# Self-Adaptive Containers: Interoperability Extensions and Cloud Integration

Wei-Chih Huang
Department of Computing, Imperial College London
London, SW7 2AZ, United Kingdom
Email: wei-chih.huang11@imperial.ac.uk

William Knottenbelt
Department of Computing, Imperial College London
London, SW7 2AZ, United Kingdom
Email: wjk@imperial.ac.uk

*Abstract*—**Driven by an ever-increasing diversity of application contexts, execution environments and scalability requirements, modern software is faced with the challenge of frequent code refactoring. To address this, we have proposed an STL-like self-adaptive container library, which dynamically changes its data structures and resource usage to meet programmer-specified Service Level Objectives relating to performance, reliability and primary memory use. A prototype of this library has been implemented and utilised in two case studies to prove its viability. In the present work, we explore a low-cost means to extend our library to satisfy wider classes of Service Level Objectives. This is achieved through the integration of third-party container frameworks, which exploit parallelism to boost performance and disk-based data offloading to reduce primary memory consumption, and the integration of cloud storage services, which offer cost-effective location-free storage. We demonstrate our library's application in a state-space exploration case study. With very low programmer overhead, experimental results show that our library can improve performance with a 76% reduction in insertion time and an 86% reduction in search time, and can also exploit out-of-core storage, including cloud storage.**

***Keywords—Self-Adaptive Systems; Containers; Standard Template Library; Out-Of-Core Storage; Cloud Storage; Parallelism***

## I. Introduction

The number of execution environments in which applications are deployed (e.g. smartphones, servers, laptops, tablets, etc.) is growing rapidly. It is a non-trivial challenge to write software which can meet Quality of Service (QoS) requirements in diverse application contexts and execution environments, especially where resources are limited and scalability requirements are high [1, 2]. Traditional software engineering techniques [3–5] cannot effectively cater for such diversity, which results in either a small code base which cannot guarantee QoS or multiple manually-optimised code bases which are difficult to maintain [6].

To address the above-mentioned challenges, we have introduced a self-adaptive container library [7] which can dynamically adapt to support large input data and can automatically change its underlying data structures in order to satisfy resource constraints and QoS requirements pertaining to the current execution environment. Our library features a mechanism for tighter functionality specification, which enables greater scope for efficiency optimisations including the exploitation of out-of-core storage, probabilistic data structures, and parallelism. Additionally, the adopted data structure is associated with desired Service Level Objectives (SLOs) rigorously specified in the WSLA [8] format. Our work focuses on containers because they are frequently the performance and capacity bottlenecks in real systems [9], and choice of underlying data structures is critical to delivered performance, memory usage, and reliability [10]. This implies that change of the adopted data structure affects QoS and resource usage [11]. The usage of our library is designed to be as similar as possible to that of the Standard Template Library (STL) [12] with the aim of reducing required levels of expertise and programmer effort. To illustrate our library's viability, applicability, and scalability, a prototype including a single value container and an associative key-value store container is implemented and utilised in two case studies [7, 13] – state-space exploration and route planning – exploiting out-of-core storage and probabilistic data structures. The results of the two studies show that our library provides a ready route to easily adapt software to each different execution environment; the library also has the ability to automatically meet programmer-specified SLOs with respect to performance, memory consumption, and reliability.

Many research teams have proposed their container frameworks to support efficient out-of-core storage (e.g. STXXL [14], TPIE [15], LEDA-SM [16], PSTL [17]) and parallelism (e.g. TBB [18], STAPL [19], HPC++ [20]). These frameworks provide well-developed functionalities and member functions, which considerably decreases development time and lowers the required level of expertise. Although these frameworks can effectively help programmers, they still need to be reconfigured when execution environments and application contexts change. However, through integration of these frameworks, our library can reduce the complexity of implementing out-of-core storage and parallel software, provide a broader class of SLOs, and automatically configure these frameworks on-the-fly.

Recent years have seen the rise of cloud storage, which carries many benefits (e.g. low cost, high security, location independence). To exploit cloud storage, the functionalities accessing stored data have to be modified. Further, because the performance of cloud storage is lower than that of disks, the overall performance is notably worsened when most data is stored in the cloud. Thus, it is critical to dynamically adjust the utilisation rates of local disks and cloud storage. Currently, to the best of our knowledge, no container library makes use of cloud storage. Our library aims to supply programmers with alternative out-of-core memory dynamically exploited when local disks are not available and with efficient access functions in an effort to reduce the degree to which code is modified.

This paper yields the following three contributions.

- We integrate a well-developed third-party out-of-core container library, STXXL [14], into our library for efficient out-of-core storage, which eliminates the need for high expertise and programmer effort of implementing out-of-core storage.
- Our library now supports automatic parallelism, which is not entailed in our previous implementation, through cooperation with Intel Threading Building Blocks (TBB) [18], which supplies widely-used containers and algorithms in order to lessen the complexity of developing concurrent software. This allows our library to satisfy a broader class of Service Level Objectives, particularly those relating to performance, without affecting existing code.
- By means of the dynamic exploitation of Amazon Simple Storage Service (S3) [21], our library can now provide alternative memory to store data when primary memory and local disks are not available. Transfer of data to and from cloud storage is transparently managed by our library, rather than being the explicit responsibility of programmers.

The remainder of this paper is organised as follows. Section II describes background and related work. Section III presents the self-adaptive container framework, while Section IV introduces the use of third-party container frameworks. Section V shows the exploitation of cloud storage. Section VI shows the experimental results, and Section VII summarises the conclusion and future work.

## II. Background and Related Context

### A. Containers

Containers are responsible for storing related objects which can be accessed through supported member functions. They are implemented as template classes, which allows programmers to specify built-in/user-defined data types. The most commonly-used container libraries are the STL [12] (implemented in C++) and Java Collections Framework [22]. Since our library is implemented in C++, this subsection will focus on the STL. The STL's containers can be categorised into sequence containers, associative containers, container adaptors, and unordered associative containers. Sequence containers store inserted elements according to their original orders, which enables programmers to specify positions of inserted elements. Falling into the category of sequence containers are *vector*, *list*, and *deque*. Associative containers maintain inserted elements in a predefined order (e.g. sorted ascending). They can be further divided into *set* (including *set* and *multiset*) and *map* (including *map* and *multimap*). Both *set* and *map* apply trees as their underlying data structures. Container adaptors, consisting of *stack*, *queue*, and *priority_queue*, are interfaces on top of other containers. For example, *stack*'s functionalities can be supported through *vector*, *list*, or *deque*. The STL's containers' final category, unordered associative containers, makes use of hash tables to store elements. Although they are used to improve performance, they have two constraints. First, when programs adopt unordered associative containers, -std=c++0x has to be set in compiler options. Second, when user-defined elements are stored, custom hash functions are required.

### B. Related Work and Context

In the late 1990s and early 2000s, IBM coined the term of autonomic computing [23], derived from human autonomic nervous systems, which can unconsciously control human bodies (e.g. heart rate, salivation, perspiration). The purpose of autonomic computing is to deal with the challenge of managing rapidly growing system complexity. To achieve this purpose, a system should be equipped with self-CHOP properties [24] (self-configuration, self-healing, self-optimisation, and self-protection), which are fulfilled by sensors, effectors, and autonomic managers. Sensors are used to retrieve required information from managed resources, and effectors are used to manage resources. Each autonomic manager contains a self-adaptive cycle composed of an observation phase, an analysis phase, and an adaptation phase [25]. The cycle starts from the observation phase, which monitors managed resources and collects required data from sensors. The analysis phase determines if an adaptation action should be taken in accordance with comparison results of the data reported from the observation phase and expectations (e.g. knowledge, rules, etc.) and plans a suitable adaptation action as necessary. The adaptation phase performs the adaptation action selected in the analysis phase by means of effectors.

Many researchers have proposed their reference models for building self-adaptive systems. MAPE-K [26, 27] is one reference model, which divides the functionalities of self-adaptive systems into five functions, Monitor, Analyse, Plan, Execute, and Knowledge. Some research work focuses on model-based approaches (e.g. [28–30]). Some researchers attempt to develop self-adaptive systems through architecture-based self-adaptations (e.g. Rainbow [31]). Some pieces of research are related to requirements-based approaches (e.g. Zanshin [32]). In addition to the above-mentioned methodologies, some researchers adopt language-level self-adaptations, which can be classified into meta-programming [33], aspect-oriented programming [34], and context-oriented programming [35] based on Salvaneschi et al.'s work [36]. The latter study lists programming language extensions which can perform adaptation actions (e.g. Iguana/J [37], JAsCo [38], ContextJ [39], ContextErlang [40]). However, compared to our library, programmer overhead, i.e. the amount of code that needs to be changed relative to a naïve implementation, is relatively high.

Among approaches automatically changing data structures to save resources are SILT [41] and OSKI [42]. SILT is a flash-based key-value store system featuring several underlying candidate data structures with data being converted between them according to the size of key fragments at run time. However, it only focuses on memory usage. Other QoS metrics (e.g. performance or reliability) are not taken into account. OSKI, which is a collection of low-level primitives, provides automatically tuned computational kernels as well as a mechanism selecting a data structure and code transformations. Similarly, the application areas of OSKI are restricted because in addition to performance it does not consider other QoS metrics. Indeed, there is no mechanism for specifying any Service Level Objectives, which leads to difficulties in adapting software to each execution environment and application context.

## III. Self-adaptive Container Framework

This section will briefly introduce our library's framework, self-adaptive mechanism, and adaptation actions. For full details, please refer to our previous publications [7, 13]. As can be seen in Figure 1, the library consists of two major components, Application Programming Interface (API) and Self-Adaptive Unit (SAU). The former provides a means through which programmers can lay down required operations and control the library, and the latter performs operations and activates the self-adaptive mechanism. API provides programmers with two template classes covering most functionalities of the STL: `ICollection`, which implements the functionalities of the STL's *vector*, *list*, *queue*, *priority_queue*, *stack*, *deque*, *set*, and *multiset* and `IKeyValue`, supporting key-value stores (i.e. the STL's *map* and *multimap*). The member functions of `ICollection` and `IKeyValue` can be divided into operation interfaces, which are a group of commonly-used operations, and configuration interfaces, which provide a way of controlling the library. The usage of the container constructors are expressed as follows.

`ICollection<`$T, Compare = less < T >$`>(`*op_desc*, *SLO_file*[, *freq*]`)`

and

`IKeyValue<`$K, V, Compare = less < K >$`>(`*op_desc*, *SLO_file*[, *freq*]`)`

where *op_desc* describes the required functionalities, *SLO_file* indicates a path to an XML file containing a description of the SLOs related to the container in WSLA format, and *freq* (optional parameter) specifies the frequency with which the self-adaptive mechanism is activated. An example of how to express SLOs for our self-adaptive containers in WSLA format is shown in [7].

The SAU comprises an Execution unit, an SLO store, an Observer, an Analyzer, and an Adaptor. The Execution unit performs container manipulation commands given by operation interfaces. The SLO store holds all SLOs laid down by configuration interfaces. The Observer monitors per operation response time (via the *clock_gettime()* system call in Linux and the *QueryPerformanceFrequency* function in Windows), computes memory consumption, and calculates reliability when a probabilistic data structure is exploited. The Analyzer determines whether an adaptation action should be taken and where appropriate plans an adaptation action. The Adaptor performs the selected adaptation action.

The self-adaptive mechanism of our library is a classical self-adaptive cycle [25], which is formed by the Observer, the Analyzer, and the Adaptor. The mechanism starts working when the Observer monitors the Execution unit to obtain operation profiles (e.g. per operation response time, memory usage, and, where appropriate, reliability). The operation profiles are then sent to the Analyzer, which compares them with SLOs to determine if any SLOs are violated. If a certain SLO is violated, the Analyzer will decide if an adaptation action is required based on the following rules:

(a) The adaptation action is expected to result in either the satisfaction of the SLO or a reduction in the degree to which the SLO is flouted.

(b) The adaptation action is not expected to violate a currently-satisfied SLO of higher priority.

We design Rule (a) because we have recognised that it may not be possible to meet all (or any) of the SLOs within resource constraints. The purpose of Rule (b) is to prevent an adaptation action taken to address one violated SLO from violating another SLO. For example, the adaptation action taken to reduce primary memory may lead to performance drop. In order to solve the above-mentioned issue, each SLO is assigned a distinct priority according to its declaration sequence. The Analyzer addresses each SLO in priority order. If the SLO being addressed is satisfied, no adaptation is necessary. If the SLO is violated, the Adaptor is called in for adaptation.

The Adaptor may perform three kinds of adaptation actions in accordance with the nature of the violated SLO and its priority ordering. If it is performance-related (e.g. an SLO related to insertion or search response time), then gains may be had from subdividing the underlying data structure. In general this will increase memory consumption but in the case of a probabilistic data structure this adaptation will also increase reliability. If the violated SLO is memory-related, then gains may be had from utilising out-of-core storage (with the side effect of hurting performance), or, should reliability and functionality requirements allow, moving to a probabilistic underlying data structure. Finally, if the violated SLO is reliability-related (e.g. the number of elements inserted into our container with only "insert" and "search" functionality has increased to such an extent that the underlying probabilistic data structure no longer meets its reliability SLO), then the data structure should be subdivided (with the side effect of improving performance and increasing memory).

## IV. The Utilisation of Third-Party Container Frameworks

In this work, our library integrates STXXL [14] and MC-STL [43] for out-of-core storage and Intel Threading Building Blocks (TBB) [18] for parallelism. STXXL provides out-of-core containers and algorithms, both of which aim to deal with a huge amount of data in out-of-core memory. We choose to integrate STXXL rather than other out-of-core libraries for the following reasons. First, STXXL gives transparent support of parallel disks and implements parallel disk algorithms. Second, it makes use of "pipelining" and "overlapping" to improve performance and resource utilisation, respectively. Third, STXXL can cooperate with MCSTL for the purpose of internal computation improvement. MCSTL is an OpenMP-based [44] algorithm library exploiting multiprocessors and the multi-cores of a processor. Software adopting MCSTL can achieve performance improvement without any changes due to the common algorithm names shared by MCSTL and the STL. In spite of these advantages, MCSTL restricts adopted compilers (i.e. gcc and g++) to lower versions (lower than version 4.2).

Despite the benefits brought by STXXL and MCSTL, there are two drawbacks which may lead to performance deterioration. First, when primary memory is sufficient to hold all data, the activation of STXXL will result in considerable performance decline owing to frequent access of out-of-core memory. Second, STXXL is not equipped with a mechanism

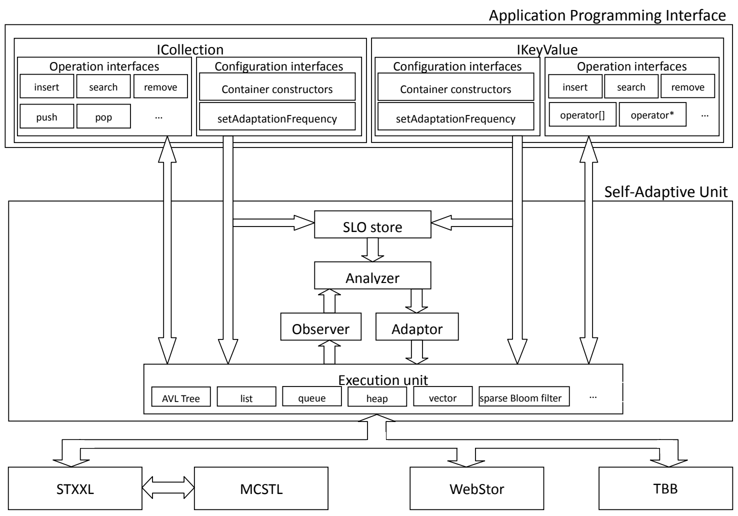

Figure 1. The self-adaptive container library architecture

for dynamically adjusting the in-core and out-of-core memory utilisation rates. When data is moved into external memory, this means more available primary memory. As a result, newly inserted data should be stored in in-core memory. Once primary memory is full, part of the data should be moved to external memory again. Because STXXL cannot specify primary memory constraints, it allocates a pre-defined amount of primary memory for caching data. This causes in-core memory usage to be hardly optimised. To reap the benefits of STXXL and avoid the disadvantages, our library acts as a controller which determines when to trigger STXXL and dynamically transfers data from in-core memory to out-of-core memory according to primary memory constraints.

Intel TBB is a concurrent library which supplies containers (e.g. *concurrent_queue*, *concurrent_hash_map*) and algorithms (e.g. *parallel_sort*, *parallel_for*) in order to reduce the complexity of developing multi-threaded software. TBB has the ability to detect the number of cores, which prevents reconfiguration when execution environments change; it adopts the technique of task stealing [45], which enables tasks to be dynamically reassigned to different cores, in order to enhance core utilisation. TBB's performance is affected by the number of threads, which implies that its performance can be dynamically adjusted by way of re-specifying the maximum number of threads. Additionally, even though TBB provides STL-like interfaces, programmers still have to learn how to control it and how to utilise the modified interfaces. For example, TBB's *concurrent_queue* does not support *front* operations and its *pop* operations require a parameter for storing a removed element.

## V. The Exploitation of Cloud Storage

Cloud storage refers to a data storage model where data is stored in the cloud and managed by a hosting company (e.g. Amazon [21], Google [46], Dropbox [47], etc.). Companies storing their data in the cloud do not need to purchase expensive disks or spend time backing up. Further, data stored in the cloud can be accessed through the internet at any time at any place (i.e. local-independent). The major downside to cloud storage is its dependency on and consumption of network bandwidth. If the internet connection is unstable or slow, the stored data may not be accessible. However, as network bandwidth is broadened and connectivity becomes ubiquitous, it is possible to overcome this barrier.

Since cloud storage services have been adopted by many companies, programmers may frequently adapt their software to meet different cloud storage services and their capacity limits. This adaptation process can be simplified by the self-adaptive mechanism our library possesses. It is capable of dynamically exploiting programmer-specified cloud storage services. Simultaneously, memory consumption is monitored in conformity with capacity limits. Our library makes use of WebStor [48] library, which provides C++ APIs, to implement the required functionalities of cloud storage. WebStor is designed for the cloud storage services supporting Cloud Storage Engine for MySQL (ClouSE), e.g. Google Cloud Storage or Amazon S3. Additionally, it supports parallel operations (e.g. put, get, delete), which can improve throughput to a large extent. To boost performance and reduce cost, our library splits data and transfers them to buckets under different accounts. There are two reasons for this design. First, most cloud storage service providers offer each customer free but limited post and get operations as well as capacity. Second, cloud storage does not come with search operations, and nor is it capable of specifying a position indicator associated with a stream, which means when an element is searched, the file where this element may be stored has to be fully downloaded in order to ascertain whether or not it exists. As a result, the subdivision of data can expeditiously reduce both the size of individual files stored in each account and the number of required post and get operations.

## VI. Case Study

The case study chosen to illustrate our library's applicability is an explicit state-space exploration algorithm, which is commonly employed in the domains of model checking [49] and concurrent performance analysis [50]. This algorithm is executed on a Linux machine with 64 Intel Xeon E5-4650 CPUs (2.70 GHz) and 528 GB memory. As can be seen in

```
void bfs (Graph G, State s)
{
    queue<State> unexplored;
    set<State> explored;

    unexplored.push(s);
    explored.insert(s);
    while (!unexplored.empty()) {
        State next = unexplored.front();
        unexplored.pop();
        for (State *w = G.first_edge(next) ; w ; w = G.next_edge(next)) {
            if (!explored.search(*w)) {
                    unexplored.push(*w);
                    explored.insert(*w);
            }
        }
    }
}
```

```
void bfs (Graph G, State s)
{
    ICollection<State> unexplored(OP_QUEUE, "UnexploredSLOs.xml");
    ICollection<State> explored (OP_INSERT|OP_SEARCH, "ExploredSLOs.xml", 100);

    unexplored.push(s);
    explored.insert(s);
    while (!unexplored.empty()) {
        State next = unexplored.front();
        unexplored.pop();
        for (State *w = G.first_edge(next) ; w ; w = G.next_edge(next)) {
            if (!explored.search(*w)) {
                    unexplored.push(*w);
                    explored.insert(*w);
            }
        }
    }
}
```

Figure 2. The naïve program (left) and the program adopting our library (right)

Figure 2, the only difference between the naïve program and the program adopting our library is the container declarations (one for the queue of unexplored states, and one for the table of explored states). The following subsections will show the library's behaviour when STXXL, MCSTL, TBB and Amazon S3 are activated.

### A. Exploiting STXXL and MCSTL for Efficient Out-of-core Storage

Figures 3 and 4 show the average insertion time and search time taken by different memory-sensitive container frameworks. As can be seen, when our library utilises MCSTL and STXXL's map, its average search time is close to that of our baseline implementation, but the average insertion time is slower than that of the baseline implementation. STXXL makes use of a B-tree to store data in out-of-core memory, which requires more I/O access to adjust the tree's height. By contrast, when our library employs MCSTL and STXXL's vector, whose elements are sorted, its insertion performance is better but its search performance is slower. That is because the number of I/O is reduced when data is transferred to out-of-core memory, the number of I/O increased when data is searched. The two figures also indicate that when STXXL is utilised alone, performance is the lowest, the reason for which is all the data stored in out-of-core memory. In summary, this experiment illustrates that through the utilisation of STXXL programmer effort and expertise required to implement out-of-core storage are reduced.

The response times for push and pop operations expended by our baseline implementation and STXXL's *queue* are depicted in Figures 5 and 6, which show that the performance of baseline implementation is similar to that of STXXL. That is because both implementations adopt the same methodology in which the head and tail blocks are kept in primary memory.

### B. Exploiting TBB for High Performance

The average insertion time and search time expended by STL's set and our library adopting TBB under 8 threads are exhibited in Figures 7 and 8. As can be seen, when our library utilises TBB's *concurrent_hash_map* exploiting 8 threads to store explored states, performance is improved. Specifically, insertion time is reduced by 76%, search time reduced by 86%.

Figures 9 and 10 depict the average push and pop times of STL's queue and our library utilising TBB under different maximum numbers of threads. Similarly, our library, adopting TBB's *concurrent_queue*, is faster than STL's queue (i.e. 54% reduction in push time and 77% reduction in pop time when the maximum number of threads is 8).

### C. Exploiting Cloud Storage

In the case study, Amazon S3 is utilised to demonstrate the dynamic exploitation of cloud storage. Figures 11 and 12 indicate the average insertion time and search time expended by our library exploiting baseline implementation and Amazon S3. As can be seen, the performance of Amazon S3 is lower than that of the baseline implementation, which is in keeping with the actual situation where disk I/O performance is faster than network performance.

Figure 13 represents primary memory consumed by the baseline out-of-core implementation and cloud storage. It shows that when the number of stored states is approximately 31,000 our library activates cloud storage in order to protect the memory-related SLO.

## VII. Conclusion and Future Work

In this work, we have shown that the self-adaptive container library not only has the ability of automatically changing its underlying data structures but dynamically deploying third-party container frameworks and cloud storage. These frameworks enable our library to reduce the complexity of developing out-of-core storage and parallel software and to provide a broader class of Service Level Objectives. Further, the dynamic deployment of cloud storage supplies programmers with alternative out-of-core memory which can be utilised when local disks are not available. The efficacy has been

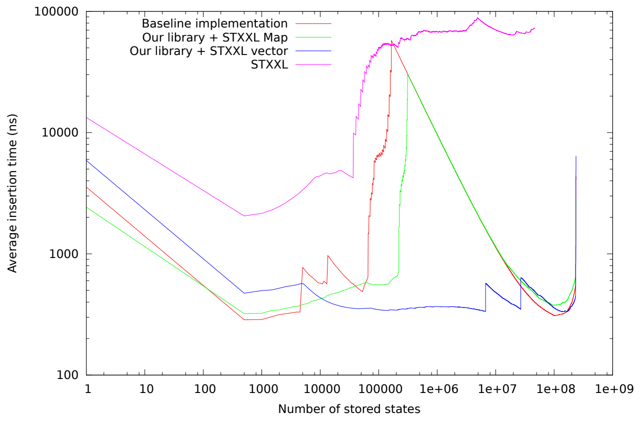

Figure 3. Average insertion time of different memory-sensitive container frameworks

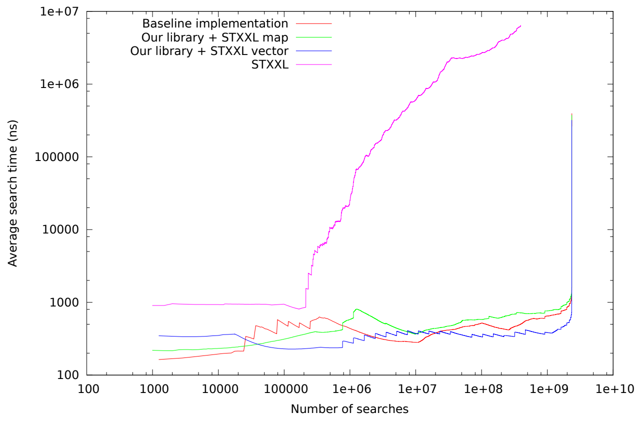

Figure 4. Average search time of different memory-sensitive container frameworks

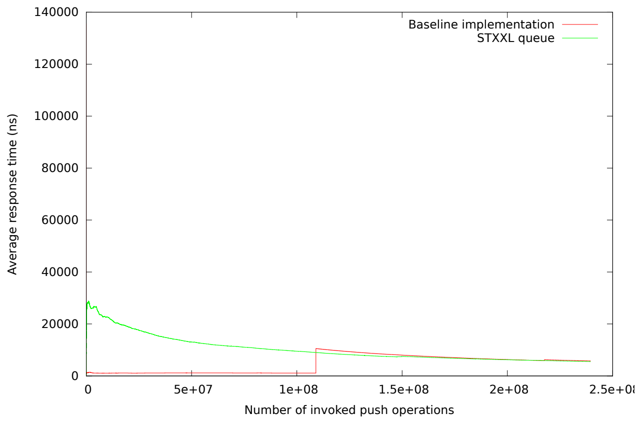

Figure 5. Average response time of push operations

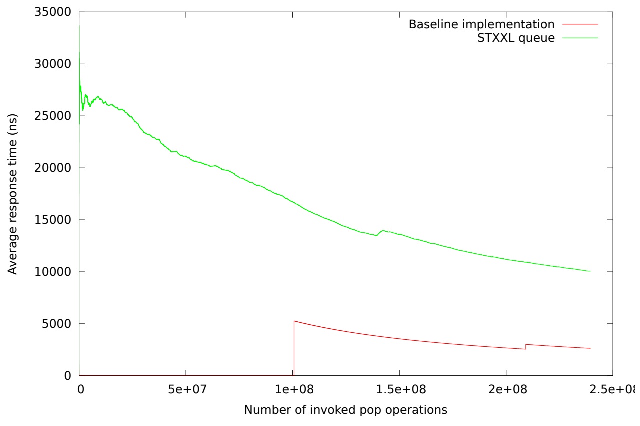

Figure 6. Average response time of pop operations

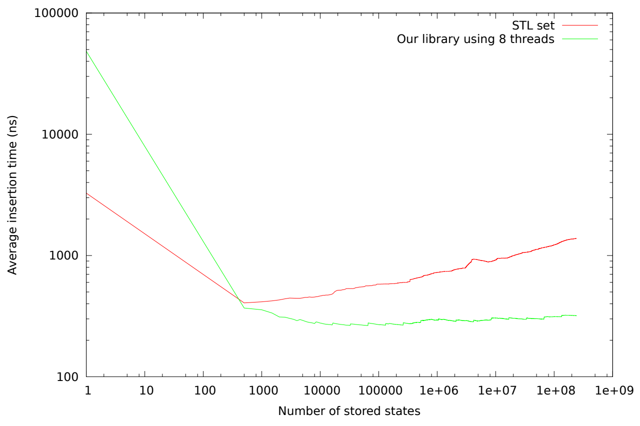

Figure 7. Average insertion time consumed by STL's set and our library adopting TBB

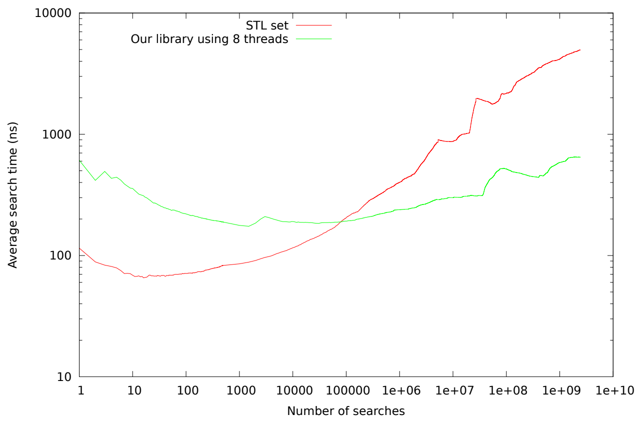

Figure 8. Average search time consumed by STL's set and our library adopting TBB

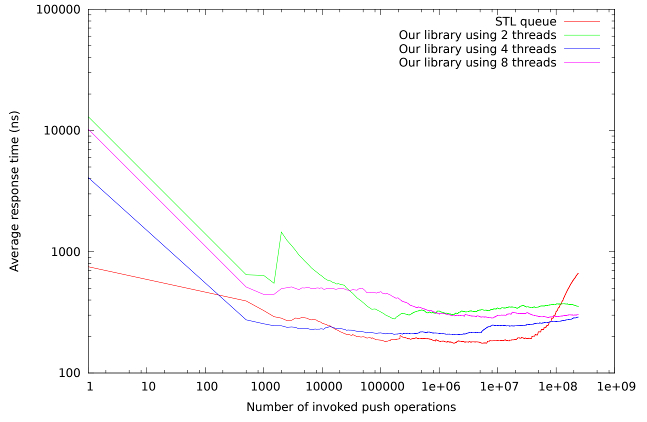

Figure 9. Average response time of invoked push operations expended by STL's queue and our library

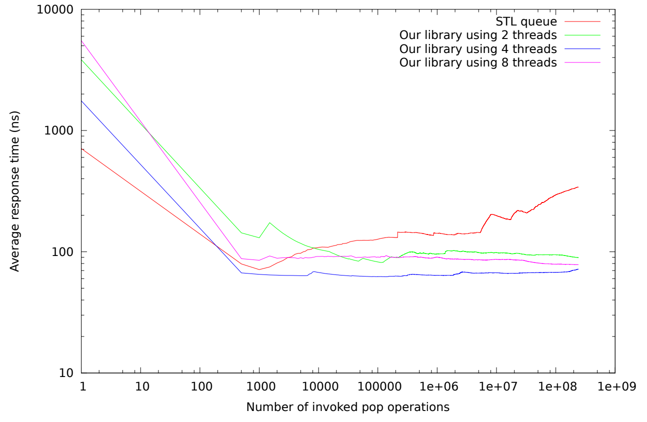

Figure 10. Average response time of invoked pop operations expended by STL's queue and our library

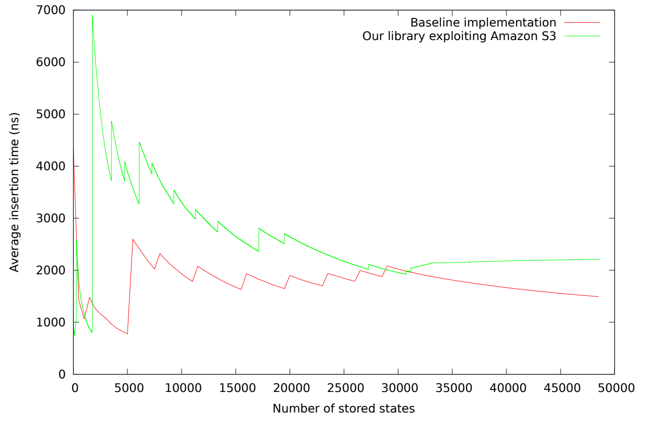

Figure 11. Average insertion time taken by our library exploiting the baseline implementation and Amazon S3

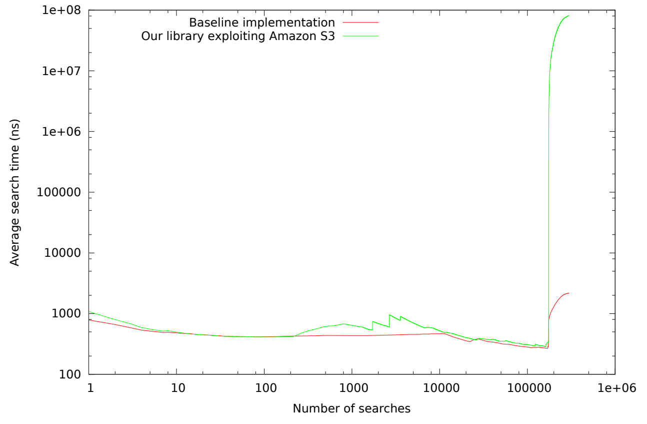

Figure 12. Average search time taken by our library exploiting the baseline implementation and Amazon S3

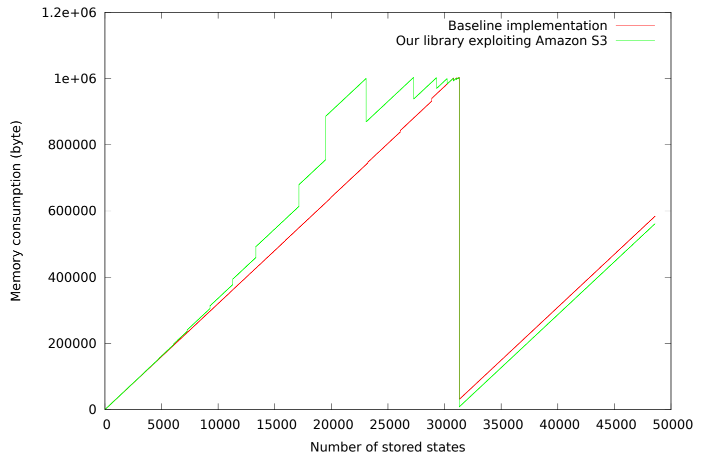

Figure 13. Memory usage consumed by our library exploiting the baseline implementation and Amazon S3

proven in a case study, which illustrates that how a naïve implementation of an algorithm utilising our library can easily become scalable, resource-efficient, and parallel. Simultaneously, the programmer overhead of migrating software from one execution environment to another is reduced to redefinition of Service Level Objectives which are suitable for the resource constraints of the new environment.

Currently, our library chooses either disks or cloud storage to store out-of-core data. In the future, a combination of disks and cloud storage will very likely be employed. This can be achieved by a mechanism which specifies different types of out-of-core storage. This will enable our self-adaptive mechanism to dynamically move data among primary memory, disks, and cloud. Further, because cloud storage service providers have different costs and features, these should also be taken into consideration when cloud storage is deployed.

## References

[1] D. Perez-Palacin, J. Merseguer, and R. Mirandola, "Analysis of bursty workload-aware self-adaptive systems," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE)*, Boston, USA, Apr. 2012, pp. 75–84.

[2] I. Boutsis and V. Kalogeraki, "Radar: Adaptive rate allocation in distributed stream processing systems under bursty workloads." in *Proceedings of the IEEE 31st Symposium on Reliable Distributed Systems (SRDS)*, Irvine, USA, Oct. 2012, pp. 285–290.

[3] A. Mili, R. Mili, and R. T. Mittermeir, "A survey of software reuse libraries," *Annals of Software Engineering*, vol. 5, no. 1, pp. 349–414, Jan. 1998.

[4] P. Clements and L. Northrop, *Software Product Lines: Practices and*

*Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[6] J. Sametinger, *Software Engineering with Reusable Components*. New York, NY, USA: Springer-Verlag New York, Inc., 1997.

[7] W.-C. Huang and W. J. Knottenbelt, "Self-adaptive containers: Building resource-efficient applications with low programmer overhead," in *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, San Francisco, USA, May 2013, pp. 123–132.

[8] A. Keller and H. Ludwig, "The WSLA framework: Specifying and monitoring service level agreements for web services," *Journal of Network and Systems Management*, vol. 11, no. 1, pp. 57–81, Mar. 2003.

[9] P. Isensee. C++ optimization strategies and techniques. [Online]. Available: http://www.tantalon.com/pete/cppopt/main.html

[10] S. Meyers, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Essex, UK, UK: Addison-Wesley Longman Ltd., 2001.

[11] A. B. Bondi, "Characteristics of scalability and their impact on performance," in *Proceedings of the 2nd International Workshop on Software and Performance*, Ottawa, Canada, Sep. 2000, pp. 195–203.

[12] N. M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, 2nd ed. Addison-Wesley Professional, 2012.

[13] W.-C. Huang and W. J. Knottenbelt, "Self-adaptive containers: Functionality extensions and further case study," in *Proceedings of the 6th International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE)*, Venice, Italy, May 2014, pp. 92–98.

[14] R. Dementiev, L. Kettner, and P. Sanders, "STXXL: Standard template library for xxl data sets," *Software- Practice & Experience*, vol. 38, no. 6, pp. 589–637, May 2008.

[15] D. E. Vengroff, "A transparent parallel i/o environment," in *Proceedings of the 3rd DAGS Symposium on Parallel Computation*, Hanover, Germany, Jul. 1994, pp. 117–134.

[16] A. Crauser and K. Mehlhorn, "LEDA-SM extending LEDA to secondary memory," in *Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE)*, London, UK, Jul. 1999, pp. 228–242.

[17] T. Gschwind, "PSTL: A c++ persistent standard template library," in *Proceedings of the 6th Conference on USENIX Conference on Object-Oriented Technologies and Systems*, San Antonio, USA, Feb. 2001, pp. 147–158.

[18] Intel. (2014) Threading building blocks. [Online]. Available: https://www.threadingbuildingblocks.org/

[19] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger, "STAPL: An adaptive, generic parallel c++ library," in *Proceedings of the 14th International Conference on Languages and Compilers for Parallel Computing*, Cumberland Falls, USA, Aug. 2001, pp. 193–208.

[20] E. Johnson and D. Gannon, "HPC++: Experiments with the parallel standard template library," in *Proceedings of the 11th International Conference on Supercomputing*, Vienna, Austria, Jul. 1997, pp. 124–131.

[21] Amazon. (2014) Amazon simple service storage. [Online]. Available: http://aws.amazon.com/s3/

[22] W. J. Collins, *Data Structures and the Java Collections Framework*. McGraw-Hill Higher Education, 2001.

[23] P. Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology," 2011, presented at AGENDA 2001, Socttsdale, Available via http://www.research.ibm.com/autonomic/.

[24] R. Murch, *Autonomic Computing*. IBM Press, 2004.

[25] M. Rohr, S. Giesecke, W. Hasselbring, M. Hiel, W.-J. van den Heuvel, and H. Weigand, "A classification scheme for self-adaptation research," in *Proceedings of the International Conference on Self-Organization and Autonomous Systems In Computing and Communications*, Erfurt, Germany, Sep. 2006, pp. 1–5.

[26] IBM Corp., *An architectural blueprint for autonomic computing*. IBM Corp., Oct. 2004.

[27] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[28] B. Chen, X. Peng, Y. Yu, B. Nuseibeh, and W. Zhao, "Self-adaptation through incremental generative model transformations at runtime," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, Hyderabad, India, May 2014, pp. 676–687.

[29] D. Garlan and B. Schmerl, "Model-based adaptation for self-healing systems," in *Proceedings of the First Workshop on Self-healing Systems*, Charleston, USA, Nov. 2002, pp. 27–32.

[30] S. H. Young, T. A. Mazzuchi, and S. Sarkani, "A model-based framework for predicting performance in self-adaptive systems," in *Proceedings of the 12th Annual Conference on Systems Engineering Research*, Los Angeles, USA, Mar. 2014, pp. 513–521.

[31] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, Oct. 2004.

[32] G. Tallabaci and V. E. S. Souza., "Engineering adaptation with Zanshin: an experience report," in *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, San Francisco, USA, May 2013, pp. 93–102.

[33] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004.

[34] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, Jyväskylä, Finland, Jun. 1997, pp. 220–242.

[35] R. Hirschfeld, P. Costanza, and O. Nierstrasz, "Context-oriented programming," *Journal of Object Technology*, vol. 7, no. 3, pp. 125–151, 2008.

[36] G. Salvaneschi, C. Ghezzi, and M. Pradella, "An analysis of language-level support for self-adaptive software," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 8, no. 2, pp. 7:1–7:29, Jul. 2013.

[37] B. Redmond and V. Cahill, "Supporting unanticipated dynamic adaptation of application behaviour," in *Proceedings of the 16th European Conference on Object-Oriented Programming*, Málaga, Spain, Jun. 2002, pp. 205–230.

[38] D. Suvée, W. Vanderperren, and V. Jonckers, "JAsCo: An aspect-oriented approach tailored for component based software development," in *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, Boston, USA, Mar. 2003, pp. 21–29.

[39] M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara, "ContextJ: Context-oriented programming with Java," *Journal of the Japan Society for Software Science and Technology on Computer Software*, vol. 28, no. 1, pp. 272–292, 2011.

[40] G. Salvaneschi, C. Ghezzi, and M. Pradella, "ContextErlang: Introducing context-oriented programming in the actor model," in *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, Potsdam, Germany, Mar. 2012, pp. 191–202.

[41] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A memory-efficient, high-performance key-value store," in *Proceedings of the 23th ACM Symposium on Operating Systems Principles*, Cascais, Portugal, Oct. 2011, pp. 1–13.

[42] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," in *Proceedings of SciDAC 2005, Journal of Physics: Conference Series*, vol. 16, Jun. 2005, pp. 521–530.

[43] J. Singler, P. Sanders, and F. Putze, "MCSTL: The multi-core standard template library," in *Proceedings of the 13th International Euro-Par Conference on Parallel Processing*, Rennes, France, Aug. 2007, pp. 682–694.

[44] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, Jan. 1998.

[45] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, "Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity," *Journal of Parallel and Distributed Computing*, vol. 27, no. 2, pp. 118–141, Jun. 1995.

[46] Google. (2014) Google cloud platform. [Online]. Available: http://cloud.google.com/

[47] Dropbox. (2014) Dropbox. [Online]. Available: https://www.dropbox.com/

[48] OblakSoft. (2014) WebStor: high-performance API for cloud storage. [Online]. Available: http://www.oblaksoft.com/downloads/

[49] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[50] W. J. Knottenbelt, "Parallel performance analysis of large Markov models," Ph.D. dissertation, Department of Computing, Imperial College of Science, Technology and Medicine. University of London., Dec. 1999.