A Systematic Methodology for Optimization of Applications Utilizing Concurrent Data Structures

Lazaros Papadopoulos, *Student Member, IEEE*, Ivan Walulya, *Student Member, IEEE*, Philippas Tsigas, *Member, IEEE*, and Dimitrios Soudris, *Member, IEEE*

Abstract—Modern multicore embedded systems often execute applications that rely heavily on concurrent data structures. The selection of efficient concurrent data structure implementations for a specific application is usually a complex and time consuming task, because each design decision often affects the performance and the energy consumption of the embedded system in various and occasionally unpredictable ways. The complexity is normally addressed by developers by adopting ad-hoc design solutions, which are often suboptimal and yield poor results. To face this problem, we propose a semi-automated methodology for the optimization of applications that utilize concurrent data structures that is based on design space exploration. The proposed approach is evaluated by using both microbenchmarks and real-world applications that are executed on multicore embedded systems with different architectural specifications. Our results show that we can identify various trade-offs between different data structure implementations that can be used to optimize applications that rely on concurrent data structures.

Index Terms—Concurrent data structures, design space exploration, embedded systems, performance, energy

1 Introduction

Multicore embedded systems are now widely available in the market and execute a wide range of applications such as image processing, video streaming, databases, etc. These applications often rely on concurrent data structures, where multiple threads store and process their data. The implementation of these data structures is crucial to the performance of the application and therefore to the QoS that the whole system provides.

The design of efficient concurrent data structures is a complex and demanding task for many reasons. First of all, application developers cannot intuitively determine in many cases, without development and testing, which concurrent data structure synchronization method is the most efficient, especially when they adopt complex and sophisticated synchronization algorithms. The reason is that the effects of the synchronization algorithms on the efficiency of a data structure are determined by a large number of parameters, such as the contention level, the number of threads, the delay of acquiring/releasing locks, the inherent parallelism of data structures, etc. As a result, developers should implement a large number of different "versions" of data structures, with the same functionality and different synchronization methods and test them one by one, in order to find a solution that adheres to the design constraints.

The above problem becomes even more complicated, if we consider the portability issue. Embedded platforms support various synchronization primitives or the same primitives implemented at low level in a different way. A concurrent data structure implemented on an embedded platform using a specific synchronization algorithm may provide different results in terms of performance and power consumption when implemented on another platform. Therefore, the porting from one system to another cannot be straightforward. There is a need for exploration and evaluation of the platform-specific synchronization options that concurrent data structures utilize in order to be efficiently ported.

Synchronization algorithms evaluated in general purpose systems cannot always be implemented in embedded systems directly. General purpose systems provide various synchronization options often different from the options that are available in embedded chips [1], [2]. Additionally, these algorithms were not originally developed to adhere to the embedded systems' design constraints, such as the energy efficiency. Therefore, they should be evaluated and customized in order to provide efficient results on embedded devices.

Developers normally address the aforementioned problems by adopting ad-hoc solutions, without consid-

1.1 Design Space Exploration

In order to select an efficient concurrent data structure for an application implemented in an embedded system, there is a need for exploration of various design options. In this work, we propose a systematic methodology for the optimization of applications that utilize concurrent data structures. The methodology is semi-automated and it is based on the exploration of the concurrent data structure design space. It is supported by a tool flow that automates many steps of the methodology and provides several features for efficient exploration and extensibility.

The flow of the methodology can be summarized as follows: The developer, after providing the application and hardware constraints that prune the proposed design space, inserts the interface of the tool flow in the application under optimization. Then, the exploration phase follows, where data structure implementations instantiated by different design options are evaluated and the results for various metrics are provided to the developer. Therefore, the developer can select the implementation that is the most efficient in relation to the design constrains. Thus, the adoption of ad-hoc solutions and the time consuming and error-prone process of developing and testing a large number of different implementations with various customizations are both avoided.

The contributions of this work are the following:

- We present a systematic classification of the concurrent data structure design decisions that form a design space, along with the interdependences between the design options and the constraints that affect the design choices. To the best of our knowledge, there exists no similar classification of the concurrent data structure design space in the literature.
- We propose a systematic semi-automated methodology, along with a tool-chain, for the evaluation of different concurrent data structure implementations, based on design space exploration. The methodology provides the identification of trade-offs between different data structure implementations for various metrics.

The rest of the paper is organized as follows: Section 2 presents the related work that is relevant to exploration methodologies and similar approaches. The description of the design space, the methodology and the tool flow are presented in Section 3. In Section 4 we present the results of applying the methodology on set of benchmarks on different embedded platforms. Finally, in Section 5 we draw the conclusions.

2 Related Work

The concept of the exploration of different data structure designs has been initially proposed in [3]. It presents the dynamic data type refinement methodology (DDTR) that focuses on the identification of trade-offs between non-concurrent data structure implementations in terms of execution time, memory footprint and energy consumption. There are major differences between the present work and the DDTR approach: DDTR methodology is mainly a library of data structures. It does not provide any systematic way to prune implementations that are unsupported in specific applications or platforms, or to prune the ones that are expected to provide poor results. For example, in the DDTR methodology, there is no formalized way to prevent the evaluation of tree implementations in a data structure with FIFO access pattern. As a result, a large number of the implementations that are evaluated in DDTR are incoherent or meaningless in specific contexts. To face this problem, DDTR is limited to simple lists and arrays. It has never been evaluated on applications that utilize data structures more complex than lists and arrays. For instance, associative arrays cannot be integrated in the DDTR library, since they would be evaluated along with arrays and lists, yielding incoherent results. Additionally, it cannot be extended in the area of concurrent data structures: It provides no support for preventing the evaluation of implementations that utilize synchronization primitives on platforms on which they are unsupported. Therefore, DDTR is an application and platform-independent high abstraction level methodology. Since it lacks the features that would allow the methodology to be extended to complex and platform-dependent data structures, it is limited to simple non-concurrent implementations.

The proposed approach is very different: We define the data structure design space and identify the application and platform constraints that prune unsupported design options or options that are expected to provide poor results. Thus, the application and platform-independent design space is "converted" to application and platform-specific for each different context. Therefore, apart from the fact that our approach reduces the size of the design space that is being explored, it can also be applied to a wide range of applications and embedded platforms that support different synchronization primitives. In contrast to DDTR, the approach presented in this work:

- supports the evaluation of only meaningful data structures in each context, instead of the evaluation of all available implementations.
- supports the evaluation of platform-specific data structures for a specific context, instead of only abstract platform-independent implementations.
- is extended by supporting various concurrent and non-concurrent data structure implementations, such as hash tables, skip-lists and trees.

Finally, as it will be explained in Section 3.1, DDTR methodology is now a small subset of the approach described in this work. A similar design space exploration approach is presented in [4]. It focuses on the optimization of dynamic memory managers for embedded systems and it is complementary to the present work.

There are many works that propose lock-based and lock-free concurrent data structure implementations for general purpose systems, such as queues, trees and hash tables. A survey can be found in [5]. In our approach, we consider these works to be design decisions for the implementation of concurrent data structures. In other words, these proposed solutions compose the design space through which developers will make the decisions that will lead to the implementation of application specific and platform-specific efficient concurrent data structures. Application developers, instead of selecting in an ad-hoc manner one of the proposed solutions, or implementing a large number of them for evaluation, can use the proposed framework to semi-automatically

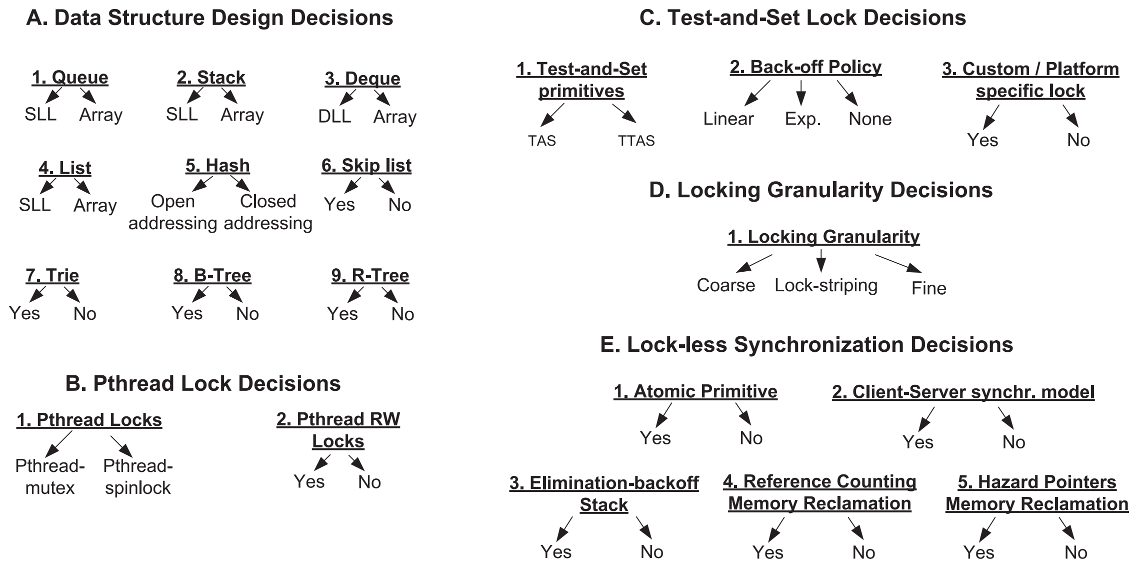


Fig. 1. Concurrent data structures design space.

select the most efficient concurrent data structure implementation for the application under optimization.

There exists a large number of concurrent data structure libraries, such as the NOBLE [6] and the practical lock-free data structures [7]. These libraries provide data structure implementations that are designed to be directly integrated in applications. In this work, we do not present a library, but a systematic methodology for identification of trade-offs between concurrent data structures. The proposed methodology is supported not only by a library of concurrent data structures (which is only one of the components of the tool-chain we provide), but also by tools that are integrated in a tool-flow that supports the exploration: tools for pruning the design space, for automating the exploration process, for collecting profiling results and for providing the identification of Pareto efficient implementations.

Several works describe trade-offs between various synchronization approaches. For instance, the authors in [8] compare the performance and the energy consumption of locks and software transactional memory. A similar comparison in a Haswell architecture is described in [9]. Moreshet et al. propose the implementation of energy efficient lock-free data structures by utilizing hardware transactional memory [1]. Other works focus on the evaluation of various synchronization options on NUMA architectures. For example, the authors in [10] evaluate message passing schemes as alternatives to shared memory data structure designs for Xeon and SPARC architectures, while Dice et al. propose efficient lock implementations for NUMA [2]. Finally, the evaluation of performance and energy consumption of optimistic techniques on Nehalem architectures has been studied in [11]. Our methodology focuses on embedded systems with architectural characteristics that are different from the aforementioned architectures and do not provide transactional memory support.

3 METHODOLOGY

In this Section we first define and describe the concurrent data structure design space. Then, we present the steps that comprise the proposed optimization methodology, the tool flow that supports it and the extensibility features it provides.

3.1 Concurrent Data Structures Design Space

The proposed concurrent data structure design space is shown in Fig. 1. The decisions taken by the application developer when implementing a concurrent data structure are modeled through a set of decision trees that represent design options of concurrent data structures. The design space covers the most common concurrent data structure design options that are proposed in the literature. It is application and platform- independent, since it includes design options that apply to various platforms and to applications with different behavior. The design options are grouped into five categories. Each category consists of one or more decision trees.

Data Structure Design Decisions category refers to the design of a concurrent data structure, without taking into consideration the synchronization algorithm that handles concurrent accesses.

Pthread Lock Decisions category consists of the simple and the readers/writer pthread locks options.

Test-and-Set Lock Decisions category groups the customization options for *TAS* and *TTAS* locks. *Back-off policy* decision determines whether or not a thread will withdraw for an amount of time (which can increase linearly or exponentially) from spinning on an occupied lock to reduce bus congestion.

Locking Granularity Decisions category refers to the granularity of locks. *Lock-striping* is a well-known locking technique proposed for concurrent hash tables, in which each lock protects a range of the data structure elements.

The last category is the *Lock-less Synchronization Decisions* that groups the design options for lock-less data structures. *Client-Server synchronization model* refers to the synchronization method proposed in [12]: An on-chip core is dedicated to play the role of the "server" and it is the only one that accesses the concurrent data structure directly. The rest of the on-chip cores are "clients", which send request

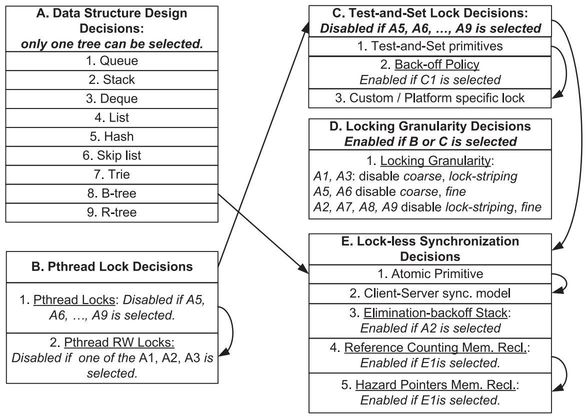

A. Data Structure Design Decisions: only one tree can be selected.
1. Queue
2. Stack
3. Deque
4. List
5. Hash
6. Skip list
7. Trie
8. B-tree
9. R-tree
B. Pthread Lock Decisions
1. Pthread Locks: Disabled if A5, A6, …, A9 is selected.
2. Pthread RW Locks: Disabled if one of the A1, A2, A3 is selected.
C. Test-and-Set Lock Decisions: Disabled if A5, A6, …, A9 is selected
1. Test-and-Set primitives
2. Back-off Policy Enabled if C1 is selected
3. Custom / Platform specific lock
D. Locking Granularity Decisions Enabled if B or C is selected
1. Locking Granularity:
A1, A3: disable coarse, lock-striping
A5, A6 disable coarse, fine
A2, A7, A8, A9 disable lock-striping, fine
E. Lock-less Synchronization Decisions
1. Atomic Primitive
2. Client-Server sync. model
3. Elimination-backoff Stack: Enabled if A2 is selected
4. Reference Counting Mem. Recl.: Enabled if E1is selected.
5. Hazard Pointers Mem. Recl.: Enabled if E1is selected.

Fig. 2. Design space interdependencies.

messages to the server for specific operations (e.g., insert, remove, etc.) and wait for the server's response. The communication takes place through on-chip message passing between the cores. The application interface for this model is implemented through a set of macros, as described in [12]. *Elimination-backoff Stack* is an algorithm for concurrent stacks proposed in [13]. Finally, memory reclamation options handle decisions related to the memory management of the dynamic memory for lock-free data structures. *Hazard pointers* are presented in [14] and *Reference Counting* in [15].

The options that compose the design space are not independent, since certain design options may affect other options. An interdependency occurs when a certain design option prevents the use of another decision tree, leaf or category. The interdependencies provide a complete set of rules on how a single concurrent data structure can be instantiated in the design space. They are integrated in the tool flow that supports the methodology and they are used to instantiate coherent concurrent data structures for evaluation. They are displayed in Fig. 2. The selection of a design option in the beginning of a directed edge prevents the selection of the pointed design option. Some interdependencies on Fig. 2 are depicted using arrows, while the rest are described with text. The interdependencies that handle the data structure implementations are an internal part of the proposed tool-chain. The developer does not have direct access to them.

The design space along with the interdependencies is an effective and systematic way to illustrate the concurrent data structure design choices: It makes a distinction between the decision trees that represent *attributes* of the data structures and leaves that represent different *values* that an attribute can have. This hierarchical representation assists the coherent design of the tool-chain and simplifies the exploration process: All the non-pruned leaves and design options are evaluated in the exploration phase. In other words, the implementations that will be evaluated finally are the ones that can be instantiated from the non-pruned leaves of the decision trees, by following the interdependencies rules. Additionally, the systematic representation of the design space through decision trees makes its extension convenient, as it is described in Section 3.4.

Thus, it assists the hierarchical development and the extensibility of the tool-chain. Based on the definition of the design space, we propose a step-by-step exploration methodology.

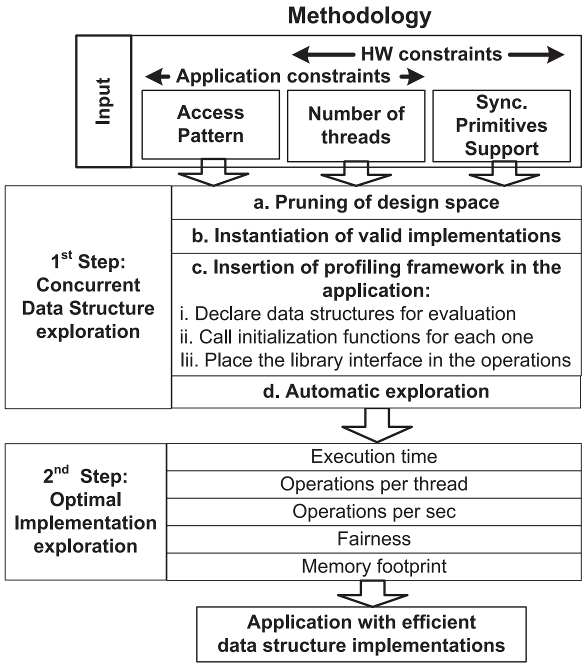

Methodology
Input
HW constraints
Application constraints
Access Pattern
Number of threads
Sync. Primitives Support
1st Step: Concurrent Data Structure exploration
a. Pruning of design space
b. Instantiation of valid implementations
c. Insertion of profiling framework in the application:
i. Declare data structures for evaluation
ii. Call initialization functions for each one
Iii. Place the library interface in the operations
d. Automatic exploration
2nd Step: Optimal Implementation exploration
Execution time
Operations per thread
Operations per sec
Fairness
Memory footprint
Application with efficient data structure implementations

Fig. 3. Concurrent data structures optimization methodology.

3.2 Description of the Methodology

The proposed methodology consists of two steps and it is presented in Fig. 3. The first step is the Concurrent Data Structure exploration and the second step is the Optimal Implementation exploration.

As stated in the previous section, the design space is platform and application independent. Therefore, not all design options are meaningful for any application, nor available in all platforms. Before the exploration, it is necessary to prune the design space, by eliminating meaningless and non-supported design options.

We identified the application and the platform constraints that enable or disable specific categories or decision trees of the design space: The *access pattern* is an application constraint. The *number of threads* can be an application or a hardware constraint, while the *synchronization primitives support* is a hardware constraint. The constraints are the input of the methodology, provided by the user that results in the design space pruning at category or decision tree level (i.e. they do not prune specific leaves of design trees). This is important, in order to retain the coherence of the methodology and the tools that support it.

The *access pattern* is defined as the sequence that the data are accessed in a data structure by the application's algorithm [3]. It refers to an abstraction level above the underlying data structure implementation. It affects the design decisions of category *A* of the design space and prunes meaningless abstract data structures. The access patterns and the corresponding enabled decision trees of Category *A* are shown in Table 1.

The *number of threads* constraint is related to the concurrency of the data structure and it can have two values: *one* and *many*. It determines if specific elements of the data structure are updated by multiple threads simultaneously. If the number is one, then categories *B*, *C*, *D*, and *E* are disabled, as shown in Table 2. In this case, the data structure is sequential and the exploration is limited to the decision

TABLE 1
Access Pattern and Corresponding Enabled Decision Trees of the Design Space

Access Pattern	Enabled Decision Trees of Category 'A'
FIFO	A1
LIFO	A2
Deque	A3
Simple storage	A4
key-value pairs storage	A5, A6, A8
key-value pairs sorted storage	A6, A8
String storage	A7
Spacial access	A9

trees of Category *A*. This is the case with the DDTR methodology, presented in [3] that is limited to the first four decision trees of Category *A*.

The last constraint is the *synchronization primitives support*, which is hardware related and is presented in Table 3. The specifications of the platform enable decision trees from categories *B*, *C*, *D*, and *E*.

It is important to distinguish between the decision tree and category pruning that occurs due to the application and hardware constraints and the interdependencies. The role of the former is to disable unsupported and non-applicable decision trees and categories due to application and the platform characteristics. In other words, this early pruning makes the generic design space to be application and platform-specific. On the other hand, the decision tree interdependences refer to the way that each *single* data structure will be instantiated in order to be evaluated in the exploration phase. It determines the way that the leaves of the design space that remained after the pruning due to the constraints will be grouped to instantiate meaningful and valid concurrent data structures for evaluation.

To apply the first step of the methodology, the user provides the application and the hardware constraints information that prevents the evaluation of non-coherent and unsupported data structures. The pruned design space is provided to a script that handles the instantiation of valid data structure implementations, through a library of concurrent data structures that the proposed tool-chain provides. All remaining design choices are used to instantiate coherent data structures, according to the interdependences.

After the instantiation of the valid data structure implementations, the developer inserts the library interface to the application manually, by replacing the application's data structures under optimization with the ones of the library. Thus, all operations (e.g., insert, remove, etc.) pass through the library's data structures. Since data structures that correspond to the same access pattern (Table 1) have the same interface, this procedure needs to be done only once.

TABLE 2
Number of Threads and Corresponding Disabled Categories of the Design Space

Number of threads	Disabled Categories
One	B, C, D, E
Many	-

TABLE 3
Synchronization Primitives Support and Corresponding Enabled Decision Trees of the Design Space

Synchronization primitives support	Enabled decision trees
Pthreads	B, D
Test-and-Set	C1, C2, D
Custom/Platf. Spec. locks	C3, D
Atomic Primitives with CAS	E1, E5
Atomic Primitives with DCAS	E1, E4
Message passing communication	E2

Then, the exploration takes place: The application is executed for all different data structure implementations that were instantiated previously. The user should provide a workload that corresponds to the one that the application is expected to exhibit in real-world. However, for applications in which the workload may vary at runtime, the user may decide to profile the application for a set of representative workloads, with each one corresponding to a different real-world scenario. In this case, step *1.c* of the methodology and *step 2* should be repeated once for each workload.

A profiler integrated into the library monitors information about the execution time, the number of operations per thread, throughput, fairness, and memory footprint for each execution. Results for other metrics, such as power consumption for each implementation that is evaluated are collected manually. To improve the accuracy of the results, the developer can configure the number of times that the application will be executed for each different implementation. In case of multiple executions per implementation, the produced results are the average values of each execution. Pareto front for two or more optimization objectives is automatically provided for results produced by the framework directly. For results that are obtained manually, the Pareto front can be identified with trivial configurations of the corresponding script.

The second step of the methodology is the Optimal Implementation exploration. The profiling results for each concurrent data structure implementation that was evaluated are provided to the developer. Taking into consideration the design constraints, the developer selects the most efficient data structure implementation for the application under optimization.

If the application contains more than one data structures, the *access pattern* and the *number of threads* inputs are required for each one. Then, the library interface is inserted in each data structure. Thus, all the combinations of implementations are evaluated by brute-force exploration. In the second step of the methodology, the developer can select the most efficient combination of data structure implementations. Finally, to further illustrate the details of the methodology, Table 4 distinguishes between the steps that are completed by the tool-chain automatically and the steps that the developer performs manually.

The overhead of applying the proposed methodology is affected mainly by two parameters: The number of data structures of the application for which different implementations are being evaluated and the size of the design space for each one. A relatively large design space results in the evaluation of a large number of data structure

TABLE 4
Non-Automated and Automated Steps of the Methodology

Non-automated steps	Automated steps
Input: declaration of constraints	**1a**: pruning of design space
1c: insertion of Library i/f. a) data structures declaration b) initialization function call c) i/f in operations	**1b**: instantiation of valid implementations
metrics of 2nd step: power consumption measurements for each execution	**1d**: config. of initialization functions before execution. Execution of valid implem.
	metrics of 2nd step: execution time oper./sec, oper./thread fairness, memory footprint identification of Pareto front

TABLE 5
Example of the Library Interface

FIFO	Deque	Key-value pairs Storage	
empty	empty	empty	iter_begin
size	size	size	iter_end
front	push_back	insert	iter_next
back	pop_back	find	iter_deref
push_back	push_front	remove	
pop_front	pop_front	count	

implementations. However, the pruning that takes place at the first step of the methodology and the pruning due to the interdependences, drastically reduces the number of the implementations that will be finally evaluated.

3.3 Tool Flow and Usage of the Methodology

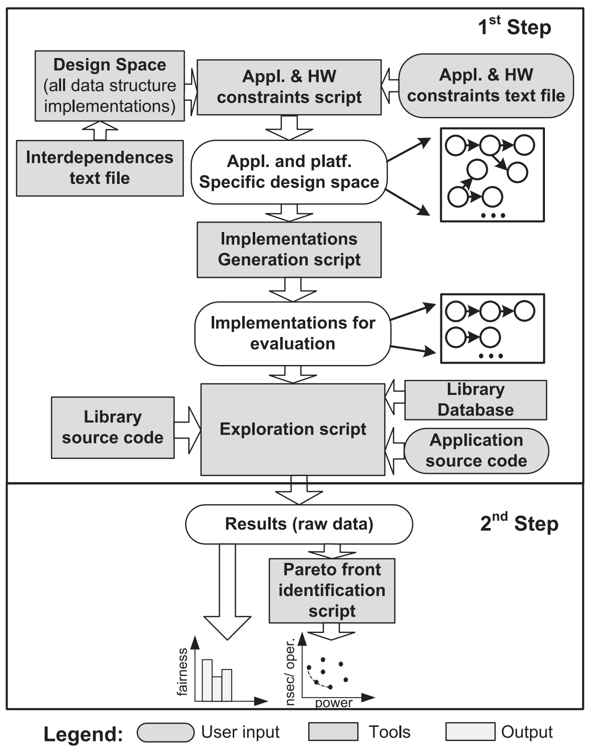
In this section we provide details on the set of tools that assist the implementation of each step of the methodology. The tool flow is presented in Fig. 4.

All different implementations that can be instantiated by the proposed design space can be represented as direct acyclic graphs, with the design options being the graph nodes and the order of decisions being the directed edges. The graphs along with the application and platform constraints (that are provided by the user in a text file) are forwarded to a script that prunes the unsupported graphs, making the design space application and platform-dependent. The pruned design space is subsequently provided to the *implementations generation script* that instantiates the valid implementations. The role of the *exploration script* is to assign the nodes of the graphs to the library's source code, using the *Library Database*. The database is implemented as a simple hash table that associates nodes with library source code and functions. The script parses the application source files and makes the appropriate changes in the library interface found in the application's source code. Then, it executes the application for each different implementation. Finally, it collects profiling data for execution time, number of operations per thread, operations per second, fairness and memory footprint for each data structure.

As stated in the previous section, the profiling data produced by the tool-chain are directly processed by the *Pareto front identification script*. In the context of this work, an implementation is Pareto efficient for two optimization objectives in a particular experiment, if no other evaluated implementation provides better results for both objectives. The algorithm is implemented following the exhaustive solution for Pareto front identification described in [16].

The library interface is inserted in the application by the user in three steps: The data structures under evaluation are declared and they are instantiated by calling an initialization function for each one in the application's source code. These functions will be automatically modified by the exploration script, in order to instantiate different implementations during the exploration. The last step is the replacement of the data structure operations with the library's interface.

The library is developed in C. It is built in view of providing operations and interface similar to the corresponding data structures of the C++ Standard Template Library (STL). The available operations for the *FIFO*, *Deque*, and *Key-value pair Storage* access patterns are displayed in Table 5. The interface of the operations of each data structure implementation is based on function pointers and it is exactly the same for all implementations that correspond to the same access pattern. For example, all the FIFO implementations that are instantiated by the design search space have exactly the same interface. Therefore, no changes are needed to be made to the operation functions (e.g., push, pop) by the developer or by any script in the application's source code, in order to evaluate different FIFO implementations during the exploration. Instead, the exploration script automatically modifies only the functions that are used to initialize each data structure before each execution. Thus, although the data structure interface remains the same, the underlying implementation changes. Data


1st Step
Design Space (all data structure implementations)
Appl. & HW constraints script
Appl. & HW constraints text file
Interdependences text file
Appl. and platf. Specific design space
Implementations Generation script
Implementations for evaluation
Library Database
Library source code
Exploration script
Application source code
Results (raw data)
2nd Step
Pareto front identification script
fairness
nsec/ oper.
power
Legend:
User input
Tools
Output
Fig. 4. Tool flow of the methodology.

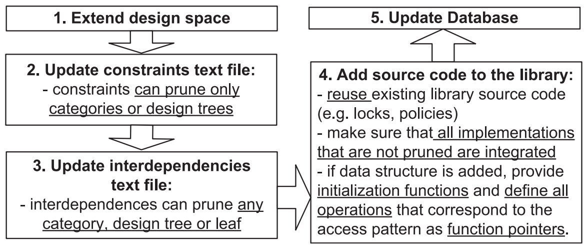

1. Extend design space
2. Update constraints text file:
- constraints can prune only categories or design trees
3. Update interdependencies text file:
- interdependences can prune any category, design tree or leaf
4. Add source code to the library:
- reuse existing library source code (e.g. locks, policies)
- make sure that all implementations that are not pruned are integrated
- if data structure is added, provide initialization functions and define all operations that correspond to the access pattern as function pointers.
5. Update Database

Fig. 5. Extending the methodology with more design options: steps and requirements.

structure implementations that have the same interface are interchangeable in the context of this work, because they provide the exact same functionality. In other words, all data structures of Table 1 that support the same access pattern have the same functionality.

The memory allocation of the data structures of the library is made dynamically. The lists are implemented as single linked lists (for the *queue* decision tree leaf) or double linked list (for the *deque*, the *Stack* and the *List* decision trees). The array implementations are similar to the STL vectors: When no more elements can be stored, their capacity is doubled by a memory-copy operation, automatically.

To increase its portability of the library, the atomic primitives are implemented using the corresponding gcc intrinsics. The client-server synchronization model is implemented as a wrapper around the message buffers used for communication between the chip cores. If another low-level communication primitive is available (e.g., interrupts), the user follows the steps described in Section 3.4 to integrate it in the library.

Although the tool provides memory footprint information, the results cannot be consistent for concurrent data structures, since they are affected by non-deterministic parameters, such as the order at which the threads acquire locks. However, these results can be used for the estimation of the order of magnitude of the memory size that each data structure requires.

3.4 Extensibility of the Methodology

The design space we propose covers the most common data structures that can be found in the literature and in real-world applications. However, it cannot be considered exhaustive. In this section we describe the steps that the user can follow in order to extend it with more design options (Fig. 5).

The first step is the extension of the design space by adding new design options. It can be made either by the addition of new leaves to existing decision trees, or by adding new decision trees to an existing or to a new category. The second step, is the extension of the design constraints. The user should determine whether or not the new design option is disabled for specific application or platform constraints and update the input constraints text file. It is important to state that a constraint cannot prune one or more decision tree leaves. It can only prune a category or a complete decision tree. This is important for ensuring the coherence of the methodology and the tools.

The third step is to identify the interdependences between the new and the existing design options and update the interdependencies text file. Then, the source code is manually added in the library. It should cover all possible data structure implementations that are not pruned by the interdependencies. If the extension is a new data structure, the supported operations must be the same with the data structures that have the same access pattern and should be defined as function pointers. Additionally, an initialization function must be provided that instantiates the new data structure declarations.

The developer should reuse components of the library: For example, if a new data structure is added, the source code of lock implementations that already exists must be integrated in the new data structure, directly. However, if the new data structure utilizes new kinds of locks, then these lock types must be added as new design options in the design space, by following again the steps described in Fig. 5. Finally, the developer updates the database with the new design option, by assigning the appropriate source files, functions and declarations to the corresponding design options.

4 Demonstration of Methodology

In this Section we evaluate the proposed methodology. We selected five realistic benchmarks that utilize concurrent data structures, running on two representative embedded platforms with very different hardware specifications. Our goal is to identify trade-offs in the applications for various metrics, by evaluating different concurrent data structure implementations.

4.1 Platform Case Studies

The two selected platforms are a board that integrates the Freescale I.MX 6 Quad and the Movidius Myriad chip. They are representative of two different categories of modern multicore embedded chips.

Freescale I.MX 6 Quad belongs to a family of multicore ARM-based chips integrated in single-board computers [17].

It integrates four ARM Cortex A9 cores operating at 1GHz and provides two cache levels. Regarding the hardware specifications that can be used for efficient synchronization, the I.MX 6 chip provides *pthread* through Linux OS, *Test-and-Set* and *Atomic Primitives with CAS* support (Table 3).

Myriad is a 65nm heterogeneous MPSoC designed by Movidius Ltd. It acts as a low power coprocessor integrated in mobile devices and it focuses on video and streaming processing.

It integrates a 32-bit RISC processor (LEON3) and 8 VLIW SHAVE cores. Regarding the memory specifications, Myriad contains 1MB of on-chip SRAM memory available to all SHAVEs, which is not cached and 64MB of off-chip DDR memory. More technical information on Myriad chip can be found in [18].

With respect to the synchronization options, Myriad provides 8 spinlocks implemented as fair locks with round-robin arbitration. It also provides a set of registers that can be used for fast and efficient message exchange between the SHAVEs. Each SHAVE has its own copy of these registers and the size of each one is 4×64 bit words. They can be used to implement a client-server synchronization model, as described in Section 3.1 and presented in [12]. Therefore, with respect to the

TABLE 6
Description of Benchmarks and Applications

Test case	Distrib. of operations or dataset	Access Pattern	# conc. d.s.
Deque	50% push 50% pop	deque	1
Database	20% write, 80% read	key-value pairs storage	1
Patricia	40% unique keys [19]	string storage	1
Dedup	[20]	key-value pairs storage	1
Streaming Aggregation	[21]	key-value pairs sorted storage	2

hardware specifications that can be used for synchronization, Myriad chip provides *Custom/Platform-specific locks* and *Message Passing communication* support (Table 3).

4.2 Application Case Studies

The aforementioned embedded chips, though they target different embedded platforms, run applications that rely heavily on concurrent data structures, such as databases and data streaming applications. To evaluate our methodology, we selected five benchmarks from different application domains, which we ported to the aforementioned platforms. These benchmarks are representative of the kind of applications that run on modern multicore embedded platforms. Two of them are custom microbencharks, while the rest are realistic applications. They are presented on Table 6.

The first benchmark is a concurrent deque microbenchmark. The experimental setup is similar to the one used in previous works [22]. We run three experiments for the specific benchmark: In the first one the deque is initialized with 100,000 elements. In the second, the deque is initialized as in the previous experiment, however, each thread executes a custom synthetic workload after each operation. Thus, in this experiment we explore the case in which the deque contention level is lower in comparison with the previous one. Finally, in the last experiment, there is no synthetic workload and the deque is not initialized with elements. The second benchmark is an in-memory, non-relational database microbenchmark. It stores key-value pairs and supports find and store operations. The database is initialized with 20,000 unique 32-bit keys. Patricia application is taken from the MiBench benchmark suite [23]. The multicore version is based on a radix tree data structure, which is lock-protected and each core executes an equal amount of the total workload [19]. Dedup is a data deduplication algorithm taken from the Parsec Benchmark Suite [20]. Finally, the last application performs a multiway data Streaming Aggregation [21]. We used this application in order to demonstrate the implementation of the methodology in applications that utilize more than one concurrent data structure. The application aggregates streams of tuples and maintains two concurrent data structures.

4.3 Evaluation Setup

In all experiments, the number of threads does not exceed the hardware limit of the physical cores. In other words, all applications in the four-core I.MX 6 use four threads, while in the 8-core Myriad chip use up to eight threads, with each thread pinned to a specific core during the whole execution. More specifically, in Myriad, we run experiments for two-cores (i.e., two threads with each one pinned to a specific core), four, six and eight cores. The only exception is the Streaming Aggregation application, in which we run experiments only for 8 cores. The duration of all experiments is at least one minute. Finally, all values presented in the following section (including the power consumption results) are the average of 10 executions, by elimination of the outliers.

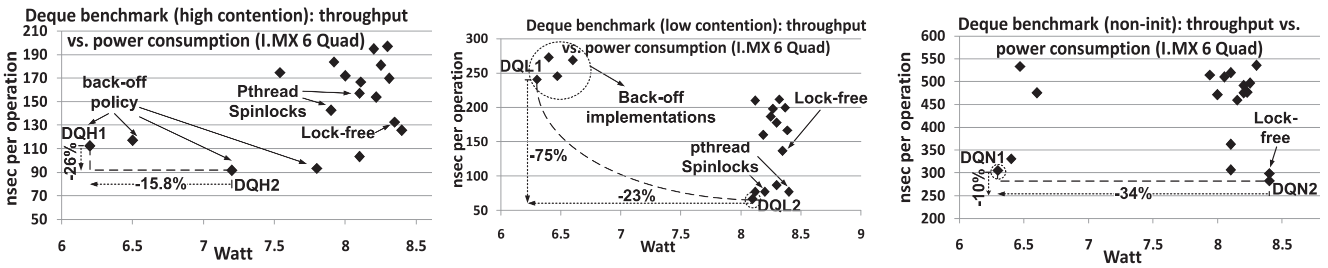
Execution time on I.MX 6 Quad was measured using the *gettimeofday* linux command, while in Myriad we used the *DrvTimer* functions that are provided by the Myriad MDK. The power consumption results were obtained through hardware instrumentation using a Watts Up PRO meter device and following a setup similar to methods proposed in the literature [11]. The device was connected directly to the power supply cable of the I.MX 6 board, while in Myriad it was connected to a shunt resistor attached to the power supply cable.

Fairness was evaluated according to the approach proposed in [24]. A prerequisite for the correct estimation of fairness by our tools is that all threads execute the same amount of workload for a specific amount of time. Fairness values close to 1 indicate fair behavior, while lower values imply unfairness and thread starvation.

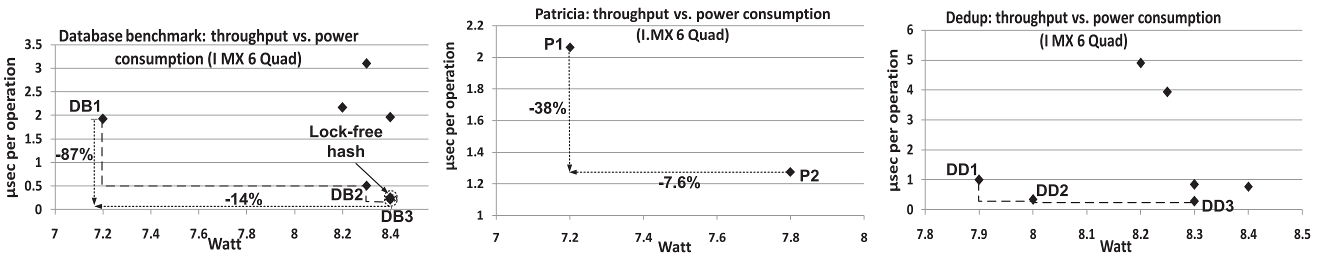
4.4 Demonstration on Freescale I.MX 6 Quad Chip

The first three subfigures of Fig. 6 present the performance vs. power consumption results for three different scenarios of the deque benchmark on I.MX 6 Quad chip. 18 different implementations were evaluated and each point in the figures corresponds to a different deque implementation. The Pareto efficient implementations are detailed in Table 7. We notice that the most efficient implementations are array-based. This is related to the fact that the I.MX 6 chip provides cache hierarchy, which favors data structures that provide high data locality. The high contention scenario presented in Fig. 6a favors, in terms of performance, implementations that utilize the back-off policy (*DQH1*, *DQH2*). However, when the contention level lowers, the back-off implementations fail to provide high performance due to lock underutilization. In this case, spinlocks and TTAS locks without back-off provide higher performance (*DQL2*). Finally, in the case of a non-initialized deque, the performance and the power consumption are affected mainly by the sequence of operations: The larger the number of times that a thread tries to pop from an empty deque and fails, the lower the performance.

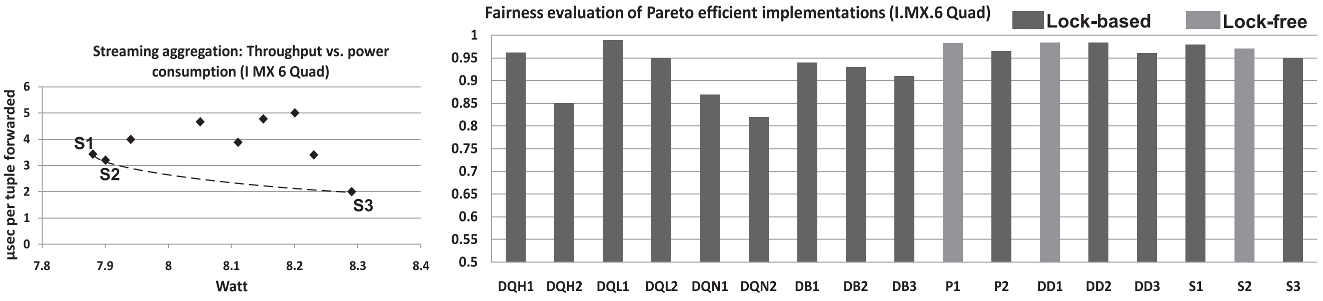
The results for the database benchmark on I.MX 6 chip are presented in Fig. 6d. A total of seven different implementations were evaluated and we identified three Pareto efficient implementations, which are a lock-based b-tree (*DB1*) and two lock-based hash tables (*DB2* and *DB3*). *DB3* is the implementation that provides highest throughput due to the lock-striping approach. However, *DB1* yields lower power consumption in comparison with the *DB3*, since in the hash table using lock-striping, more threads are active simultaneously than in the b-tree implementation. In



(a) Deque under high contention (b) Deque under low contention (c) Non-initialized deque



(d) Database benchmark (e) Patricia application (f) Dedup application



(g) Streaming aggregation application (h) Fairness of Pareto efficient implementations

Fig. 6. Demonstration on I.MX 6 Quad chip (four threads).

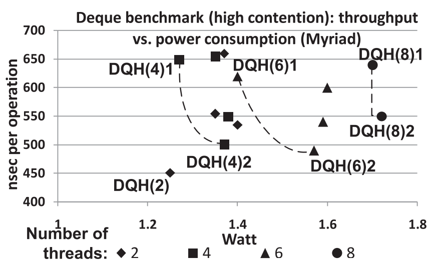
the latter, the threads are often suspended by the kernel, while they wait for the lock to be released.

There are two valid implementations for the Patricia benchmark for the I.MX 6 chip. String storage access pattern enables the decision tree *A7* (Table 1) and there are two available implementations for the trie data structure: a lock-based radix tree that utilizes *pthread-rw locks* (*P1*) and a lockless Ctrie that uses atomic primitives (*P2*) [25]. Ctrie is designed for efficient cache utilization and therefore provides high performance on the I.MX 6 Quad chip. However, the fact that in lock-free implementations the threads that access the data structures are constantly active (either by making retries when atomic operations fail, or by accessing and updating the data structure) explains the Ctrie implementation's increased power consumption.

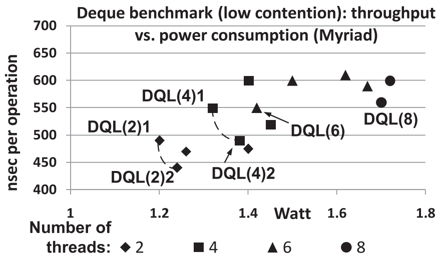
A set of seven different implementations were evaluated for the Dedup application on I.MX 6 Quad. We identified trade-offs for throughput versus power consumption, displayed in Fig. 6f: The *DD1* implementation yields 4.8 percent lower power consumption in comparison with *DD3* that provides 71 percent higher throughput. As shown in

TABLE 7
Pareto Efficient Implementations on I.MX 6 Quad Chip

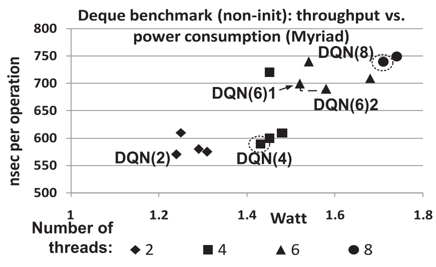
Pareto Point	Data Structure Implementation	Pareto Point	Data Structure Implementation
DQH1	A3(DLL), C1(TTAS), C2(exp), D1(fine)	P2	A7(trie), E1(atomic), E5(hazard)
DQH2	A3(array), C1(TTAS), C2(exp), D1(fine)	DD1	A8(b-tree), B2(RW locks), D1(coarse)
DQL1	A3(array), C1(TTAS), C2(linear), D1(fine)	DD2	A5(hash-closed), E1(atomic), E5(hazard)
DQL2	A3(array), C1(TTAS), C2(none), D1(fine)	DD3	A5(hash-closed), B2(RW locks), D1(lock-str.)
DQN1	A3(array), C1(TTAS), C2(linear), D1(fine)	S1	A6(skip list), E1(atomic), E5(hazard)/
DQN2	A3(array), C1(TAS), C2(none), D1(fine)		A6(skip-list), B2(RW-locks), D1(lock-str.)
DB1	A8(b-Tree), B2(RW locks), D1(coarse)	S2	A6(skip list), B2(RW locks), D1(lock-str.) /
DB2	A5(open), B2(RW locks), D2(lock-str.)		A6(skip list), E1(atomic), E5(hazard)
DB3	A5(closed), B2(RW locks), D2(lock-str.)	S3	A6(skip list), E1(atomic), E5(hazard) /
P1	A7(trie), B2(RW locks), D1(coarse)		A6(skip list), E1(atomic), E5(hazard)



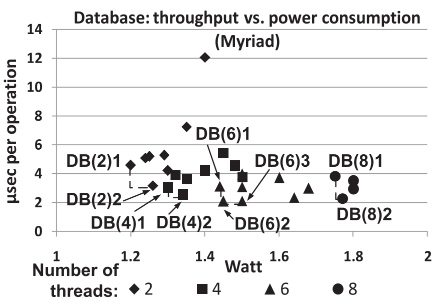
(a) Deque under high contention



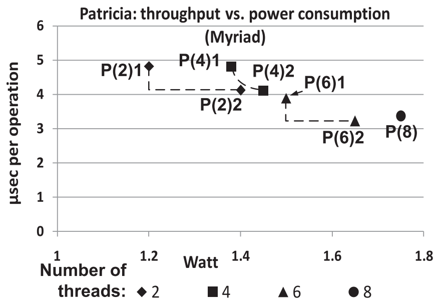
(b) Deque under low contention



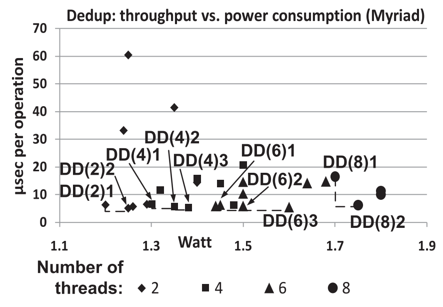
(c) Non-initialized deque



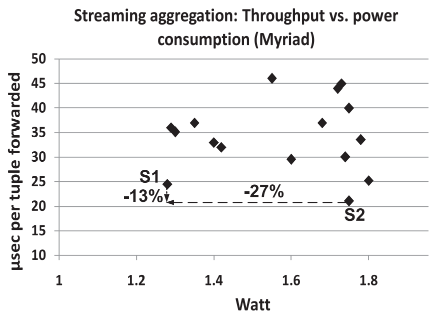
(d) Database benchmark



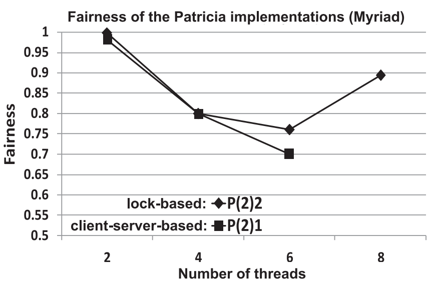
(e) Patricia application



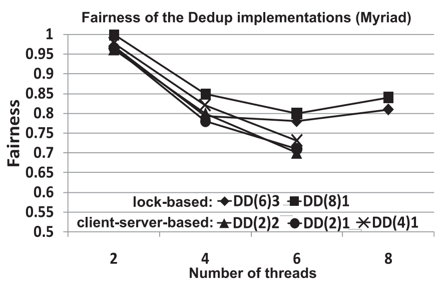
(f) Dedup application



(g) Streaming aggregation application



(h) Fairness: Patricia implement.



(i) Fairness: Dedup implementations

Fig. 7. Demonstration on Myriad chip.

Table 7, *DD1* is a lock-based b-tree implementation of the Dedup concurrent data structure, while *DD3* is a lock-based hash table. The hash table implementation provides higher concurrency than the b-tree implementation, due to the lock-striping granularity. However, more cores are active simultaneously and perform intensive calculations. Therefore the power consumption is increased. *DD2* is a lock-free hash. It provides slightly higher performance than the lock-based hash table, however it consumes more power due to the fact that the cores are constantly active. Similar conclusions about the power consumption of the lock-free data structures can be found in the literature [11]. The lock-based b-tree implementation (*DD1*) is the one with the lowest power consumption, since the threads are often suspended by the kernel.

The Streaming Aggregation application is heavily data driven with relatively low computational workload. Nine concurrent data structure combinations were evaluated (three different implementations for each data structure) and three Pareto efficient points were identified. *S1* corresponds to a lock-free skip list for the first data structure (in which the tuples are inserted) and a lock-based skip list for the data structure where the aggregated value is calculated. *S2* is the exact opposite combination, while *S3* corresponds to the lock-free implementations for both data structures. We notice that the latter implementation provides the highest throughput, along with the highest power consumption (42 and 5.1 percent higher than *S1* respectively). Similar results related to the efficiency in terms of throughput of lock-free data structures for streaming aggregation implementations can be found in the literature [21].

Fig. 6h presents the fairness values for the Pareto efficient data structure implementations on I.MX. 6 chip for all the applications. In general, most implementations provide high fairness, above or very close to 0.9. The same applies to the lock-free implementations that provide fairness equal or close to the corresponding lock-based. An interesting observation that is especially visible in the deque and in the database benchmarks is the fact that the implementations that provide relatively high throughput (e.g., *DQH2*, *DB3*), tend to provide smaller degree of fairness. This trend is a result of lock contention. Indeed, it is less intense in the deque benchmark in the low contention scenario (*DQL1* and *DQL2*).

4.5 Demonstration on Myriad Chip

The results of the demonstration of the methodology on Myriad for the deque benchmark are presented in the first three subfigures of Fig. 7 and the sequence of decisions for

TABLE 8
Pareto Efficient Implementations on Myriad Chip

Pareto Point	Data Structure Implementation	Pareto Point	Data Structure Implementation
DQH(2), DQH(8)1	A3(array), C3(custom), D1(fine)	DB(6)3, DB(8)2	A5(hash-closed), C3(custom), D1(lock-str.)
DQH(4)1	A3(array), E2(client-server)	DB(8)1	A8(b-tree), C3(custom), D1(coarse)
DQH(4)2, DQH(6)1	A3(DLL), E2(client-server)	P(2)1, P(4)1, P(6)1	A7(trie), E2(client-server)
DQH(6)2, DQH(8)2	A3(DLL), C3(custom), D1(fine)	P(2)2, P(4)2, P(6)2, P(8)	A7(trie), C3(custom), D1(coarse)
DQL(2)1	A3(array), C3(custom), D1(fine)	DD(2)1, DD(4)3, DD(6)1	A8(b-tree), E2(client-server)
DQL(2)2, DQL(8)	A3(DLL), C3(custom), D1(fine)	DD(2)2, DD(4)2, DD(6)2	A5(hash-closed), E2(client-server)
DQL(4)1, DQL(6)	A3(DLL), E2(client-server)	DD(4)1	A5(hash-open), E2(client-server)
DQL(4)2	A3(array), E2(client-server)	DD(6)3, DD(8)2	A5(hash-closed), C3(custom), D1(lock-str.)
DQN(2)	A3(DLL), E2(client-server)	DD(8)1	A8(b-tree), C3(custom), D1(coarse)
DQN(4), DQN(6)2	A3(array), E2(client-server)	S1	A6(skip-list), E2(client-server) /
DQN(6)1, DQN(8)	A3(DLL), C3(custom), D1(fine)		A6(skip-list), E2(client-server)
DB(2)1, DB(4)1, DB(6)1	A8(b-tree), E2(client-server)	S2	A6(skip-list), C3(custom), D1(lock-str.) /
DB(2)2, DB(4)2, DB(6)2	A5(hash-closed), E2(client-server)		A6(skip-list), C3(custom), D1(lock-str.)

the Pareto efficient points is shown in Table 8. Four implementations were evaluated for 2, 4, 6 and 8 threads. We notice that, in contrast with the I.MX. 6 experiments, many Pareto points that provide high performance are list-based deque implementations (e.g., *DQH(4)2*, *DQH(6)2*, *DQL(2)2*), since Myriad chip does not provide cache memory or prefetching mechanisms that tend to favor implementations with high data locality. Also, the client-server based implementations provide not only low energy consumption, but also, in some cases, high performance, mainly in the experiments with small number of threads (e.g., *DQH(4)2*, *DQL(4)2*, and *DQN(2)*). Additionally, as the contention level decreases, we notice in Fig. 7b that the throughput gap between the client-server based implementations and the lock-based becomes smaller. For instance, *DQH(4)2* provides 31 percent higher throughput than *DQH(4)1*, while the difference between *DQL(4)2* and *DQL(4)1* is reduced to 11 percent. Finally, the experiment with the non-initialized deque is presented in Fig. 7c. As in the corresponding experiment on I.MX.6 chip, the number of times that the threads try to pop from an empty deque affects the performance of each implementation.

Eight different implementations for the Database benchmark were evaluated on Myriad chip. The corresponding Pareto curves are presented in Fig. 7d. For the eight-thread experiment, the Pareto efficient implementations are similar to the ones that were identified in the I.MX 6 experiment (lock-based hash table and lock-based b-tree). Pareto point *DB(6)3* is a lock-based hash table implementation, which yields 32.6 percent higher throughput in comparison with *DB(6)1* and 4 percent higher power consumption. *DB(6)2* is the client-server version of the same implementation. We notice that as the number of threads increases, the lock-based hash table implementation begins to outperform the client-server model, since it provides more parallelism.

The results of the demonstration of the methodology in Patricia application are presented in Fig. 7e. There are two valid implementations for the *string storage* access pattern: A lock-based (*P(2)2*, *P(4)2*, *P(6)2*, *P(8)*) and a client-server (*P(2)1*, *P(2)3*, *P(2)5*). Performance versus power consumptions trade-offs are identified for two, four and six-thread experiments. For instance, in the 6-thread version of Patricia, the lock-based *P(6)2* trie implementation yields 17.2 percent higher throughput, while the client-server *P(6)1* yields 10.7 percent lower power consumption in comparison with *P(6)2*.

Throughput versus power consumption results for the Dedup application are displayed in Fig. 7f. We identified trade-offs in all experiments. In the eight-thread experiment using a b-tree lock-based implementation (*DD(8)1*) the power consumption is lower by 1.2 percent in comparison with the lock-based hash implementation (*DD(8)2*). However, the lock-based hash implementation yields increased throughput by 40 percent, since it provides more parallelism on Myriad due to the lock-striping technique. We notice that for relatively small number of threads, where the lock contention level is low, the client-server implementations dominate in both performance and power consumption. However, in the six and eight-thread experiments the lock-based hash implementation provides higher throughput.

Streaming Aggregation application is implemented for 8 threads only. Sixteen data structures combinations were evaluated (four for each data structure). *S1* corresponds to the client-server version for both data structures, while *S2* is lock-based versions. Since only three threads access each data structure, the client-server implementations provide comparable performance with the corresponding the lock-based ones.

We present fairness results for two real-world applications: Patricia in Fig. 7h and Dedup in Fig. 7i. In both applications, although the client-server based implementations provide fairness similar to the lock-based for up to four threads, for the six-thread experiment the fairness is significantly lower; close to 0.7. On the contrary, the lock-based implementations maintain fairness higher than 0.75 in all cases. Indeed, in contrast with the Myriad mutexes, no fairness mechanism is provided for the message-based communication that it is used in the client-server model. Therefore, client-server model implementations tend to provide lower fairness as the number of threads increases. The same conclusion can be reached for the rest of the test cases that are not presented, due to lack of space.

4.6 Discussion of Experimental Results

In this section, by studying the evaluation results, we identify the hardware and application characteristics that do not prune the design space, however they affect the

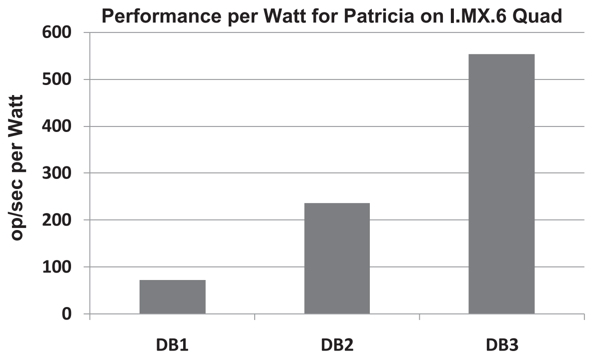


Fig. 8. Performance per watt of pareto efficient implementations for patricia on I.MX.6 Quad.

performance, the power consumption and the fairness of various decision trees.

The *rate* at which the operations are performed by the application's algorithm may impact the lock contention and therefore it affects the performance of the decision tree leaves of the categories *B*, *C*, and *D*. High operation rate that may cause increased level of lock contention favors back-off implementations, as shown in the high contention deque experiment in Fig. 6a on I.MX. 6 chip. The operation rate impacts the fairness, as well: As shown in Fig. 6h for the deque implementations, the fairness when the contention is increased, is lower than in the low contention experiment. This is consistent with the results of other related works that evaluate fairness [24].

The *memory hierarchy* and more specifically the existence of cache and prefetching techniques affects the leaves of Category *A*, by favoring implementations that provide high data locality (e.g., in the deque benchmark for I.MX. 6 in Fig. 6a). Additionally, the existence of cache affects the performance of the leaves of categories *B*, *C*, and *D*. Although, spinlocks can provide high performance in case of low operation rate (e.g., deque benchmark in Fig. 6b), the combination of high operation rate and the existence of cache yields low performance. The performance may drop even more in case of a fine grain locking implementation (decision tree *D1*) under high lock contention [24]. However, this is not the case in Myriad, as shown in the high contention experiment for six threads in Fig. 7a, since Myriad does not provide cache memory.

Finally, it is important to state that the results provided by the proposed tool-chain can be used by developers to produce results for more complex metrics. For example, Fig. 8 shows performance per Watt results for the Pareto efficient implementations of the Database benchmark on I.MX.6 Quad. These results are produced by the ones of Fig. 6e.

5 Conclusion

We described and demonstrated a systematic methodology for the optimization of applications that utilize concurrent data structures. The methodology provides the identification of trade-offs of different implementations for various metrics. It is based on design space exploration and it is supported by a tool-chain that partially automates the exploration process. The methodology assists developers in the selection of efficient data structure implementations for applications under optimization and on the effective porting of applications on platforms with various specifications. In the future, we plan to extend the design space of the methodology with more design options and evaluate our approach on more hardware platforms.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and Movidius Ltd. for providing us with Myriad chip in material transfer to conduct this research. This work was partially supported by the EU H2020 project AEGLE (http://www.aegle-uhealth.eu) and the EU FP7 project EXCESS (http://www.excess-project.eu).

References

[1] T. Moreshet, R. I. Bahar, and M. Herlihy, "Energy-aware microprocessor synchronization: Transactional memory vs. locks," in *Proc. 4th Annu. Boston-Area Archit. Workshop*, 2006, p. 21.
[2] D. Dice, V. J. Marathe, and N. Shavit, "Lock cohorting: A general technique for designing numa locks," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 247–256, 2012.
[3] L. Papadopoulos, C. Baloukas, and D. Soudris, "Exploration methodology of dynamic data structures in multimedia and network applications for embedded platforms," *J. Syst. Archit.*, vol. 54, no. 11, pp. 1030–1038, 2008.
[4] S. Xydis, A. Bartzas, I. Anagnostopoulos, D. Soudris, and K. Pekmestzi, "Custom multi-threaded dynamic memory management for multiprocessor system-on-chip platforms," in *Proc. IEEE Int. Conf. Embedded Comput. Syst.*, 2010, pp. 102–109.
[5] N. Shavit, "Data structures in the multicore age," *Commun. ACM*, vol. 54, no. 3, pp. 76–84, 2011.
[6] H. Sundell and P. Tsigas, "Noble: Non-blocking programming support via lock-free shared abstract data types," *ACM SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 80–87, 2009.
[7] K. Fraser, "Practical lock-freedom," Ph.D. dissertation, Kings College, Univ. Cambridge, Cambridge, U.K., 2004.
[8] A. Gautham, K. Korgaonkar, P. Slpsk, S. Balachandran, and K. Veezhinathan, "The implications of shared data synchronization techniques on multi-core energy efficiency," in *Proc. USENIX Conf. Power-Aware Comput. Syst.*, 2012, p. 6.
[9] A. Skyrme and N. Rodriguez, "From locks to transactional memory: Lessons learned from porting a real-world application," in *Proc. 8th ACM SIGPLAN Workshop Trans. Comput.*, 2013, http://transact2013.cse.lehigh.edu/.
[10] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. Marathe, and M. Moir, "Message passing or shared memory: Evaluating the delegation abstraction for multicores," in *Proc. 17th Int. Conf. Principles Distrib. Syst.*, 2013, pp. 83–97.
[11] N. Hunt, P. S. Sandhu, and L. Ceze, "Characterizing the performance and energy efficiency of lock-free data structures," in *Proc. Interaction between Compilers Comput. Archit.*, 2011, pp. 63–70.
[12] L. Papadopoulos, I. Walulya, P. Renaud-Goud, P. Tsigas, D. Soudris, and B. Barry, "Performance and power consumption evaluation of concurrent queue implementations in embedded systems," *Comput. Sci.-Res. Develop.*, vol. 30, no. 2, pp. 165–175, 2014.
[13] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable Lock-free stack algorithm," in *Proc. 16th Annu. ACM Symp. Parallelism Algorithms Archit.*, 2004, pp. 206–215.
[14] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, Jun. 2004.
[15] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele Jr, "Lock-free reference counting," *J. Distrib. Comput.*, vol. 15, no. 4, pp. 255–271, 2002.
[16] T. Givargis, F. Vahid, and J. Henkel, "System-level exploration for Pareto-optimal configurations in parameterized system-on-a-chip," *IEEE Trans. VLSI Syst.*, vol. 10, no. 4, pp. 416–422, Aug. 2002, http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1177338
[17] "Freescale I.MX 6 quad application processors for industrial products data manual," Freescale Semiconductor Inc., 2015, http://cache.freescale.com/files/32bit/doc/data_sheet/IMX6DQIEC.pdf.

[18] D. Moloney, “1tops/w software programmable media processor,” *HotChips HC23*, 2011.

[19] C. Ferri, S. Wood, T. Moreshet, R. I. Bahar, and M. Herlihy, “Embedded-tm: Energy and complexity-effective hardware transactional memory for embedded multicore systems,” *J. Parallel Distrib. Comput.*, vol. 70, no. 10, pp. 1042–1052, 2010.

[20] C. Bienia and K. Li, “Parsec 2.0: A new benchmark suite for chip-multiprocessors,” in *Proc. 5th Annu. Workshop Model., Benchmarking Simul.*, 2009, http://www-mount.ece.umn.edu/~jjyi/MoBS/2009/MoBS_2009_Advance_Program.html.

[21] D. Cederman, V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, “Brief announcement: Concurrent data structures for efficient streaming aggregation,” in *Proc. 26th ACM Symp. Parallelism Algorithms Archit.*, 2013, pp. 76–78.

[22] H. Sundell and P. Tsigas, “Lock-free and practical doubly linked list-based deques using single-word compare-and-swap,” in *Proc. 8th Int. Conf. Principles Distrib. Syst.*, 2005, pp. 240–255.

[23] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “Mibench: A free, commercially representative embedded benchmark suite,” in *Proc. IEEE Int. Workshop Workload Characterization*, 2001, pp. 3–14.

[24] D. Cederman, B. Chatterjee, N. Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, “A study of the behavior of synchronization methods in commonly used languages and systems,” in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 1309–1320.

[25] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, “Concurrent tries with efficient non-blocking snapshots,” *ACM SIGPLAN Not.*, vol. 47, no. 8, 2012, pp. 151–160.

Lazaros Papadopoulos received the diploma in electrical and computer engineering at Democritus University of Xanthi in Greece. He is currently working toward the PhD degree at the National Technical University of Athens. His research interests include low power embedded systems design and data structures optimizations for embedded systems’ applications. He is a student member of the IEEE.

Ivan Walulya received the Bsc degree in electrical engineering at Makerere University Kampala in 2010 and the Msc degree in computer systems and networks from the Chalmers University of Technology in 2013. He is currently working toward the PhD degree in the Networks and Systems Division, Department of Computer Science and Engineering at Chalmers. His main research interest is concurrent data structures, especially the energy aspects in multicore or many core processors. He is a student member of the IEEE.

Philippas Tsigas received the BSc degree in mathematics and the PhD degree in computer engineering and informatics from the University of Patras, Greece. He was at the National Research Institute for Mathematics and Computer Science, Amsterdam, The Netherlands (CWI), and at the Max-Planck Institute for Computer Science, Saarbrucken, Germany, before. He is currently a professor in the Department of Computing Science at Chalmers University of Technology, Sweden. His research interests include concurrent data structures and algorithmic libraries for multiprocessor and many-core systems, communication and coordination in parallel systems, power aware computing, fault-tolerant computing, autonomic computing, and information visualization. He is a member of the IEEE.

Dimitrios Soudris received the Diploma and PhD degree in electrical & computer engineering from the University of Patras, Greece, in 1987 and 1992, respectively. Since 1995 and for 13 years, he has served as a professor in the Department of Electrical and Computer Engineering, Democritus University of Thrace, Greece. He is currently an associate professor with the School of Electrical and Computer Engineering, National Technical University of Athens, Greece. His research focuses on embedded systems design, low power VLSI design, and reconfigurable architectures. He has published more than 350 papers and is the coauthor/coeditor of six Kluwer/Springer books. He is a leader and principal investigator in numerous research projects funded from the Greek Government and Industry, European Commission and European Space Agency. He is a member of the VLSI Systems and Applications Technical Committee of IEEE CAS and ACM. He is a member of the IEEE.