Automating the Exchangeability of Shared Data Abstractions*

Jiange Zhang[1], Qian Wang[2], Qing Yi[1], and Huimin Cui[3]

[1] University of Colorado, Colorado Springs, USA, {jzhang3,qyi}@uccs.edu
[2] NVidia, China, traz0824@gmail.com
[3] Institute of Computing, Chinese Academy of Science, China, huimin.cui@gmail.com

Abstract. This paper presents a framework to support the automated exchange of data abstractions in multi-threaded applications, together with an empirical study of their uses in PARSEC. Our framework was able to speedup six of the benchmarks by up to 2x on two platforms.

1 Introduction

Software applications need to use synchronous data abstractions, e.g., queues and hash maps, to store shared data. The relative efficiency of these abstractions are not easily predictable when used in different scenarios. To demonstrate, Figure 1 shows the measured speedups when using C11 queue, TBB concurrent queue, and Boost deque, to replace a default ring-buffer task-queue on two hardware platforms. On both platforms, the TBB concurrent queue performs the best when the batch size is 1 but poorly when batch size is 20, where the C11 queue is the best on the AMD and the Boost deque the best on the Intel. There is not a single implementation that always performs the best.

This paper aims to support the automated exchange of abstractions in multi-threaded applications. Figure 2 shows our overall workflow, which includes (1) an *abstraction adapter interface* that documents the relations between different abstraction implementations and (2) an *abstraction replacement compiler* that automatically substitutes abstractions in applications with alternative ones based on the adapter specifications. Offline profiling is used to drive the optimizations.

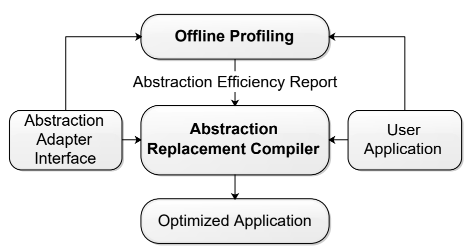
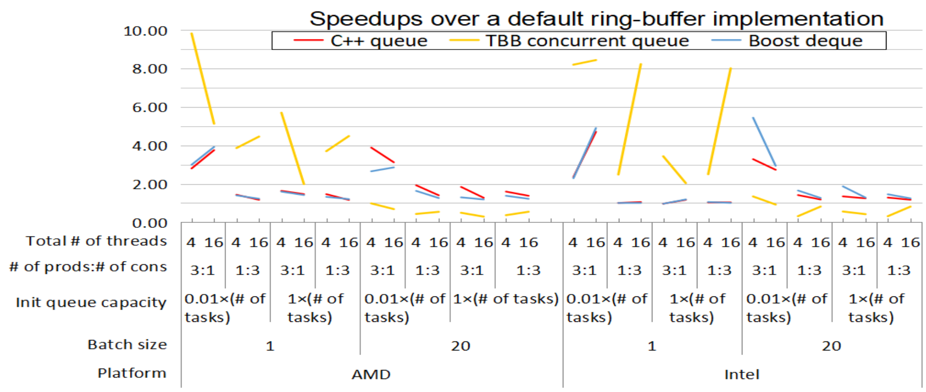


Fig. 2: Overall Workflow

The abstraction adapter interface, manually written by developers, is used to ensure correct optimization. Our technical contributions include the following.

- A programming interface for documenting the relations between different abstraction, thus allowing them to be used interchangeably in applications.

* This research is funded by NSF through award CCF-1261584.



*batch size: the number of tasks a thread can push into or pop from the queue each time;

# of prods: # of cons: the number of threads pushing into vs popping from the queue

Fig. 1: Efficiencies of three task queues on a 12-core Intel and 24-core AMD

– A source-to-source compiler that automatically replaces existing uses of abstractions in multi-threaded applications with alternative implementations;
– An empirical study of optimizing the use of data abstractions in PARSEC [4].

The rest of the paper presents each of the above components in more detail.

2 The Abstraction Adapter Interface

Figure 3 shows some example adapters defined using our interface, each in the form of *adapt* x *as* y { body }, where x is an existing abstraction being adapted; y is an abstract type name; and *body* is a sequence of interface functions, each defined by borrowing a subset of C++, enhanced with the following notations,

– *this*, which refers to the abstraction object being adapted;
– *val_type*, which refers to the type of values stored in abstraction x;
– $ref(t)$, which defines a pointer type to objects of type t;
– $array(t, n)$, which defines an array type with n elements of type t;
– the () notation, which refers to an empty type (the void type);
– $t_1 \to t_2$, which defines a function type that maps type t_1 to t_2;
– the | operator, which connects multiple implementations of a function;
– $syn.\_mutex\_lock(v)\{s\}$, which uses mutex lock v to synchronize block s;
– $syn.\_wait(c, v)$, which blocks a thread until the condition variable c is set;
– $syn.\_broadcast(c)$, which wakes up threads blocked on condition variable c;
– *foreach v in lower .. upper .. step do s enddo*, which repetitively evaluates statement s while setting variable v from *lower* to *upper* by *step*.

For two existing abstractions x_i and x_j to be exchangeable, two adapters a_i and a_j must be defined to respectively adapt them to a common abstract type. Further, the common interface functions in both a_i and a_j must be sufficient to cover all uses of x_i in the application. Our compiler checks these requirements and performs the substitution only when all the requirements are satisfied.

```
(1) adapt struct ::_ringbuffer_t {int head=0; int tail=0; int size=CONFIG;
                    val_type data[size]; } from dedup/{queue.h,queue.c} as task_queue {
  _empty = () -> (this.tail == this.head);
  _full = () -> (this.head == (this.tail - 1 + this.size) % this.size)
              | (this.tail == (this.head+1) % this.size);
  _erase_1 = (val : ref(val_type)) -> syn._mutex_lock(&this.mutex) {
              val = this.data[this.tail];
              this.tail = this.tail  +1; if (this.tail == this.size) this.tail=0;
            }
  _insert_1 = (x : val_type) -> syn._mutex_lock(&this.mutex) {
              this.data[this.head] = x;
              this.head = this.head + 1; if (this.head == this.size) this.head = 0;
            }
  _syn_erase_n = (val : array(val_type,1), n:int, lock : mutex, f1 : ()->(), f2 : ()->())
       ->  syn._mutex_lock(lock) { f1;
                                   foreach i in 0 .. n ..1 do
                                     this._erase_1(val[i]); if (this._empty()) { i=i+1; break; }
                                   enddo
                                   f2; return i; }
    _syn_insert_n = ......  };
(2) adapt tbb::concurrent_queue as task_queue {
   _empty = () -> this.empty();
   _full = ()->false;
   _try_insert_1 = (x : val_type) -> this.try_push(x);
   _try_erase_1=(val : ref(val_type))-> this.try_pop(val);
   _syn_erase_n = (val : array(val_type,1),n:int, lock : mutex, f1 : ()->(), f2 : ()->())
     -> {syn._mutex_lock(lock) { f1; }
         foreach i in 0 .. n ..1 do if (!this._try_erase_1(val[i])) break; enddo
         syn._mutex_lock(lock) { f2; }
         return i;  }
    _syn_insert_n = ......  };
```

Fig. 3: Example: abstraction adapter interface

3 The Abstraction Replacement Compiler

Our abstraction compiler takes three inputs: the user application to modify, the adapter interface that relates different abstractions, and a set of optimization configurations. The developer is expected to invoke our compiler with the same configurations on all files to ensure consistency of the substitution results. Each configuration instructs the compiler to convert an abstraction x_i to x_j, based on their adapters a_i and a_j. To do this, the compiler first finds the abstraction type and the adapter definitions to make sure they are consistent with each other. It then tries to convert each variable v_i of type x_i in each function f of the input application, by first outlining all uses of v_i into invocations of abstract interface functions in a_i. Then, it modifies the type definition of x_i: if only a subset of its member variables are used in a_i, a new member variable of type x_j is added to x_i to replace these member variables; otherwise, the type of v_i is simply changed from x_i to x_j. Finally, it inlines each abstract interface operation over v_i with implementations defined in adapter a_j over the new v_j variable.

The key of the compiler is its outlining algorithm, which includes three steps: (1) normalize the input code to use higher-level notations defined in the adapter interface; (2) sort all interface functions in increasing granularity and convert each interface function f_a into a set of patterns, where variables, e.g., *val*, *n*, *lock*, *f*1, *f*2, and *this* in _syn_erase_n of the *task_queue* in Figure 3, are converted to pattern parameters that can be matched to different expressions and statements; and (3) use each implementation pattern generated in step (2) to match against existing input code, while outlining each matched code fragment

```
struct queue {
    int head, tail, size;  void** data;    int count, threads;
    pthread_mutex_t mutex; pthread_cond_t empty, full;
};
int dequeue(struct queue *que, int *fetch_count, void **to_buf) {
    pthread_mutex_lock(&que→mutex);
    while((que→tail==que→head)&&(que→count<que→threads))
        {pthread_wait(&que→empty,&que→mutex);}
    if((que→tail==que→head)&&(que→count==que→threads)) {
        pthread_cond_broadcast(&que→empty); pthread_mutex_unlock(&que→mutex); return -1;}
    for((*fetch_count)=0; (*fetch_count)<16; (*fetch_count)++) {
        to_buf[(*fetch_count)]=que→data[que→tail]; que→tail++;
        if (que→tail==que→size) que→tail = 0;
        if (que→tail==que→head){(*fetch_count)++; break;}}
    pthread_cond_signal(&que→full); pthread_mutex_unlock(&que→mutex); return 0;}
```

Fig. 4: An example queue abstraction

into an invocation of the corresponding interface function. Figure 5(a-b) illustrate the results of these steps when outlining the dequeue function in Figure 4, with the result of instantiating the outlined code by using the TBB concurrent queue adapter shown in (c). Here the original mutex protected critical section has been split into three subsections, with the middle section no longer protected by the lock and instead invoking the already synchronous *try_erase* function of the TBB queue. Such algorithmic changes are enabled by the adapter definitions, which can be made quite powerful by integrating knowledge from developers.

Our compiler follows two steps to outline each implementation pattern from an input code. First, it traverses all statements in the input code while matching each of them against all parts of the given pattern, with each successful match remembering the required values for each pattern parameter. Then, it examines the saved matches to see whether they can be outlined without violating dependences of the original function, while performing the outlining transformation only when safe. Specifically, each outlining transformation requires a sequence of statements in the input code that are matched precisely to the sequence of statements in the given pattern, without any conflicting assignments of values to the pattern parameters, and with no dependence cycle involving any other intervening statements in the input code. Note that single pattern parameters such as variables $f1$ and $f2$ in *_syn_erase_n* of adapter (1) in Figure 3 can be matched to a sequence of statements in the input code, to enhance effectiveness.

4 Experimental Evaluation

We have implemented our infrastructure using the POET language [16] on top of the ROSE C/C++ open-source compiler [12]. We used our adapter interface to manually document a set of queue and map implementations from the PARSEC benchmarks [4] and from C++11 std [2], TBB [13] and Boost [1] libraries. We also identified a number of simple mutex-based synchronization patterns and automatically correlated them with equivalent non-blocking synchronizations, illustrated in Figure 6. We then tried to optimize PARSEC [4] 3.0, by replacing their existing uses of queue, map, and synchronization abstractions. We used offline profiling to determine the performance of different abstractions in different use cases.

We evaluated all benchmarks on two platforms, shown in Table 1. All benchmarks were compiled using *icc* with *-O3* on the Intel machine and using *g++* with *-O3* on the

```
int dequeue(struct queue *que, int *fetch_count, void **to_buf) {
syn._mutex_lock(&que→mutex): {
   while((que._empty()&&(que→count<que→threads)) {syn._wait(&que→empty,&que→mutex);}
   if((que._empty()&&(que→count==que→threads)) { syn._broadcast(&que→empty); return -1;}
   foreach i in 0 .. 16 .. 1 do
      to_buf[i]=que→data[que→tail]; que→tail=que→tail+1; if (que→tail==que→size) que→tail = 0;
      if (que._empty()){i=i+1; break;} enddo
   (*fetch_count)=i; syn._signal(&que→full);}; return 0;}
```

(a) after normalization and outlining _empty

```
int dequeue(struct queue *que, int *fetch_count, void **to_buf) {
(*fetch_count) = _syn_erase_n(to_buf, 16, &que→mutex,
   /*f1*/{ while((que._empty()&&(que→count<que→threads)) {syn._wait(&que→empty,&que→mutex);}
      if((que._empty()&&(que→count==que→threads)) { syn._broadcast(&que→empty); return -1;}},
   /*f2*/ { syn._signal(&que→full);}); return 0;}
```

(b) after outlining _erase_1 and _syn_erase_n

```
struct queue {
   tbb::concurrent_queue<void*> *tbb_que;   int count, threads;
   pthread_mutex_t mutex; pthread_cond_t empty, full; };
int dequeue(struct queue *que, int size, void **to_buf) {
   pthread_mutex_lock(&que→mutex);
   while ((que→tbb_que→empty())&&(que→count<que→threads))
       {pthread_cond_wait(&que→empty,&que→mutex);}
   if ((que→tbb_que→empty())&&(que→count==que→threads)) {
       pthread_cond_broadcast(&que→empty); pthread_mutex_unlock(&que→mutex); return -1;}},
   pthread_mutex_unlock((&que→mutex);
   for(int i=0; i<size; i+=1) { if (!que→tbb_que→try_pop(to_buf[i])) break;  }
   pthread_mutex_lock(&que→mutex); pthread_cond_signal(&que→full);
   pthread_mutex_unlock(&que→mutex); (*fetch_count) = i; return 0;}
```

(c) after replacement

Fig. 5: Example: substitute the queue in Figure 4 with TBB concurrent_queue

```
adapt { x : val_type; pt : syn.mutex; } as atomic_var {
(1) _syn_fetch_add = (incr: val_type) →
      { syn._mutex_lock(this.pt) { tmp : val_type =this.x; this.x=this.x + incr; } return tmp;}
(2) _syn_add_fetch = (inc:val_type) → { syn._mutex_lock(this.pt) { thix.x=this.x + inc; } return this.x;}
(3) _syn_set_value = (v : val_type) → { syn._multex_lock(this.pt) { this.x=v; } }
(4) _syn_set_and_broadcast = (pc : syn.cond_var) →
      { syn._multex_lock(this.pt) { this.x=v; syn._broadcast(pc) } }
(5) _syn_wait_cond = (cond : bool, pc : syn.cond_var) →
      { syn._multex_lock(this.pt) { while (cond) syn._wait(pc, this,pt); } } }
(6) adapt ::pthread_barrier_t as thread_barrier {
    _barrier_init = (n_threads : int) → {pthread_barrier_init(this, NULL, n_threads);}
    _barrier_wait = () → {pthread_barrier_wait(this);}
    _barrier_destroy = () → {pthread_barrier_destroy(this);} }
```

Fig. 6: Example adapters for synchronization operations

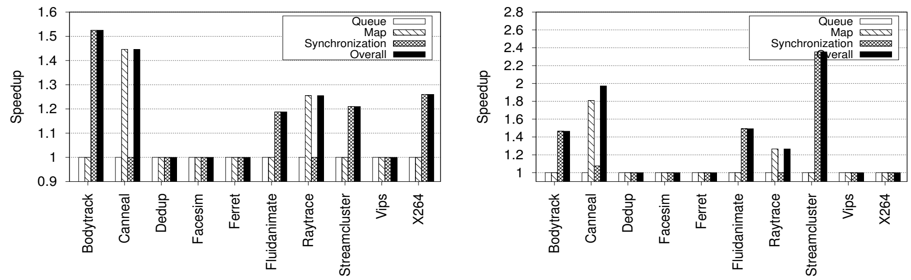
AMD. Each benchmark is evaluated by using its *native* input (the largest input set) and with a thread configuration that provides the best performance. Each measurement is repeated 10 times, and the average used to calculate performance speedups. The variations across different runs of the same code are $\leq 5\%$.

Our framework is able to support the exchange of all uses of pre-defined queue, map, and synchronization abstractions in PARSEC (they are used in 10 of the 13 available benchmarks). Figure 7 shows the overall performance speedups attained by our compiler, together with a breakdown of the speedups from tuning only the queue, map, and synchronization abstraction implementations respectively.

Four PARSEC benchmarks (Dedup, Bodytrack, Ferret and Facesim) use the queue abstraction. However, they are all designed to minimize contention among the threads over the queue operations. Due to low contention, a better synchronized queue implementation does not produce any speedup, unless the overall application is modified to increase concurrency among the threads. The map abstraction is also used in four

CPU	Freq.	L1 Cache sz	L2 Cache sz	# of cores
Intel E5-2420	1.9GHz	32KB	256KB	12
AMD Opteron-6128	2GHz	64KB	512KB	24

Table 1: Platform Configurations



(a) on the Intel Platform (b) on the AMD Platform

Fig. 7: Performance speedups attained by our compiler

PARSEC benchmarks: Canneal, Dedup, Raytrace and Vips. Speedups of 1.255-1.806x are achieved for Canneal and Raytrace, by replacing their uses of the C++ std::map, which is internally a red-black tree. with the faster C++ std::unordered_map, which is internally a hash table. No speedups were attained for Dedup and Vips because their maps are already quite efficient. Most speedups (1.08-2.35x) are attained by replacing the underlying implementations of synchronizations in Canneal, Bodytrack, Fluidanimate, and Streamcluster. All four benchmarks benefited from replacing their uses of Pthread barriers with a lighter weight implementation using atomic operations followed by spin waiting. Bodytrack and X264 also benefited from using atomic operations to replace their mutex-based synchronizations over single global shared variables. The results across platforms are mostly consistent. We have observed from tuning these applications that their uses of abstractions are tightly connected with other aspects of application design, and replacing a single abstraction in isolation is often not rewarding, unless the abstraction itself is complex enough to offer significant opportunities.

5 Related Work

The idea of automated data structure selection originated in the context abstract data types [9]. More recent work has studied the automatic selection of abstraction implementations for performance optimizations [14, 10, 5] and the use of nonblocking synchronizations in multi-threaded applications to enable better load balancing and scalability [3, 7, 11, 8, 15]. In this paper, we develop compiler support to automate the deployment of alternative abstraction implementations. Existing frameworks on abstraction-aware optimizations mostly focus on optimizing a specific type of data abstraction, e.g., matrices [6] and arrays [17]. Our framework aims to support the automated selection of general-purpose abstractions in multi-threaded applications.

6 Conclusion

This paper presents a framework for automatically exchanging abstraction implementations in multi-threaded applications to enhance performance portability. The framework is used to optimize the use of queues, maps, and synchronization abstractions in the PARSEC benchmarks.

References

1. Boost 1.56.0 Library Documentation (2014), http://www.boost.org/doc
2. Standard C++ Library reference (2014), http://www.cplusplus.com
3. Barrington, A., Feldman, S.D., Dechev, D.: A scalable multi-producer multi-consumer wait-free ring buffer. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015. pp. 1321–1328 (2015)
4. Bienia, C., Li, K.: Parsec 2.0: A new benchmark suite for chip-multiprocessors. In: Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation (June 2009)
5. Cho, D., Pasricha, S., Issenin, I., Dutt, N., Paek, Y., Ko, S.: Compiler driven data layout optimization for regular/irregular array access patterns. In: Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08), Tucson, AZ, USA, June 12-13, 2008. pp. 41–50 (2008)
6. Cui, H., Yi, Q., Xue, J., Feng, X.: Layout-oblivious compiler optimization for matrix computations. ACM Trans. Archit. Code Optim. **9**(4), 35:1–35:20 (Jan 2013), http://doi.acm.org/10.1145/2400682.2400694
7. Dechev, D., LaBorde, P., Feldman, S.D.: LC/DC: lockless containers and data concurrency a novel nonblocking container library for multicore applications. IEEE Access **1**, 625–645 (2013)
8. Feldman, S.D., Bhat, A., LaBorde, P., Yi, Q., Dechev, D.: Effective use of non-blocking data structures in a deduplication application. In: Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity. pp. 133–142. SPLASH '13, ACM, New York, NY, USA (2013)
9. Low, J.R.: Automatic data structure selection: An example and overview. Commun. ACM **21**(5), 376–385 (May 1978). https://doi.org/10.1145/359488.359498, http://doi.acm.org/10.1145/359488.359498
10. Majeti, D., Barik, R., Zhao, J., Grossman, M., Sarkar, V.: Compiler-driven data layout transformation for heterogeneous platforms. In: Euro-Par 2013: Parallel Processing Workshops - BigDataCloud, DIHC, FedICI, HeteroPar, HiBB, LSDVE, MHPC, OMHI, PADABS, PROPER, Resilience, ROME, and UCHPC 2013, Aachen, Germany, August 26-27, 2013. Revised Selected Papers. pp. 188–197 (2013)
11. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996. pp. 267–275 (1996)
12. Quinlan, D., Schordan, M., Yi, Q., Saebjornsen, A.: Classification and utilization of abstractions for optimization. In: ISOLA'04: The First International Symposium on Leveraging Applications of Formal Methods. Paphos, Cyprus (Oct 2004)
13. Reinders, J.: Intel threading building blocks. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edn. (2007)
14. Rubin, S., Bodík, R., Chilimbi, T.M.: An efficient profile-analysis framework for data-layout optimizations. In: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002. pp. 140–153 (2002)

15. Tsigas, P., Zhang, Y.: Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies. In: Workshop on Software and Performance. pp. 55–66 (2002)
16. Yi, Q.: POET: A scripting language for applying parameterized source-to-source program transformations. Software: Practice & Experience pp. 675–706 (May 2012)
17. Yi, Q., Quinlan, D.: Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In: LCPC04: The 17th International Workshop on Languages and Compilers for Parallel Computing. West Lafayette, Indiana, USA (Sep 2004)