Building Fast Concurrent Data Structures through Data Structure Families

Brendan Lynch[1] Peter Pirkelbauer[2] Damian Dechev[1]

[1]Computer Software Engineering - Scalable and Secure Systems Lab
University of Central Florida
Orlando, FL 32816

[2]Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294

brendan.lynch@knights.ucf.edu pirkelbauer@uab.edu dechev@eecs.ucf.edu

ABSTRACT

Choosing a suitable data structure is hard in sequential applications and harder in parallel applications. In this paper, we describe a novel methodology that selects an optimal data structure implementation from a repository. Users describe the data structure (e.g., queue, linked list), their requirements (ABA free, relaxed linearizability), and used operations (e.g., enqueue, dequeue). The repository contains multiple instances for each data structure. The most generic instance, known as parent, implements a full set of operations. In addition, the repository contains children data structures optimized for a subset of operations. Users write code in terms of the generic data structure, specify requirements, and required operations. Through code analysis, we determine whether any child data structure can be used in place of the parent. This selection allows for fast, correct, scalable implementations that do not lock the user into a single approach. By allowing the user's code base to evolve without fear of implementation boundaries, we can provide a major benefit in complex software design. Our approach is built on the idea that change is inevitable in software design. Selecting the best data structure implementation is a hard problem, but should not distract programmers from their work.

1. INTRODUCTION

Concurrent data structures are critical for performance in modern systems [13]. While C++ is widely used for the development of high-performance computing applications, there is a lack of available and scalable data structures for concurrent systems. Future exascale architectures will demand efficient utilization of intra-node parallelism. Applications depending on significant inter-thread communication and synchronization will benefit from light-weight concurrent data structures.

In this paper, we describe a solution to concurrent data structure selection using what we call data structure families (DSF). DSFs contain a concurrent data structure known as the parent. The family is selected by the user (e.g. queue, vector, hash-map,) along with certain use case constraints (i.e. single writer, multiple reader). Parents are correct but possibly slow implementations of a container. Our goal is for parents to contain as close to the full range of operations available in the C++ standard library (STL) [1]. With concurrent data structures it is often the case that this interface must be slightly modified. Each parent data structure contains a library of children data structures such that each child implements a subset of the parent's operations. These children data structures are case-specific algorithm optimizations of the parent data structure. These algorithms can be optimized around specific correctness guarantees such as ABA-freedom or relaxed consistency, we call these the optimization dimensions.

Motivation for our approach stems from past research that shows data structure selection to be a hard problem [15]. Moreover, the selection of the right data structure is one of the most important performance aspects of application development [10]. When the selection of the best data structure in sequential code is already difficult for expert programmers, then choosing the best available concurrent data structure in parallel codes poses a challenge to performance and correctness. Many one-size-fits-all lock-free solutions do not provide performance gains, because implementing for all concurrent use cases requires expensive synchronizations among many operations. Lightweight implementations for specific use cases do not offer some functionality thereby eliminating and optimizing synchronization operations. Through code analysis we can determine if a highly specialized (lightweight) child implementation of a parent container could be used as a faster substitute. The expense of reduced functionality is in the unused operations, creating the illusion of a fully implemented container. When no child solution can meet

the use criteria the parent is always the default solution.

By removing the burden of selection from the user, we create an environment that is less prone to errors. Errors are negated by ensuring implementations are only called on when conditions match their case-specific algorithm, and that correctness is maintained as the code evolves. This is especially helpful in environments where many users will be contributing to the same source code — users do not need to understand the implementation boundaries to contribute. Black boxing the implementation selection allows for scalable software solutions and reduces avenues of possible software debt and implementation lock-in. Freeing developers to base their code around the solution and not around an implementation.

We evaluate the effectiveness of our approach using a micro-benchmark. We compare data structure family members to parent data structures in a series of performance tests. Each test contains either a typical distribution of container function calls, or a distribution relevant to a specialized family member. The families selected for testing include a family of lock-free queues and one of lock-free vectors. Our results illustrate two things: either the overhead of the data structure is greater than the cost of alternative solutions, but provides an ABA and lock-free solution that is otherwise unobtainable; or a faster (child) solution is employed. In our results, we compare as many solutions as possible to show how much the additional synchronization overhead affects each data structure.

The rest of the paper is organized as follows. Section 2 provides a brief summary of atomic primitives and lock-free data structures. Section 3 reviews some related works. Section 4 covers the design theory of data structure families. Section 5 shows two use case scenarios of data structure families. Section 6 presents the experimental evaluation of our approach. Section 7 concludes with a summary of results and perspectives on future work.

2. BACKGROUND

In this paper, we refer to optimization techniques and correctness guarantees found in concurrent programming. This section, provides a summary of the concepts used in our work.

2.1 Atomic Primitives and Progress guarantees

Atomic primitives are the building blocks of non-blocking programming. Our algorithms rely on `compare_and_swap` and `fetch_and_add`. The `compare_and_swap(addr, expected_value, new_value)` instruction also known as CAS, always returns the original value at the specified address but only writes `new_value` to `addr` if the original value matches `expected_value`. CAS is an atomic operation with an infinite consensus number [9]. The `fetch_and_add(addr, added_value)` instruction atomically increments the value stored in `addr` by the `added_value` and returns the value that was held in `addr` before the operation.

Lock-freedom is a progress guarantee associated with non-blocking algorithms. An algorithm is said to be lock-free if it guarantees that the system makes progress in a finite number of steps, independent of the system scheduler [8]. Lock-freedom does not guarantee starvation-freedom; although some thread will always make progress there is no control over which one it will be. Each CAS operation that fails, fails because a different CAS succeeded during the same time interval [8].

2.2 Linearizability

An objects linearization point defines the moment in time when that operation is said to have occurred, regardless of how long the operation takes as a whole. When reasoning about linearization every linearizable concurrent history is equivalent to some sequential history. A history is linearizable if the sequential history produced is legal. Another important attribute of linearization is that linearizability is compositional [8]. This allows us to treat linearizable objects as the basic building blocks of a concurrent system, to maintain the correctness guarantee.

So when building a concurrent library we can reason that as long as all components of the library are linearizable, so is any construction made from the components. Our data structure families are built out of linearizable objects in such a fashion.

2.3 ABA-Freedom

ABA-freedom is the result of solving the ABA problem. The ABA problem occurs in CAS algorithms, with a false positive execution of a CAS-based speculation on a shared location [2]. This scenario can be resolved by ensuring each item is unique, since this is generally not possible, a popular solution includes attaching a version tag to each object. Since a good solution is algorithm-specific and computationally expensive, we take advantage of the case when an ABA occurrence is impossible to return a light weight algorithm. In the cases when an ABA occurrence is possible we return an algorithm with an ABA avoidance scheme. When no specialized avoidance scheme exists for a data structure we rely on version tagging each object with the MCAS operation as a general avoidance solution.

2.4 MCAS

Multi-word Compare and Swap (MCAS) is a software algorithm that provides a linearizable way to execute multiple CAS operations. If one of the CAS operations fails, then the whole operation is rolled back, in an all or nothing completion strategy. This gives the user the illusion of performing an arbitrary number of CAS operations in one atomic step. Data Structure Families uses MCAS to extend version tagging for ABA-prevention to algorithms that do not have a well defined ABA-prevention strategy. We use the MCAS implementation-based on the work of Fraser et. al. [4].

2.5 Relaxed Consistency

Many concurrent data structures follow a sequential API, creating a bottleneck for scalability. Data structures such as queues, lists, stacks, share contention for one or two memory locations corresponding to the head or tail. One area of research focuses on increasing the performance of data structures by relaxing the linearizability requirement on the data structure. These algorithms balance scalability

and correctness. By relaxing the consistency and allowing K enqueue and dequeue points, a data structure's correctness becomes K - consistent [7].

3. RELATED WORK

Our work aims to bring specific algorithmic optimizations to data structures without losing the generality of a library. We focus on abstracting when and where the algorithmic optimizations are applied. We allow users to specify data structure transitions that follow the natural phases of execution within each program. Our contributions includes tailoring the chosen data structure to the use case.

Hawkins et al. create a process to synthesize high level concurrent relationships, choose concrete data structures, and select a locking strategy that is suitable for the context [6]. They tackle the problem of data structure selection in a concurrent setting, requiring programmers to use the operations provided to build concurrent relations, instead of actual data structures. This approach could be used to extend our current work; universal constructions such as concurrent relations usually lack optimization in algorithm design. Their analysis focuses on high-level relationships between data structures, while we focus on the best implementation of a specific data structure.

Vechev and Yahav built *Paraglider*, a tool to explore large areas of algorithms, while using a model checker to verify the linearizability of these concurrent objects [14]. This tools helps developers focus in on specific algorithmic implementations while rejecting ones that are not linearizable in the given use case. Paraglider focuses on the high-level skeleton structure of an algorithm, and attempts to optimize this skeleton against available design choices. They also attempt to return linearizable designs with the minimum space overhead. The key difference in our approach is that our designs are optimized around an extra decision space that is, defined by the user (i.e. relaxed linearizability and ABA freedom). Another difference is that paraglider returns a set of available algorithms to the developer, we return a set of algorithms in our search, and only return the most optimal data structure in the set to the developer.

Jung et al. designed *Brainy*, a program analysis tool that uses machine learning, code analysis, and memory management features, to determine the best data structure choice in sequential code [10]. Their work was motivated by findings that suboptimal data structure selection can reduce performance by up to 17% [11]. Brainy compares the frequency of function calls in each data structure. Given the distribution of function calls, it is possible to recommend similar data structures, that are more performant (e.g. a vector over a set for high iteration speed). Drawing in data from used function calls was inspirational to our work, although Brainy is bounded to the realm of sequential code. Our work examines function calls made in a particular phase of execution to determine what minimal overhead algorithms are available. The motivation is that the more unique function calls a concurrent data structure implementation maintains, the heavier the overhead for the algorithm.

4. METHODOLOGY

Our approach to choosing the appropriate data structure implementation in a concurrent system requires a strictly defined notation. The notation is designed to scale with added complexity in problem size. Our solution taxonomy relies on data structure families that are hierarchically stored in a binary search tree (BST). User specifications are represented as tuples that can be supplied through pragmas. With our tuple we traverse the tree of data structure families and return the best implementation available.

4.1 Data Structure Family Tree

The BST is a tree containing a collection of data structures as shown in Fig. 1. Due to the size in which the tree in Fig. 1 would grow if each child data structure was represented by a node, this total representation is not shown in our figures. Section 5.1 and Section 5.2 will display subtrees of Fig. 1 mapped all the way down to the child as it is traced.

The root node is level zero in the tree. The first level of the tree contains the different types of data structures that exist within our library (i.e. vector, queue, stack). Our second level contains the solution dimensions; dimensions refer to the solution spaces for which each data structure family is optimized. These dimensions include relaxed linearizability, ABA freedom, and hardware-specific optimizations. Each dimension exists as a requirement for optimizations. Any optimization that cannot guarantee the requirements specified in the dimension, cannot form a valid child node. These invalid children are not included in the branch. However, these children may appear in other branches, where the optimization does not violate the requirements of that branches dimension. Some dimensions exist to inform the tree that there is room for speed up, such as a single-writer-multiple-reader dimension. Other dimensions inform the tree of required safety or progress guarantees that are harder to optimize for correctly and have less children. More about each dimension can be found in Section 2.

The third level of our tree contains the parent structures for each dimension. The interfaces of the data structures are similar to their STL counterparts. Tree levels underneath the parent data structure contain all available children. Children data structures implement a subset of the parent interface. By offering a narrower interface, the available member functions can be implemented more efficiently. The specific data structure is selected based on the information specified by the user in form of pragmas. An optimal data structure is chosen according to the specification.

If we consider each parent node to be the root node of its family we can consider this to be level 0 of the family tree. A family tree has at most n levels, level one contains all of the data structures with n-1 functions that existed within the parent data structure. This reduction continues to the n-1 level at most, where each data structure has one function left (Iff this is an optimization). Levels are skipped (do not exist) if the level provides no data structures that yield an optimization over the parent level, this is why n-1 children levels is an upper bound and not a constant.

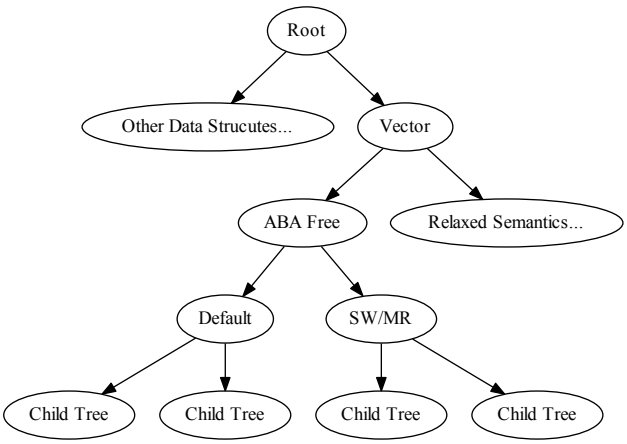

Figure 1: Overview of the Data Structure Family Tree

4.2 Tuple

In order to traverse our tree of data structure families we need to collect traversal rules from our user's code. We combine these rules into a tuple that mirrors the structure of our tree. Reading our tuple members from left to right corresponds with a top down traversal of our tree. This relationship can be seen in figure 1, symbols defined for members of the tuple are mapped to levels in the tree. Our tuple consists of the following members:

$$(\tau, \delta, \{\mu\}, \{\nu\})$$

1. A data structure, denoted as τ.
2. The dimension, that we refer to the scope of optimization we are applying to the data structure, denoted as δ (examples in 5.1 and 5.2).
3. User defined pragmas known as special rules, denoted as μ.
4. Rules generated from code analysis, denoted as ν.

4.3 Tree Traversal

Each member of the tuple in Section 4.2 contains the data required to traverse one level, or in the case of members that are sets, one set of levels in the tree. The tuple is parsed from left to right executing the instructions found in each member of the tuple. Individual items such as τ and δ are not null-able and are required to perform the tree traversal. However, the item sets such as the set of μ rules, and the set of ν rules, are allowed to be the empty set. If μ or ν is the empty set our implementation will return the full parent data structure containing no special rules from inserted pragmas.

If a member of the set is seen but there is no valid tree branch to accommodate that rule, from our current position in the tree, then the rule is treated as empty. When we return a data structure to the user, a most optimal data structure from the given rule set that exists in our library is returned. Once the entire tuple is parsed we return the data structure represented by the node in the tree that we end at. So if we parse the first rule of ν and descend into our child tree to find that there is no valid branch for the next rule in the set then we will return the current child. We guarantee the correctness of the user's selection:

1. Our parent node represents the full data structure, so each child is a subset of operations from the parent layer.
2. We transition to child nodes as long as the child supports each of the operations that exist in ν.
3. Nodes in ν are a one-to-one mapping of real data structures in our library; if there is no valid child branches for our current set of rules, that means the node we are on is the most optimal valid node in the current branch.

Tuples are parsed for each branch that exists in the tree, in a breadth-first search. Multiple valid data structures will exist for each tuple (i.e. every data structure above a valid child is also valid, but less optimal). To choose the best data structure to return for a use case we track the depth of each query into the tree, the query containing the largest depth is returned. Since each level of the tree contains one less function than the previous level, the deepest query will contain the least functionality, and be the best candidate for the use case. In the event of the tie an arbitrary branch will be returned to the user.

The vector trace event shown in Figure 2 is a possible traversal of the shown vector tree. In the traversal example shown the user has indicated through the use of pragmas that the only function they are interested in is the push function. This type of scenario can be commonly seen in many programs where a period of time goes by populating a data structure from another source. Each box in a node represents the functions that the node supports. Random access read is denoted by RAR, and random access write is denoted by RAW.

Each node in the figure is a vector implementation, with the root existing as the parent. The solid green arrows on the right indicate a probe into the tree that returns the push only vector with a depth value of four. The dotted black lines down the center represent an unexplored region because the first child does not match the criteria we are looking for. The red dotted lines to the left indicate a valid probe that was returned, the depth value of the child is beaten out by the value of the push only vector.

5. EXAMPLES

Examples consist of Section 5.1 and Section 5.2. In Section 5.2 using data structure families results in each phase of code execution paying the minimum algorithmic cost for maintaining an ABA prevention scheme. While Section 5.1 shows that depending on the data structure's use case, we can scale the correctness in terms of linearizability to increase performance.

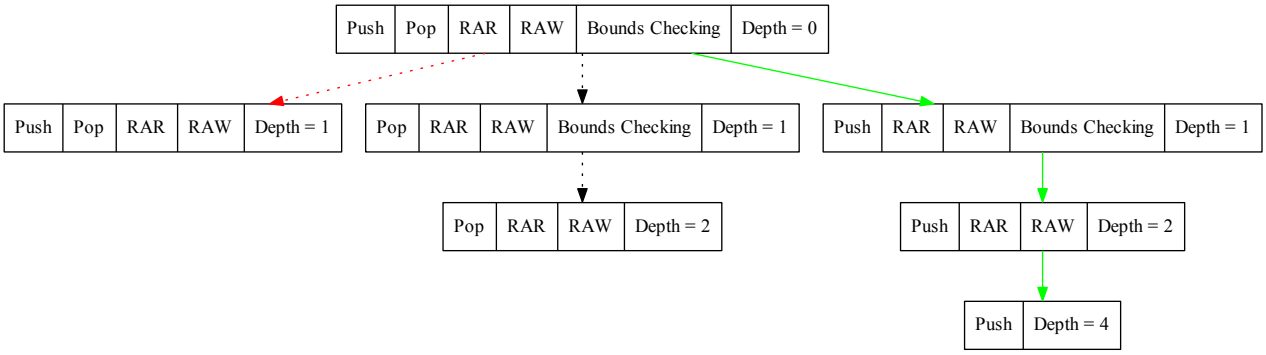

Figure 2: Vector Trace Event

5.1 Relaxed Linearizable Queues

In a complex system such as the operating system of a distributed server, data structures are used at various points to assist with the communication of data. When examining use cases of a queue in these systems, it becomes clear that there are two distinct points where a queue is needed:

1. When a scheduler receives tasks that requires time using a shared system resource.
2. When individual processes pass messages to each other, each of these messages is delivered to the recipients receive buffer, this buffer is a queue.

The case of a scheduler receiving tasks would require a queue that enforces strict ordering on its contents, to maintain the fairness of the chosen scheduling algorithm [12]. This queue should be able to quickly receive task requests, without violating the queue's first in first out ordering. Each runnable process entering the queue is waiting for a turn to use shared system resources and will block until those resources are available.

However, in the receive buffer case, a process who is sending a message to another process is doing so asynchronously and does not want to block. In this instance, it is more important that the inbox queue be able to receive a high volume of requests without blocking the senders on insert than it is to enforce total ordering. Many times these operations can include single process to many process messages. Using a queue that adheres to stricter progress guarantees can result in longer-lasting system-wide contention as the sender will block until all receivers have received the message [3]. Greater concurrency can be achieved with relaxed linearizble queues [5]. With DSF, and the proper use of pragmas, it is possible to build both the scheduler queue and the process receive buffer queues with a single library.

5.2 Process ID Map Using ABA-free Vectors

The ABA problem is a condition that can occur in highly concurrent CAS-based algorithms, the problem is explained in detail in Section 2 of the paper. There are several ABA prevention schemes available to CAS algorithms, these approaches generally require more expensive operations. Some existing approaches include using double-word CAS with a time stamp, or an algorithm that inserts descriptors into the data structure. The data structure family library uses both of these approaches. The queue relies on a software double-word CAS known as MCAS, the vector relies on a specialized descriptor algorithm.

These ABA prevention schemes require expensive overhead, in the form of additional CAS operations. MCAS requires 2N+1 CAS calls where N is the number of single CAS calls required; the descriptor approach requires 3 CAS operations for each operation that would otherwise require a single CAS. An approach to ABA prevention that is harder to implement is to not use CAS algorithms in scenarios that could lead to an ABA problem. This is not always possible, in programs that have multiple phases of execution it might only be possible during certain phases of execution. Considering a vector, one phase of execution could be unique elements being pushed into the vector, which is not ABA prone. A second phase could consist of concurrent push and pop operations prone to the ABA problem. A phase of execution is defined by the user with a phase barrier pragma, acting as a memory barrier for the current data structure implementation. When a phase is complete, the data structure reevaluates the requirements for implementation. The DSF library will examine function calls used in each phase of the execution and return the fastest available data structure. Which allows the user to always receive the fastest available CAS algorithm that maintains an ABA-free solution, at each step of the execution. Without such a library the user who wants an ABA-free solution would continue to pay the cost of the solution through the life time of the process, even when it is not necessary.

Process IDs are allocated in Linux on a sequential basis; they go up to a maximum value and then roll back to the minimum. These process IDs could be associated with a concurrent vector. Ideally, the concurrent vector would protect against the ABA problem to avoid data loss, while benefiting from concurrency.

An operating system provides a great example of multi-phase execution:

1. During the start-up phase of execution the operating system performs necessary steps to bringing the system online for the user. While performing the boot sequence the operating system is not listening to input from the user. As processes are launched and added to the process ID list, the list is in `add only` mode, and there is no fear of a possible ABA occurrence.

2. Once the boot phase is complete, the operating system is listening for input from users. Users interact with the operating system, launching and ending processes. During this phase it is possible to encounter an ABA scenario.

If the phase change is passed as a pragma to the DSF, the ABA scenario can be avoided, without paying the price of an ABA avoidance scheme during boot.

6. EXPERIMENT

All experiments are run on a 64 core Linux machine. For each graph the Y axis refers to the benchmark completion time and the X axis refers to the number of threads. All experiments are run over 2, 4, 8, 16, 32, 64 threads. The experiments show the execution time of parent data structures in relation to their children in common use case scenarios. Each data structure in the experiment is optimized to enforce ABA-freedom.

Fig. 3 is a family of lock-free queues that enforce ABA-freedom. The parent data structure represented by version three supports atomic size, and version tagging by employing MCAS. Version two is a lock-free queue with version tagging to eliminate the ABA problem, but does atomically track size, reducing the overhead of each operation. This version is representative of eliminating size when the user does not make use of the function, because it adds to each operation cost, for unused functionality. Version one is a lock-free queue that does not solve for the ABA-problem. When the queue is working with all unique elements, or only performing enqueue or dequeue, the ABA-problem does not manifest in the data structure.

Fig. 4 shows a family of lock-free vectors. This particular experiment shows the family in the use case where 100% of the operations are `push_back()`. This use case is not prone to ABA-problems and is common in programs with multiple phases of execution. Many vector use cases consist of a phase for populating array values from another source. As shown in the Figure, the optimized push only vector out performs the vectors not specialized for this use case.

Fig. 5 and Fig. 6 display a family of lock-free vectors optimized for ABA-freedom over two normal distributions of vector operations. In both distributions the vector that does not implement an ABA-avoidance scheme outperforms the ABA-free vector algorithm. The ABA-avoidance scheme creates expensive overhead for the vector, users can avoid paying this cost in code sections that are not ABA-prone.

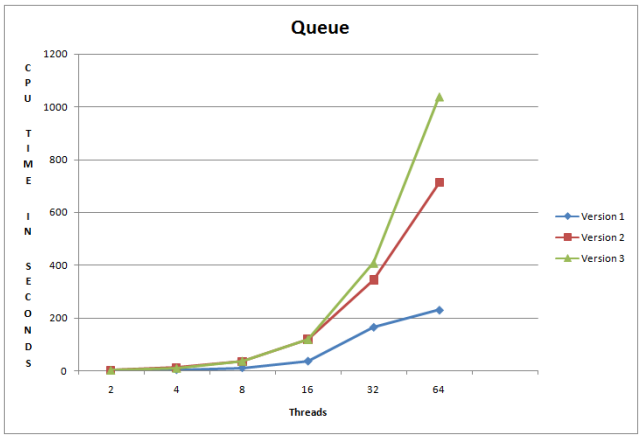

Figure 3: Displays the performance of the queue slices over interleaved pushes and pops

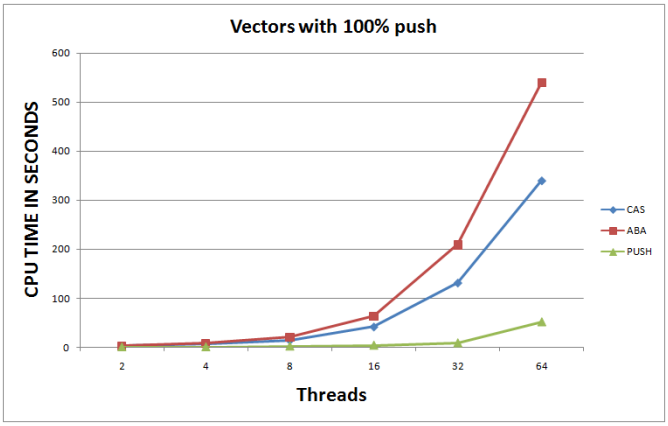

Figure 4: Shows the performance benefit of using specialized vectors in a push only environment

Our experiments show that different algorithms can create very large speedups over out-of-the-box solutions in Fig. 4. With a smart library that can shift between data structure implementations, we never pay for functionality that we do not use.

Large performance gaps between implementations makes shifting a desirable quality, especially when the need for a heavy implementation is in a short, but critical code block.

7. CONCLUSIONS

In this paper we present the concept of concurrent Data Structure Families (DSF), a novel abstraction mechanism for assisting programmers with concurrent algorithms' optimization and selection. One-size-fits-all libraries limit concurrency, and correctly implementing concurrent data structures is a hard problem. Typically, the most optimal algorithm for a given architecture is tightly coupled to the variations in the degree of parallelism, the specifics of the communication, and the available system resources. The existing mainstream compilers and programming libraries are not designed to

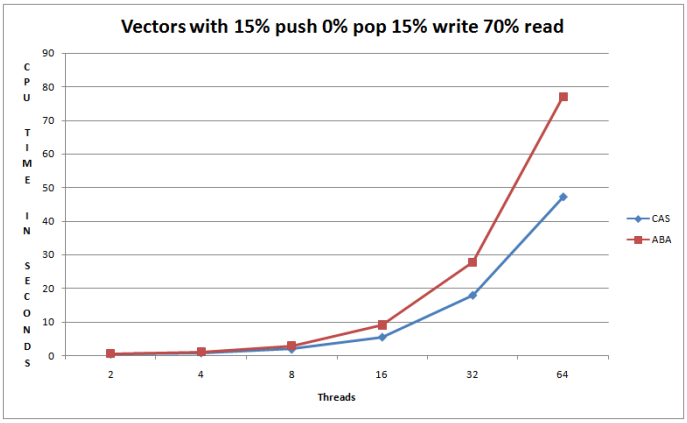

Figure 5: Displays the performance of the vectors over varied operation thresholds

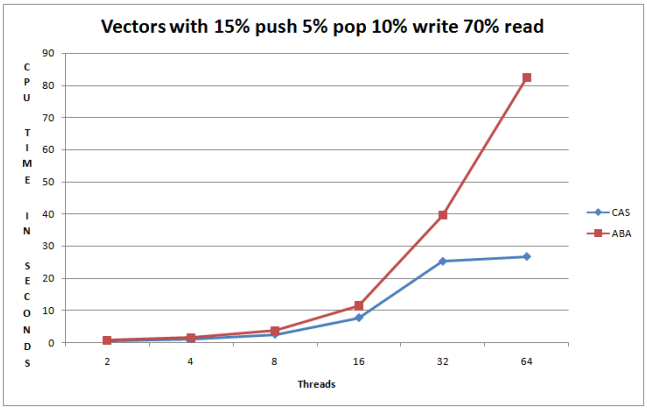

Figure 6: Displays the performance of the vectors over varied operation thresholds

handle algorithmic choice. Our proposed approach offers adaptable concurrent programming that can easily be integrated with current mainstream compilers and programming libraries and without requiring the developer to be an expert in concurrent data structures and algorithms. The DSF containers approach allows for significant increase of throughout and performance gains because: 1) it provides for a flexible use of concurrent objects as the application can be adapted to make use of the most optimal version of a data structure at each phase of execution and 2) the cost of providing for progress and correctness guarantees in multiprocessor programming is high; the data structure selection mechanism strives to satisfy only the minimal set of progress and correctness invariants needed by the user application. This paper introduces a first step towards the design and implementation of an automatic mutiprocessor algorithm selection capability. Possible future work in the evolution of the DSF concept involves the automation of the algorithm selection process through the use of program analysis and the automatic generation of application-specific data structure versions based on the application's properties and invariants.

Acknowledgment

This material is based upon work supported by the National Science Foundation under CCF Award No.1218100.

References

[1] Matthew H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[2] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. In *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '10, pages 185–192, Washington, DC, USA, 2010. IEEE Computer Society.

[3] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, Sep. 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[4] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), May 2007.

[5] Andreas Haas, Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. How fifo is your concurrent fifo queue? In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*, RACES '12, pages 1–8, New York, NY, USA, 2012. ACM.

[6] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Concurrent data representation synthesis. *SIGPLAN Not.*, 47(6):417–428, June 2012.

[7] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. *SIGPLAN Not.*, 48(1):317–328, January 2013.

[8] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier Science, 2011.

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

[10] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. Brainy: effective selection of data structures. *SIGPLAN Not.*, 47(6):86–97, June 2011.

[11] Lixia Liu and Silvius Rus. Perflint: A context sensitive performance advisor for c++ programs. In *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009*, pages 265–274. IEEE Computer Society, 2009.

[12] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[13] Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, March 2011.

[14] Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. *SIGPLAN Not.*, 43(6):125–135, June 2008.

[15] Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. *SIGPLAN Not.*, 45(6):160–173, June 2010.