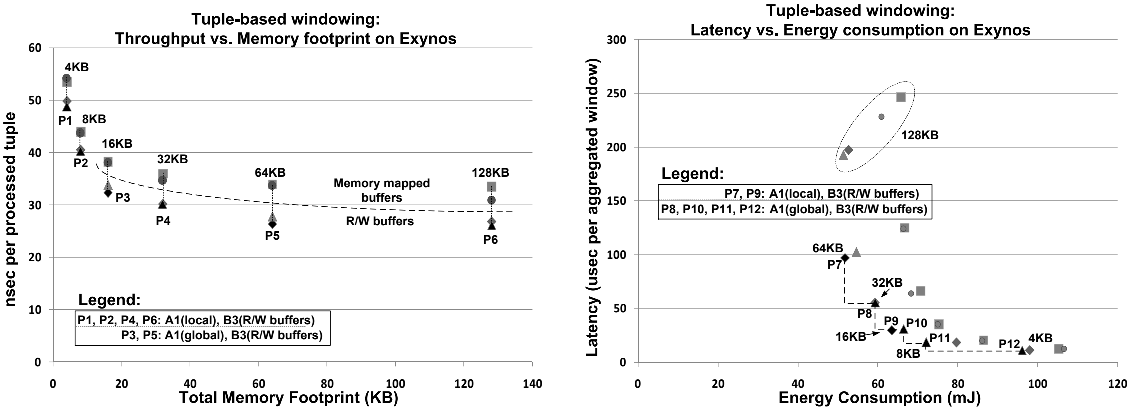
(a) Throughput evaluation on Myriad1

(b) Latency vs. energy consumption on Myriad1

(c) Throughput evaluation on Myriad2

(d) Latency vs. energy consumption on Myriad2

Figure 24: Evaluation results of count-based streaming aggregation implementations.



(a) Throughput evaluation on Exynos

(b) Latency vs. energy consumption on Exynos

Figure 25: Evaluation results of count-based streaming aggregation implementations.

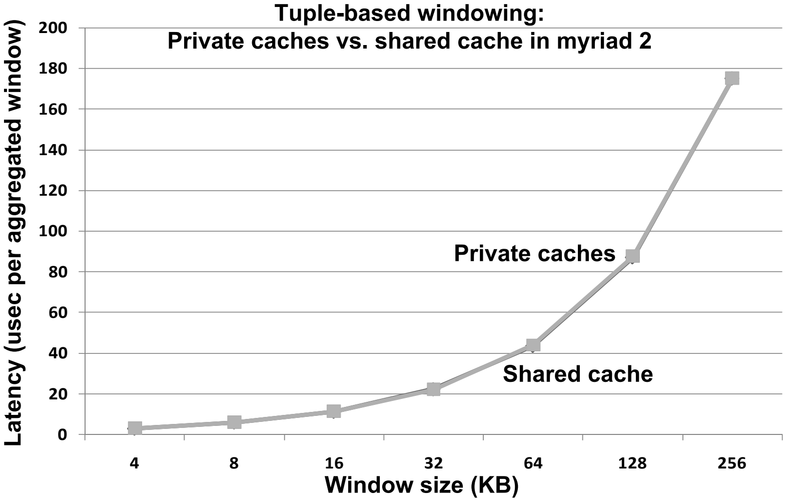


Figure 26: Latency vs. window size on Myriad2 for count-based streaming aggregation.

3.5.4.2 Demonstration on Myriad2

In Myriad2, we first evaluate latency vs. window size for two different cache configurations. As shown in Fig. 26, utilization of shared cache provides slightly lower latency than private caches (less than 1%). Therefore, implementations that utilize private caches are pruned and the design space is reduced.

As in Myriad1, implementations that provide higher throughput are the ones in which tuples are transferred through DMA and processed in local memory. Fig. 24c shows that throughput increases up to 59% using the aforementioned implementation, in comparison with the implementation in which tuples are processed in global memory, with window size 64KB. Also, we notice that larger windows provide slightly higher throughput. For instance, increasing window size from 4KB to 128KB, yields throughput increase about 10% (P1 to P6).

Implementations that utilize local memory and DMA transfers provide both low latency and energy efficiency, as shown in Fig. 24d. Processing in global or in local memory affects both latency and energy consumption results. For instance, tuples in local memory and utilization of DMA with 4KB window size provides 31.4% lower energy consumption than the corresponding implementation with tuple processing in global memory.

3.5.4.3 Demonstration on Exynos 5

Throughput vs. memory footprint results are displayed in Fig. 25a. Larger window sizes provide higher throughput. Implementations that utilize R/W buffers yield higher performance than corresponding implementations with memory mapped data buffers: up to 21% for 64KB window size.

Regarding latency vs. energy consumption, displayed in Fig. 25b, 6 Pareto points are identified. Smaller window sizes provide lower latency, but higher energy consumption, due to the increased rate of data transfers. Utilization of R/W buffers is more efficient than memory mapped ones, both in terms of latency and energy consumption. Due to the relatively small buffer size, the overhead of utilizing R/W buffers is also small.

3.5.5 Performance per watt evaluation

One of the goals of this work is to compare performance per watt of streaming aggregation mapped on low power embedded platforms with the corresponding results on an HPC CPU and a GPGPU. In this subsection, we first provide details on the implementation of the operator on the aforementioned platforms and then we present the evaluation results.

We implemented the time-based streaming aggregation scenario on an Intel Xeon E5 CPU with 8 cores operating at 3.4GHz, with 16GB RAM, running Ubuntu Linux 12.04. Compiler is gcc v.4.9.2 and optimization flag is *-O3*. Power consumption was measured through hardware instrumentation and refers to dynamic CPU Power. Throughput and latency were measured similarly to the embedded implementations. Data transfer was based on *memcpy()* operations and synchronization based on semaphores.

The results are presented on Table 8. The values for Myriad1, Myriad2 and I.MX.6 correspond to the implementation that provides the best results for each specific metric. To ensure fair comparison, all values for all platforms utilize 5 input queues. Performance per watt is calculated as number of tuples forwarded per second, per watt.

In Table 8, we notice that in terms of performance, latency on Intel Xeon is 62.3% lower than in Myriad2, while it is 3.8 and 9.3 times lower than in I.MX.6 and Myriad1, respectively. In terms of throughput, Xeon provides more than two times higher throughput than Myriad2, 2.8 than I.MX.6 and 8.3 times higher than Myriad1. The high performance of Intel Xeon is related with the higher computational power it provides and the fact that it operates in much higher frequency than the embedded architectures. However, in terms of performance per watt, embedded platforms outperform Intel Xeon. Because the Myriad processors consume very low power, they achieve higher performance per watt: 54 times higher in Myriad2, while in Myriad1 it is 20 times higher. Finally, I.MX.6 provides 24 times higher performance per watt in comparison with Intel Xeon.

Count-based aggregation scenario was implemented in OpenCL 1.1 and evaluated in AMD Radeon HD 6450 general purpose GPU [1]. The host runs Ubuntu Linux 12.04 with gcc v.4.9.2. Throughput and latency were measured similarly to the corresponding embedded implementations, while power consumption is estimated based on GPU's specifications. Device data accessing is based on R/W buffers.

The results are presented in Table 9. Embedded platforms provide lower throughput and higher latency than Radeon GPGPU. However, both Myriad boards yield higher performance per watt than GPGPU, due to the very low power that they require. More specifically, Myriad2 provides about 14 higher performance per watt, while Myriad1 7 times.

Table 8: Time-based streaming aggregation: Comparison between latency, throughput and performance per watt on embedded and Intel Xeon architectures.

	Latency (usec)	Throughput (t/sec)	(t/sec)/watt
Myriad1	140.38	132,622	379,041
Myriad2	39.8	497,154	1,004,766
I.MX.6	58	384,952	446,787
Xeon	15	1,105,221	18,412

Table 9: Count-based streaming aggregation: Comparison between latency, throughput and performance per watt on embedded and Radeon HD 6450.

	Latency (usec)	Throughput (Mt/sec)	(Mt/sec)/watt
Myriad1	17.98	151.8	593
Myriad2	3.04	505.4	1286
Exynos	7.5	47.4	7.93
GPGPU	1.94	2576.3	85.87

3.5.6 Discussion of Experimental Results

In this subsection we summarize the conclusions we draw from the demonstration of the methodology that is presented in the previous subsections. The trade-offs we demonstrated in the experimental results can be used to draw conclusions about the relation between the customization options and the evaluation metrics.

3.5.6.1 Time-based streaming aggregation conclusions

Observation 1: Streaming aggregation should be customized differently, not only between I.MX.6 Quad and Myriad architectures, but also between Myriad1 and Myriad2.

For example, in Myriad1, in the first experiment, in the implementation that provides the lowest latency, data transfer is based on hardware buffers. On the contrary, in Myriad2 it is based on busy waiting mechanism. In the implementation that provides the highest throughput, the queue is 256B in Myriad1, while it is 512B in Myriad2.

Observation 2: There is a threshold in the queue size, below which latency is very high. Very large queue sizes may also negatively affect latency.

We notice that in both Myriad and I.MX.6, latency is very high for small queue sizes, which is due to the high overhead of constantly fetching data for refilling the queues with new tuples. In these cases, the thread that executes the merge-sort phase, often finds the queues to be empty. As the queue size increases latency drops drastically. However, in Myriad1 and

Myriad2 experiments, we notice that as the queue size increases, latency tends to increase, as well (Fig. 18 and Fig. 21). The reason is the fact that the larger the queue, the more cycles it takes to complete a DMA transfer of data from the DDR to the local memory and start refilling the queue with new tuples. Thus, the tuples that entered the update phase before a new DMA transfer and exit the output phase after it, they have higher latency than the rest ones. In contrast with Myriad, on I.MX.6 we can use much bigger queues, since the available memory is much larger. However, beyond a specific queue size, throughput and latency on I.MX.6 do not seem to be significantly affected any more (Fig. 23a).

Observation 3: Throughput is mainly affected by either the data transfer mechanism (in Myriad2) or by the signaling mechanism (in Myriad1).

In general, in Myriad1 and Myriad2, throughput drops when the queue size becomes smaller, due to overhead of the DMA transfers, which is added more frequently when the queues are small (e.g. Fig. 19a and Fig. 22a). However, latency becomes lower in that case, as stated earlier. In Myriad2, throughput is mainly determined by whether *memcpy* or DMA data transfer mechanism is used. Indeed, data transfer options seem to have major impact on throughput (Fig. 22a and Fig. 22d). On the other hand, in Myriad1 the utilization of hardware buffer or of busy waiting scheme is the dominant factor that affects throughput (Fig. 19a and Fig. 19d). In Myriad2 signaling design options have much lower impact in comparison with data transfer options. On the contrary, in Myriad1, data transfer mechanism has relatively small effect on throughput in comparison with the signaling mechanism (*memcpy* however is slightly more efficient). In I.MX.6, the utilization of freelists to avoid the frequent system calls improves throughput and latency results. However, the main factor that improves performance is the utilization of semaphores instead of monitors (Fig. 23b).

Observation 4: Latency is affected by the synchronization mechanism. Different mechanism should be used in Myriad1 than in Myriad2.

The synchronization mechanism is the main design option that affects latency and energy in both Myriad architectures. Busy waiting mechanism provides lower latency in Myriad2 and slightly lower energy consumption. On the contrary, the utilization of hardware buffers in Myriad1 is more efficient it terms of latency. The data transfer mechanism has much lower impact in both architectures in terms of latency and energy.

Observation 5: The frequency by which data movements are performed from global to local memory affects energy consumption in Myriad. We notice that larger queue sizes are more energy efficient in both Myriad1 and Myriad2, due to the lower rate by which data are fetched in the local memory (e.g. Fig. 19b and Fig. 22b). On I.MX.6 Quad, energy consumption is determined mainly the by synchronization scheme that it is used.

Finally, an interesting observation is the fact that the memory allocation of the input queues affects neither the performance nor the energy consumption in Myriad significantly. The reason is the fact that both Myriad architectures provide cache memory and the rate of cache misses for accessing the queues by the PE that performs the merge-sort operation is relatively small. On the other hand, the allocation of the windows data structure in global memory has major impact in both performance and energy consumption. For instance, in Myriad2, by allocating the windows data structure in global memory, latency increases about 9%, throughput drops by 7% and energy consumption increases by 20% in comparison with the allocation in local memory.

The above observations can be used to draw more general conclusions on how the streaming aggregation should be customized on embedded platforms. When the optimization target is performance, the following considerations should be taken into account:

- The queue size should be large enough to decrease the rate by which data transfers are instructed. Frequent small data transfers lower performance. However, for implementations that are very sensitive to latency, it should be noted that too large queue sizes may increase latency.

- Window *size* and *advance* values affect a lot the maximum size of the windows data structure and therefore the memory allocation design options and the performance. Platforms with very small local memory may be not suitable for implementing streaming aggregation, since they would limit the window configuration values that can be used, if allocation of the data structure in global memory is not a option, due to very strict performance requirements.

- Platform-specific options for efficient communication between cores (such as the hardware buffers on Myriad) should be evaluated, when the streaming aggregation is implemented at low level. In some cases (such as in Myriad1) they can provide increased performance.

On the other hand, if the main goal is energy efficiency, the following issues should be considered:

- The queues should be as large as possible to avoid the energy consumption overhead of frequent small data transfers.

- For window *size* and *advance* values apply the same that are stated earlier: Window configuration that forces the allocation of the windows data structure in global memory has negative impact in energy consumption.

- Finally, developers should try to evaluate features that set the PEs in a low-energy mode when they are forced to wait (such as the hardware buffers in Myriad1).

3.5.6.2 Count-based streaming aggregation conclusions

Observation 1: Both throughput and latency in Myriad implementations are affected by the memory allocation of the processed tuples. In Exynos implementations, they are mainly affected by the data accessing method by the device.

In general, throughput is apparently affected by the window size. Apart from that, design choices such as the allocation of the window in local memory and R/W buffers in OpenCL implementations, yield increased throughput.

In contrast with throughput, smaller window sizes provide lower latency. Implementations in which tuples are processed in local memories in Myriad and utilize R/W buffers in mobile GPU provide the lowest latency.

Observation 2: Energy consumption is mainly affected by the memory allocation and the window size.

Energy consumption in Myriad is affected by both the type of memory in which tuples are processed and the size of the window (Fig. 24d). In Exynos, window size has the highest impact in energy (Fig. 25b). Since the rate of data transfers is increased when smaller windows are used, energy consumption is also increased.

To summarize, when the optimization target is performance, DMA transfers and R/W OpenCL buffers provide higher throughput than the rest of the design choices. Large windows yield increased throughput, while smaller ones provide low latency. Finally, window sizes that allow processing in local memory benefit both performance and energy.

The methodology we propose in this work provides a systematic approach to the efficient customization of the streaming aggregation on embedded platforms. Instead of trying to tune the application and hardware parameters arbitrary to achieve the desired results, the proposed methodology provides a set of customization solutions from which developers can select the one that is more suitable according the design constraints.

Finally, it is important to state that the methodology is not fundamentally limited to streaming aggregation. The design space could be adapted to be applicable to other streaming operators, as well (such as join, filter etc.) and to embedded platforms with various other features. New attributes can be integrated in the design space for exploration as new decision trees, leaves or categories. The application and hardware constraints should be updated accordingly to retain the coherency of the customized implementations.

3.6 Conclusion

We proposed a customization methodology for the implementation of streaming aggregation in modern embedded devices. The methodology was demonstrated in 4 different embedded architectures, 2 aggregation scenarios and a real-world data set. The customized implementations provided by the methodology can be utilized by developers to perform trade-offs between several parameters, taking into consideration the design constraints that are imposed by both the application requirements and the embedded architecture. In the future, we intend to extend the design space by integrating more streaming aggregation operators and evaluate the approach in embedded platforms with various features.

4 Energy Model on CPU for Lock-free Data-structures in Dynamic Environments

4.1 Introduction

Here, we consider the modeling and the analysis of the performance of lock-free data structures. Then, we combine the perfomance analysis with our power model that is introduced in D2.1 [75] and D2.3 [73] to estimate the energy efficiency of lock-free data structures that are used in various settings.

Lock-free data structures are based on retry loops and are called by application-specific routines. In contrast to the model and analysis provided in D2.3, we consider here the lock-free data structures in dynamic environments. The size of each of the retry loops, and the size of the application routines invoked in between, are not constant but may change dynamically.

During the last two decades, lock-free data structures have received a lot of attention in the literature, and have been accepted in industrial applications, *e.g.* in the Intel's Threading Building Blocks Framework [87], the Java concurrency package [4] and the Microsoft .NET Framework [5]. Lock-free implementations provide indeed a way out of several limitations of their lock-based counterparts, in robustness, availability and programming flexibility. Last but not least, the advent of multi-core processors has pushed lock-freedom on top of the toolbox for achieving scalable synchronization.

Naturally, the development of lock-free data structures was accompanied by studies on the performance of such data structures, in order to characterize their scalability. Having no guarantee on the execution time of an individual operation, the time complexity analyses of lock-free algorithms have turned towards amortized analyses. The so-called amortized analyses are thus interested in the worst-case behavior over a sequence of operations, which can be seen as a worst-case bound on the average time per operation. In order to cover various contention environments, the time complexity of the algorithms is often parametrized by different contention measures, such as point [23], interval [14] or step [24] contention. Nonetheless these investigations are targeting worst-case asymptotic behaviors. There is a lack of analytical results in the literature capable of describing the execution of lock-free algorithms on top of a hardware platform, and providing predictions that are close to what is observed in practice. Asymptotic bounds are particularly useful to rank different algorithms, since they rely on a strong theoretical background, but the presence of potentially high constants might produce misleading results. Yet, an absolute prediction of the performance can be of great importance by constituting the first step for further optimizations.

The common measure of performance for data structures is throughput, defined as the number of operations on the data structure per unit of time. To this end, this performance measure is usually obtained by considering an algorithm that strings together a pure sequence of calls to an operation on the data structure. However, when used in a more realistic context, the calls to the operations are mixed with application-specific code (that we call here parallel work). For instance, in a work-stealing environment designed with deques, a thread basically runs one of the following actions: pushing a new-generated task in its deque, popping a task from a deque or executing a task. The modifications on the deques are thus interleaved with deque-independent work. There exist some papers that consider in their experiments local computations between calls to operations during their respective evaluations, but the amount of local computations follows a given distribution varying from paper to paper, *e.g.* constant [107], uniform [81], exponential [135].

In this work, we derive a general approach for unknown distributions of the size of the application-specific code, as well as a tighter method when it follows an exponential distribution.

As for modeling the data structure itself, we use as a basis the universal construction described by Herlihy in [82], where it is shown that any abstract data type can get such a lock-free implementation, which relies on one retry loop. Moreover, we have particularly focused our experiments on data structures that present a low level of disjoint-access parallelism [89] (stack, queue, shared counter, deque). Coming back to amortized analyses, the time complexity of an operation is often expressed as a contention-free time complexity added with a contention overhead. In this work, we want to model and analyze the impact of contention, whether nonexistent, mediocre or high. So that the contention overhead is not hidden, we focus on data structures with low contention-free complexity, that can also provide very high contention without bringing hundreds of threads into play.

We propose two different approaches that analyze the performance of such data structures. On the one hand, we derive an average-based approach invoking queuing theory, which provides the throughput of a lock-free algorithm without any knowledge about the distribution of the parallel work. This approach is flexible but allows only a coarse-grained analysis, and hence a partial knowledge of the contention that stresses the data structure. On the other hand, we exhibit a detailed picture of the execution of the algorithm when the parallel work is instantiated with an exponential distribution, through a second complementary approach. We prove that the multi-threaded execution follows a Markovian process and a Markov chain analysis allows us to pursue and reconstruct the execution, and to compute a more accurate throughput.

We finally show several ways to use our analyses and we evaluate the validity of our ideas by experimental results. Those two analysis approaches give a good understanding of the phenomena that drive the performance of a lock-free data structure, at a high-level for the average-based approach, and at a detailed level for the constructive method. We also emphasize that there exist several concrete paths to apply our analyses. To this end, based on the knowledge about the application at hand, we implement two back-off strategies. We show the applicability of these strategies by tuning a Delaunay triangulation application [62] and a streaming pipeline component which is fed with trade exchange workloads [2]. We also design a new adaptive memory management mechanism for lock-free data structures in dynamic environments which surpasses the traditional scheme and which is such that the loss in performance, when compared to a static data structure without memory management, is largely leveraged. This memory management mechanism is based on the analyses presented in this work.

Lastly, we show how these results can be used to obtain the energy consumption of the lock-free data structures.

The rest is organized as follows: we start by presenting related work in Section 4.2, then we define the algorithm and the platform that we consider, together with concepts that are common to our both approaches in Section 4.3. The average-based approach is described in Section 4.4, while the constructive analysis is exposed in Section 4.5, both methods are evaluated in the experiment part that is presented in Section 4.6 and the energy model with the evaluations is given in Section 4.9.

4.2 Previous Work

In D2.3, performance impacting factors are illustrated for a subset of the lock-free structures that we consider in this work. In the former paper, the analysis is built upon properties that arise only when the sizes of the critical work and the parallel work are constant. There, we show that the execution is not memoryless due to the natural synchrony provided by the retry loops; at the end of the line, we prove that the execution is cyclic and use this property to bound the rate of failed retries. This work is complementary to that work, not only because of the difference in the analysis tools but also because they altogether exhibit the impact of the size distributions of the parallel work on the performance of lock-free data structures. Moreover, owing to our assumptions on the size of the parallel and critical works, the results of this paper can be applied to a larger variety of data structures running on a larger variety of environments.

4.3 Preliminaries

We describe in this subsection the structure of the algorithm that is covered by our model. We explain how to analyze the execution of an instance of such an algorithm when executed by several threads, by slicing this execution into a sequence of adjacent success periods, where a success period is an interval of time during which exactly one operation returns. Each of the success periods is further split into two by the first access to the data structure in the considered retry loop. This execution pattern reflects fundamental phases of both analyses, whose first steps and general direction are outlined at the end of the subsection.

4.3.1 System Settings

All threads call Procedure AbstractAlgorithm (see Figure 27) when they are spawned. So each thread follows a simple though expressive pattern: a sequence of calls to an operation on the data structure, interleaved with some parallel work during which the thread does not try to modify the data structure. For instance, it can represent a work-stealing algorithm, as described in the introduction.

The algorithm is decomposed in two main sections: the *parallel section*, represented on line 2, and the *retry loop* (which represents one operation on the shared data structure) from line 3 to line 6. A *retry* starts at line 4 and ends at line 6. The outer loop that goes from line 1 to line 6 is designated as the *work loop*.

In each retry, a thread tries to modify the data structure and does not exit the retry loop until it has successfully modified the data structure. It firstly reads the access point AP of the data structure, then, according to the value that has been read, and possibly to other previous computations that occurred in the past, the thread prepares, during the critical work, 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 *CAS* primitive. If it succeeds, *i.e.* if the access point has not been changed by another thread between the first *Read* and the *CAS*, then it goes to the next parallel section, otherwise it repeats the process. The retry loop is composed of at least one retry (and the first iteration of the retry loop is strictly speaking not a retry, but a try).

We denote by cc the execution time of a *CAS* when the executing thread does not own the cache line in exclusive mode, in a setting where all threads share a last level cache. Typically, there exists a thread that touches the data between two requests of the same thread, therefore this cost is paid at every occurrence of a *CAS*. As for the *Read*s, rc holds for the execution time of a cache miss. When a thread executes a failed *CAS*, it immediately reads the same cache line (at the beginning of the next retry), so the cache line is not missing, and the execution time of the *Read* is considered as null. However, when the thread comes back from the parallel section, a cache miss is paid. To conclude with the parameters related to the platform, we dispose of P cores, where the *CAS* (resp. the *Read*) latency is identical for all cores, *i.e.* cc (resp. rc) is constant.

The algorithm is parametrized by two execution times. In the general case, the execution time of an occurrence of the parallel section (application-specific section) is a random variable that follows an unknown probability distribution. In the same way, the execution time of the critical work (specific to a data structure) can vary while following an unknown probability distribution. The only provided information is the mean value of those two execution times: cw for the critical work, and pw for the parallel work. These values will be given in units of work, where 1 u.o.w. = 50 cycles.

4.3.2 Execution Description

It has been underlined in [73] that there are two main conflicts that degrade the performance of the data structures which do not offer a great degree of disjoint-access parallelism: logical and hardware conflicts.

Logical conflicts occur when there are more than one thread in the retry loop at a given time (happens typically when the number of threads is high or when the parallel section is small). At any time, considering only the threads that are in the retry loop, there is indeed at most one thread whose retry will be successful (*i.e.* whose ending *CAS* will succeed), which implies the execution of more retries for the failing threads. In addition, after a thread executes successfully its final *CAS*, the other threads of the retry loop have first to finish their current retry before starting a potentially successful retry, since they are not informed yet that their current retry is doomed to failure. This creates some “holes” in the execution

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

Figure 27: Thread procedure

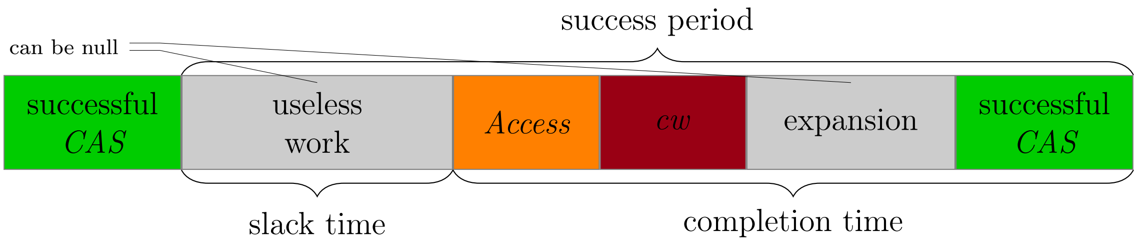


Figure 28: Success Period

where all threads are executing useless work.

The threads will also experience *hardware conflicts*: if several threads are requesting for the same data, so that they can operate a *CAS* on it, a single thread will be satisfied. All the other threads will have to wait until the current *CAS* is finished, and give a new try when this *CAS* is done. While waiting for the ownership of the cache line, the requesting threads cannot perform any useful work. This waiting time is referred to as *expansion*.

We now refine the description of the execution of the algorithm. The timeline is initially decomposed into a sequence of success periods that will define the throughput. A success period is an interval of time of the execution that (i) starts after a successful *CAS*, (ii) contains a single successful *CAS*, (iii) finishes after this successful *CAS*. As explained in the previous subsection, to be successful in its retry, a thread has first to access the data structure, then modify it locally, and finally execute a *CAS*, while no other thread performs changes on the data structure. That is why each success period is further cut into two main phases (see Figure 28). During the first phase, whose duration is called the *slack time*, no thread is accessing the data structure. The second phase, characterized by the *completion time*, starts with the first access to the data structure (by any thread). Note that this *Access* could be either a *Read* (if the concerned thread just exited the parallel section) or a failed *CAS* (if the thread was already in the retry loop). The next successful *CAS* will come at least after *cw* (one thread has to traverse the critical work anyway), that is why we split the latter phase into: *cw*, then expansion, and finally a successful *CAS*.

4.3.3 Our Approaches

In this work, we propose two different approaches to compute the throughput of a lock-free algorithm, which we name as average-based and constructive. The average-based approach relies on queuing theory and is focused on the average behavior of the algorithm: the throughput is obtained through the computation of the expectation of the success period at a random time. As for the constructive approach, it describes precisely the instants of accesses and modifications to the data structure in each success period: in this way, we are able to deconstruct and reconstruct the execution, according to observed events. The constructive approach leads to a more accurate prediction at the expense of requiring more information about the algorithm: the distribution functions of the critical and parallel works have indeed to be instantiated.

In both cases, we partition the domain space into different levels of contention (or *modes*); these partitions are independent across approaches, even if we expect similarities, but in each case, cover the whole domain space (all values of critical work, parallel work and number of threads).

4.3.4 Average-based Analysis

We distinguish two main modes in which the algorithm can run: contended and non-contended. In the non-contended mode, *i.e.* when the parallel work is large or the number of threads is low, concurrent operations are not likely to collide. So every retry loop will count a single retry, and atomic primitives will not delay each other. In the contended mode, any operation is likely to experience unsuccessful retries before succeeding (logical conflicts), and a retry will last longer than in the non-contended mode because of the collision of atomic primitives (hardware conflicts).

Once all the parameters are given, the analysis is centered around the calculation of a single variable $\overline{P_{rl}}$, which represents the expectation of the number of threads inside the retry loop at a random instant. Based on this variable, we are able to express the expected expansion $\overline{e}\left(\overline{P_{rl}}\right)$ at a random time. As a next step, we show how this expansion can be used to estimate the expected slack time $\overline{st}\left(\overline{P_{rl}}\right)$ and the expected completion time $\overline{ct}\left(\overline{P_{rl}}\right)$, and at the end, the expected time of a success period $\overline{sp}\left(\overline{P_{rl}}\right)$.

4.3.5 Constructive Method

The previous average-based reasoning is founded on expected values at a random time, while in the constructive approach, we study each success period individually, based on the number of threads at the beginning of the considered success period. So we are able to exhibit more clearly the instants of occurrences of the different accesses and modifications to the data structure, and thus to predict the throughput more accurately.

We rely on the same set of values used in the average-based approach, but these values are now associated with a given success period. Thus the number of threads inside the retry loop P_{rl}, as well as the slack time and the completion time are evaluated at the beginning of each success period. We denote these times in the same way as in the first approach, but remove the bar on top since these values are not expectations any more.

The different contention modes do not characterize here the steady-state of the data structure as in the previous approach but are associated with the current success period. Accordingly, the contention can oscillate through different modes in the course of the execution. First, a success period is not contended when $P_{rl} = 0$, *i.e.* when there is no thread in the retry loop after a successful *CAS*. In this case, the first thread that exits the parallel section will be successful, and the *Access* of the sequence will be a *Read*. Second, the contention of a success period is high when at any time during the success period, there exists a thread that is executing a *CAS*. In other words, at the end of each *CAS*, there is at least one thread that is waiting for the cache line to operate a *CAS* on it. This implies that the first access of the success period is a *CAS* and occurs immediately after the preceding successful *CAS*: the slack time is null. Third, the medium contention mode takes place when $P_{rl} > 0$, while at the same time, there are not enough requesting threads to fill the whole success period with *CAS*'s (which implies a non-null slack time). Since these requesting threads have synchronized in the previous success period, *CAS*'s do not collide in the current success period, and because of that, the expansion is null.

4.4 Average-based Approach

We propose in this section our coarse-grained analysis to predict the performance of lock-free data structures. Our approach utilizes fundamental queuing theory techniques, describing the average behavior of the algorithm. In turn, we need only a minimal knowledge about the algorithm: the mean execution time values *cw* and *pw*. As explained in Section 4.3.4, the system runs in one of the two possible modes: either contended or uncontended.

4.4.1 Contended System

We first consider a system that is contended. When the system is contended, we use Little's law to obtain, at a random time, the expectation of the success period, which is the interval of time between the last and the next successful *CAS*'s (see Figure 28).

The stable system that we observe is the parallel section: threads are entering it (after exiting a successful retry loop) at an average rate, stay inside, then leave (while entering a new retry loop). The average number of threads inside the parallel section is $\overline{P_{ps}} = P - \overline{P_{rl}}$, each thread stays for an average duration of pw, and in average, one thread is exiting the retry loop every success period $\overline{sp}\left(\overline{P_{rl}}\right)$, by definition of the success period. According to Little's law [103], we have:

$$\overline{P_{ps}} = pw \times \frac{1}{\overline{sp}\left(\overline{P_{rl}}\right)}, \ \textit{i.e.}$$

$$\frac{1}{pw} \times \overline{sp}\left(\overline{P_{rl}}\right) = \frac{1}{P - \overline{P_{rl}}} \tag{1}$$

As explained in Section 4.3.2, we further decompose a success period into two parts, separated by the first access to the data structure after a successful *CAS*. We can then write the average success period as the sum of: (i) the expected time before some thread starts its *Access* (the slack time), and (ii) the expected completion time. We compute these two expectations independently and gather them into the success period thanks to:

$$\overline{sp}\left(\overline{P_{rl}}\right) = \overline{st}\left(\overline{P_{rl}}\right) + \overline{ct}\left(\overline{P_{rl}}\right). \tag{2}$$

When the data structure is contended, a thread is likely to be successful after some failed retries. Therefore a thread that is successful was already in the retry loop when the previous successful *CAS* occurred. This implies that the *Access* to the data structure will be due to a failed *CAS*, instead of a *Read*.The time before a thread starts its *Access* is then the time before a thread finishes its current critical work since there is a thread currently executing a *CAS*.

4.4.2 Expected Completion time

Since the data structure is contended, numerous threads are inside the retry loop, and, due to hardware conflicts, a retry can experience expansion: the more threads inside the retry loop, the longer time between a *CAS* request and the actual execution of this *CAS*. The expectation of the completion time can be written as:

$$\overline{ct}\left(\overline{P_{rl}}\right) = cc + cw + \overline{e}\left(\overline{P_{rl}}\right) + cc, \tag{3}$$

where $\overline{e}\left(\overline{P_{rl}}\right)$ is the expectation of expansion when there are $\overline{P_{rl}}$ threads inside the retry loop, in expectation. This expansion can be computed in the same way as in [73], through the following differential equation:

$$\begin{cases} \overline{e}'\left(\overline{P_{rl}}\right) & = & cc \times \dfrac{\frac{cc}{2} + \overline{e}\left(\overline{P_{rl}}\right)}{cc + cw + cc + \overline{e}\left(\overline{P_{rl}}\right)} \\ \overline{e}\left(1\right) & = & 0 \end{cases},$$

by assuming that the expansion starts as soon as strictly more than 1 thread are in the retry loop, in expectation.

4.4.3 Expected Slack Time

Concerning the slack time, we consider that, at any time, the threads that are running the retry loop have the same probability to be anywhere in their current retry. However, when a thread is currently executing a *CAS*, the other threads cannot execute as well a *CAS*. The other threads are thus in their critical work or expansion. For every thread, the time before accessing the data structure is then uniformly distributed between 0 and $cw + \overline{e}\left(\overline{P_{rl}}\right)$.

According to Lemma 1, we conclude that

$$\overline{st}\left(\overline{P_{rl}}\right) = \left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) / (\overline{P_{rl}} + 1). \tag{4}$$

Lemma 1. *Let an integer n, a real positive number a, and n independent random variables $X_1, X_2, \ldots, X_n$, uniformly distributed within $[0, a[$. Let then X be the random variable defined by: $X = \min_{i \in [\![1,n]\!]} X_i$. The expectation of X is:*

$$\mathbb{E}\left(X\right) = \frac{a}{n+1}.$$

Proof. Let a positive real number x be such that $x < a$. We have

$$\begin{aligned}\mathbb{P}\left(X > x\right) &= \mathbb{P}\left(\forall i : X_i > x\right) \\ &= \prod_{i=1}^{n} \mathbb{P}\left(X_i > x\right) \\ \mathbb{P}\left(X > x\right) &= \left(\frac{a-x}{a}\right)^n\end{aligned}$$

Therefore, the probability distribution of X is given by:

$$t \mapsto \frac{n}{a}\left(\frac{a-x}{a}\right)^{n-1},$$

and its expectation is computed through

$$\begin{aligned}\mathbb{E}\left(X\right) &= \frac{n}{a}\int_0^a x \times \left(\frac{a-x}{a}\right)^{n-1} dx \\ &= \frac{n}{a}\int_0^a (a-u) \times \left(\frac{u}{a}\right)^{n-1} du \\ &= \frac{n}{a^n}\int_0^a (a-u) \times u^{n-1}\, du \\ &= \frac{n}{a^n}\left(a \times \frac{a^n}{n} - \frac{a^{n+1}}{n+1}\right) \\ \mathbb{E}\left(X\right) &= \frac{a}{n+1}.\end{aligned}$$

□

4.4.4 Expected Success Period

We just have to combine Equations 2, 3, and 4 to obtain the general expression of the expected success period:

$$\overline{sp}\left(\overline{P_{rl}}\right) = \left(1 + \frac{1}{\overline{P_{rl}}+1}\right)\left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) + 2cc,$$

which leads, according to Equation 1, to

$$\frac{1}{pw} \times \left(\frac{\overline{P_{rl}}+2}{\overline{P_{rl}}+1}\left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) + 2cc\right) = \frac{1}{P - \overline{P_{rl}}}. \tag{5}$$

4.4.5 Non-contended System

When the system is not contended, logical conflicts are not likely to happen, hence each thread succeeds in its retry loop at its first *retry*. *A fortiori*, no hardware conflict occurs. Each thread still performs one success every work loop, and the success period is given by

$$\overline{sp}\left(\overline{P_{rl}}\right) = \frac{pw + rc + cw + cc}{P}. \tag{6}$$

Moreover, a thread spends in average $rc + cw + cc$ units of time in the retry loop within each work loop. As this holds for every thread, we can obtain the following expression for the total average number of threads inside the retry loop:

$$\overline{P_{rl}} = \frac{rc + cw + cc}{pw + rc + cw + cc} \times P = \frac{rc + cw + cc}{\overline{sp}\left(\overline{P_{rl}}\right)} \tag{7}$$

Equation 6 also gives $rc + cw + cc = P \times \overline{sp}\left(\overline{P_{rl}}\right) - pw$, hence, thanks to Equation 7,

$$\overline{P_{rl}} = \frac{P \times \overline{sp}\left(\overline{P_{rl}}\right) - pw}{\overline{sp}\left(\overline{P_{rl}}\right)}, \quad i.e. \quad \frac{\overline{sp}\left(\overline{P_{rl}}\right)}{pw} = \frac{1}{P - \overline{P_{rl}}}, \tag{8}$$

where $\overline{sp}\left(\overline{P_{rl}}\right) = \frac{rc+cw+cc}{\overline{P_{rl}}}$.

4.4.6 Unified Solving

It remains to decide whenever the data structure is under contention or not, and to find the corresponding solution. Concerning the frontier between contended and non-contended system, we can remark that Equations 5 and 8 are equivalent if and only if

$$\frac{rc + cw + cc}{\overline{P_{rl}}} = \frac{\overline{P_{rl}} + 2}{\overline{P_{rl}} + 1}\left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) + 2cc, \tag{9}$$

which leads to Lemma 2.

Lemma 2. *The system switches from being non-contended to being contended at* $\overline{P_{rl}} = P_{rl}^{(0)}$, *where*

$$P_{rl}^{(0)} = \frac{-(cc + cw - rc) + \sqrt{(cc + cw - rc)^2 + 4(rc + cw + cc)(cw + 2cc)}}{2(cw + 2cc)}.$$

Proof. We show that:

- $P_{rl}^{(0)}$ is the unique positive solution of Equation 9 if the expansion is set to 0,
- $P_{rl}^{(0)} \leq 1$,

- there is no solution of Equation 9 with a non-null expansion.

If the expansion is set to 0, then Equation 9 can be turned into the second order equation

$$\overline{P_{rl}}^2(cw + 2cc) + \overline{P_{rl}}\,(cw + cc - rc) - (rc + cw + cc) = 0,$$

that has a single positive solution: $P_{rl}^{(0)}$.

While instantiating the binomial with $\overline{P_{rl}} = 1$, we obtain $cw + 2(cc - rc)$, which is not negative, since $cc \geq rc$ in all the architectures that we are aware of. As the second order equation has also a negative solution, and $cw + 2cc$ is positive, we have that $1 \geq P_{rl}^{(0)}$. This implies that $P_{rl}^{(0)}$ is a solution of the former Equation 9: the expansion is indeed a non-decreasing function, thus $0 \leq \overline{e}\left(P_{rl}^{(0)}\right) \leq \overline{e}\,(1) = 0$. Still we could have other solutions with a non-null expansion.

However, Equation 9 can be rewritten as:

$$rc + cw + cc = \frac{\overline{P_{rl}} + 2}{\overline{P_{rl}} + 1} \times \overline{P_{rl}} \times \left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) + 2cc. \tag{10}$$

The left-hand side of Equation 10 is constant, while the right-hand side is increasing, which discards any other solution, hence the lemma. □

Thanks to Lemma 2, we can unify the success period as:

$$\overline{sp}\left(\overline{P_{rl}}\right) = \begin{cases} \left(rc + cw + cc\right)/\overline{P_{rl}} & \text{if } \overline{P_{rl}} \leq P_{rl}^{(0)} \\ \left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) \times \frac{\overline{P}_{rl+2}}{\overline{P}_{rl+1}} + 2cc & \text{otherwise.} \end{cases}$$

The unified success period obeys to the following equation

$$\overline{sp}\left(\overline{P_{rl}}\right) = \frac{pw}{P - \overline{P_{rl}}}. \tag{11}$$

We show in the following theorem how to compute the throughput estimate; the proof manipulates equations in order to be able to use the fixed-point Knaster-Tarski theorem.

Theorem 1. *The throughput can be obtained iteratively through a fixed-point search, as* $T = \left(\overline{sp}\left(\lim_{n\to+\infty} u_n\right)\right)^{-1}$, *where*

$$\begin{cases} u_0 = \frac{rc+cw+cc}{pw+rc+cw+cc}P \\ u_{n+1} = \frac{u_n\overline{sp}(u_n)}{pw+u_n\overline{sp}(u_n)} \times P \qquad \textit{for all } n \geq 0. \end{cases}$$

Proof. Let us note $\mathrm{f}_1\left(\overline{P_{rl}}\right) = \overline{sp}\left(\overline{P_{rl}}\right) \times \overline{P_{rl}}$ and $\mathrm{f}_2\left(\overline{P_{rl}}\right) = pw \times \overline{P_{rl}}/(P - \overline{P_{rl}})$; then Equation 11 is equivalent to $\mathrm{f}_1\left(\overline{P_{rl}}\right) = \mathrm{f}_2\left(\overline{P_{rl}}\right)$, and we have some properties on f_1 and f_2.

Firstly, since $x \mapsto x(x+2)/(x+1)$ is non-decreasing on $[0, +\infty[$, as well as the expected expansion, we know that f_1 is a non-decreasing function. Secondly, f_2 is increasing on $[0, P[$, and is bijective from $[0, P[$ to $[0, +\infty[$. We can thus rewrite Equation 11 as:

$$\overline{P_{rl}} = \mathrm{f}_2^{-1}\left(\mathrm{f}_1\left(\overline{P_{rl}}\right)\right). \tag{12}$$

Moreover, ${\mathrm{f}_2}^{-1} \circ \mathrm{f}_1$ is a non-decreasing function, as a composition of two non-decreasing functions. Thirdly, ${\mathrm{f}_2}^{-1}$ can be obtained through $x = \mathrm{f}_2\left({\mathrm{f}_2}^{-1}(x)\right) = pw \times {\mathrm{f}_2}^{-1}(x) / (P - {\mathrm{f}_2}^{-1}(x))$, which leads to

$${\mathrm{f}_2}^{-1}(x) = \frac{x}{pw + x} P.$$

In addition, we know by construction that if $\overline{P_{rl}} > P_{rl}^{(0)}$, then

$$\left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) \times \frac{\overline{P_{rl}} + 2}{\overline{P_{rl}} + 1} + 2cc \geq \frac{rc + cw + cc}{\overline{P_{rl}}}. \tag{13}$$

Indeed, on the one hand,

$$\lim_{\overline{P_{rl}} \to 0^+} \frac{rc + cw + cc}{\overline{P_{rl}}} = +\infty,$$

and on the other hand, $(cw + \overline{e}\left(\overline{P_{rl}}\right)) \times (\overline{P_{rl}} + 2)/(\overline{P_{rl}} + 1) + 2cc$ remains bounded. According to Lemma 2, those two functions cross only once, hence Equation 13.

Since $\overline{sp}\left(\overline{P_{rl}}\right) = (rc + cw + cc)/\overline{P_{rl}}$ if $\overline{P_{rl}} \leq P_{rl}^{(0)}$, we have $\overline{sp}\left(\overline{P_{rl}}\right) \geq (rc + cw + cc)/\overline{P_{rl}}$ for any $\overline{P_{rl}}$, and then

$$\mathrm{f}_1\left(\overline{P_{rl}}\right) \geq rc + cw + cc.$$

Let then

$$P_{rl}^{(\mathrm{i})} = \frac{rc + cw + cc}{pw + rc + cw + cc} P.$$

We have seen that ${\mathrm{f}_2}^{-1} \circ \mathrm{f}_1$ is a non-decreasing function, hence

$$\begin{aligned} {\mathrm{f}_2}^{-1}\left(\mathrm{f}_1\left(P_{rl}^{(\mathrm{i})}\right)\right) &\geq {\mathrm{f}_2}^{-1}(rc + cw + cc) \\ &\geq \frac{rc + cw + cc}{pw + rc + cw + cc} \times P \\ {\mathrm{f}_2}^{-1}\left(\mathrm{f}_1\left(P_{rl}^{(\mathrm{i})}\right)\right) &\geq P_{rl}^{(\mathrm{i})}. \end{aligned}$$

Since ${\mathrm{f}_2}^{-1}$ is bounded, Equation 12 admits a solution.

We are interested in the solution whose $\overline{P_{rl}}$ is minimal since it corresponds to the first attained solution when the expansion grows, starting from 0. The current theorem comes then from the application of the Knaster-Tarski theorem. □

4.5 Constructive Approach

In this section, we instantiate the probability distribution of the parallel work with an exponential distribution. We have therefore a better knowledge of the behavior of the algorithm, particularly in medium contention cases, which allows us to follow a fine-grained approach that studies individually each successful operation together with every *CAS* occurrence. We provide an elegant and efficient solution that relies on a Markov chain analysis.

4.5.1 Process

We have seen in Section 4.3.5 that the success period can run in one of the three modes: no contention, medium contention or high contention. The main idea is to start from a configuration with a given number of threads P_{rl} just after a successful *CAS*, and to describe what will happen until the next successful *CAS*: what will be the mode of the next success period, and even more precisely, which will be the number of threads at the beginning of the next success period.

As a basis, we consider the execution that would occur without any other thread exiting the parallel section (then entering the retry loop); we call this execution the *internal execution.* This execution follows the success period pattern described in Figure 28 (with an infinite slack time if the system is not contended). On top of this basic success period, we inject the threads that can exit the parallel section, which has a double impact. On the one hand, they increase the number of threads inside the retry loop for the next success period. On the other hand, if the first thread that exits the parallel section starts its retry during the slack time of the success period of the internal execution, then this thread will succeed its *Access*, which is a *Read*, and will shrink the actual slack time of the current success period.

According to the distribution probability of the arrival of the new threads, we can compute the probability for the next success period to start with any number of threads. The expression of this stochastic sequence of success periods in terms of Markov chains results in the throughput estimate.

4.5.2 Expansion

The expansion, as before, represents the additional time in the execution time of a retry, due to the serialization of atomic primitives. However, in contrary to Section 4.4.2, we compute here this additional time in the current success period, according to the number of threads P_{rl} inside the retry loop at the beginning of the success period. The expansion only appears when the success period is highly contended, *i.e.* when we can find a continuous sequence of *CAS*'s all through the success period. We assume that for the rest of the section.

The expansion is highly correlated with the way the cache coherence protocol handles the exchange of cache lines between threads. We rely on the experiments of the research report associated with [16], which show that if several threads request for the same cache line in order to operate a *CAS*, while another thread is currently executing a *CAS*, they all have an equal probability to obtain the cache line when the current *CAS* is over.

We draw an illustrative example in Figure 29. The green *CAS*'s are successful while the red *CAS*'s fail. To lighten the picture, we hide what happened for the threads before they experience a failed *CAS*. The horizontal dash lines represent the time where a thread wants to access the data in order to operate a *CAS* but has to wait because another thread owns the data in exclusive mode. We can observe in this example that the first thread that accesses the data structure is not the thread whose operation returns.

We are given that P_{rl} threads are inside the retry loop at the end of the previous successful *CAS*, and we only consider those threads. When such a thread executes a *CAS* for the first

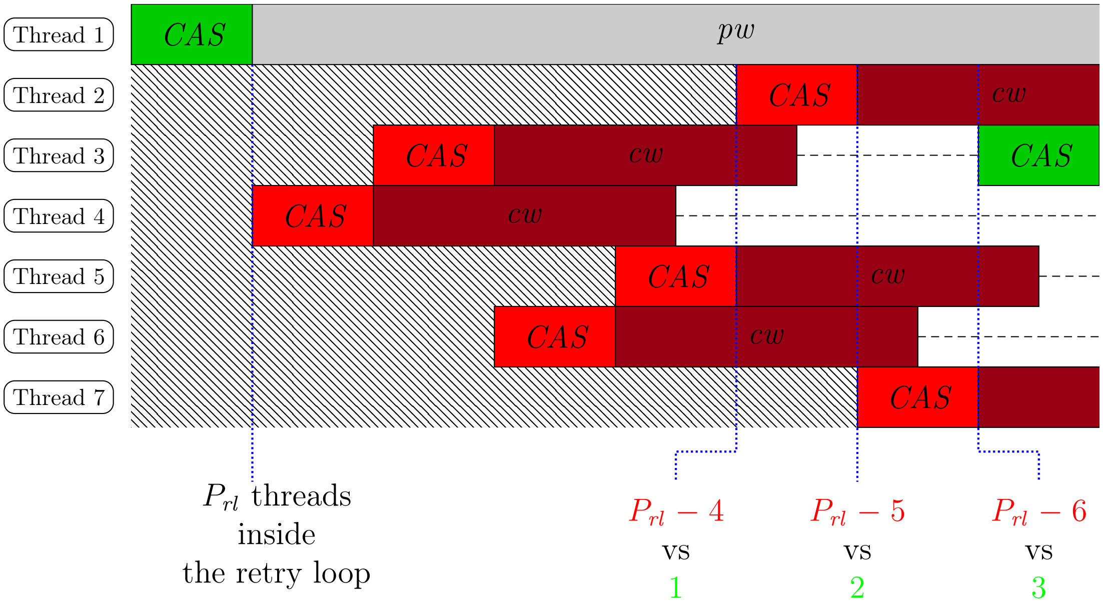


Figure 29: Highly-contended execution

time, this *CAS* is unsuccessful. The thread was in the retry loop when the successful *CAS* has been executed, so it has read a value that is not up-to-date anymore. However, this failed *CAS* will bring the current version of the value (to compare-and-swap) to the thread, a value that will be up-to-date until a successful *CAS* occurs.

So we have firstly a sequence of failed *CAS*'s until the first thread that operated its *CAS* within the current success period finishes its critical work. At this point, there exists a thread that is executing a *CAS*. When this *CAS* is finished, some threads compete to obtain the cache line. We have two bags of competing threads: in the first bag, the thread that just ended its critical work is alone, while in the second bag, there are all the threads that were in the retry loop at the beginning of the success period, and did not operate a *CAS* yet. The other, non-competing, threads are running their critical work and do not yet want to access the data.

As described before, every thread has the same probability to become the next owner of the cache line. If a thread from the first bag is drawn, then the *CAS* will be successful and the success period ends. Otherwise, the *CAS* is a failure, and we iterate at the end of this failed *CAS*. However, the thread that just failed its *CAS* is now executing its critical work, and does not request for a new *CAS* until this work has been done, thus it is not anymore in the second bag. In addition, the thread that had executed its *CAS* after the thread of the first bag is now back from its critical work and falls into the first bag. The process iterates until a thread is drawn from the first bag.

As a remark, note that we do not consider threads that are not in the retry loop at the beginning of the success period since even if they come back from the parallel section during the success period, their *Read* will be delayed and their *CAS* is likely to occur after the end of the success period.

Theorem 2 gives the explicit formula for the expansion, based on the previous explanations.

Theorem 2. *The expected time between the end of the critical work of the first thread that operates a CAS in the success period and the beginning of a successful CAS is given by:*

$$e\left(P_{rl}\right) = \lceil cw/cc \rceil cc - cw + \sum_{i=1}^{P_{com}} \frac{i(i-1)}{\left(P_{com}\right)^{i}} \frac{(P_{com}-1)!}{(P_{com}-i)!} \times cc,$$

where $P_{com} = P_{rl} - \lceil cw/cc \rceil + 1$.

Proof. Let us set the timeline so that at the beginning of the success period, *i.e.* just after a successful *CAS*, we are at $t = 0$. Firstly, a success cannot start before $t = t_0$, where $t_0 = cc + \lceil cw/cc \rceil cc$. The quickest thread indeed starts a failed *CAS* at $t = 0$ and comes back from critical work at $t = cc + cw$. It has then to wait for the current *CAS* to finish before being able to obtain the cache line. At $t = t_0$, $P_{rl} - t_0/cc + 1$ threads are competing for the data. Among them, 1 thread will lead to a successful *CAS*, while the $P_{rl} - t_0/cc$ other threads will end up with a failed *CAS*. If a failed *CAS* occurs, then at $t = t_0 + cc$, the same number of threads compete, but now there is one more potential success and one less potential failure. In the worst case, it will continue until all competing threads will lead to a successful *CAS*.

Let $P_{com} = P_{rl} - t_0/cc + 1$ the number of threads that are competing at each round, and let, for all $i \in [\![1, P_{com}]\!]$, $p_i = i/P_{com}$ the probability to draw a thread that will execute a successful *CAS*.

The expected number of failed *CAS*'s that occurs after the first thread comes back is then given by

$$\mathbb{E}\left(F\right) = p_1 \times 0 + (1-p_1)p_2 \times 1 + \cdots + (1-p_1)(1-p_2) \times \cdots \times (1-p_{P_{com}-1}) \times p_{P_{com}} \times (P_{com}-1).$$

More formally,

$$\begin{aligned}
\mathbb{E}\left(F\right) &= \sum_{i=1}^{P_{com}} \prod_{j=1}^{i-1} (1-p_j) p_i \times (i-1) \\
&= \sum_{i=1}^{P_{com}} \prod_{j=1}^{i-1} (1 - \frac{j}{P_{com}}) \frac{i}{P_{com}} \times (i-1) \\
&= \sum_{i=1}^{P_{com}} \frac{1}{\left(P_{com}\right)^{i}} \prod_{j=1}^{i-1} (P_{com}-j) i(i-1) \\
\mathbb{E}\left(F\right) &= \sum_{i=1}^{P_{com}} \frac{i(i-1)}{\left(P_{com}\right)^{i}} \frac{(P_{com}-1)!}{(P_{com}-i)!}
\end{aligned}$$

□

4.5.3 Formalization

The parallel work follows an exponential distribution, whose mean is pw. More precisely, if a thread starts a parallel section at the instant t_1, the probability distribution of the execution time of the parallel section is

$$t \mapsto \lambda e^{-\lambda(t-t_1)} \mathbb{1}_{[t_1,+\infty[}(t)\,, \text{ where } \lambda = \frac{1}{pw}.$$

This probability distribution is memoryless, which implies that the threads that are executing their parallel section cannot be differentiated: at a given instant, the probability distribution of the remaining execution time is the same for all threads in the parallel section, regardless of when the parallel section began. For all threads, it is defined by:

$$t \mapsto \lambda e^{-\lambda t}, \text{ where } \lambda = \frac{1}{pw}.$$

For the behavior in the retry loop, we rely on the same approximation as in the previous section, *i.e.* when a successful thread exits its retry loop, the remaining execution time of the retry of every other thread that is still in the retry loop is uniformly distributed between 0 and the execution time of a whole retry. We have seen that the expectation of this remaining time is the size of the execution time of a retry divided by the number of threads inside the retry loop plus one. Here, we assume that a thread will start a retry at this time. This implies another kind of memoryless property: the behavior of a thread that is in the retry loop does not depend on the moment that it entered its retry loop.

To tackle the problem of estimating the throughput of such a system, we use an approach based on Markov chains. We study the behavior of the system over time, step by step: a state of the Markov chain represents the state of the system when the current success period began (*i.e.* just after a successful *CAS*) and (thus) the system changes state at the end of every successful *CAS*. According to the current state, we are able to compute the probability to reach any other state at the beginning of the next success period. In addition, the two memoryless properties render the description of a state easy to achieve: the number of threads inside the retry loop when the current success begins, indeed fully characterizes the system.

We recall that P_{rl} is the number of threads inside the retry loop when the success period begins. The Markov chain is strongly connected with P_{rl}, since it composed of P states $\mathcal{S}_0, \mathcal{S}_1, \dots, \mathcal{S}_{P-1}$, where, for all $i \in [\![0, P-1]\!]$, the success period is in state $\mathcal{S}_i$ iff $P_{rl} = i$. For all $(i,j) \in [\![0, P-1]\!]^2$, $\mathbb{P}\left(\mathcal{S}_i \to \mathcal{S}_j\right)$ denotes the probability that a success characterized by $\mathcal{S}_j$ follows a success in state $\mathcal{S}_i$. $st\left(\mathcal{S}_i \to \mathcal{S}_j\right)$ denotes the slack time that passed while the system has gone from state $\mathcal{S}_i$ to state $\mathcal{S}_j$. This slack time can be expressed based on the slack time $st\,(i)$ of the internal execution, *i.e.* the execution that involves only the i threads of the retry loop and ignores the other threads (see Section 4.5.1). Recall that we consider that the slack time of the internal execution with 0 thread is infinite, since no thread will access the data structure. In the same way, we denote by $ct\,(i)$ the completion time of the internal execution, hence $ct\,(i) = cc + cw + e\,(i) + cc$.

We have seen that the level of contention (mode) is determined by P_{rl}, hence the interval $[\![0, P-1]\!]$ can be partitioned into

$$[\![0, P-1]\!] = \mathcal{I}_{\text{noc}} \cup \mathcal{I}_{\text{mid}} \cup \mathcal{I}_{\text{hi}},$$

where the partitions correspond to the different contention levels. So, by definition, $\mathcal{I}_{\text{noc}} = \{0\}$, and for all $i \in \mathcal{I}_{\text{noc}} \cup \mathcal{I}_{\text{mid}}$, $e\,(i) = 0$ (see Section 4.3.5).

The success period is highly-contended, *i.e.* we have a continuous sequence of *CAS*'s in the success period, if the sum of the execution time of all the *CAS*'s that need to be operated exceeds the critical work. Hence $\mathcal{I}_{\text{hi}} = [\![i_{\text{hi}}, P-1]\!]$, where

$$i_{\text{hi}} = \min\{i \in [\![1, P-1]\!] \mid i \times cc > cw\}.$$

In addition, as the sequence of *CAS*'s is continuous when the contention is high, the slack time is null when the success period is highly contended, *i.e.*, for all $i \in \mathcal{I}_{\text{hi}}$, $st\,(i) = 0$, and *a fortiori*, $st\,(\mathcal{S}_i \rightarrow \mathcal{S}_\star) = 0$.

Otherwise, the success period is in medium contention, hence $\mathcal{I}_{\text{mid}} = [\![1, i_{\text{hi}} - 1]\!]$. Moreover, if $i \in \mathcal{I}_{\text{mid}}$, $st\,(i) > 0$, and $e\,(i) = 0$, because the *CAS*'s synchronized during the previous success period and will not collide any more in the current success period.

4.5.4 Transition Matrix

We consider here that the system is in a given state, and we compute the probability that the system will next reach any other state. Without loss of generality, we can choose the origin of time such that the current success period begins at $t = 0$.

Let us first look at the core cases, *i.e.* let $i \in \mathcal{I}_{\text{mid}} \cup \mathcal{I}_{\text{hi}}$ and $k \in [\![0, P-i-1]\!]$; we assume that the system is currently in state $\mathcal{S}_i$, and we are interested in the probability that the system will switch to $\mathcal{S}_{i+k}$ at the end of the current state. In other words, we want to find the probability that, given that the current success period started when i threads were in the retry loop, the next success period will begin while $i+k$ threads are in the retry loop.

As the successful thread will exit the retry loop at the end of the current success period, there is at least one thread that enters the retry loop during the current success period. Two non-overlapping events can then occur (see Figure 30): either the first thread exiting the parallel section starts within $[0, st\,(i)\,[$, *i.e.* in the slack time of the internal execution, and this event is written E_{ext}, or the first thread entering the retry loop starts after $t = st\,(i)$, and this event is denoted by E_{int}. Therefore, we have $\mathbb{P}\,(\mathcal{S}_i \rightarrow \mathcal{S}_{i+k}) = \mathbb{P}\,(E_{\text{ext}}) + \mathbb{P}\,(E_{\text{int}})$.

First note that E_{ext} cannot happen when the success period is highly contended; in this case, the slack time is indeed null, and we conclude $\mathbb{P}\,(E_{\text{ext}}) = 0$. In addition, we have seen in Section 4.5.2 that external threads, *i.e.* threads that are in the parallel section at the beginning of the success period, do not participate to the game of expansion, so they cannot be successful. Under high-contention, E_{int} happens, and the successful *CAS* that ends the success period is operated by an internal thread, *i.e.* a thread that was already in the retry loop when the success period began.

Under medium contention, E_{ext} can occur. In this case, an external thread accesses the data structure before any internal thread does. We have also seen that the expansion is

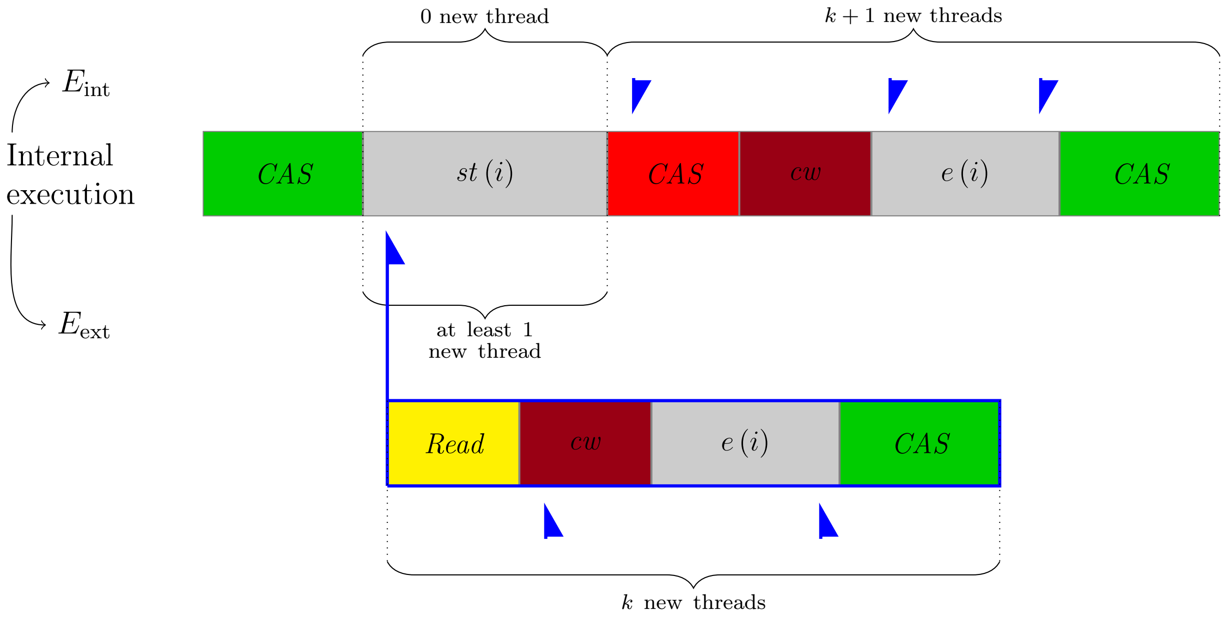


Figure 30: Possible executions

null in medium contention level, thus the external thread will execute its critical work, and especially its *CAS* without being delayed; this implies that the first external thread that accesses the data structure will end the current success period with the end of its *CAS*. If however E_{int} occurs, an internal thread succeeds, but is not necessarily the first thread that accessed the data structure during the success period.

The two possible events are pictured in Figure 30, where the blue arrows represent the threads that exit the parallel section. Recall, we aim at computing the probability to start the next success period with $i+k$ threads inside the retry loop. We formalize the idea drawn in the figure by using $X_{[a,b[}$, which is defined as a random variable indicating the number of threads exiting the parallel section during the time interval $[a,b[$. The probability of having E_{int} is then given by

$$\mathbb{P}(E_{\text{int}}) = \mathbb{P}\left(X_{[0,st(i)[} = 0 \quad | \quad P_{rl} = i \text{ at } t = 0^+\right) \times \mathbb{P}\left(X_{[st(i),st(i)+ct(i)[} = k+1 \quad | \quad P_{rl} = i \text{ at } t = st(i)^+\right).$$

Concerning E_{ext}, we know that if $i \in \mathcal{I}_{\text{hi}}$, then $\mathbb{P}(E_{\text{ext}}) = 0$. Otherwise, if we denote by t_3 the starting time of the first thread that exits the parallel section, we obtain

$$\begin{aligned}\mathbb{P}(E_{\text{ext}}) =& \mathbb{P}\left(X_{[0,st(i)[} > 0 \quad | \quad P_{rl} = i \text{ at } t = 0^+\right) \\ & \times \mathbb{P}\left(X_{[t_3,t_3+rc+cw+cc[} = k \quad | \quad P_{rl} = i+1 \text{ at } t = t_3^+\right)\end{aligned}$$

To simplify the reasoning, and given that the costs of *Read* and *CAS* are approximately the same, we approximate $t_3 + rc + cw + cc$ with $t_3 + cc + cw + cc$, leading to

$$\begin{aligned}\mathbb{P}(E_{\text{ext}}) =& \mathbb{P}\left(X_{[0,st(i)[} > 0 \quad | \quad P_{rl} = i \text{ at } t = 0^+\right) \\ & \times \mathbb{P}\left(X_{[t_3,t_3+ct(i+1)[} = k \quad | \quad P_{rl} = i+1 \text{ at } t = t_3^+\right)\end{aligned}$$

According to the exponential distribution, given a thread that is in the parallel section at $t = a$, the probability to exit the parallel section within $[a, b[$ is:

$$\int_a^b \lambda e^{-\lambda(t-a)}\, dt = \int_0^{b-a} \lambda e^{-\lambda u}\, du.$$

It is also the probability, given a thread that is in the parallel section at $t = 0$, to exit the retry loop within $[a, b-a[$. This implies:

$$\begin{aligned}\mathbb{P}\left(E_{\text{int}}\right) =&\mathbb{P}\left(X_{[0,st(i)[} = 0 \quad | \quad P_{rl} = i \text{ at } t = 0^+\right)\\ &\times \mathbb{P}\left(X_{[0,ct(i)[} = k+1 \quad | \quad P_{rl} = i \text{ at } t = 0^+\right)\end{aligned}$$

and

$$\begin{aligned}\mathbb{P}\left(E_{\text{ext}}\right) =&\mathbb{P}\left(X_{[0,st(i)[} > 0 \quad | \quad P_{rl} = i \text{ at } t = 0^+\right)\\ &\times \mathbb{P}\left(X_{[0,ct(i)[} = k \quad | \quad P_{rl} = i+1 \text{ at } t = 0^+\right).\end{aligned}$$

To lighten the notations, let us define

$$\left\{\begin{array}{l} a_{i,k} = \mathbb{P}\left(X_{[0,ct(i)[} = k \quad | \quad P_{rl} = i \text{ at } t = 0\right) \\ b_i = \mathbb{P}\left(X_{[0,st(i)[} = 0 \quad | \quad P_{rl} = i \text{ at } t = ct\left(i\right)^+\right). \end{array}\right. \tag{14}$$

In addition, given a thread that is in the parallel section at $t = 0$, the probability to exit the parallel section within $[0, b-a[$ is $\int_0^{b-a} \lambda e^{-\lambda u}\, du$. By counting the number of threads that need to exit the parallel section, we obtain:

$$\left\{\begin{array}{l} a_{i,k} = \binom{P-i}{k}\left(1 - e^{-\lambda ct(i)}\right)^k \left(e^{-\lambda ct(i)}\right)^{P-i-k} \\ b_i = \left(\exp\left(-\lambda st\left(i\right)\right)\right)^{P-i}. \end{array}\right. \tag{15}$$

Altogether, we have that

$$\mathbb{P}\left(\mathcal{S}_i \to \mathcal{S}_{i+k}\right) = b_i \times a_{i,k+1} + \left(1 - b_i\right) \times a_{i+1,k}.$$

The situation is slightly different if $k = -1$; in this case, no thread should exit the parallel section during the slack time and no thread should exit during the retry of the first thread that accessed the data structure during the success period neither. This shows that

$$\mathbb{P}\left(\mathcal{S}_i \to \mathcal{S}_{i-1}\right) = b_i \times a_{i,0}.$$

When the success period is not contended, *i.e.* if $i = 0$, the slack time of the execution that ignores external threads can be seen as infinite, hence we can define $b_0 = 0$ (the probability that a thread exits its parallel section during an infinite interval of time is 1). As for the $a_{i,k}$'s, they can be defined in the same way as earlier.

We have obtained the full transition matrix $\left(M_{i,j}\right)_{(i,j)\in[\![0,P-1]\!]^2}$, which is a triangular matrix, augmented with a subdiagonal:

$$\left\{\begin{array}{lll} M_{i,i+k} & = b_i a_{i,k+1} + (1-b_i) a_{i+1,k} & \text{if } k \in [\![0, P-i-1]\!] \\ M_{i,i-1} & = b_i \times a_{i,0} & \text{if } i > 0 \\ M_{i,j} & = 0 & \text{otherwise} \end{array}\right.$$

Lemma 3. *M is a right stochastic matrix.*

Proof. First note that, by definition of $a_{i,k}$, for all $i \in [\![0, P-1]\!]$,

$$\sum_{k=0}^{P-i} a_{i,k} = 1.$$

If i threads are indeed inside the retry loop at $t = 0$, then, within $[0, st\,(i)\,[$, at least 0 thread, and at most $P - i$ threads (inclusive) will exit their parallel section.

We have first

$$\sum_{j=0}^{P-1} M_{0,j} = \sum_{k=0}^{P-1} a_{0+1,k} = 1.$$

In the same way, for all $i \in [\![1, P-1]\!]$,

$$\begin{aligned}\sum_{j=0}^{P-1} M_{i,j} &= \sum_{k=-1}^{P-1-i} M_{i,i+k}\\ &= b_i \times a_{i,0} + \sum_{k=0}^{P-1-i} b_i a_{i,k+1} + (1-b_i) a_{i+1,k}\\ &= b_i \times \sum_{k=-1}^{P-1-i} a_{i,k+1} + (1-b_i) \sum_{k=0}^{P-1-i} a_{i+1,k}\\ \sum_{j=0}^{P-1} M_{i,j} &= 1.\end{aligned}$$

□

Lemma 4. *The transition matrix has a unique stationary distribution, which is the unique left eigenvector of the transition matrix with eigenvalue 1 and sum of its elements equal to 1.*

Proof. Note that the Markov chain is irreducible and aperiodic. Let $X \geq P-1$, $i \in [\![0, P-1]\!]$ and $j \in [\![i, P-1]\!]$.

$$\begin{aligned}\mathbb{P}\left(\mathcal{S}_j \to \mathcal{S}_i \text{ in X steps}\right) \geq & \mathbb{P}\left(\mathcal{S}_j \to \mathcal{S}_{j-1} \to \cdots \to \mathcal{S}_i\right)\\ & \times \mathbb{P}\left(\mathcal{S}_i \to \mathcal{S}_i\right)^{X-(j-i)}\\ \mathbb{P}\left(\mathcal{S}_j \to \mathcal{S}_i \text{ in X steps}\right) > & 0\end{aligned}$$

As

$$\mathbb{P}\left(\mathcal{S}_i \to \mathcal{S}_j \text{ in X steps}\right) \geq \mathbb{P}\left(\mathcal{S}_i \to \mathcal{S}_j\right) > 0,$$

the Markov chain is irreducible. Since $\mathcal{S}_1$ is clearly aperiodic, and the chain is irreducible, the chain is aperiodic as well.

This implies that the Markov chain has a unique stationary distribution, which is the unique left eigenvector of the transition matrix with eigenvalue 1 and sum of its elements equal to 1. □

4.5.5 Stationary Distribution

Theorem 3. *Given the transition matrix, the stationary distribution can be found in* $(P+1)P-1$ *operations.*

Proof. As the Markov chain is irreducible, the stationary distribution does not contend any zero. The space of the left eigenvectors with unit eigenvalue is uni-dimensional; therefore, for any v_0, there exists a vector $v = (v_0\ v_1\ \dots\ v_{P-1})$, such that v spans this space.

Let v_0 a real number; necessarily, v fulfills $v \cdot M = v$, hence for all $i \in [\![0, P-2]\!]$

$$\sum_{k=0}^{i+1} v_k M_{k,i} = v_i,$$

which leads to, for all $i \in [\![0, P-2]\!]$:

$$v_{i+1} = \frac{1}{M_{i+1,i}} \left((1 - M_{i,i}) v_i - \sum_{k=0}^{i-1} v_k M_{k,i} \right).$$

So we obtain the $v_1, \dots, v_{P-1}$ iteratively (we know that $M_{i+1,i} = b_{i+1} \times a_{i+1,0}$, which is not null), with $2 \times i + 1$ operations needed to compute v_{i+1}.

The elements of the stationary distribution should sum to one, so we start from any v_0, compute the whole vector, and then normalize each element by their sum, hence the theorem. □

4.5.6 Slack time and Throughput

In order to compute the final throughput, we have to compute the expectation of the slack time, when the system goes from state $\mathcal{S}_i$ to any other state, that we note $\mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right)$. Also, we will be able to exhibit a vector $s = (s_0, s_1, \dots, s_{P-1})$ of expected success period, where s_i is the expectation of the execution time of the success period if i threads are in the retry loop when the success period begins:

$$\begin{cases} s_i = \mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right) + cc + cw + e\left(i\right) + cc & \text{if } i \notin \mathcal{I}_{\text{noc}} \\ s_i = \mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right) + rc + cw + cc & \text{otherwise.} \end{cases}$$

Finally, the expected throughput (inverse of the success period) is calculated through

$$T = \frac{1}{v \cdot s},$$

where v is the stationary distribution of the Markov chain.

We know already that if $i \in \mathcal{I}_{\text{hi}}$, then $\mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_{i+k}\right)\right) = 0$.

In the other extreme case, *i.e.* if $i \in \mathcal{I}_{\text{noc}}$, we rely on the following lemma.

Lemma 5. *Let an integer n, a real number λ, and n independent random variables $X_1, X_2, \ldots, X_n$, following an exponential distribution of mean λ^{-1}. Let then X be the random variable defined by: $X = \min_{i \in [\![1,n]\!]} X_i$. The expectation of X is:*

$$\mathbb{E}\left(X\right) = \frac{1}{\lambda n}.$$

Proof. We have

$$\begin{aligned}
\mathbb{P}\left(X > x\right) &= \mathbb{P}\left(\forall i : X_i > x\right) \\
&= \prod_{i=1}^{n} \mathbb{P}\left(X_i > x\right) \\
&= \left(\int_{x}^{+\infty} \lambda e^{-\lambda t}\right)^{n} \\
\mathbb{P}\left(X > x\right) &= e^{-\lambda n x}
\end{aligned}$$

Therefore, the probability distribution of X is given by:

$$t \mapsto \lambda n e^{-\lambda n t},$$

and its expectation is computed through

$$\begin{aligned}
\mathbb{E}\left(X\right) &= \int_{0}^{+\infty} \lambda n t e^{-\lambda n t}\, dt \\
&= \left[e^{-\lambda n t} t\right]_{+\infty}^{0} + \int_{0}^{+\infty} e^{-\lambda n t}\, dt \\
&= \left[\frac{1}{\lambda n} e^{-\lambda n t}\right]_{+\infty}^{0} \\
\mathbb{E}\left(X\right) &= \frac{1}{\lambda n}
\end{aligned}$$

□

This proves that

$$\mathbb{E}\left(st\left(\mathcal{S}_0 \to \mathcal{S}_\star\right)\right) = \frac{1}{pw \times P}.$$

Let now $i \in \mathcal{I}_{\text{mid}}$, and $k \in [\![-1, P-i-1]\!]$; we are interested in $\mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_{i+k}\right)\right)$. The slack time is less immediate, and we use the following reasoning. First note that the probability distribution of the first thread exiting the parallel section is given by $t \mapsto \lambda(P-i)e^{-\lambda(P-i)t}$.

If this thread comes back during $]0, st\,(i)\,[$, the time that passed since the beginning of the success period is the slack time, otherwise, it is $st\,(i)$.

$$\begin{aligned}
\mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right) &= \int_0^{st(i)} \lambda(P-i)e^{-\lambda(P-i)t} t \, dt + \int_{st(i)}^{+\infty} \lambda(P-i)e^{-\lambda(P-i)t} st\,(i) \; dt \\
&= \left[e^{-\lambda(P-i)t} t\right]_{st(i)}^{0} + \left[\frac{1}{\lambda(P-i)} e^{-\lambda(P-i)t}\right]_{st(i)}^{0} + st\,(i) \left[e^{-\lambda(P-i)t}\right]_{+\infty}^{st(i)} \\
\mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right) &= -st\,(i)\, e^{-\lambda(P-i)st(i)} + \frac{1 - e^{-\lambda(P-i)st(i)}}{\lambda(P-i)} + st\,(i) \left(e^{-\lambda(P-i)st(i)}\right)
\end{aligned}$$

We conclude that

$$\mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right) = \frac{1 - e^{-\frac{(P-i)st(i)}{pw}}}{P-i} pw.$$

Putting all together, we obtain

$$\begin{cases}
\mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right) = \frac{1-e^{-\frac{(P-i)st(i)}{pw}}}{P-i} pw & \text{if } i \in \mathcal{I}_{\text{noc}} \cup \mathcal{I}_{\text{mid}} \\
\mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right) = 0 & \text{if } i \in \mathcal{I}_{\text{hi}}.
\end{cases}$$

4.5.7 Number of Failed Retries

Another metric to estimate the quality of the model is the number of failed retries per successful retry. We compute it by counting the number of failed retries within the current success period, where a retry is billed to a given success period if its failed *CAS* occurs during this success period. We denote by $\mathbb{E}\left(f_i\right)$ the expected number of failed *CAS* during a success period that begins with i threads, where $i \in [\![0, P-1]\!]$.

If the success period is not contended, *i.e.* if $i \in \mathcal{I}_{\text{noc}}$, no failure will occur since the first *CAS* of the success period will be a success; hence $\mathbb{E}\left(f_i\right) = 0 = i$.

If the success period is medium contended, *i.e.* if $i \in \mathcal{I}_{\text{mid}}$, every thread that is in the retry loop in the beginning of the success period will execute at least one *CAS* during this success period, and exactly two if the thread is the successful one. We know indeed that, even if a thread exits its parallel section during the slack time, and is then successful, the failed *CAS*'s will occur before the thread entering the retry loop executes its successful *CAS*. As any thread that exits its parallel section during the success period either is successful at its first *CAS*, or does not operate the *CAS* during the success period, we conclude that: $\mathbb{E}\left(f_i\right) = i$.

If the success period is highly contended, *i.e.* if $i \in \mathcal{I}_{\text{hi}}$, then we know that we have an uninterrupted sequence of failed *CAS*'s, from the beginning of the success period to the last ending successful *CAS*. The expected number of failed *CAS*'s is then directly related to the expected duration of the success period. Recalling that the expansion is given in Theorem 2, we obtain:

$$\mathbb{E}\left(f_i\right) = 1 + \frac{cw + e\,(i)}{cc}.$$

4.6 Experiments

To validate our analysis results, we use two main types of lock-free algorithms. In the first place, we consider a set of algorithms that follow the pattern in AbstractAlgorithm. This set of algorithms includes: (i) synthetic designs, that cover the design space of possible lock-free data structures; (ii) several fundamental designs of data structure operations such as lock-free stacks [130] (`Pop`, `Push`), queues [107] (`Dequeue`), counters [53] (`Increment`, `Decrement`). As a second step, we consider more advanced lock-free operations that involve helping mechanisms, and show how to use our analysis in this context. Finally, in order to highlight the benefits of the analysis framework, we show how it can be applied to i) determine a beneficial back-off strategy and ii) optimize the memory management scheme used by a data structure, in the context of an application.

We also give insights about the strengths of our two approaches. On the one hand, the constructive approach exhibits better predictions due to the tight estimation of the failing retries. On the other hand, the average-based approach is applicable to a broader spectrum of algorithmic designs as it leaves room to abstract complicated algoritmic designs, which do not follow the pattern of AbstractAlgorithm.

4.6.1 Setting

We have conducted experiments on an Intel ccNUMA workstation system. The system is composed of two sockets equipped with Intel Xeon E5-2687W v2 CPUs with frequency band 1.2-3.4. GHz The physical cores have private L1, L2 caches and they share an L3 cache, which is 25 MB. In a socket, the ring interconnect provides L3 cache accesses and core-to-core communication. Due to the bi-directionality of the ring interconnect, uncontended latencies for intra-socket communication between cores do not show significant variability.Our model assumes uniformity in the *CAS* and *Read* latencies on the shared cache line. Thus, threads are pinned to a single socket to minimize non-uniformity in *Read* and *CAS* latencies. In the experiments, we vary the number of threads between 4 and 8 since the maximum number of threads that can be used in the experiments are bounded by the number of physical cores that reside in one socket. We show the experimental results with 8 threads.

In all figures, the y-axis shows both the throughput values, *i.e.* number of operations completed per second, and the ratio of failing to successful retries (multiplied by 10^6, for readability), while the mean of the exponentially distributed parallel work pw is represented on the x-axis. The number of failures per success in the average-based approach is computed as $\overline{P_{rl}} - 1$ and is described in Section 4.5.7 for the constructive approach.

We have also added a straightforward upper bound as a baseline approach, which is defined as the minimum of $1/(rc + cw + cc)$ (two successful retries cannot overlap) and $P/(pw + rc + cw + cc)$ (a thread can succeed only once in each work loop).

4.6.2 Basic Data Structures

Here, we consider lock-free algorithms that strictly follow the pattern in AbstractAlgorithm and provide predictions using both the average-based and the constructive approach together with the theoretical upper bound.

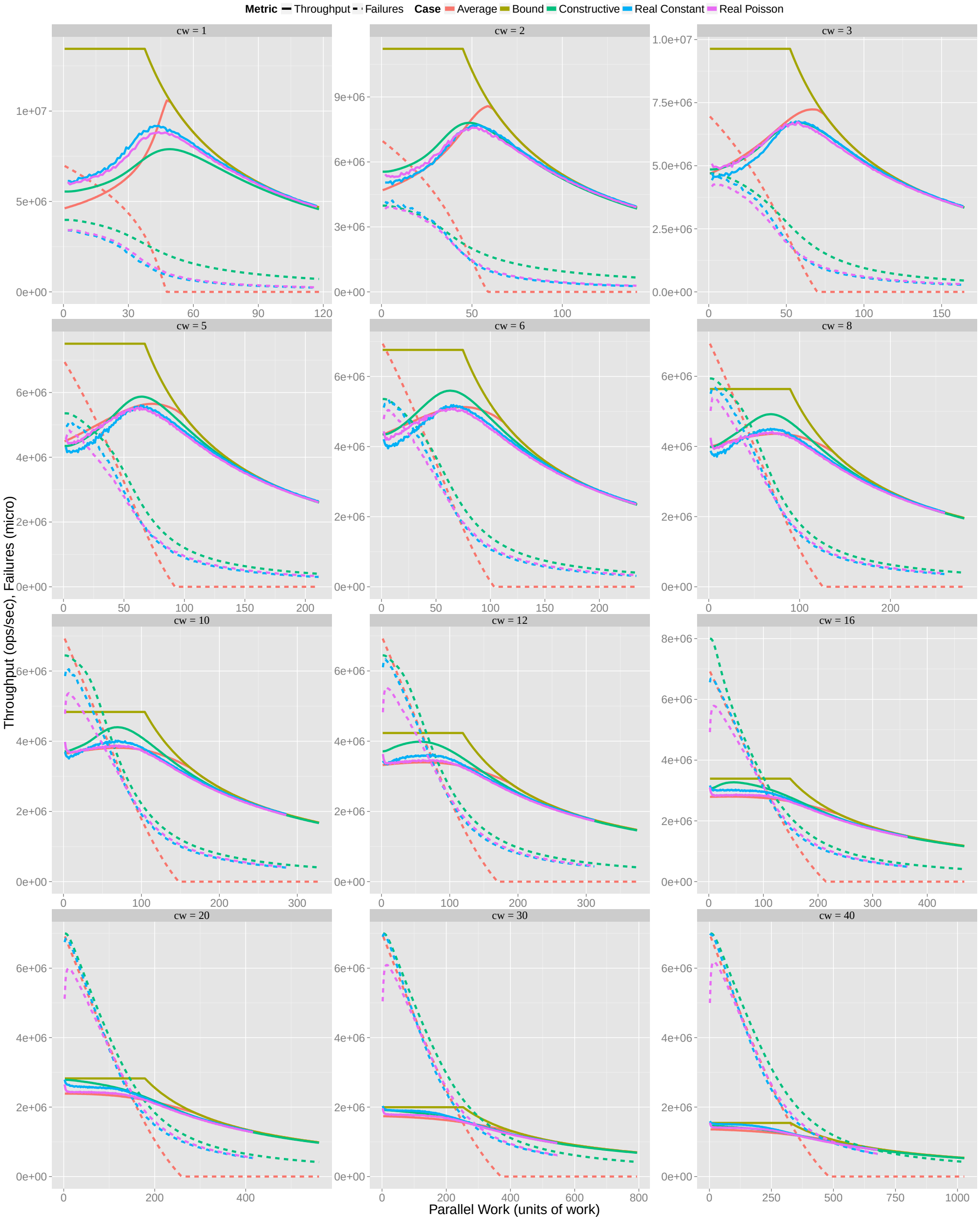


Figure 31: Synthetic program with exponentially distributed parallel work

4.6.3 Synthetic Tests

We first evaluate our models using a set of synthetic tests that have been constructed to abstract different possible design patterns of lock-free data structures (value of *cw*) and

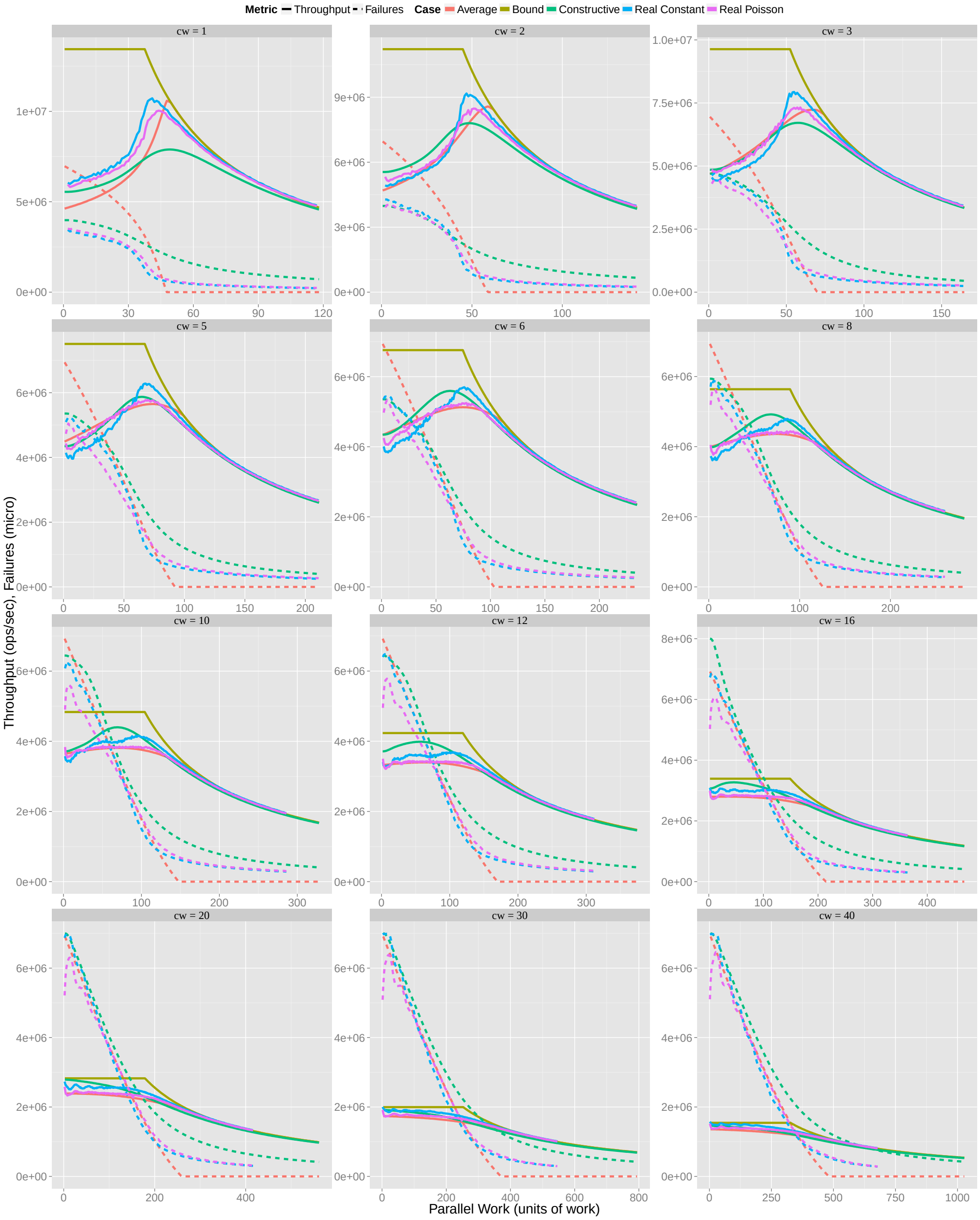


Figure 32: Synthetic program with parallel work following Poisson

different application contexts (value of pw). The critical work is either constant, or follows a Poisson distribution; in Figure 31, its mean value cw is indicated at the top of the graphs.

A steep decrease in throughput, as pw gets low, can be observed for the cases with low cw, that mainly originates due to expansion. When cw is high, performance continues to

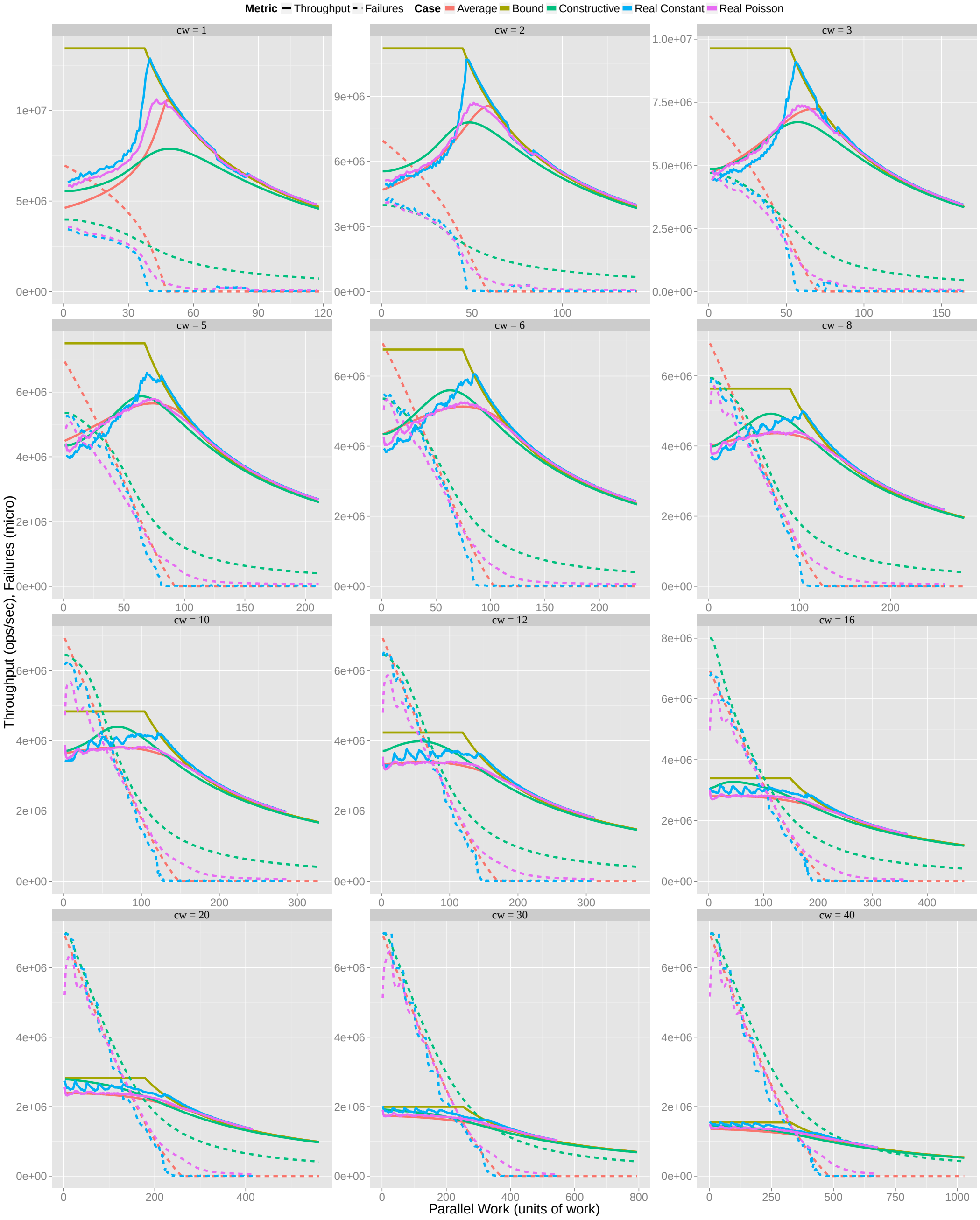


Figure 33: Synthetic program with Constant parallel work

increase when *pw* decreases, though slightly. The expansion is indeed low but the slack time, which appears as a more dominant factor, decreases as the number of threads inside the retry loop increases.

When looking into the differences between the constructive and the average-based approach: the average-based approach estimations come out to be less accurate for mid-contention cases as it only differentiates between contended and non-contended modes. In addition, it fails to capture the failing retries when measured throughput starts to deviate from the theoretical upper bound, as *pw* gets lower. In contrast, the constructive approach provides high accuracy in all metrics for almost every case.

We have also run the same synthetic tests with a parallel work that follows a Poisson distribution (Figure 32) or is constant (Figure 33), in order to observe the impact of the distribution nature of the parallel work. Compared to the exponential distribution, a better throughput is achieved with a Poisson distribution on the parallel work. The throughput becomes even better with a constant parallel work, since the slack time is minimized due to the synchronization between the threads, as explained in [73].

4.6.4 Treiber's Stack

The lock-free stack by Treiber [130] is a fundamental data structure that provides `Pop` and `Push` operations. To `Pop` an element, the top pointer is read and the next pointer of the initial element is obtained. The latter pointer will be the new value of the *CAS* that linearizes the operation. So, accessing the next pointer of the topmost element represents *cw* as it takes place between the *Read* and the *CAS*. We initialize the stack by pushing elements with or without a stride from a contiguous chunk of memory. By this way, we are able to introduce both costly or not costly cache misses. We also vary the number of elements popped at the same time to obtain different *cw*; the results, with different *cw* values are illustrated in Figure 34.

4.7 Towards Advanced Data Structure Designs

Advanced lock-free operations generally require multiple pointer updates that cannot be done with a single *CAS*. One way to design such operations, in a lock-free manner, is to use helping mechanisms: an inconsistency will be fixed eventually by some thread. Here we consider two data structures that apply immediate helping, the queue from [107] and the deque designed in [105]. In the queue experiment (Figure 35), we run the `Enqueue` operation on the queue with and without memory management; in the deque experiment, each thread is dedicated to an end of the deque (equally distributed), while we vary the proportion of push operations (colors in Figure 36).

Here, we consider data structures that apply immediate helping, where threads help for the completion of a recently linearized operation until the data structure comes into a stable state in which a new operation can be linearized. The crucial observation is that the data structure goes through multiple stages in a round robin fashion. The first stage is the one where the operation is linearized. The remaining ones are the stages in which other threads,

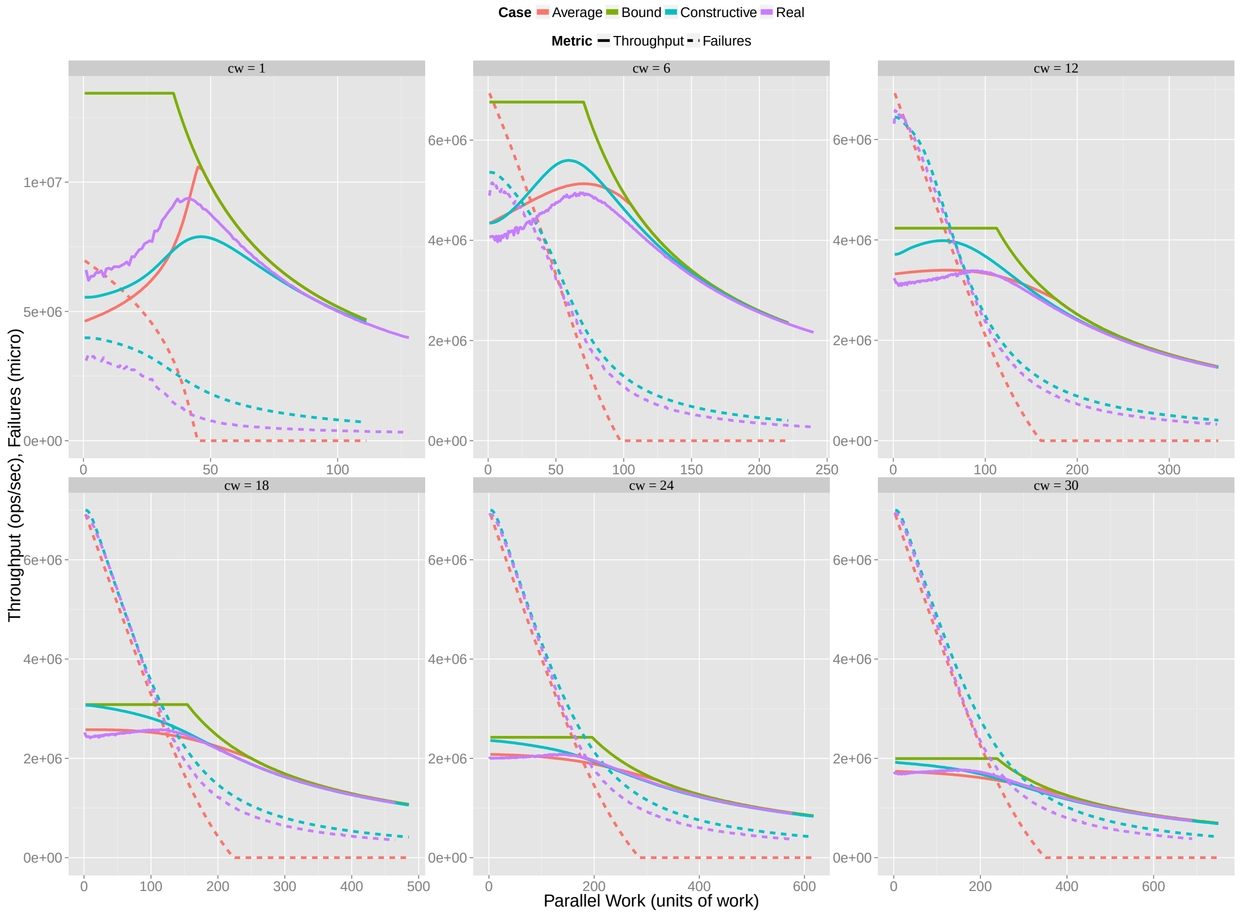


Figure 34: Treiber's Stack

that execute another operation, might help for the completion of the linearized operation, before attempting to linearize their own operations. Thus, the success period (ignoring the slack time) can be seen as the sum of the execution time of these stages, each ending with a *CAS* that updates a pointer. The *CAS* in the first stage might be expanded by the threads that are competing for the linearization of their operation, and consequent *CAS*'s might be expanded by the helper threads, which are still trying to help an already completed operation. Also, there might be slack time before the start of the first stage as the other stages will start immediately due to the thread that has completed the previous stage.

Although it is hard to stochastically reconstruct the executions with Markov chains, our average-based approach provides the flexibility required to estimate the performance by plugging the expected success period, given the number of threads inside the retry loop, into the Little's Law. As the impacting factors are similar, we estimate the success period in the same vein as in Section 4.4; with a minor adaptation of the expansion formula and by slightly adapting the slack time estimation based on the same arguments.

4.7.1 Expected Expansion for the Advanced Data Structures

Consider an operation such that, the success period (ignoring the slack time) is composed of S stages (denoted by $\mathit{Stage}_1, \dots, \mathit{Stage}_S$) where each stage represents a step towards the completion of the operation. Let CAS_i denote the CAS operation at the end of the Stage_i. From a system-wide perspective, $\{\mathit{CAS}_1, \dots, \mathit{CAS}_S\}$ is the set of CAS's that have to be successfully and consecutively executed to complete an operation, assuming all threads are executing the same operation. This design enforces that CAS_i can be successful only if the last successful CAS is a CAS_{i-1}. And, CAS_1 can be successful only if the last successful CAS is a CAS_S. In other words, another operation can not linearize before the completion of the linearized but incomplete operation.

Now, let e_i denote the expected expansion of CAS_i. If the data structure is in the stable state (*i.e.* is in Stage_1, where a new operation can be linearized), then we have to consider the probability, for all threads except one, to expand the successful CAS_1 which linearizes the operation. After the linearization, this operation will be completed in the remaining stages where again the successful CAS's at the end of the stages are subject to the same expansion possibility by the threads in the retry loop, as they might be still trying to help for the completion of the previously completed operation.

Similar to [73], our assumption here is that any thread that is in the retry loop, can launch CAS_i, with probability h, that might expand the successful CAS_i. We consider, the starting point of a failing CAS_i is a random variable which is distributed uniformly within the retry loop, which is composed of expanded stages of the operation. This is because an obsolete thread can launch a CAS_i, regardless of the stage in which the data structure is in (equally, regardless of the last successful CAS). Due to the uniformity assumption, the expansion for the successful CAS's in all stages, would be equal. Similar to the [73], we estimate the expansion e_i by considering the impact of a thread that is added to the retry loop. Let the cost function delay_i provide the amount of delay that the additional thread introduces, depending on the point where the starting point of its CAS_i hits. By using these cost functions, we can formulate the total expansion increase that each new thread introduces and derive the differential equation below to calculate the expected total expansion in a success period, where $\overline{e}\left(\overline{P_{rl}}\right) = \sum_{i=1}^{S} \overline{e_i}\left(\overline{P_{rl}}\right)$. Note that, we assume that the expansion starts as soon as strictly more than 1 thread are in the retry loop, in expectation.

Lemma 6. *The expansion of a CAS operation is the solution of the following system of equations, where* $\mathit{rlw} = \sum_{i=1}^{S} \mathit{rlw}_i = \sum_{i=1}^{S} (\mathit{rc}_i + \mathit{cw}_i + \mathit{cc}_i)$*:*

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

Proof. We compute $\overline{e}\left(\overline{P_{rl}} + h\right)$, where $h \leq 1$, by assuming that there are already $\overline{P_{rl}}$ threads in the retry loop, and that a new thread attempts to CAS during the retry, within a probability h. For simplicity, we denote $a_j^i = (\sum_{j=1}^{i-1} \mathit{rlw}_j + e_j(\overline{P_{rl}})) + \mathit{rc}_i + \mathit{cw}_i$.

$$
\begin{aligned}
\overline{e}\left(\overline{P_{rl}}+h\right) &= \overline{e}\left(\overline{P_{rl}}\right) + h \times \sum_{i=1}^{S} \int_{0}^{rlw^{(+)}} \frac{delay_i\left(t_i\right)}{rlw^{(+)}}\, dt_i \\
&= \overline{e}\left(\overline{P_{rl}}\right) + h \times \sum_{i=1}^{S} \Bigg(\int_{0}^{a_j^i-cc} \frac{delay_i\left(t_i\right)}{rlw^{(+)}}\, dt_i + \int_{a_j^i-cc}^{a_j^i} \frac{delay_i\left(t_i\right)}{rlw^{(+)}}\, dt_i \\
&\qquad + \int_{a_j^i}^{a_j^i+\overline{e_i}\left(\overline{P_{rl}}\right)} \frac{delay_i\left(t_i\right)}{rlw^{(+)}}\, dt_i + \int_{a_j^i+\overline{e_i}\left(\overline{P_{rl}}\right)}^{rlw^{(+)}} \frac{delay_i\left(t_i\right)}{rlw^{(+)}}\, dt_i \Bigg) \\
&= \overline{e}\left(\overline{P_{rl}}\right) + h \times \sum_{i=1}^{S} \Bigg(\int_{a_j^i-cc}^{a_j^i} \frac{t_i}{rlw^{(+)}}\, dt_i + \int_{a_j^i}^{a_j^i+\overline{e_i}\left(\overline{P_{rl}}\right)} \frac{cc}{rlw^{(+)}}\, dt_i \Bigg) \\
\overline{e}\left(\overline{P_{rl}}+h\right) &= \overline{e}\left(\overline{P_{rl}}\right) + h \times \frac{\left(\sum_{i=1}^{S} \frac{cc^2}{2}\right) + \overline{e}\left(\overline{P_{rl}}\right) \times cc}{rlw^{(+)}}
\end{aligned}
$$

This leads to

$$\frac{\overline{e}\left(P_{rl}+h\right) - \overline{e}\left(\overline{P_{rl}}\right)}{h} = \frac{S \times \frac{cc^2}{2} + \overline{e}\left(\overline{P_{rl}}\right) \times cc}{rlw^{(+)}}.$$

When making h tend to 0, we finally obtain

$$\overline{e}'\left(\overline{P_{rl}}\right) = cc \times \frac{S \times \frac{cc}{2} + \overline{e}\left(\overline{P_{rl}}\right)}{rlw + \overline{e}\left(\overline{P_{rl}}\right)}. \qquad \square$$

In addition, if a set S_k of CAS's are operating on the same variable var_k, then $CAS_i \in S_k$ can be expanded by the $CAS_j \in S_k$. In this case, we can obtain $\overline{e_k}\left(\overline{P_{rl}}\right)$ by using the reasoning above. The calculation simply ends up as follows: Consider the problem as if no CAS shares a variable and denote expansion in $Stage_i$ with $\overline{e_i}\left(\overline{P_{rl}}\right)^{(old)}$. Then, $\overline{e_k}\left(\overline{P_{rl}}\right) = \sum_{CAS_i \in S_k} \overline{e_i}\left(\overline{P_{rl}}\right)^{(old)}$.

4.7.2 Expected Slack Time for the Advanced Data Structures

We assume here the slack time can only occur after the completion of an operation (*i.e.* before stage 1), as the other stages are expected to start immediately due to the thread that completes the previous stage. Similar to Section 4.4.3, we consider that, at any time, the threads that are running the retry loop have the same probability to be anywhere in their current retry. Thus, a thread can be in any stage just after the successful CAS that completes the operation. So, we need to consider the thread which is closest to the end of its current stage when the operation is completed. We denote the execution time of the expanded retry loop with $rlw^{(+)}$ and the number of stages with S. For a thread executing $Stage_i$ when the operation completes, the time before accessing the data structure is then uniformly distributed between 0 and $rlw_i^{(+)}$.

Here, we take another assumption and consider all stages can be completed in the same amount of time (*i.e.* for all (i, j) in $\{1, \ldots, S\}^2$, $rlw_i^{(+)} = rlw_j^{(+)} = rlw^{(+)}/S$). This assumption does not diverge much from the reality and provides a reasonable approximation. With these assumption and using Lemma 1, we conclude that:

$$\overline{st}\left(\overline{P_{rl}}\right) = \frac{rlw^{(+)}}{S \times \left(\overline{P_{rl}} + 1\right)}. \tag{16}$$

4.7.3 Enqueue on Michael-Scott Queue

As a first step, we consider the `Enqueue` operation of the MS queue to validate our approach. This operation requires two pointer updates leading to two stages, each ending with a *CAS*. The first stage, that linearizes the operation, updates the next pointer of the last element to the newly enqueued element. In the next and last stage, the queue's head pointer is updated to point to the recently enqueued element, which could be done by a helping thread, that brings the data structure into a stable state.

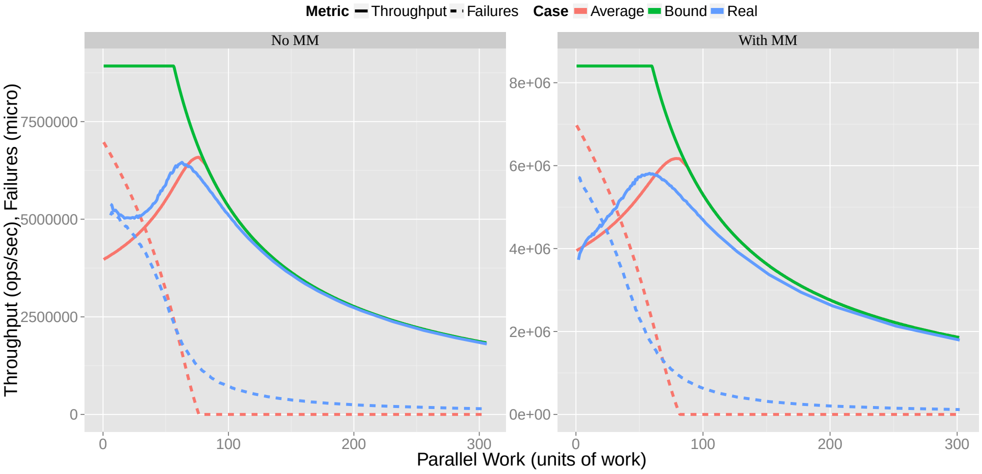


Figure 35: Enqueue on MS Queue

We estimate the expansion in the success period as described above and throughput as explained in Section 4.4. The results for the `Enqueue` experiments where all threads execute `Enqueue` are presented in Figure 35.

4.7.4 Deque

We consider the deque designed in [105]. `PushLeft` and `PushRight` (resp. `PopLeft` and `PopRight`) operations are exactly the same, except that they operate on the different ends of the deque. The status flags, which depict the state of the deque, and the pointers to the leftmost element and the rightmost element are together kept in a single double-word variable, so-called *Anchor*, which could be modified by a double-word *CAS* atomically.

A `PopLeft` operation linearizes and even completes in one stage that ends with a double-word *CAS* that just sets the left pointer of the anchor to the second element from left.

A `PushLeft` operation takes three stages to complete. In the first stage, the operation is linearized by setting the left pointer of the *Anchor* to the new element and at the same time changing the status flags to "left unstable", to indicate the status of the incomplete but linearized `PushLeft` operation. In the second stage, the left pointer of the leftmost element is redirected to the recently pushed element. In the third stage, a *CAS* is executed on *Anchor* to bring the deque status flags into "stable state". Every operation can help an incomplete `PushLeft` or `PushRight` until the deque comes into the stable state; in this state, the other operations can attempt to linearize anew.

As noticed, the first and the third stage execute a *CAS* on the same variable (*Anchor*) so it is possible to delay the third stage of the success period by executing a *CAS* in the first stage. This implies that the expansion in stage one should also be considered when the delay in the third stage is considered, and the other way around. This can be done by summing expansion estimates of the stages that run the *CAS* on the same variable and using this expansion value in all these stages. Again, it just requires simple modifications in the expansion formula by keeping assumptions unchanged.

We first run pop-only and push-only experiments where dedicated threads operate on both ends of the deque, in a half-half manner. We provide predictions by plugging the slightly modified expansion estimate, as explained above, into the average-based approach. Then, we take one step further and mix the operations, assigning the threads inequally among push and pop operations. And, we obtain estimates for them by simply taking the weighted average (depending on the number of threads running each operation) of the success period of pop-only and push-only experiments, with the corresponding *pw* value.

In Figure 36, results are illustrated; they are satisfactory for the push-only and pop-only cases. For the mixed-case experiments, the results are mixed: our analysis follows the trend and becomes less accurate when the *pw* gets lower, as experimental curves tend toward push-only success period. This, presumably, happens because the first stage of a `PushLeft` (or `PushRight`) operation is shorter than the first stage of a `PopLeft` (or `PopRight`) operation. This brings indeed an advantage to push operations, under contention: they have higher chances to linearize before pop operations after the data structure comes into the stable state. It also provides an interesting observation which highlights the lock-free nature of operations: it is improbable to complete a pop operation if numerous threads try to push, due to the difference of work inside the first stage of their retry loop.

4.8 Applications

4.8.1 Back-off Optimizations

When the parallel work is known, we can deduce from our analysis a simple and efficient back-off strategy: as we are able to estimate the value for which the throughput is maximum,

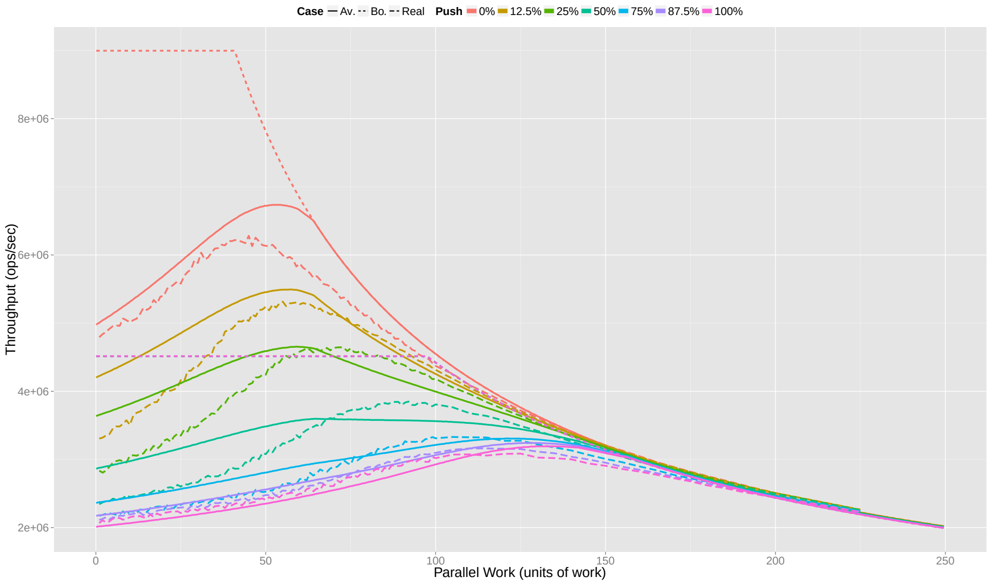


Figure 36: Operations on deque

we just have to back-off for the time difference between the peak pw and the actual pw. In Figure 38, we compare, on a synthetic workload, this constant back-off strategy against widely known strategies, namely exponential and linear, where the back-off amount increases exponentially or linearly after each failing retry loop starting from a 115 cycles step size. In Figure 37, we apply our constant back-off on a Delaunay triangulation application [62], provided with several workloads. The application uses a stack in two phases, whose first phase pushes elements on top of the stack without delay. We are able to estimate a corresponding back-off time, and we plot the results by normalizing the execution time of our back-offed implementation with the execution time of the initial implementation.

A measure or an estimate of pw is not always available (and could change over time, see next section), therefore we propose also an adaptive strategy: we incorporate in the data structure a monitoring routine that tracks the number of failed retries, employing a sliding window. As our analysis computes an estimate of the number of failed retries as a function of pw, we are able to estimate the current pw, and hence the corresponding back-off time like previously.

We test our adaptive back-off mechanism on a workload originated from [2], where global operators of exchanges for financial markets gather data of trades with a microsecond accuracy. We assume that the data comes from several streams, each of them being associated with a thread. All threads enqueue the elements that they receive in a concurrent queue, so that they can be later aggregated. We extract from the original data a trade stream distribution that we use to generate similar streams that reach the same thread; varying the number of streams to the same thread leads to different workloads. The results, represented as the normalized throughput (compared to the initial throughput) of trades that are en-

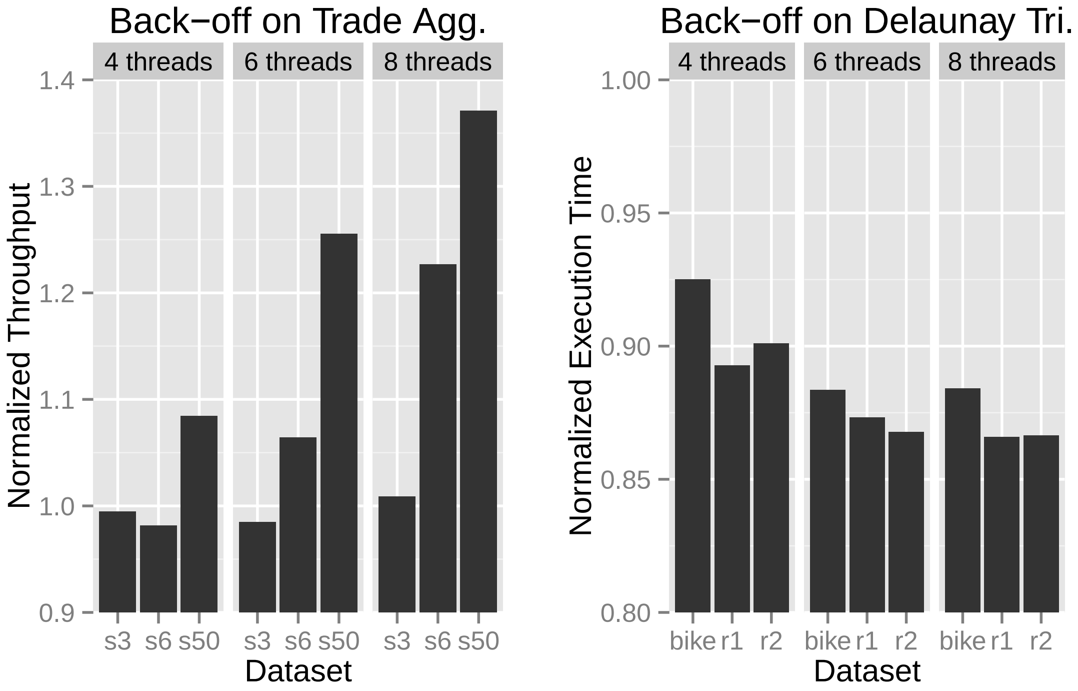


Figure 37: Performance impact of our back-off tunings

queued when the adaptive back-off is used, are plotted in Figure 37. For any number of threads, the queue is not contended on workload s3, hence our improvement is either small or slightly negative. On the contrary, the workload s50 contends the queue and we achieve very significant improvement.

4.8.2 Memory Management Optimization

Memory Management (MM) is an inseparable part of dynamic concurrent data structures. In contrary to lock-based implementations, a node that has been *removed* from a lock-free data structure can still be accessed by other threads, *e.g.* if they have been delayed. Collective decisions are thus required in order to *reclaim* a node in a safe manner. A well-known solution to deal with this problem is the hazard pointers technique [106].

A traditional design to implement this technique works as follows. Each thread $\mathcal{T}_i$, maintains two lists of nodes: $\mathcal{N}_i$ contains the nodes that $\mathcal{T}_i$ is currently accessing, and $\mathcal{D}_i$ stores the nodes that have been removed from the data structure by $\mathcal{T}_i$. Once a threshold on the size of $\mathcal{D}_i$ is reached, $\mathcal{T}_i$ calls a routine that: (i) collects the nodes that are accessed by any other thread, *i.e.* $\mathcal{N}_j$ for $j \neq i$ (collection phase), and (ii) for each element in $\mathcal{D}_i$, checks whether someone is accessing the element, *i.e.* whether it belongs to $\cup_{j \neq i} \mathcal{N}_j$, and if not, reclaims it (reclamation phase).

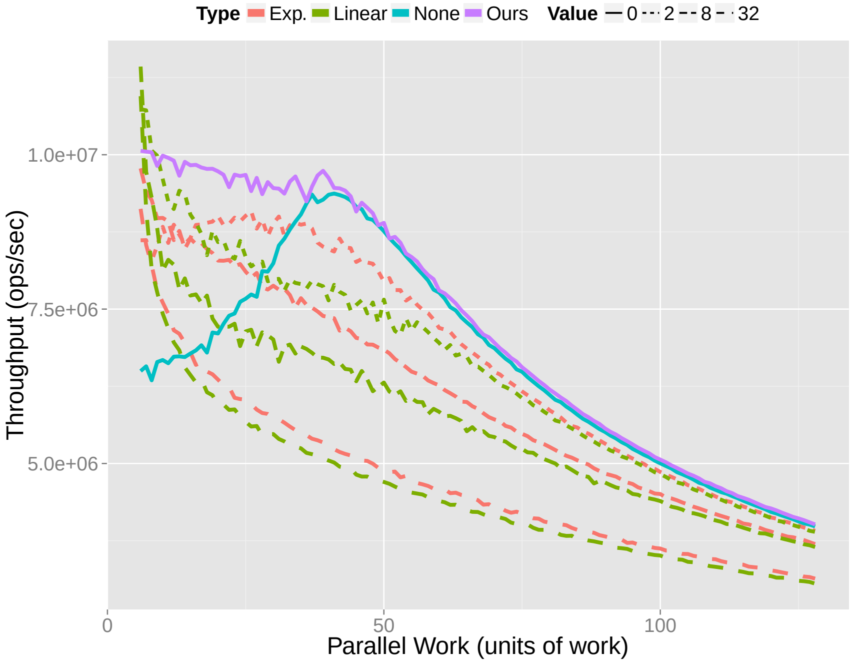


Figure 38: Back-off Tuning on Treiber's Stack

The main goal of our adaptive MM scheme is to distribute this extra-work in a way that the loss in performance is largely leveraged, knowing that additional work can be an advantage under high-contention (see previous section). The optimization is based on two main modifications. First, the granularity has to be finer, since the additional quantum that the back-off mechanism uses, has to be rather small (hundreds of cycles for a queue). Second, we need to track the contention level on the data structure in order to be able to inject the work at a proper execution point.

Fine-grain Memory Management Scheme: We divide the routine (and further the phases) of the traditional MM mechanism into quanta (equally-sized chunks).One quantum of the collection phase is the collection of the list of one thread, while three nodes are reclaimed during one quantum of the reclamation phase. The traditional MM scheme was parameterized by a threshold based on the number of the removed nodes; the fine-grain MM scheme is parameterized by the number of quanta that are executed at each call.

We apply different MM schemes on the `Dequeue` operation of the Michael-Scott queue, and plot the results in Figure 39. We initialize the queue with enough elements. Threads execute `Dequeue`, which returns an element, then call the MM scheme. On the left side, we compare a pure queue (without MM), a queue with the traditional MM (complete reclamation once in a while) and a queue with fine-grain MM (according to the numbers of quanta that are executed at each call). Note that the performance of the traditional MM is also subject to the tuning of the threshold parameter. We have tested and kept only the best parameter on the studied domain. First, unsurprisingly, we can observe that the pure queue outperforms the others as its *cw* is lower (no need to maintain the list of nodes that a thread is accessing). Second, as the fine-grain MM is called after each completed `Dequeue`, adding a constant work, the MM can be seen as a part of the parallel work. We highlight this idea on the second experiment (on the right side). We first measure the work done in a quantum. It follows that, for each value of the granularity parameter, we are able to estimate the effective parallel work as the sum of the initial *pw* and the work added by the fine-grain MM. Finally, we run the queue with the fine-grain MM, and plot the measured throughput, according to the effective parallel work, together with our two approaches instantiated with the effective *pw*. The graph shows the validity of the model estimations for all values of the granularity parameter.

Adaptive Memory Management Scheme: We build the adaptive MM scheme on top of the fine-grain MM mechanism by adding a monitoring routine that tracks the number of failed retry loops, employing a sliding windows. Given a granularity parameter and a number of failed retry loops, we are able to estimate the parallel work and the throughput, hence we can decide a change in the granularity parameter to reach the peak performance. Note that one can avoid memory explosion by specifying a threshold like the traditional implementation in case the application provides a durable low contention; in the worst case, it performs like the traditional MM.

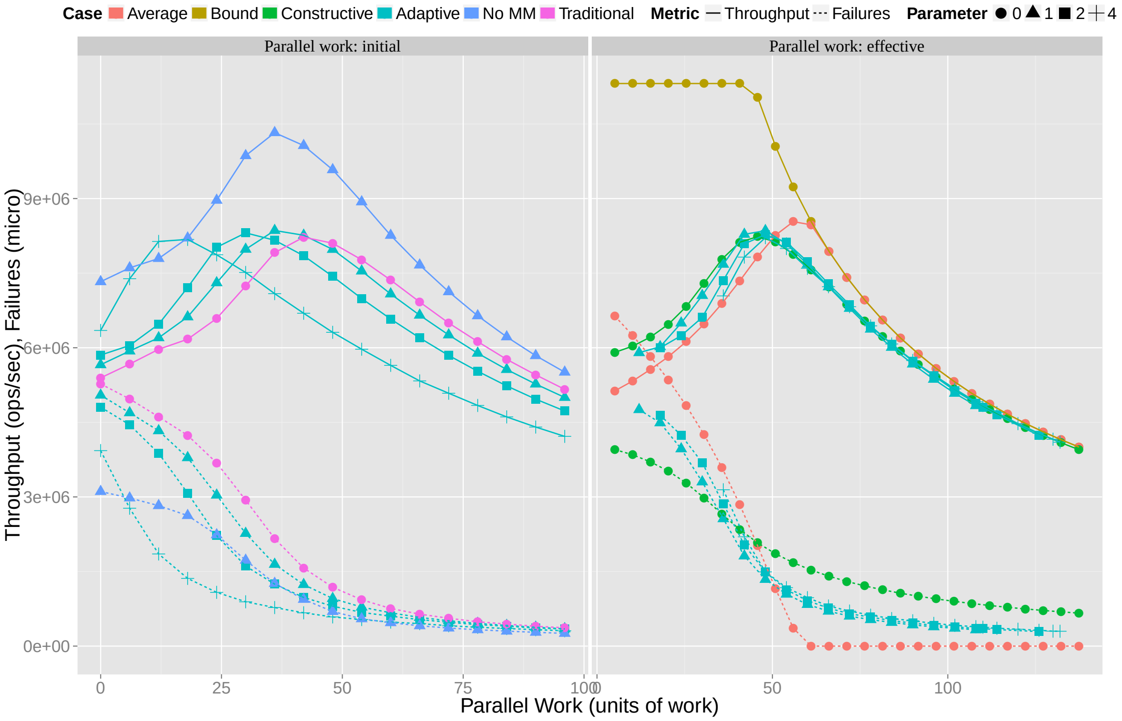


Figure 39: Performance of memory management mechanisms

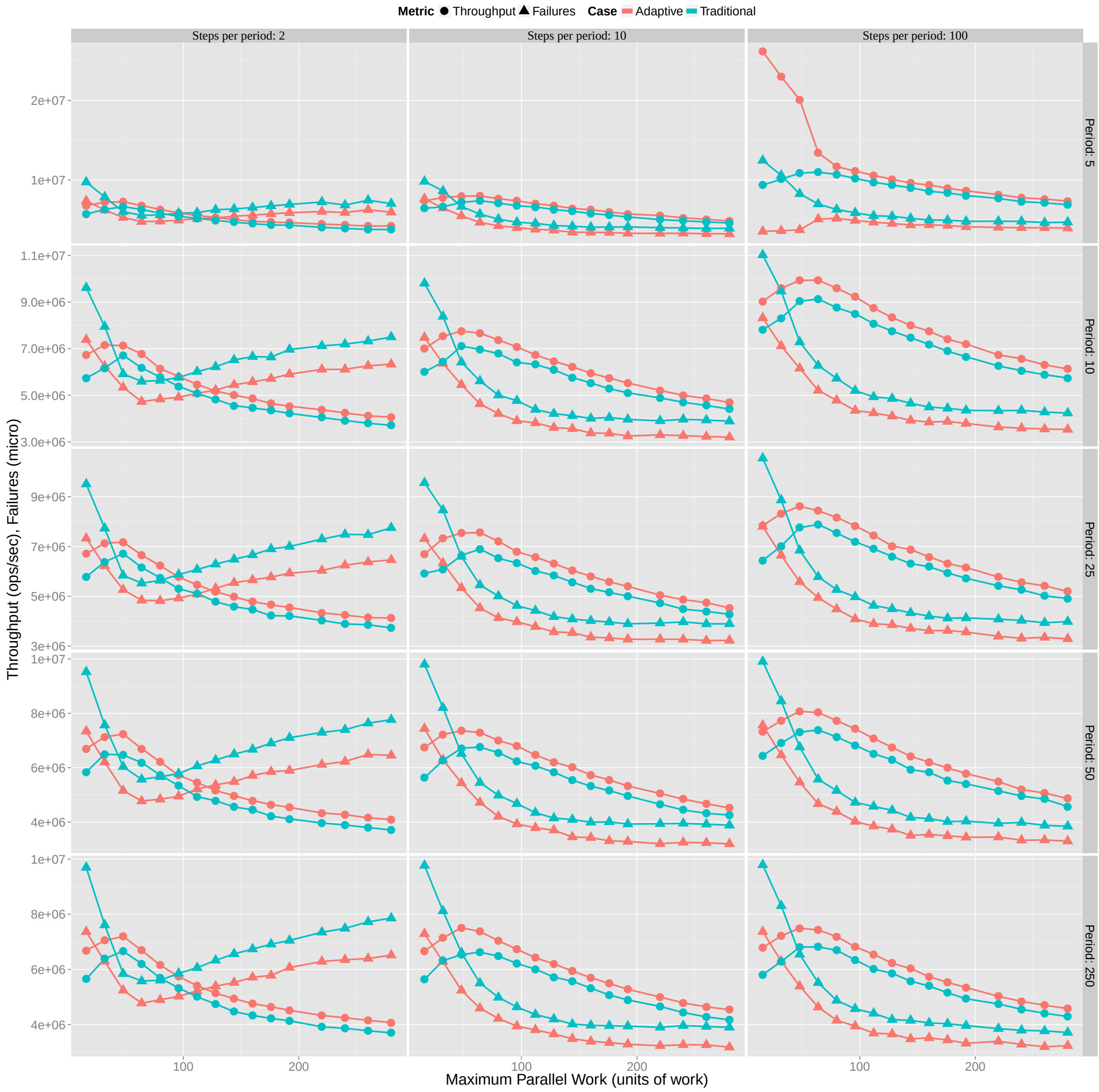


Figure 40: Adaptive MM with varying mean pw

Numerous scientific applications are built upon a pattern of alternating phases, that are communication- or computation-intensive. If the application involves data structures, it is expected that the rate of the modifications to the data structures is high in the data-oriented phases, and conversely. These phases could be clearly separated, but the application can also move gradually between phases. The rate of modification to a data structure will anyway oscillate periodically between two extreme values. We place ourselves in this context, and evaluate the two MMs accordingly. The parallel work still follows an exponential distribution of mean pw, but pw varies in a sinusoidal manner with time, in order to emulate the numerical phases. More precisely, pw is a step approximation of a sine function. Thus, two additional parameters rule the experiment: the period of the oscillating function represents the length of the phases, and the number of steps within a period depicts how continuous are the phase changes.

In Figure 40, we compare our approach with the traditional implementation for different periods of the sine function, on the `Dequeue` of the Michael-Scott queue [107]. The adaptive MM, that relies on the analysis presented in this work, outperforms the traditional MM because it provides an advantage both under low contention due to the costless (since delayed) invocation of the MM and under high contention due to the back-off effect.

4.9 Energy Modelling and Empirical Evaluation

We introduced our power model and the power impacting factors in D2.1 [75]. Then, we combined it with our initial performance model in D2.3 [73] to obtain the average power consumption in the static parallel programs that uses the fundamental lock-free data structures (*i.e.* the size of the parallel work that is executed in between data structure operations is constant).

Here, we take one step further and aim to obtain the energy efficiency of a wider range of lock-free data structure implementations that are used in the dynamic environments (*i.e.* the parallel work that is executed in between data structure operations follows a probability distribution). The performance analysis, that is presented above, can be used to predict the performance of such data structures in such environments. For the energy consumption estimations, we apply the methodology that was provided in D2.3, where we combine the power model presented in D2.1 with the performance estimations to obtain the average power consumption estimations.

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

In D2.3, we explain in detail the methodology to obtain the energy consumption estimations that span the whole parallel work and the number of threads domain. Here, we use the same power model that relies on the variation of dynamic components of the power in between the execution of the data structure operations and parallel work. Different from D2.3, here we back this power model with a more extensive performance analysis, presented above, in order to find the ratio of time that the parallel programs spend executing the data structure operations. Thanks to our performance analysis, we are able to estimate the

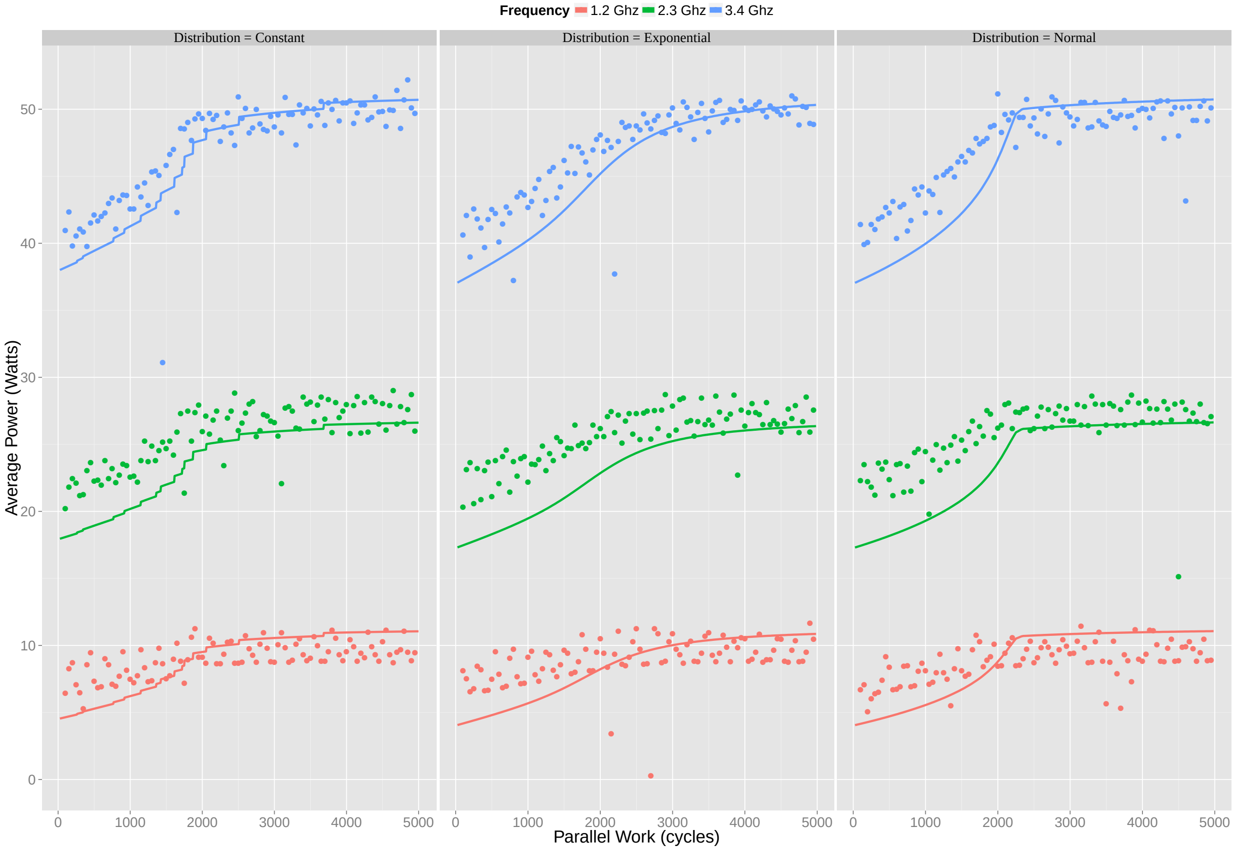


Figure 41: Average Power Consumption for Treiber's Stack (Pop operation)

energy consumption of lock-free data structures in dynamic environments where the size of parallel work, denoted by pw, is either constant or follows a probability distribution.

We present the results for a set of fundamental lock-free data structure operations, namely for Micheal and Scott Queue (Enqueue and Dequeue operations), Treiber's Stack (Pop operation) and Shared Counter (Increment operation). In the figures, x-axis provides the mean of pw which follows a probability distribution. Lines and points represent predictions and actual measurements, respectively. The performance estimations, for the different probability distributions, are conducted by making use of different approaches. We used, respectively, the approach presented in D2.3, the constructive approach in Section 4.5, the average-based approach in Section 4.4 for the cases where pw is constant, follows exponential distribution and follows normal distribution.

In the figures, we observe a similar behaviour. Dynamic CPU power decreases when pw decreases. We know that pw is a key aspect that influences the contention on the data structure, equally with the the ratio of time that threads spend in the data structure operation. Also, we can observe, though slightly, the difference between the different probability distributions of pw. For instance, the variation of the average power occurs more smoothly

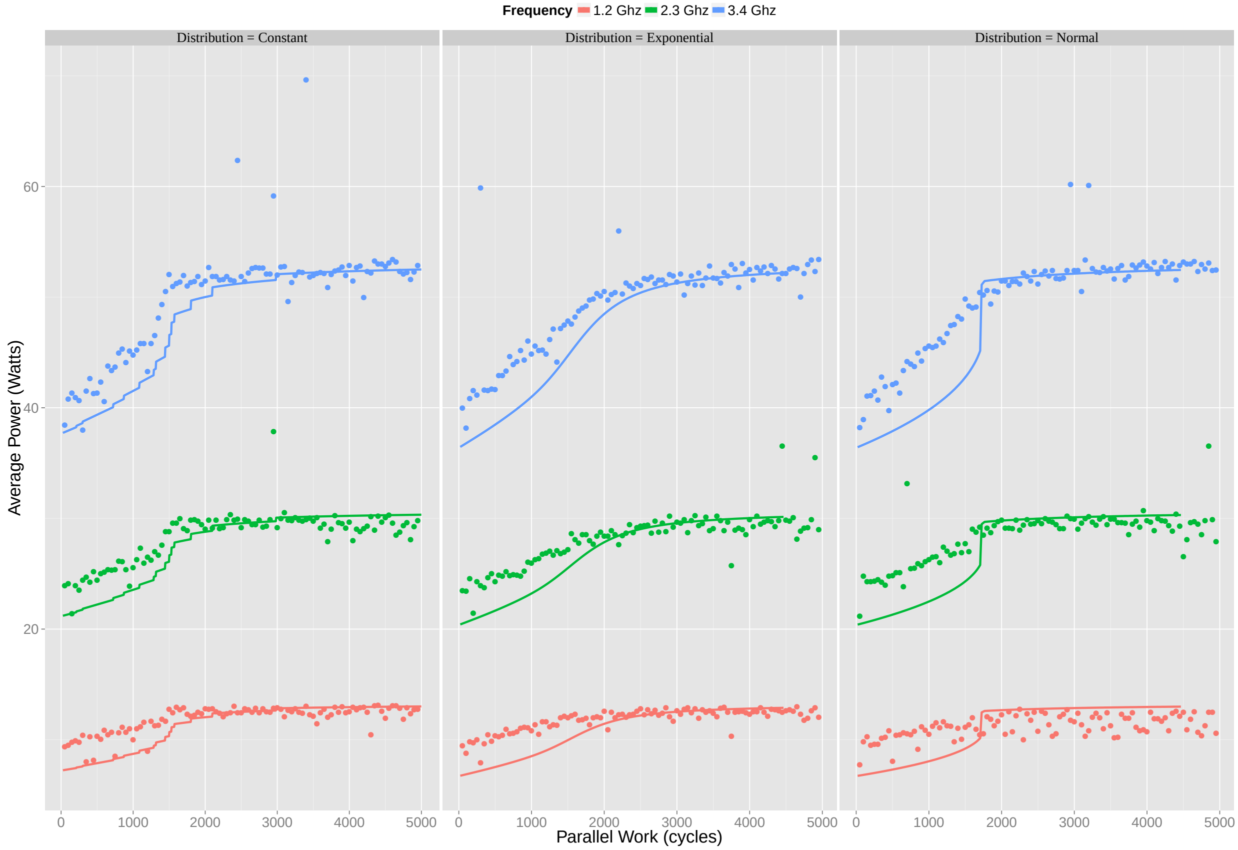


Figure 42: Average Power Consumption for Shared Counter (Increment operation)

when *pw* follows exponential distribution, similar to what is estimated by our model.

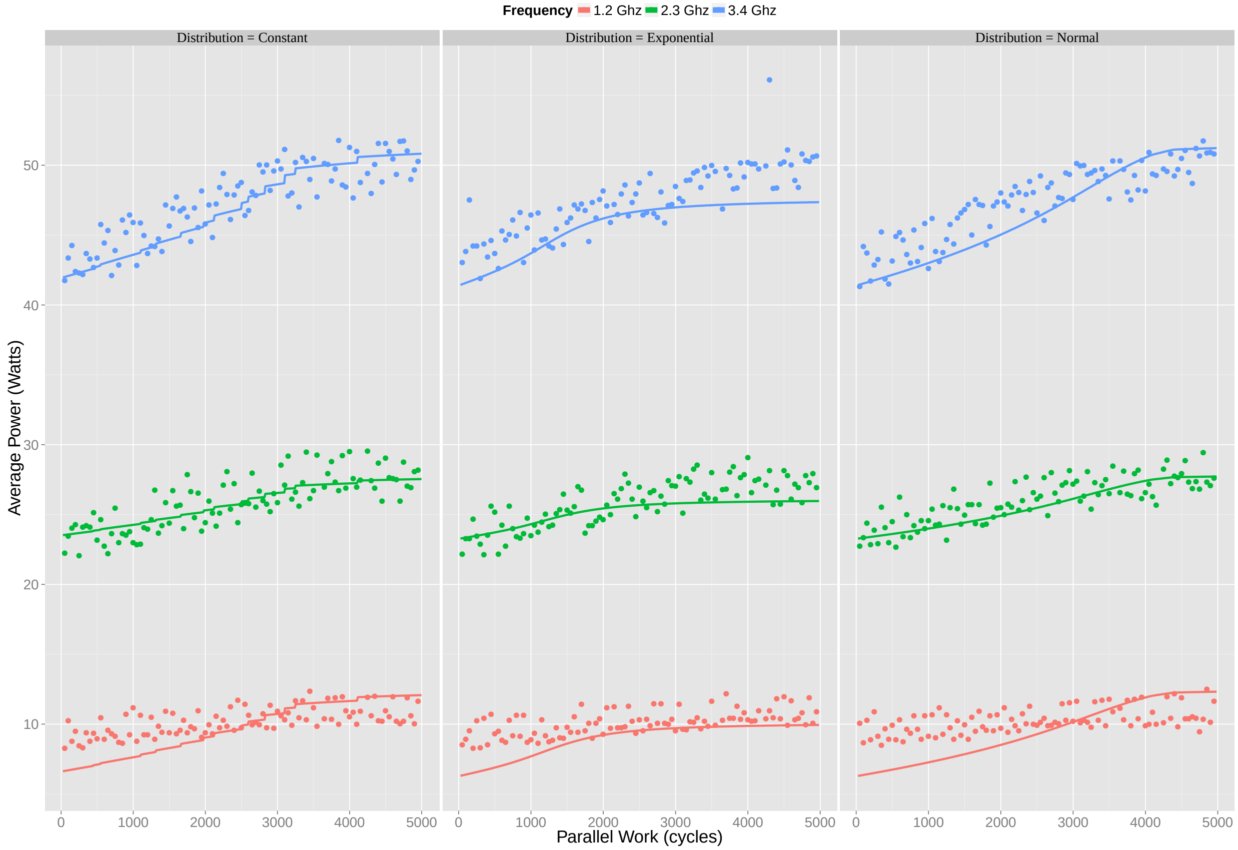


Figure 43: Average Power Consumption for MS Queue (Enqueue operation)

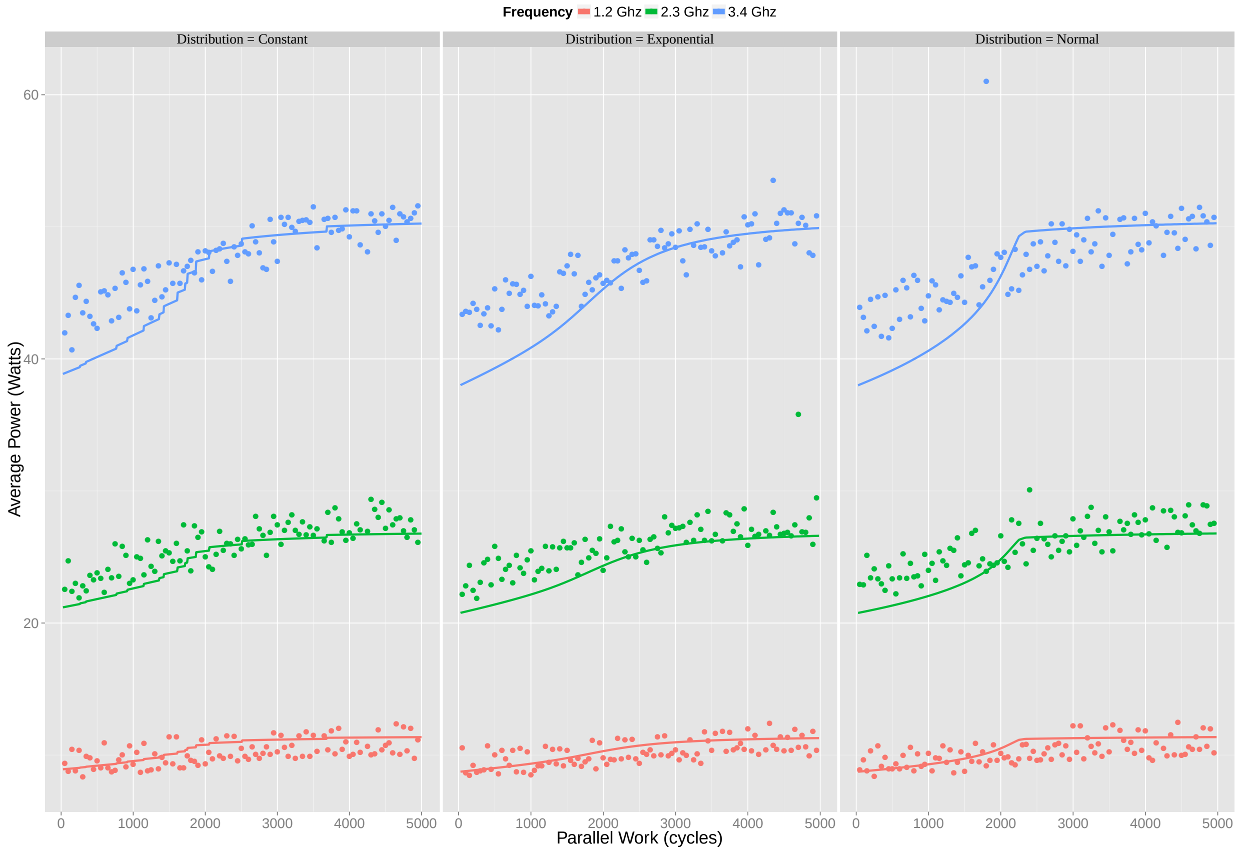


Figure 44: Average Power Consumption for MS Queue (Dequeue operation)

4.10 Conclusion

In this work we have presented two analyses for calculating the performance of lock-free data structures in dynamic environments. The first analysis has its roots in queuing theory, and gives the flexibility to cover a large spectrum of configurations. The second analysis makes use of Markov chains to exhibit a stochastic execution; it gives better results, but it is restricted to simpler data structures and exponentially distributed parallel work. We have evaluated the quality of the prediction on basic data structures like stacks, as well as more advanced data structures like optimized queues and deques. Our results can be directly used by algorithmicians to gain a better understanding of the performance behavior of different designs, and by experimentalists to rank implementations within a fair framework. We have also shown how to use our results to tune applications using lock-free codes. These tuning methods include: (i) the calculation of simple and efficient back-off strategies whose applicability is illustrated in application contexts; (ii) a new adaptative memory management mechanism that acclimates to a changing environment.

Moreover, we have integrated the performance estimations with the power model to estimate the energy efficiency of lock-free data structure by using only a restricted amount of information about the application at hand.

The main differences between the data structures of this work and linked lists, skip lists and trees occur when the size of the data structure grows. With large sizes, the performance is dominated by the traversal cost that is ruled by the cache parameters. The reduction in the size of the data structure decreases the traversal cost which in turn increases the probability of encountering an on-going CAS operation that delays the threads which traverse the link. The expansion, which can additionally be supported unfavorably by helping mechanisms, appears then as the main performance degrading factor. While the analysis becomes easier for high degrees of parallelism (large data structure size), being able to describe the behavior of lock-free data structures as the degree of parallelism changes constitutes the main challenge of our future work.

5 A General and Validated Energy Complexity Model for Multi-threaded Algorithms

In this Deliverable D2.4, we report the ICE (Ideal Cache Energy) complexity model for analyzing the energy complexity of a wide range of multi-threaded algorithms [126]. Compared to the EPEM model reported in D2.3, this model proposed using Ideal Cache memory model to compute I/O complexity of the algorithms. Besides a case study of SpMV to demonstrate how to apply the ICE model to find energy complexity of parallel algorithms which is described in Deliverable D2.3, Deliverable D2.4 also reports a case study to apply the ICE model to Dense Matrix Multiplication (matmul) (cf. Section 5.6). The model is then validated with both data-intensive (i.e., SpMV) and computation-intensive (i.e., matmul) algorithms according to three aspects: different algorithms, different input types/sizes and different platforms (cf. Section 5.7). In order to make the reading flow easy to follow, we include in this report a complete study of ICE model along with latest results.

5.1 Introduction

As described in Deliverable D2.3, understanding the energy complexity of algorithms is crucially important to improve the energy efficiency of algorithms and reduce the energy consumption of computing systems [74, 96]. One of the main approaches to understand the energy complexity of algorithms is to devise energy models.

Significant efforts have been devoted to developing power and energy models in literature [17, 41, 40, 92, 93, 86, 109, 123]. However, there are no analytic models for multithreaded algorithms that are both applicable to a wide range of algorithms and comprehensively validated yet (cf. Table 10). The existing *parallel* energy models are either theoretical studies without validation or only applicable for specific algorithms. Modeling energy consumption of *parallel* algorithms is difficult since the energy models must take into account the complexity of both parallel algorithms and parallel platforms. The algorithm complexity results from parallel computation, concurrent memory accesses and inter-process communication. The platform complexity results from multicore architectures with deep memory hierarchy.

The existing models and their classification are summarized in Table 10. To the best of our knowledge, the proposed ICE (Ideal Cache Energy) complexity model is the first energy model that covers all three aspects: i) ability to analyze the energy complexity of parallel algorithms (i.e. Energy complexity analysis for parallel algorithms), ii) applicability to a wide range of algorithms (i.e., Algorithm generality), and iii) model validation (i.e., Validation). Section 5.2 describes how the ICE model complements the other currently used models.

The energy complexity model ICE proposed in this study is for general multithreaded algorithms and validated on three aspects: different algorithms for a given problem, different input types and different platforms. The proposed model is an analytic model which characterizes both algorithms (e.g., representing algorithms by their *work*, *span* and *I/O* complexity) and platforms (e.g., representing platforms by their static and dynamic energy

Table 10: Energy Model Summary

Study	**Energy complexity analysis for parallel algorithms**	**Algorithm generality**	**Validation**
LEO [109]	No	General	Yes
POET [86]	No	General	Yes
Koala [123]	No	General	Yes
Roofline [41, 40]	No	General	Yes
Energy scalability [92, 93]	Yes	General	No
Sequential energy complexity [116]	No	General	Yes
Alonso et al. [17]	Yes	Algorithm-specific	Yes
Malossi et al. [104]	Yes	Algorithm-specific	Yes
ICE model (this study)	**Yes**	**General**	**Yes**

of memory accesses and computational operations). By considering *work*, *span* and *I/O* complexity, the new ICE model is applicable to any multithreaded algorithms.

The new ICE model is designed for analyzing the energy *complexity* of algorithms and therefore the model does not provide the estimation of absolute energy consumption. The goal of the ICE model is to answer energy complexity question: "Given two parallel algorithms A and B for a given problem, which algorithm consumes less energy analytically?". Hence, the details of underlying systems (e.g., runtime and architectures) are abstracted away to keep ICE model simple and suitable for complexity analysis. O-notation represents an *asymptotic upper-bound* on energy complexity.

In this work, the following contributions have been made.

- Devising a new general energy model ICE for analyzing the energy complexity of a wide range of multithreaded algorithms based on their *work*, *span* and *I/O* complexity (cf. Section 5.4). The new ICE model abstracts away possible *multicore platforms* by their static and dynamic energy of computational operations and memory access. The new ICE model complements previous energy models such as energy roofline models [41, 40] that abstract away possible *algorithms* to analyze the energy consumption of different multicore platforms.

- Conducting two case studies (i.e., SpMV and matmul) to demonstrate how to apply the ICE model to find energy complexity of parallel algorithms. The selected parallel algorithms for SpMV are three algorithms: Compressed Sparse Column(CSC), Compressed Sparse Block(CSB) and Compressed Sparse Row(CSR)(cf. Section 5.5). The selected parallel algorithms for matmul are two algorithms: a basic matmul algorithm and a cache-oblivious algorithm (cf. Section 5.6).

- Validating the ICE energy complexity model with both data-intensive (i.e., SpMV) and computation-intensive (i.e., matmul) algorithms according to three aspects: different algorithms, different input types and different platforms. The results show the precise prediction on which validated SpMV algorithm (i.e., CSB or CSC) consumes more energy when using different matrix input types from Florida matrix collection [50] (cf. Section 5.7.5). The results also show the precise prediction on which validated matmul algorithm (i.e., basic or cache-oblivious) consumes more energy (cf. Section 5.7.6). The model platform-related parameters for 11 platforms, including x86, ARM and GPU, are provided to facilitate the deployment of the ICE model.

5.2 Related Work - Overview of energy models

We also included the related work of the most well-known energy models in this report to show why we need the new proposed ICE model. Energy models for finding energy-optimized system configurations for a given application have been recently reported [12, 16, 19]. Imes et al. [86] used controller theory and linear programming to find energy-optimized configurations for an application with soft real-time constraints at runtime. Mishra et al. [109] used hierarchical Bayesian model in machine learning to find energy-optimized configurations. Snowdon et al. [123] developed a power management framework called Koala which models the energy consumption of the platform and monitors an application' energy behavior. Although the energy models for finding energy-optimized system configurations have resulted in energy saving in practice, they focus on characterizing system platforms rather than applications and therefore are not appropriate for analyzing the energy complexity of application algorithms.

Another direction of energy modeling study is to predict the energy consumption of applications by analyzing applications without actual execution on real platforms which we classify as analytic models.

Among energy and power models for different architectures [41, 40, 99, 101, 125, 129], energy roofline models [41, 40] are some of the comprehensive energy models that abstract away possible algorithms in order to analyze and characterize different multicore platforms in terms of energy consumption. Our new energy model, which abstracts away possible multicore platform and characterize the energy complexity of algorithms based on their *work, span* and *I/O* complexity, complements the energy roofline models.

Validated energy models for *specific* algorithms have been reported recently [17, 104]. Alonso et al. [17] provided an accurate energy model for three key dense matrix factorizations. Malossi et al. [104] focused on basic linear-algebra kernels and characterized the kernels by the number of arithmetic operations, memory accesses, reduction and barrier steps. Although the energy models for specific algorithms are accurate for the target algorithms, they are not applicable for other algorithms and therefore cannot be used as general energy complexity models for parallel algorithms.

The *energy scalability* of a parallel algorithm has been investigated by Korthikanti et al. [92, 93]. Unlike the energy scalability studies that have not been validated on real platforms, our new energy complexity model is validated on HPC and accelerator platforms, confirming

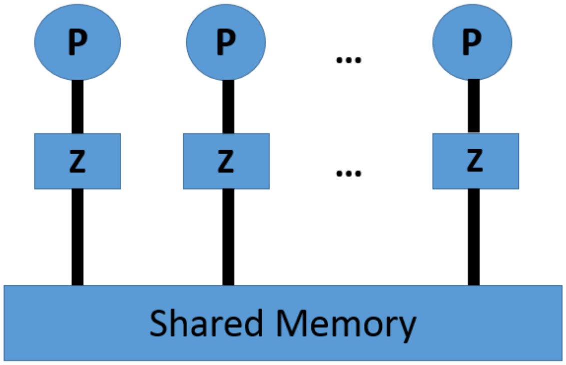


Figure 45: A Shared Memory Machine Model with Private Caches

its usability and accuracy.

The energy complexity of *sequential* algorithms on a *uniprocessor* machine with *several memory banks* has been studied by Roy et al. [116]. Our energy complexity studies complement Roy et al.'s studies by investigating the energy complexity of *parallel* algorithms on a *multiprocessor* machine with *a shared memory bank* and private caches, a machine model that has been widely adopted to study parallel algorithms [60, 21, 93].

5.3 ICE Shared Memory Machine Model

Generally speaking, the energy consumption of a parallel algorithm is the sum of i) static energy (or leakage) E_{static}, ii) dynamic energy of computation E_{comp} and iii) dynamic energy of memory accesses E_{mem}. The static energy E_{static} is proportional to the execution time of the algorithm while the dynamic energy of computation and the dynamic energy of memory accesses are proportional to the number of computational operations and the number of memory accesses of the algorithm, respectively [93]. As a result, in the new ICE complexity model, the energy complexity of a multithreaded algorithm is analyzed based on its *span complexity* [45] (for the static energy), *work complexity* [45] (for the dynamic energy of computation) and *I/O complexity* (for the dynamic energy of memory accesses) (cf. Section 5.4). This section describes shared-memory machine models supporting I/O complexity analysis for parallel algorithms.

The first memory model we consider is parallel external memory (PEM) model [21], an extension of the Parallel Random Access Machine (PRAM) model that includes a two-level memory hierarchy. In the PEM model, there are n cores (or processors) each of which has its own *private* cache of size Z (in bytes) and shares the main memory with the other cores (cf. Figure 45). When n cores access n distinct blocks from the shared memory *simultaneously*, the I/O complexity in the PEM model is $O(1)$ instead of $O(n)$. Although the PEM model is appropriate for analyzing the I/O complexity of parallel algorithms in terms of time performance [21], we have found that the PEM model is not appropriate for analyzing parallel algorithms in terms of the dynamic energy of memory accesses. In fact, even when the n cores can access data from the main memory simultaneously, the *dynamic* energy consumption of the access is proportional to the number n of accessing cores (because of the load-store unit activated within each accessing core and the energy compositionality of parallel computations [69, 100]), rather than a constant as implied by the PEM model.

As a result, we consider the ideal distributed cache (IDC) model [60] to analyze I/O complexity of multithreaded algorithms in terms of dynamic energy consumption. Since the cache complexity of m misses is $O(m)$ regardless of whether or not the cache misses are incurred simultaneously by the cores, the IDC model reflects the aforementioned dynamic energy consumption of memory accesses by the cores.

However, the IDC model is mainly designed for analyzing the cache complexity of divide-and-conquer algorithms, making it difficult to apply to general multi-threaded algorithms targeted by our new ICE model. Constraining the new ICE model to the IDC model would limit the applicability of the ICE model to a wide range of multithreaded algorithms.

In order to make our new ICE complexity model applicable to a wide range of multithreaded algorithms, we show that the cache complexity analysis using the traditional (sequential) ideal cache (IC) model [59] can be used to find an upper bound on the cache complexity of the same algorithm using the IDC model (cf. Lemma 7). As the sequential execution of multithreaded algorithms is a valid execution regardless of whether they are divide-or-conquer algorithms, the ability to analyze the cache complexity of multithreaded algorithms via their sequential execution in the ICE complexity model improves the usability of the ICE model.

Let $Q_1(Alg, B, Z)$ and $Q_P(Alg, B, Z)$ be the cache complexity of a parallel algorithm Alg analyzed in the (uniprocessor) ideal cache (IC) model [59] with block size B and cache size Z (i.e, running Alg with a single core) and the cache complexity analyzed in the (multicore) IDC model with P cores each of which has a private cache of size Z and block size B, respectively. We have the following lemma:

Lemma 7. *The cache complexity $Q_P(Alg, B, Z)$ of a parallel algorithm Alg analyzed in the ideal distributed cache (IDC) model with P cores is bounded from above by the product of P and the cache complexity $Q_1(Alg, B, Z)$ of the same algorithm analyzed in the ideal cache (IC) model. Namely,*

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

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

$$Q_P^i(Alg, B, Z) \leq Q_1(Alg, B, Z) \tag{18}$$

or

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

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

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

From Equations 19 and 20, we have

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

□

We also make the following assumptions regarding platforms.

- Algorithms are executed with the best configuration (e.g., maximum number of cores, maximum frequency) following the race-to-halt strategy.
- The I/O parallelism is bounded from above by the computation parallelism. Namely, each core can issue a memory request only if its previous memory requests have been served. Therefore, the work and span (i.e., critical path) of an algorithm represent the parallelism for both I/O and computation [45].

5.4 Energy Complexity in ICE model

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

5.4.1 Platform-supporting Energy Complexity Model

This section describes a methodology to find energy complexity of algorithms. The energy complexity model considers three groups of parameters: machine-dependent, algorithm-dependent and input-dependent parameters. The reason to consider all three parameter-categories is that only operational intensity [138] is insufficient to capture the characteristics of algorithms. Two algorithms with the same values of operational intensity might consume different levels of energy. The reasons are their differences in data accessing patterns leading to performance scalability gap among them. For example, although the sequential version and parallel version of an algorithm may have the same operational intensity, they may have different energy consumption since the parallel version would have less static energy consumption because of shorter execution time.

The energy consumption of a parallel algorithm is the sum of i) static energy (or leakage) E_{static}, ii) dynamic energy of computation E_{comp} and iii) dynamic energy of memory accesses

Table 11: ICE Model Parameter Description

Machine	Description
ϵ_{op}	dynamic energy of one operation (average)
$\epsilon_{I/O}$	dynamic energy of a random memory access (1 core)
π_{op}	static energy when performing one operation
$\pi_{I/O}$	static energy of a random memory access
Algorithm	**Description**
$Work$	Number of work in flops of the algorithm [45]
$Span$	The critical path of the algorithm [45]
I/O	Number of cache line transfer of the algorithm [45]

E_{mem}: $E = E_{static} + E_{comp} + E_{mem}$ [41, 92, 93]. The static energy E_{static} is the product of the execution time of the algorithm and the static power of the whole platform. The dynamic energy of computation and the dynamic energy of memory accesses are proportional to the number of computational operations $Work$ and the number of memory accesses I/O, respectively. Pipelining technique in modern architectures enables overlapping computation with memory accesses [69]. Since computation time and memory-access time can be overlapped, the execution time of the algorithm is assumed to be the maximum of computation time and memory-access time [41]. Therefore, the energy consumption of algorithms is computed by Equation 22, where the values of ICE parameters, including ϵ_{op}, $\epsilon_{I/O}$, π_{op}, and $\pi_{I/O}$ are described in Table 11 and computed by the Equation 23, 24, 25, and 26, respectively.

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

$$\epsilon_{op} = P^{op} \times \frac{F}{Freq} \tag{23}$$

$$\epsilon_{I/O} = P^{I/O} \times \frac{M}{Freq} \tag{24}$$

$$\pi_{op} = P^{sta} \times \frac{F}{Freq} \tag{25}$$

$$\pi_{I/O} = P^{sta} \times \frac{M}{Freq} \tag{26}$$

The dynamic energy of one operation by one core ϵ_{op} is the product of the consumed power of one operation by one active core P^{op} and the time to perform one operation. Equation 23 shows how ϵ_{op} relates to frequency $Freq$ and the number of cycles per operation F. Similarly, the dynamic energy of a random access by one core $\epsilon_{I/O}$ is the product of the

Table 12: Platform parameter summary. The parameters of the first nine platforms are derived from [40] and the parameters of the two new platforms are found in this study.

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

consumed power by one active core performing one I/O (i.e., cache-line transfer) $P^{I/O}$ and the time to perform one cache line transfer computed as $M/Freq$, where M is the number of cycles per cache line transfer (cf. Equation 24). The static energy of operations π_{op} is the product of the whole platform static power P^{sta} and time per operation. The static energy of one I/O $\pi_{I/O}$ is the product of the whole platform static power and time per I/O, shown by Equation 25 and 26.

In order to compute *work*, *span* and *I/O* complexity of the algorithms, the input parameters also need to be considered. For example, SpMV algorithms consider input parameters listed in Table 13. Cache size is captured in the ICE model by the *I/O complexity* of the algorithm. Note that in the ICE machine model (Section 5.3), cache size Z is a constant and may disappear in the *I/O complexity* (e.g., O-notation).

The details of how to obtain the ICE parameters of recent platforms are discussed in Section 5.7.1. The actual values of ICE platform parameters for 11 recent platforms are presented in Table 12.

The computation time of parallel algorithms is proportional to the span complexity of the algorithm, which is $T^{comp} = \frac{Span \times F}{Freq}$ where $Freq$ is the processor frequency, and F is the number of cycles per operation. The memory-access time of parallel algorithms in the ICE model is proportional to the I/O complexity of the algorithm divided by its I/O parallelism, which is $T^{mem} = \frac{I/O}{I/O-parallelism} \times \frac{M}{Freq}$. As I/O parallelism, which is the average number of I/O ports that the algorithm can utilize per step along the span, is bounded by the computation parallelism $\frac{Work}{Span}$, namely the average number of cores that the algorithm can utilize per step along the span (cf. Section 5.3), the memory-access time T^{mem} becomes: $T^{mem} = \frac{I/O \times Span \times M}{Work \times Freq}$ where M is the number of cycles per cache line transfer. If an algorithm has T^{comp} greater than T^{mem}, the algorithm is a CPU-bound algorithm. Otherwise, it is a memory-bound algorithm.

CPU-bound Algorithms

If an algorithm has computation time T^{comp} longer than data-accessing time T^{mem} (i.e., CPU-bound algorithms), the ICE energy complexity model becomes Equation 27 which is simplified as Equation 28.

$$E = \epsilon_{op} \times Work + \epsilon_{I/O} \times I/O + P^{sta} \times \frac{Span \times F}{Freq} \tag{27}$$

or

$$E = \epsilon_{op} \times Work + \epsilon_{I/O} \times I/O + \pi_{op} \times Span \tag{28}$$

Memory-bound Algorithms

If an algorithm has data-accessing time longer than computation time (i.e., memory-bound algorithms): $T^{mem} \geq T^{comp}$, energy complexity becomes Equation 29 which is simplified as Equation 30.

$$E = \epsilon_{op} \times Work + \epsilon_{I/O} \times I/O + P^{sta} \times \frac{I/O \times Span \times M}{Work \times Freq} \tag{29}$$

or

$$E = \epsilon_{op} \times Work + \epsilon_{I/O} \times I/O + \pi_{I/O} \times \frac{I/O \times Span}{Work} \tag{30}$$

5.4.2 Platform-independent Energy Complexity Model

This section describes the energy complexity model that is platform-independent and considers only algorithm characteristics. When the platform parameters (i.e., ϵ_{op}, $\epsilon_{I/O}$, π_{op}, and $\pi_{I/O}$) are unavailable, the energy complexity model is derived from Equation 22, where the platform parameters are constants and can be removed. Assuming $\pi_{max} = max(\pi_{op}, \pi_{I/O})$, after removing platform parameters, the platform-independent energy complexity model are shown in Equation 31.

$$E = O(Work + I/O + max(Span, \frac{I/O \times Span}{Work})) \tag{31}$$

5.5 A Case Study of Sparse Matrix Multiplication

SpMV is one of the most common application kernels in Berkeley dwarf list [22]. It computes a vector result y by multiplying a sparse matrix A with a dense vector x: $y = Ax$. SpMV is a data-intensive kernel and has irregular memory-access patterns. The data access patterns for SpMV is defined by its sparse matrix format and matrix input types. There

Table 13: SpMV Input Parameter Description

SpMV Input	Description
n	Number of rows
nz	Number of nonzero elements
nr	Maximum number of nonzero in a row
nc	Maximum number of nonzero in a column
β	Size of a block

are several sparse matrix formats and SpMV algorithms in literature. To name a few, they are Coordinate Format (COO), Compressed Sparse Column (CSC), Compressed Sparse Row (CSR), Compressed Sparse Block (CSB), Recursive Sparse Block (RSB), Block Compressed Sparse Row (BCSR) and so on. Three popular SpMV algorithms, namely CSC, CSB and CSR are chosen to validate the proposed energy complexity model. They have different data-accessing patterns leading to different values of I/O, work and span complexity. Since SpMV is a memory-bound application kernel, Equation 30 is applied. The input matrices of SpMV have different parameters listed in Table 13.

5.5.1 Compressed Sparse Row

CSR is a standard storage format for sparse matrices which reduces the storage of matrix compared to the tuple representation [94]. This format enables row-wise compression of A with size $n \times n$ (or $n \times m$) to store only the non-zero nz elements. Let nz be the number of non-zero elements in matrix A. The *work* complexity of CSR SpMV is $\Theta(nz)$ where $nz >= n$ and *span* complexity is $O(nr + \log n)$ [38], where nr is the maximum number of non-zero elements in a row. The *I/O* complexity of CSR in the sequential I/O model of row-major layout is $O(nz)$ [32] namely, scanning all non-zero elements of matrix A costs $O(\frac{nz}{B})$ I/Os with B is the cache block size. However, randomly accessing vector x causes the total of $O(nz)$ I/Os. Applying the proposed model on CSR SpMV, their total energy complexity are computed as Equation 32.

$$E_{CSR} = O(\epsilon_{op} \times nz + \epsilon_{I/O} \times nz + \pi_{I/O} \times (nr + \log n)) \tag{32}$$

5.5.2 Compressed Sparse Column

CSC is the similar storage format for sparse matrices as CSR. However, it compresses the sparse matrix in column-wise manner to store the non-zero elements. The *work* complexity of CSC SpMV is $\Theta(nz)$ where $nz >= n$ and *span* complexity is $O(nc + \log n)$, where nc is the maximum number of non-zero elements in a column. The *I/O* complexity of CSC in the sequential I/O model of column-major layout is $O(nz)$ [32]. Similar to CSR, scanning all non-zero elements of matrix A in CSC format costs $O(\frac{nz}{B})$ I/Os. However, randomly

$$E_{CSB} = O(\epsilon_{op} \times (\frac{n^2}{\beta^2} + nz) + \epsilon_{I/O} \times (\frac{n^2}{\beta^2} + \frac{nz}{B}) + \pi_{I/O} \times \frac{(\frac{n^2}{\beta^2} + \frac{nz}{B}) \times (\beta \times \log\frac{n}{\beta} + \frac{n}{\beta})}{(\frac{n^2}{\beta^2} + nz)}) \quad (34)$$

Table 14: SpMV Complexity Analysis

Complexity	CSC-SpMV	CSB-SpMV	CSR-SpMV
Work	$\Theta(nz)$ [38]	$\Theta(\frac{n^2}{\beta^2} + nz)$ [38]	$\Theta(nz)$ [38]
I/O	$O(nz)$ [32]	$O(\frac{n^2}{\beta^2} + \frac{nz}{B})$ [this study]	$O(nz)$ [32]
Span	$O(nc + \log n)$ [38]	$O(\beta \times \log\frac{n}{\beta} + \frac{n}{\beta})$ [38]	$O(nr + \log n)$ [38]

updating vector y causing the bottle neck with total of $O(nz)$ I/Os. Applying the proposed model on CSC SpMV, their total energy complexity are computed as Equation 33.

$$E_{CSC} = O(\epsilon_{op} \times nz + \epsilon_{I/O} \times nz + \pi_{I/O} \times (nc + \log n)) \quad (33)$$

5.5.3 Compressed Sparse Block

Given a sparse matrix A, while CSR has good performance on SpMV $y = Ax$, CSC has good performance on transpose sparse matrix vector multiplication $y = A^T \times x$, Compressed sparse blocks (CSB) format is efficient for computing either Ax or $A^T x$. CSB is another storage format for representing sparse matrices by dividing the matrix A and vector x, y to blocks. A block-row contains multiple chunks, each chunks contains consecutive blocks and non-zero elements of each block are stored in Z-Morton-ordered [38]. From Beluc et al. [38], CSB SpMV computing a matrix with nz non-zero elements, size $n \times n$ and divided by block size $\beta \times \beta$ has span complexity $O(\beta \times \log\frac{n}{\beta} + \frac{n}{\beta})$ and *work* complexity as $\Theta(\frac{n^2}{\beta^2} + nz)$.

I/O complexity for CSB SpMV is not available in the literature. We do the analysis of CSB manually by following the master method [45]. The *I/O* complexity is analyzed for the algorithm CSB_SpMV(A,x,y) from Beluc et al. [38]. The I/O complexity of CSB is similar to *work* complexity of CSB $O(\frac{n^2}{\beta^2} + nz)$, only that non-zero accesses in a block is divided by B: $O(\frac{n^2}{\beta^2} + \frac{nz}{B})$, where B is cache block size. The reason is that non-zero elements in a block are stored in Z-Morton order which only requires $\frac{nz}{B}$ I/Os. The energy complexity of CSB SPMV is shown in Equation 34.

From the complexity analysis of SpMV algorithms using different layouts, the complexity of CSR-SpMV, CSC-SpMV and CSB-SpMV are summarized in Table 14.

5.6 A Case Study of Dense Matrix Multiplication

Besides SpMV, we also apply the ICE model to dense matrix multiplication (matmul). Unlike SpMV, a data-intensive kernel, matmul is a computation-intensive kernel used in

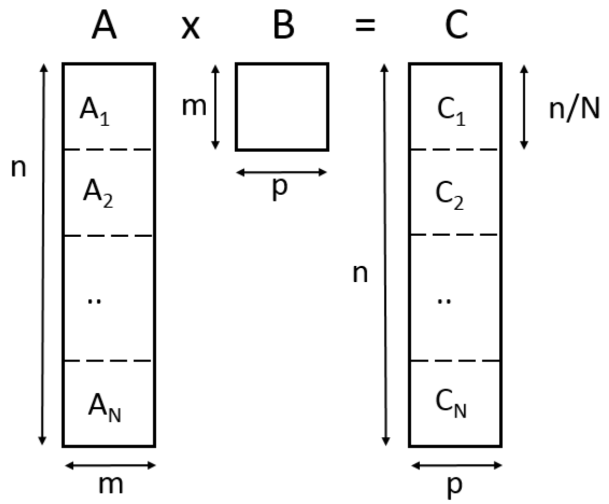


Figure 46: Partition approach for parallel matmul algorithms. Each sub-matrix A_i has size $\frac{n}{N} \times m$ and each sub-matrix C_i has size $\frac{n}{N} \times p$.

high performance computing. It computes output matrix C (size n x p) by multiplying two dense matrices A (size n x m) and B (size m x p): $C = A \times B$. In this work, we implemented two matmul algorithms (i.e., a basic algorithm and a cache-oblivious algorithm [59]) and apply the ICE analysis to find their energy complexity. Both algorithms partition matrix A and C equally to N sub-matrices (e.g., A_i with i=(1,2,..,N)), where N is the number of cores in the platform. The partition approach is shown in Figure 46. Each core computes a sub-matrix C_i: $C_i = A_i \times B$. Since matmul is a computation-bound application kernel, Equation 28 is applied.

5.6.1 Basic Matmul Algorithm

The basic matmul algorithm is described in Listing 1. Its work complexity is $\Theta(2nmp)$ [140] and span complexity is $\Theta(\frac{2nmp}{N})$ because the computational work is divided equally to N cores due to matrix partition approach. When matrix size of matrix B is bigger than the platform cache size, the basic algorithm loads matrix B n times (i.e., once for computing each row of C), results in $\frac{nmp}{B}$ cache block transfers, where B is cache block size. In total, I/O complexity of the basic matmul algorithm is $\Theta(\frac{nm+nmp+np}{B})$. Applying the ICE model on this algorithm, the total energy complexity are computed as Equation 35.

Listing 1: Simple Matmul

```
for i = 1 to n
    for j = 1 to p
        for k = 1 to m
            C (i,j) = C (i,j) + A(i,k) * B(k,j)
```

$$E_{basic} = O(\epsilon_{op} \times 2nmp + \epsilon_{I/O} \times \frac{nm + nmp + np}{B} + \pi_{op} \times \frac{2nmp}{N}) \qquad (35)$$

$$E_{CO} = O(\epsilon_{op} \times 2nmp + \epsilon_{I/O} \times (n + m + p + \frac{nm + mp + np}{B} + \frac{nmp}{B\sqrt[2]{Z}}) + \pi_{op} \times \frac{2nmp}{N}) \quad (36)$$

Table 15: Matmul Complexity Analysis

Complexity	Cache-oblivious Algorithm	Basic Algorithm
Work	$\Theta(2nmp)$ [59]	$\Theta(2nmp)$ [140]
I/O	$\Theta(n + m + p + \frac{nm+mp+np}{B} + \frac{nmp}{B\sqrt[2]{Z}})$ [59]	$\Theta(\frac{nm+nmp+np}{B})$ [this study]
Span	$\Theta(\frac{2nmp}{N})$ [this study]	$\Theta(\frac{2nmp}{N})$ [this study]

5.6.2 Cache-oblivious Matmul Algorithm

The cache-oblivious matmul (CO-matmul) algorithm [59] is a divide-and-conquer algorithm. It has work complexity the same as the basic matmul algorithm $\Theta(2nmp)$. Its span complexity is also $\Theta(\frac{2nmp}{N})$ because of the used matrix partition approach shown in Figure 46. The I/O complexity of CO-matmul, however, is different from the basic algorithm: $\Theta(n + m + p + \frac{nm+mp+np}{B} + \frac{nmp}{B\sqrt[2]{Z}})$ [59]. Applying the ICE model to CO-matmul, the total energy complexity are computed as Equation 36.

5.7 Validation of ICE Model

This section describes the experimental study to validate the ICE model, including: introducing the two experimental platforms and how to obtain their parameters for the ICE model, describing input types, and discussing the validation results of SpMV and matmul.

5.7.1 Experiment Set-up

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

5.7.2 Identifying Platform Parameters

We apply the energy roofline approach [41, 40] to find the platform parameters for the two new experimental platforms, namely Intel Xeon E5-2650l v3 and Xeon Phi 31S1P. Moreover, the energy roofline study [40] has also provided a list of other platforms including CPU, GPU, embedded platforms with their parameters considered in the Roofline model. Thanks to authors Choi et al. [40], we extract the required values of ICE parameters for nine platforms presented in their study as follows: $\epsilon_{op} = \epsilon_d$, $\epsilon_{I/O} = \epsilon_{mem} \times B$, $\pi_{op} = \pi_1 \times \tau_d$, $\pi_{I/O} = \pi_1 \times \tau_{mem}$, where B is cache block size, ϵ_d, ϵ_d, τ_d, τ_{mem} are defined by [40] as energy per flop, energy per byte, time per flop and time per byte, respectively.

The ICE parameter values of the two new HPC platforms (i.e., Xeon and Xeon-Phi 31S1P) used to validate the ICE model are obtained by using the same approach as energy roofline study [41]. We create micro-benchmarks for the two platforms and measure their energy consumption and performance. The ICE parameter values of each platform are obtained from energy and performance data by regression techniques. Along with the two HPC platforms used in this validation, we provide parameters required in the ICE model for a total of 11 platforms. Their platform parameters are listed in Table 12 for further uses.

5.7.3 SpMV Implementation

We want to conduct complexity analysis and experimental study with two SpMV algorithms, namely CSB and CSC. Parallel CSB and sequential CSC implementations are available thanks to the study from Buluç et al. [38]. Since the optimization steps of available parallel SpMV kernels (e.g., pOSKI [7], LAMA[57]) might affect the work complexity of the algorithms, we decided to implement a simple parallel CSC using Cilk and pthread. To validate the correctness of our parallel CSC implementation, we compare the vector result y from $y = A * x$ of CSC and CSB implementation. The comparison shows the equality of the two vector results y. Moreover, we compare the performance of the our parallel CSC code with Matlab parallel CSC-SpMV kernel. Matlab also uses CSC layout as the format for their sparse matrix [63] and is used as baseline comparison for SpMV studies [38]. Our CSC implementation has out-performed Matlab parallel CSC kernel when computing the same targeted input matrices. Figure 47 shows the performance comparison of our CSC SpMV implementation and Matlab CSC SpMV kernel. The experimental study of SpMV energy consumption is then conducted with CSB SpMV implementation from Buluç et al. [38] and our CSC SpMV parallel implementation.

5.7.4 SpMV Matrix Input Types

We conducted the experiments with nine different matrix-input types from Florida sparse matrix collection [50]. Each matrix input has different properties listed in Table 13, including size of the matrix $n \times m$, the maximum number of non-zero of the sparse matrix nz, the maximum number of non-zero elements in one column nc. Table 16 lists the matrix types used in this experimental validation with their properties.

5.7.5 Validating ICE Using Different SpMV Algorithms

From the model-estimated data, CSB SpMV consumes less energy than CSC SpMV on both platforms. Even though CSB has higher work complexity than CSC, CSB SpMV has less I/O

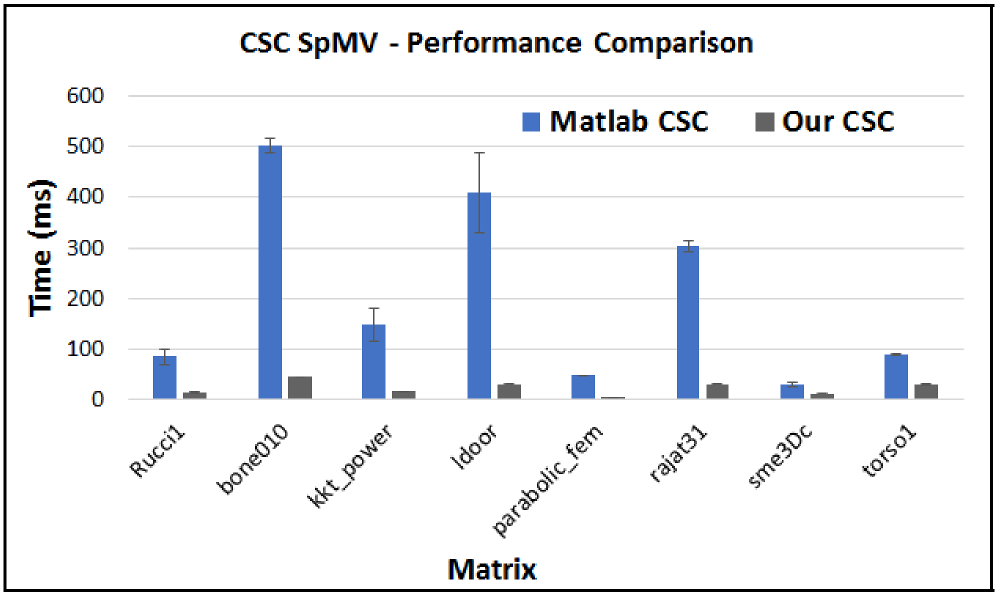


Figure 47: Performance (time) comparison of two parallel CSC SpMV implementations. For a set of different input matrices, the parallel CSC SpMV using Cilk out-performs Matlab parallel CSC.

complexity than CSC SpMV. Firstly, the dynamic energy cost of one I/O is much greater than the energy cost of one operation (i.e., $\epsilon_{I/O} >> \epsilon_{op}$) on both platforms. Secondly, CSB has better parallelism than CSC, computed by $\frac{Work}{Span}$, which results in shorter execution time. Both reasons contribute to the less energy consumption of CSB SpMV. The measurement data confirms that CSB SpMV algorithm consumes less energy than CSC SpMV algorithm, shown by the energy consumption ratio between CSC-SpMV and CSB-SpMV greater than 1 in the Figure 48 and 49. For all input matrices, the ICE model has confirmed that CSB SpMV consumes less energy than CSC SpMV algorithm.

Validating ICE Using Different Input Types

To validate the ICE model regarding input types, the experiments have been conducted with nine matrix types listed in Table 16. The model can capture the energy-consumption relation among different inputs. The increasing order of energy consumption of different matrix-input types are shown in Table 17, from both model estimation and experimental study.

For instance, in order to validate the comparison of energy consumption for different input types, a validated table as Table 18 is created for CSC SpMV on Xeon to compare model prediction and experimental measurement. For nine input types, there are $\frac{9 \times 9}{2} - 9 = 36$ input relations. If the relation is correct, meaning both experimental data and model data are the same, the relation value in the table of two inputs is 1. Otherwise, the relation value is 0. From Table 18, there are 34 out of 36 relations are the same for both model

Table 16: Sparse matrix input types. The maximum number of non-zero elements in a column *nc* is derived from [38].

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

Table 17: Comparison of Energy Consumption of Different Matrix Input Types.

Algorithm	CSB	CSB	CSC	CSC	CSB	CSB	CSC	CSC
Platform	Xeon	Xeon	Xeon	Xeon	Xeon-Phi	Xeon-Phi	Xeon-Phi	Xeon-Phi
Model/Exprmt	model	exprmt	model	exprmt	model	exprmt	model	exprmt
Increasing	sme3Dc	pds-100	pds-100	pds-100	sme3Dc	pds-100	pds-100	parabolic
Energy	torso1	parabolic	sme3Dc	parabolic	torso1	parabolic	sme3Dc	pds-100
Consumption	pds-100	sme3Dc	parabolic	sme3Dc	pds-100	Rucci1	parabolic	Rucci1
Order	parabolic	Rucci1	Rucci1	Rucci1	parabolic	sme3Dc	Rucci1	sme3Dc
	Rucci1	kkt	torso1	kkt	ldoor	kktr	torso1	rajat31
	kkt	torso1	kkt	torso1	bone010	torso1	kkt	kkt
	ldoor	rajat31	rajat31	rajat31	Rucci1	rajat31	rajat31	ldoor
	bone010	ldoor	ldoor	ldoor	kkt	ldoor	ldoor	torso1
	rajat31	bone010	bone010	bone010	rajat31	bone010	bone010	bone010

Table 18: CSC Energy Comparison of Different Input Matrix Types on Xeon

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

Table 19: Comparison accuracy of SpMV energy consumption computing different input matrix types

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

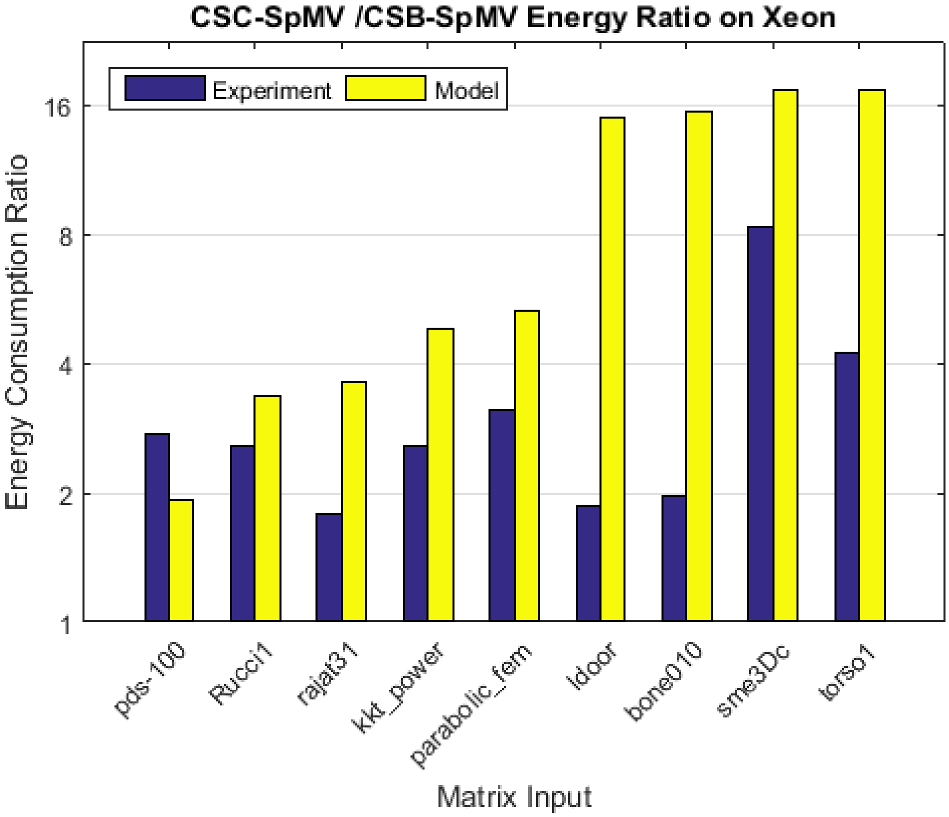


Figure 48: Energy consumption comparison between CSC-SpMV and CSB-SpMV on the Intel Xeon platform, computed by $\frac{E_{CSC}}{E_{CSB}}$. Both the ICE model estimation and experimental measurement on Intel Xeon platform show the consistent results that $\frac{E_{CSC}}{E_{CSB}}$ is greater than 1, meaning CSC SpMV algorithm consumes more energy than the CSB SpMV algorithm on different input matrices.

and experiment, which gives 94% accuracy on the relation of the energy consumption of different inputs. Similarly, the input validation for CSC and CSB on both Xeon and Xeon Phi platforms is provided in Table 19.

Validating The Applicability of ICE on Different Platforms

The energy comparison of CSB and CSC SpMV is concluded for eleven platforms listed in Table 12. Like two Xeon and Xeon Phi 31S1P platforms used in experiments, Figure 50 shows the prediction that CSB SpMV consumes less energy than CSC SpMV, on all platforms listed in Table 12. This confirms the applicability of ICE model to compare the energy consumption of algorithms on different platforms with different input types.

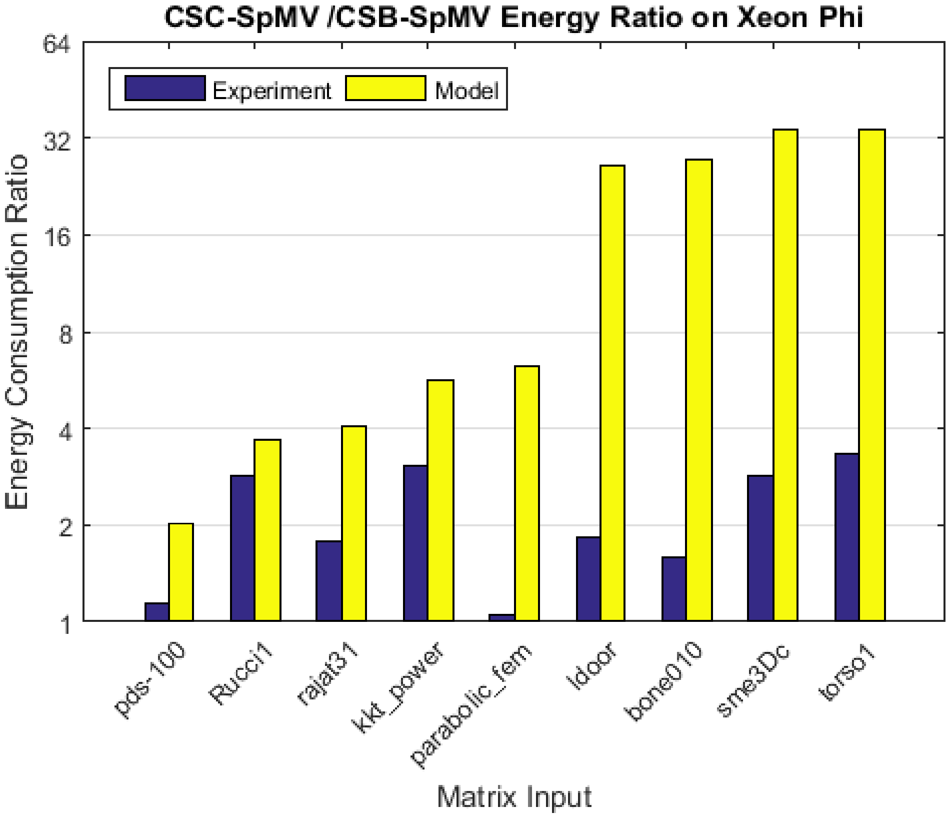


Figure 49: Energy consumption comparison between CSC-SpMV and CSB-SpMV on the Intel Xeon Phi platform, computed by $\frac{E_{CSC}}{E_{CSB}}$. Both the ICE model estimation and experimental measurement on Intel Xeon Phi platform show the consistent results that $\frac{E_{CSC}}{E_{CSB}}$ is greater than 1, meaning CSC SpMV algorithm consumes more energy than the CSB SpMV algorithm on different input matrices.

Validating the Platform-independent Energy Complexity Model

From Equation 33 and 34, the platform-independent energy complexity for CSC and CSB SpMV are derived as Equation 37 and 38, respectively.

$$E_{CSC} = O(2 \times nz + (nc + \log n)) \tag{37}$$

$$E_{CSB} = O(2 \times \frac{n^2}{\beta^2} + nz \times (1 + \frac{1}{B}) + \beta \times \log\frac{n}{\beta} + \frac{n}{\beta}) \tag{38}$$

We validate the platform-independent energy complexity of CSC and CSB SpMV. The platform-independent energy complexity also shows the accurate comparison of CSC and CSB SpMV computing different matrix types shown in Figure 51. Both platform-independent and platform-supporting models show that CSC-SpMV algorithm consumes more energy than CSB-algorithm. However, the difference gap between the energy complexity of CSC

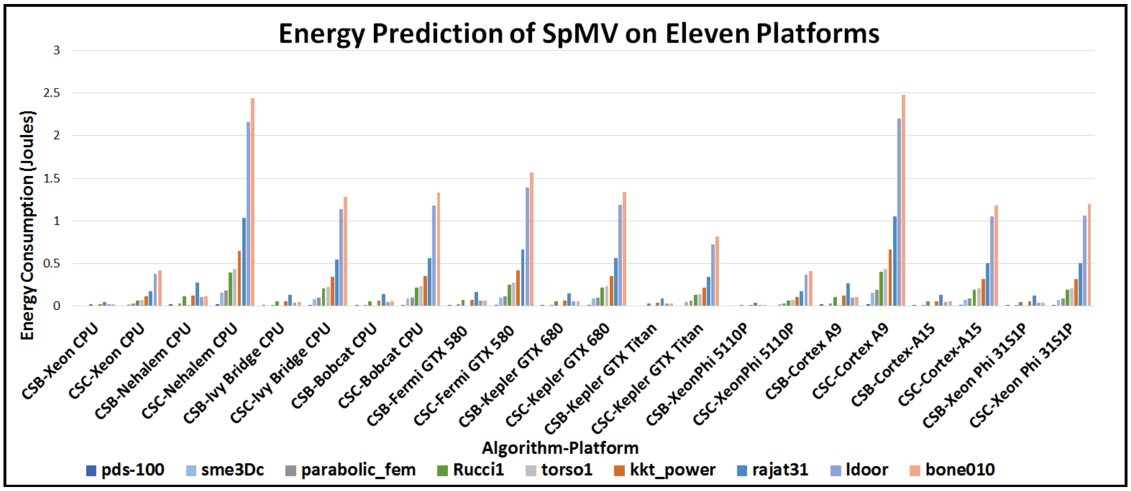


Figure 50: Energy Comparison of CSB and CSC SpMV on eleven different platforms.

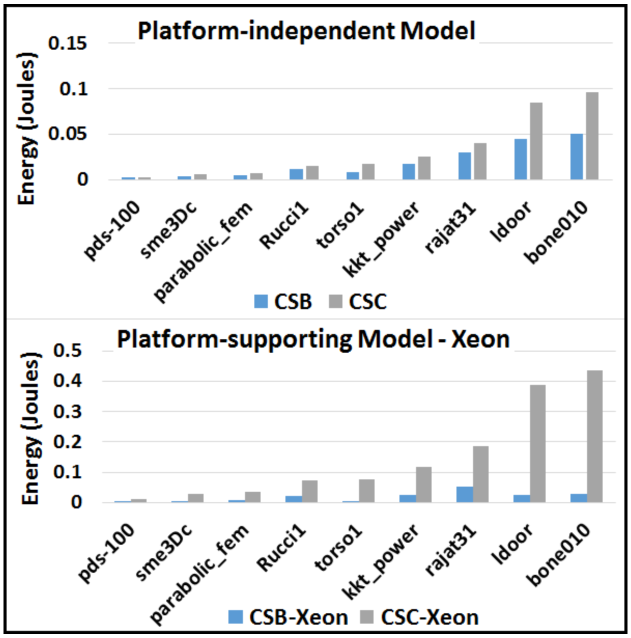


Figure 51: Comparison of platform-dependent and platform-supporting energy complexity model. Both models show that CSC SpMV consumes more energy than CSB SpMV.

and CSB using the platform-independent model is not clear for all input types except "ldoor" and "bone010" while in the platform-supporting model, the difference gap is clearer and consistent with the experiment results in terms of which algorithm consumes less energy for different input types. Comparing energy consumption of different input types requires more detailed information of the platforms. Therefore, the platform-independent model is only applicable to predict which algorithm consumes more energy.

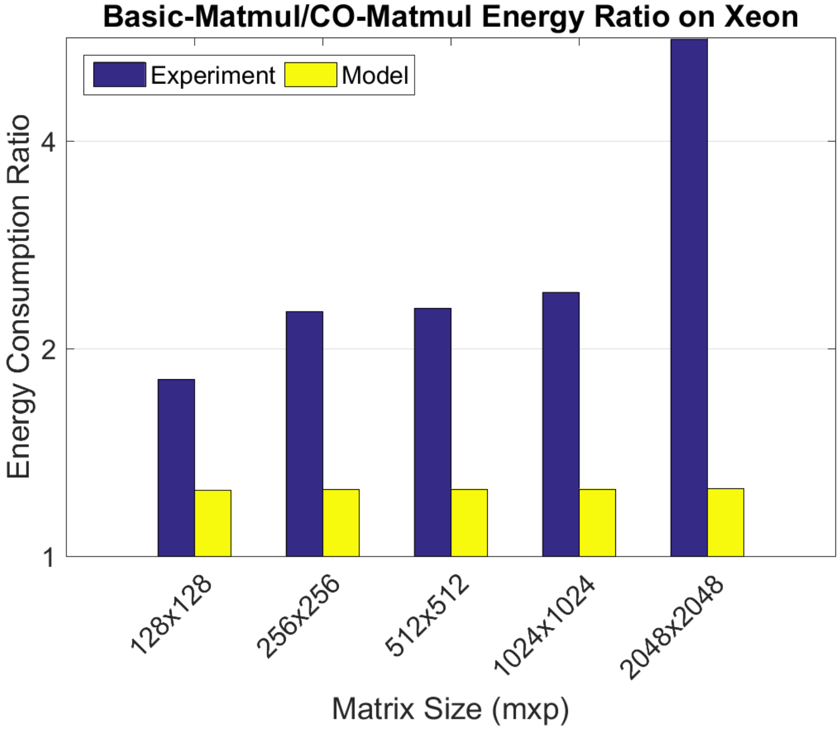


Figure 52: Energy consumption comparison between Basic-Matmul and CO-Matmul on the Intel Xeon platform, computed by $\frac{E_{Basic}}{E_{CO}}$. Both the ICE model estimation and experimental measurement on Intel Xeon platform show that $\frac{E_{Basic}}{E_{CO}}$ is greater than 1, meaning Basic-Matmul algorithm consumes more energy than the CO-Matmul algorithm.

5.7.6 Validating ICE With Matmul Algorithms

The validation of ICE model with Matmul algorithm is another new result of this study in Deliverable D2.4 as compared to Deliverable D2.3. This makes the validation of the ICE model more complete with both data-intensive and computation-intensive algorithms. From the model-estimated data, Basic-Matmul consumes more energy than CO-Matmul on both platforms. Even though both algorithms have the same work and span complexity, Basic-Matmul has more I/O complexity than CO-Matmul, which results in greater energy consumption of Basic-Matmul compared to CO-Matmul algorithm. The measurement data confirms that Basic-Matmul algorithm consumes more energy than CO-Matmul algorithm, shown by the energy consumption ratio between Basic-Matmul and CO-Matmul greater than 1 in the Figure 52 and 53. For all input matrices, the ICE model has confirmed that Basic-Matmul consumes more energy than CO-Matmul algorithm.

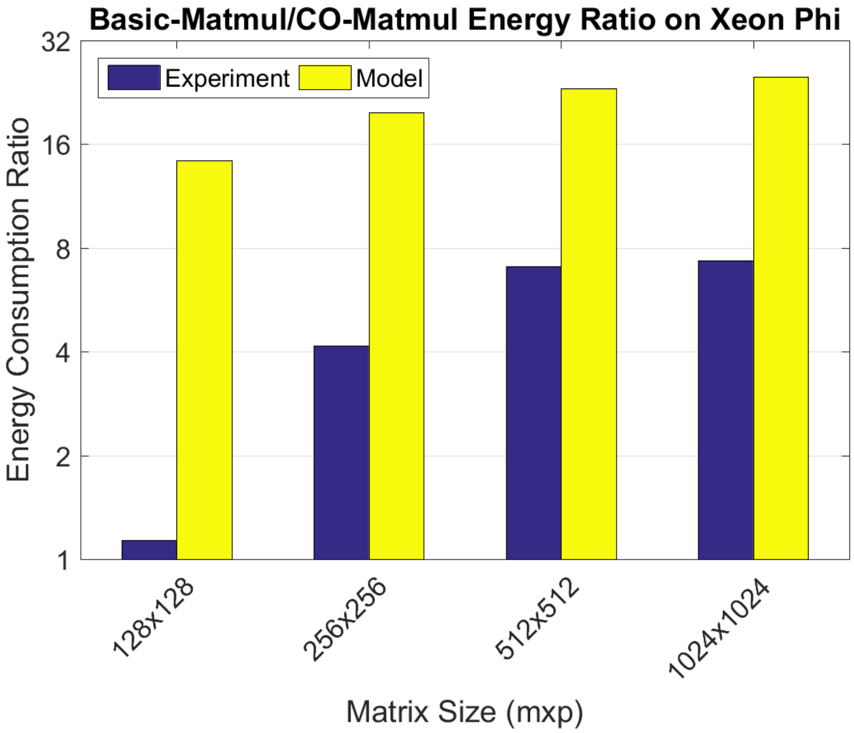


Figure 53: Energy consumption comparison between Basic-Matmul and CO-Matmul on the Intel Xeon Phi platform, computed by $\frac{E_{Basic}}{E_{CO}}$. Both the ICE model estimation and experimental measurement on Intel Xeon Phi platform show that $\frac{E_{Basic}}{E_{CO}}$ is greater than 1, meaning Basic-Matmul algorithm consumes more energy than the CO-Matmul algorithm.

6 Conclusions

In this Deliverable D2.4, we have reported our current results on the new energy/power models modeling the trade-off of energy efficiency and performance of data structures and algorithms; as well as the final prototype of libraries and programming abstractions.

- We have presented a detailed description of GreenBST, an energy-efficient concurrent search tree that is briefly described in D2.3. We have evaluated GreenBST with new state-of-the-art concurrent search trees and showed that GreenBST is portable and has a better energy efficiency and throughput than the state-of-the-art. We have developed GreenBST for Myriad2 and have experimentally evaluated our implementation.

- We developed a methodology for the customization of streaming aggregation implemented in modern low power embedded devices. We further compared the proposed embedded system implementations of the streaming aggregation operator with the corresponding HPC and GPGPU implementations in terms of performance per watt.

- We have introduced two new frameworks that can be used to the capture the performance of a wide set of lock-free data structures in dynamic environments. Then, we have integrated these performance analyses to our previous power model to obtain energy efficiency.

- We have validated the ICE model, a new energy complexity model for multithreaded algorithms with both data-intensive and computation-intensive kernels. This new energy complexity model is general for parallel (multithreaded) algorithms. The ICE model derives the energy complexity of a given algorithm from its *work*, *span* and *I/O* complexity. We also showed that *I/O* complexity in energy complexity is computed based on the Ideal Cache memory model.

References

[1] AMD Radeon HD 6450 GPU:. http://www.amd.com/en-us/products/graphics/desktop/6000/6450.

[2] Daily trades from 2015-08-05. http://www.nyxdata.com/Data-Products/Daily-TAQ#155. Accessed: 2016-05-05.

[3] Hardkernel. Odroid-xu:. http://www.hardkernel.com/.

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

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

[6] Movidius Ltd.:. http://www.movidius.com.

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

[8] Project Tango:. https://www.google.com/atap/project-tango/.

[9] SoundCloud:. https://www.soundcloud.com.

[10] Freescale i.mx 6 quad application processors for industrial products data manual. Technical report, Freescale Semiconductor Inc., 2014.

[11] Customization methodology for implementation of streaming aggregation in embedded systems. *Journal of Systems Architecture*, 66âĂŞ67:48 – 60, 2016.

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

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

[14] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long lived adaptive splitter and applications. 15(2):67–86, 2002.

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

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

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

[18] A. Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37th Annual Symp. on Foundations of Computer Science*, FOCS '96, pages 135–141, Oct 1996.

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

[20] 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.

[21] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 197–206, New York, NY, USA, 2008. ACM.

[22] 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.

[23] Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *Journal of the ACM*, 50(4):444–468, 2003.

[24] Hagit Attiya, Rachid Guerraoui, and Petr Kouznetsov. Computing with reads and writes in the absence of step contention. In *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, pages 122–136, 2005.

[25] Cagri Balkesen, Nesime Tatbul, and M Tamer Özsu. Adaptive input admission and management for parallel stream processing. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 15–26. ACM, 2013.

[26] Christos Baloukas, Jose L. Risco-Martin, David Atienza, Christophe Poucet, Lazaros Papadopoulos, Stylianos Mamagkakis, Dimitrios Soudris, J. Ignacio Hidalgo, Francky

Catthoor, and Juan Lanchares. Optimization methodology of dynamic data structures based on genetic algorithms for multimedia embedded systems. *J. Syst. Softw.*, 82(4):590–602, April 2009.

[27] Brendan Barry, Cormac Brick, Fergal Connor, David Donohoe, David Moloney, Richard Richmond, Martin O'Riordan, and Vasile Toma. Always-on vision processing unit for mobile applications. *IEEE Micro*, (2):56–66, 2015.

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

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

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

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

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

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

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

[35] 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.

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

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

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

[39] Daniel Cederman, Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. Brief announcement: Concurrent data structures for efficient streaming aggregation. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 76–78. ACM, 2014.

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

[41] 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.

[42] Yoonseo Choi, Yuan Lin, Nathan Chong, Scott Mahlke, and Trevor Mudge. Stream compilation for real-time embedded multicore systems. In *Code generation and optimization, 2009. CGO 2009. International symposium on*, pages 210–220. IEEE, 2009.

[43] Hongsuk Chung, Munsik Kang, and Hyun-Duk Cho. Heterogeneous multi-processing solution of exynos 5 octa with arm® big. littleâĎć technology.

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

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

[46] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

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

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

[49] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proc. 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 631–644, 2015.

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

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

[52] Erik D. Demaine. Cache-oblivious algorithms and data structures. 2002.

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

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

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

[56] Rolf Fagerberg. *Cache-Oblivious Model*, pages 1–99. Springer US, Boston, MA, 2008.

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

[58] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

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

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

[61] Karl Fürlinger, Christof Klausecker, and Dieter Kranzlmüller. Towards energy efficient parallel computing on consumer electronic devices. In *Information and Communication on Technology for the Fight against Global Warming*, pages 1–9. Springer, 2011.

[62] Tanmay Gangwani, Adam Morrison, and Josep Torrellas. CASPAR: breaking serialization in lock-free multicore synchronization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 789–804, 2016.

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

[64] Tony Givargis, Frank Vahid, and Jörg Henkel. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 25–30, Piscataway, NJ, USA, 2001. IEEE Press.

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

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

[67] Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 1–10, 2015.

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

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

[70] 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.

[71] 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.

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

[73] Phuong Ha, Vi Tran, Ibrahim Umar, Aras Atalar, Anders Gidenstam, Paul Renaud-Goud, Philippas Tsigas, and Ivan Walulya. D2.3 power models, energy models and libraries for energy-efficient concurrent data structures and algorithms. Technical Report FP7-611183 D2.3, EU FP7 Project EXCESS, February 2016.

[74] Phuong Ha, Vi Tran, Ibrahim Umar, Aras Atalar, Anders Gidenstam, Paul Renaud-Goud, Philippas Tsigas, and Ivan Walulya. Power models, energy models and libraries for energy-ecient concurrent data structures and algorithms. Technical Report D2.3, EU FP7 project EXCESS, 2016. http://www.excess-project.eu.

[75] 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.

[76] 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.

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

[78] 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.

[79] 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.

[80] 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.

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

[82] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[83] 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.

[84] D. Richard Hipp. Sqlite, 2015.

[85] Nicholas Hunt, Paramjit Singh Sandhu, and Luis Ceze. Characterizing the performance and energy efficiency of lock-free data structures. In *Interaction between Compilers and Computer Architectures (INTERACT), 2011 15th Workshop on*, pages 63–70. IEEE, 2011.

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

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

[88] Mircea Horea Ionica and David Gregg. The movidius myriad architecture's potential for scientific computing. *Micro, IEEE*, 35(1):6–14, 2015.

[89] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 151–160, 1994.

[90] 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.

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

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

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

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

[95] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[96] Jeremie Lagraviere, Johannes Langguth, Mohammed Sourouri, Phuong H. Ha, and Xing Cai. On the performance and energy efficiency of the pgas programming model on multicore architectures. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, 2016.

[97] 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.

[98] 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.

[99] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 487–498, New York, NY, USA, 2013. ACM.

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

[101] Sheng Li, Jung Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480, Dec 2009.

[102] 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.

[103] John D. C. Little. A proof for the queuing formula: L= λ w. *Operations research*, 9(3):383–387, 1961.

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

[105] Maged M. Michael. Cas-based lock-free algorithm for shared deques. pages 651–660, 2003.

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

[107] 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.

[108] Chi Cao Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46, Sept 2008.

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

[110] D Moloney et al. 1tops/w software programmable media processor. 2011.

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

[112] Ryan R Newton, Lewis D Girod, Michael B Craig, Samuel R Madden, and John Gregory Morrisett. Design and evaluation of a compiler for embedded stream programs. In *ACM Sigplan Notices*, volume 43, pages 131–140. ACM, 2008.

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

[114] Nikola Rajovic, Alejandro Rico, Nikola Puzovic, Chris Adeniyi-Jones, and Alex Ramirez. Tibidabo: Making the case for an arm-based hpc system. *Future Generation Computer Systems*, 36:322–334, 2014.

[115] Ohad Rodeh. B-trees, shadowing, and clones. *Trans. Storage*, 3(4):2:1–2:27, February 2008.

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

[117] Scott Schneidert, Henrique Andrade, BuÇğra Gedik, Kun-Lung Wu, and Dimitrios S Nikolopoulos. Evaluation of streaming aggregation on parallel hardware architectures. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, pages 248–257. ACM, 2010.

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

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

[120] Gaurav Singh, Greg Favor, and Alfred Yeung. Appliedmicro x-gene2. In *HotChips*, 2014.

[121] Karan Singh, Major Bhadauria, and Sally A McKee. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Computer Architecture News*, 37(2):46–55, 2009.

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

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

[124] Phillip Stanley-Marbell and Victoria Caparrós Cabezas. Performance, power, and thermal analysis of low-power processors for scale-out systems. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 863–870. IEEE, 2011.

[125] V. N. N. Tran, B. Barry, and P. H. Ha. Power models supporting energy-efficient co-design on ultra-low power embedded systems. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 39–46, 2016.

[126] V. N. N. Tran and P. H. Ha. Ice: A general and validated energy complexity model for multithreaded algorithms. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1041–1048, 2016.

[127] Vi Tran, Brendan Barry, and Phuong H. Ha. RTHpower: Accurate fine-grained power models for predicting race-to-halt effect on ultra-low power embedded systems. In *Proc. 17th IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '16, 2016. pages to appear.

[128] Vi Tran, Brendan Barry, and Phuong H. Ha. Supporting energy-efficient co-design on ultra-low power embedded systems. In *Proc. 2016 Intl. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS XVI, 2016. pages to appear.

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

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

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

[132] Ibrahim Umar, Otto Anshus, and Phuong Ha. Greenbst: Energy-efficient concurrent search tree. In *Proceedings of Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing*, pages 502–517, 2016.

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

[134] Ibrahim Umar, Otto Johan Anshus, and Phuong Hoai Ha. Effect of portable fine-grained locality on energy efficiency and performance in concurrent search trees. In *Proc. 21th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 36:1–36:2, 2016.

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

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

[137] Uri Verner, Assaf Schuster, and Mark Silberstein. Processing data streams with hard real-time constraints on heterogeneous systems. In *Proceedings of the international conference on Supercomputing*, pages 120–129. ACM, 2011.

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

[139] S. Xydis, A. Bartzas, I. Anagnostopoulos, D. Soudris, and K. Pekmestzi. Custom multi-threaded dynamic memory management for multiprocessor system-on-chip platforms. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 102–109, July 2010.

[140] Kathy Yelick. Cs 267 parallel matrix multiplication, Sept 2004.

Appendix A The tree library

We have developed concurrent search tree libraries that contain the implementation of the concurrent search tree algorithms described in Table 2.2.

A.1 Getting the source 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.

A.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`).

```
$ ./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.

A.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. Note that the valid `<libname>` is `dtree` for DeltaTree, `gbst` for GreenBST, and `cbtree` for CBTree. 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;
}
```

A.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.

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

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)