Self-Adaptive Containers: Functionality Extensions and Further Case Study

Wei-Chih Huang and William J. Knottenbelt
Department of Computing, Imperial College London
{wei-chih.huang11, wjk}@imperial.ac.uk

Abstract—**As the number of execution environments and application contexts rises exponentially, ever-changing non-functional requirements can lead to repeated code refactoring. In addition, scaling up software to support large input sizes may require major modification of code. To address these challenges, we have previously proposed a framework of self-adaptive containers which can automatically adjust their resource usage to meet Service Level Objectives and dynamically deploy the techniques of out-of-core storage and probabilistic data structures. A prototype with limited functionalities was implemented and applied to explicit state space exploration to prove the viability of our framework. In this paper, we broaden the library's functionalities through support for the important container class of key-value stores and integration of priority queues' functionalities into our previously-developed container class. We then utilise them in a new case study centred on route planning, adopting Dijkstra's shortest path algorithm. For this, a graph representing the full USA road network, which contains approximately 24 million nodes and 58 million arcs, is input to the algorithm so as to find the shortest paths from a random node to all the other nodes. The experimental results have shown that, under particular Service Level Objectives our library reduces update time by 21.4%, primary memory usage of node storage by 85.3%, and primary memory consumption required by the priority queue by 78%, compared with the Standard Template Library.**

Keywords-Self-Adaptive Systems; Containers; Standard Template Library; Probabilistic Data Structures.

I. INTRODUCTION

Traditional software engineering methodologies are facing a new challenge – a rapidly growing number of system environments, where software is executed (e.g., tablets, servers, smartphones, laptops, routers). The applications may operate under different resource constraints and Quality of Service (QoS) requirements [1], based on the environments in which they are executed. Adapting software to each possible execution environment and application context in order to maintain QoS requirements is not a trivial job, especially in the situation where bursty and/or high-intensity workloads may frequently exhaust system resources [2] [3]. Further, this may take months or years of programmer effort to modify the majority of program code and may entail considerable programmer expertise [4][5]. These new challenges cannot be dealt with simply through use of traditional software engineering techniques [6–8], which results in either one of the two possible scenarios: a small code base which cannot guarantee QoS or multiple manually-optimised code bases which are difficult to maintain.

We have previously presented a framework of self-adaptive containers [9], which attempts to tackle the above-mentioned challenges via change of data structures. Instead of manually choosing a container and its underlying data structure, our self-adaptive containers provide two classes which automatically take such actions and dynamically change their underlying data structures in accordance with programmer-specified Service Level Objectives (SLOs) and the required functionalities. The former aims to easily satisfy ever-changing QoS requirements through modification of SLO specification, and the latter intends to provide a greater scope for efficiency optimisation. Conventional standardised container libraries are built for general purpose contexts, where all functionalities are always ready to be supplied, which restricts the possibility of optimisation. Through tighter functionality specification, our containers are able to exploit the techniques which can only be utilised when certain functionalities are entailed, including out-of-core storage and probabilistic data structures. To illustrate the viability of our framework, a prototype that fulfilled partial functionalities, highlighted in yellow in Figure 1, was implemented and utilised by the breadth first search algorithm, which explored up to 240 million states. The experimental results showed that our containers could not only dynamically boost their performance but save substantial memory space. Further, the containers' behaviour varied according to assigned SLOs, indicating that the self-adaptive containers could easily adapt to different execution environments.

Our framework with limited functionalities has been implemented to prove that it is feasible. In this paper, we add support for key-value stores (`IKeyValue`) and priority queues into our previously-built container class (`ICollection`). As described more fully in Sections IV and V, both of these data structures are widely-used in industry and are fundamental to many core computer science algorithms. `IKeyValue` supports commonly used member functions such as insert and the direct access operator. The functionalities of priority queues are supported by `ICollection`, which provides the required member functions and automatic deployment of out-of-core storage. The instances of either `ICollection` or `IKeyValue` can be assigned SLOs specified in the standard Web Service Level Agreement (WSLA) [10] format, which allows programmers to clearly and easily define resource constraints and QoS requirements. When currently-consumed resources violate the SLOs, our library's self-adaptive mechanism will determine if an adaptation action is needed in order to either satisfy the violated SLOs or reduce the degree to which the SLOs are contravened.

Our library is applied to a new case study, route planning, which adopts Dijkstra's algorithm, in order to show its enhanced functionalities. The experimental results suggest that our containers can effortlessly be adopted and considerably enhance both performance and memory efficiency. They also illustrate that the containers are capable of responding to programmer-specified SLOs.

This paper yields the following contributions:

- The functionalities of our self-adaptive container library are broadened through support for the fundamental container class of key-value stores and implementation of the functionalities of priority queues in our previously-developed container class, which expands application areas where our library may be applied.
- A new case study is investigated to illustrate our library's applicability. It shows how a naïve implementation of a core computer science algorithm in combination with our library can become resource-efficient and can achieve different programmer-specified Service Level Objectives.

The remainder of this paper is organised as follows. Section II introduces self-adaptive systems, reference models for building such systems and resource-aware systems. Section III describes our library's architecture and self-adaptive mechanism. While Section IV presents the design and implementation of key-value stores, Section V describes the out-of-core priority queue's implementation. Section VI presents the case study. Section VII concludes this paper and points out possibilities of future work.

II. Self-adaptive Systems, Reference Models, and Resource-aware systems

The foundation of the research regarding modern self-adaptive systems arose in the late 1990s and early 2000s, when IBM coined the term of autonomic computing [11], derived from human autonomic nervous systems, which could unconsciously control human bodies (e.g., heart rate, salivation, perspiration). A system adopting autonomic computing should involve the properties of self-configuration, self-healing, self-optimisation, and self-protection. To be equipped with these properties, a system should contain a self-adaptive cycle composed of an observation phase, an analysis phase, and an adaptation phase [12]. The observation phase is responsible for monitoring and collecting required data. The analysis phase determines if an adaptation action should be taken in accordance with the data reported from the observation phase and chooses a suitable adaptation action. The adaptation phase performes the adaptation action selected in the analysis phase.

After autonomic computing is introduced, a reference model for building self-adaptive systems, MAPE-K (monitor, analyse, plan, execute, and knowledge), is put forward [13] and implemented in several projects [14–16]. MAPE-K contains a cycle formed by the five functions, which are used to observe and collect data from managed resources, analyse the data, plan an adaptation action, perform the adaptation action to adjust managed resources, and store managed resources' goals, respectively. Garlan et al. [17] present another framework, Rainbow, which utilises an external approach for building self-adaptive systems.

To adapt to execution environments with different resource constraints, software should have the ability to detect its current resource usage. Sumatra [18], introduced by Acharya et al., is a Java extension, which provids four programming abstractions for monitoring resources and building resource-aware programs. In their work, they also suggest the awareness requirement, the agility requirement, and the authority requirement should be satisfied in the context of mobile agent software. However, the programmer overhead is relatively high in terms of adapting existing code to a tightly-specified mobile software architecture.

Among approaches automatically changing data structures to save resources is SILT [19] , which 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. Indeed, there is no mechanism for sepcifying any Service Level Objectives, which leads to difficulties in adapting software to each execution environment and application context.

III. Library Architecture and Self-adaptive Mechanism

This section will briefly introduce our library's framework and self-adaptive mechanism. For full details, please refer to our previous publication [9]. As can be seen in Figure 1, the library consists of two major components, Application Programming Interface (API) and Self-Adaptive Unit (SAU). The API provides programmers with two template classes covering most functionalities of the Standard Template Library (STL) [20]: `ICollection`, which has been partially implemented in our previous prototype, and `IKeyValue`, supporting key-value stores. The member functions of `ICollection` and `IKeyValue` can be divided into operation interfaces and configuration interfaces. The former is a group of commonly-used operations. The latter acts as the means through which functionality requirements, SLOs, and the frequency with which the SLO compliance should be checked are imparted to the library.

The SAU, which performs operations and manages the self-adaptive mechanism, is composed of an Execution unit, a 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 times, computes memory consumption, and calculates reliability when a probabilistic data structure is exploited. These operation profiles are then reported to the Analyzer, which determines whether an adaptation action should be taken. If an adaptation action is required, the Adaptor will be invoked to perform an adaptation action.

The self-adaptive mechanism of our library is a classical self-adaptive cycle [12], 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 times, 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 (e.g., the subdivision of the underlying data structure, the activation of out-of-core technique, or the deployment of probabilistic data structures) 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 and (b) the adaptation action is not expected to violate a currently-satisfied SLO of higher priority. We design these rules for two reasons. First, it may not be possible to meet all (or any) of the SLOs within resource constraints. Second, an adaptation action taken to address one violated SLO may cause the violation of another SLO. To solve these issues, 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 an adaptation action.

IV. Key-value Stores Design and Implementation

Key-value stores, which represent data stored in pairs of keys and values, have been adopted in many industries managing large-scale data (e.g., Amazon [21], Facebook [22], Twitter [23][24], Linkedin [25]). As stored data accumulate, relational databases, which offer general purpose data stores, are incapable of providing acceptable data manipulation time. Many programming language data structures (e.g., map of the STL, HashMap of Java, and the dictionary data type in Python) and libraries (sparkey [26], LevelDB [27], YDB [28]) can be used to implement key-value stores and of course, recent years have seen the rise of persistent counterparts in the form of NoSQL databases (e.g., Cassandra [29], Riak [30], Tokyo Cabinet [31], Aerospike [32]). Our library supports the functionalities of key-value stores in `IKeyValue`, which chooses either a tree data structure, e.g. red black tree or AVL (Adelson-Velskii and Landis) tree or, where appropriate, a sparse Bloom filter [33], which transforms larger-sized elements into smaller-sized keys via hashing techniques to save considerable memory space, as the underlying data structure. As can be seen in Figure 1, `IKeyValue` API contains configuration interfaces and operation interfaces. The operation interfaces support the member functions which are needed to manipulate key-value stores, and the configuration interfaces including the constructor and `setAdaptationFrequency` are used for management purposes. The usage of `IKeyValue`'s constructor is illustrated as follows:

`IKeyValue<`K, V`>` (*op_desc*, *SLO_file*[, *freq*])

where *op_desc* specifies the required set of functionalities (so-called operation descriptors), *SLO_file* shows a path to an XML file containing a description of SLOs in WSLA format, and *freq* is an optional parameter defining the frequency with which the self-adaptive mechanism is activated.

`IKeyValue` is also capable of dynamically and automatically adjusting its underlying data structure through the SAU in order to meet SLOs. If its performance has to be improved, the underlying data structure will be subdivided. For example, when a sparse Bloom filter, which utilises a forest of AVL trees, is selected as the currently-used data structure, the number of AVL trees is increased to reduce the number of comparisons. If the reliability of `IKeyValue` needs to be increased, its underlying data structure will be subdivided as well. When the consumed memory space exceeds resource constraints, out-of-core storage may be activated. However, when the activation commences, some of the stored elements may be allocated to external memory, which makes the direct access operator (i.e., *operator[]*) unable to return a reference to the mapped value. To solve this issue, `IKeyValue`'s direct access operator will return a reference to a proxy class, which overloads the assignment operator (i.e., *operator=*) and the cast operator (i.e., *operator()*) to satisfy the functionalities of assignment and retrieval, respectively.

V. Out-of-core Priority Queue

Priority queues, whose underlying data structures are heaps, provide push operations as well as pop and top operations, which manipulate the largest (or smallest) element. As the number of stored elements increases, primary memory may be unable to store new elements, which leads to the utilisation of external memory. Because the performance of external memory is orders of magnitude slower than that of internal memory, out-of-core priority queues require I/O efficient algorithms [34–36]. In our library, the functionalities of priority queues specified by operation descriptors are embedded in `ICollection`, which now accepts a custom comparison operator as an optional template parameter, which defaults to less-than operator, to decide that either the largest or smallest element should be manipulated. When the internal memory limit has not been reached, `ICollection` behaves like the STL's *priority_queue*. Once the self-adaptive mechanism computes the memory consumption and sees it exceeds the primary memory limit, the mechanism will then take the following actions. First, it will sort the priority queue in primary memory and move the sorted elements to external memory. Next, the priority queue's largest (or smallest) element is inserted into a max (or min) heap which is intended to reduce response times of pop and top operations. These actions may be performed many times to keep memory consumption lower than the primary memory limit. When out-of-core storage is activated, pop and top operations should access not only the priority queue in internal memory but the root of the heap. If the root of the heap has to be removed, it will be deleted and the next larger (or smaller) element is then inserted into the heap.

Some researchers have proposed implementations of out-of-core priority queues (e.g., Standard Template Library for Extra Large Data Sets [37]), which considerably enhance I/O efficiency. However, they cannot dynamically determine when to trigger out-of-core storage, which deteriorates the performance of priority queues when in-core memory is sufficient. We have seen this drawback and aimed for our library to act as a controller which decides when to make use of out-of-core priority queues.

VI. Case Study

The case study chosen to illustrate our library's applicability is Dijkstra's shortest path algorithm, which has been extensively applied to route planning [38] and social network

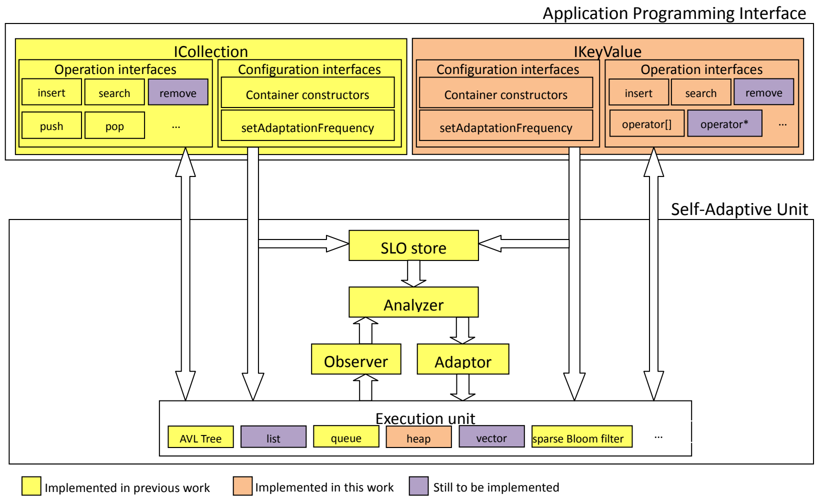

Figure 1. The highlight components of the library

analysis [39]. A naïve implementation of this algorithm is shown in Figure 2. Figure 3 displays the same algorithm via use of our library. As can be seen, the only difference between the two programs is the declaration of the key-value store variable, `Distance` (which stores the shortest distances from a given random node to all the other nodes), and of priority queue variable, `PQ` (which is used to locate the node with the shortest distance). To observe the library's behaviour under different SLOs, the following SLOs were assigned to `Distance`:

1) 80% of insertion times should be less than 1350 ns, and 90% of search times should be less than 500 ns.
2) Reliability should be higher than 0.995.
3) Memory use should be no more than 500 MB.

The above-mentioned SLOs were stored in *DistanceSLOs.xml* in WSLA format. An example of how to express SLOs fit for our self-adaptive containers in WSLA format is shown in [9]. Similarly, an SLO that requires the primary memory consumption of `PQ` to remain below 300 KB was assigned to *PQSLO.xml*. For `Distance`, the value of the optional parameter, `AdaptationFrequency`, was 100. In other words, the Analyzer was triggered every 100 operations. Naturally, the value of `AdaptationFrequency` may affect response times. In our previous case study, we have assigned different values to see the influence, which shows that when values of `AdaptationFrequency` are small, response times are lessened on account of a reduction in the Analyzer's activation times. As values rise, response times begin to increase due to delay of adaptation actions.

In this case study, a graph depicting the full USA road network [40], which contains approximately 23 million vertices and 58 million edges, was input. The performance and memory consumption from a given random node to all the others were compared, using the STL's class and our library. The SLOs of `Distance` were then input to different sequences to observe the library's behaviour. Finally, the memory consumed by both the STL's *priority_queue* and our library were displayed, showing improvement of memory efficiency.

A. *Comparison with STL's map*

To evaluate our library's effectiveness, Dijkstra's shortest path algorithm utilised to compute the shortest paths from a random node to all the other nodes was executed using the STL's map and our library. Figures 4 and 5 display average insertion and update times for `Distance`. The insertion time consumed by our library was close to that consumed by the STL's map. That was because our library spent extra time performing adaptation actions when elements were inserted. The sudden rises in insertion times indicated that our library changed its underlying data structures to boost performance or reliability. Although adaptation actions initially added to insertion times, they considerably improve scalability going forward. Indeed, insertion time and update time SLO are both subsequently maintained with only occasional adaptations.

Figure 6 depicts the memory space consumed by the STL's map and our library. Our library used an order of magnitude less memory space than the STL's map.

B. *Influence of SLO priority*

Figures 7 and 8 illustrate the performance-related SLOs and the time spent by our library under different priority orderings. For example, PerRelMem means that performance is the highest in order of priority, reliability has the next highest priority, and memory consumption's priority is the lowest. These figures indicate that when the given SLOs specified performance has higher priority over memory consumption, our library expends considerably less insertion time and update time. This phenomenon can be seen in the following orders of SLO metrics: PerMemRel, PerRelMem, and RelPerMem. When their performances cannot achieve the performance-related SLOs, the library still improves performance even if it means violating the memory limit. By contrast, when the memory-related SLO is the highest in order of priority, it causes our library to consume substantial insertion time and update time due to frequent out-of-core memory access.

The library's memory consumption conditions under the six priority sequences are depicted in Figure 9, which shows two different types of behaviour in accordance with priority in

```
void Dijkstra_algorithm(Graph G, Node s)
{
    priority_queue< pair<Node, double>,    compare > PQ;
    map<Node, double> Distance;
    Node u, v ;
    double cost;


    for (Node *w = G.start_node() ; w != G.end_node() ; w = G.next_node()) {
        Distance.insert(pair<Node, double>(*w, numeric_limits<double>::infinity()));
    }

    Distance[s] = 0;
    PQ.push(pair<Node, double>(s, Distance[s]));

    while (!PQ.empty()) {
        u = PQ.top().first;
        PQ.pop();
        pair<Node, double> *z = G.first_edge(u);

        for (; z ; z = G.next_edge(u)) {
            v = (*z).first ;
            cost = (*z).second;
            if (Distance[v] > Distance[u]+cost) {
                Distance[v] = Distance[u] + cost ;
                PQ.push(pair<Node, double>(v, Distance[v]));
            }
        }
    }
}
```

Figure 2. The naïve shortest-path algorithm

```
void Dijkstra_algorithm(Graph G, Node s)
{
    ICollection< pair<Node, double>,    compare > PQ(OP_PQUEUE, "PQSLO.xml");
    IKeyValue<Node, double> Distance(OP_INSERT|OP_INDEX, "DistanceSLO.xml", 100);
    Node u, v ;
    double cost;


    for (Node *w = G.start_node() ; w != G.end_node() ; w = G.next_node()) {
        Distance.insert(pair<Node, double>(*w, numeric_limits<double>::infinity()));
    }

    Distance[s] = 0;
    PQ.push(pair<Node, double>(s, Distance[s]));

    while (!PQ.empty()) {
        u = PQ.top().first;
        PQ.pop();
        pair<Node, double> *z = G.first_edge(u);

        for (; z ; z = G.next_edge(u)) {
            v = (*z).first ;
            cost = (*z).second;
            if (Distance[v] > Distance[u]+cost) {
                Distance[v] = Distance[u] + cost ;
                PQ.push(pair<Node, double>(v, Distance[v]));
            }
        }
    }
}
```

Figure 3. The resource-aware algorithm using self-adaptive containers

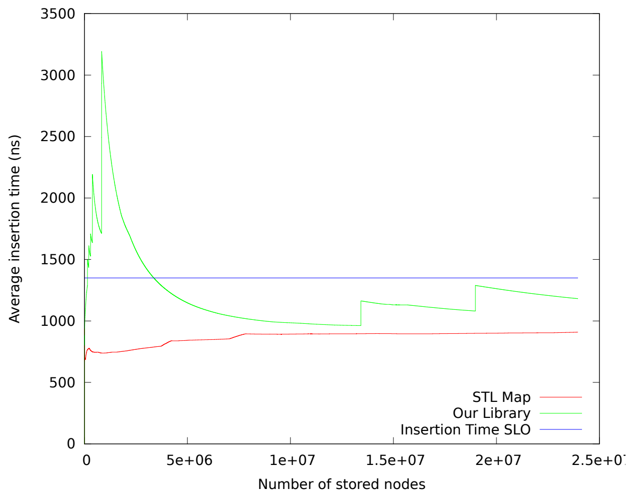

Figure 4. Average insertion time

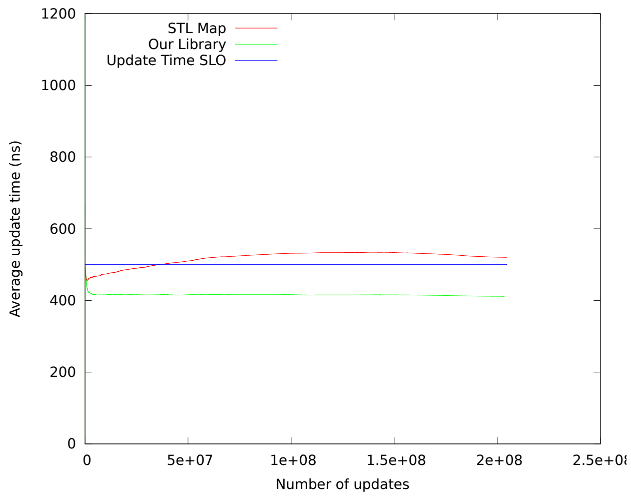

Figure 5. Average update time

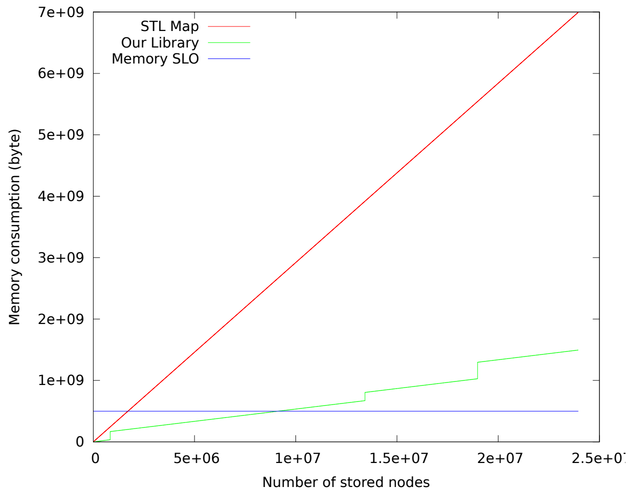

Figure 6. Memory consumption

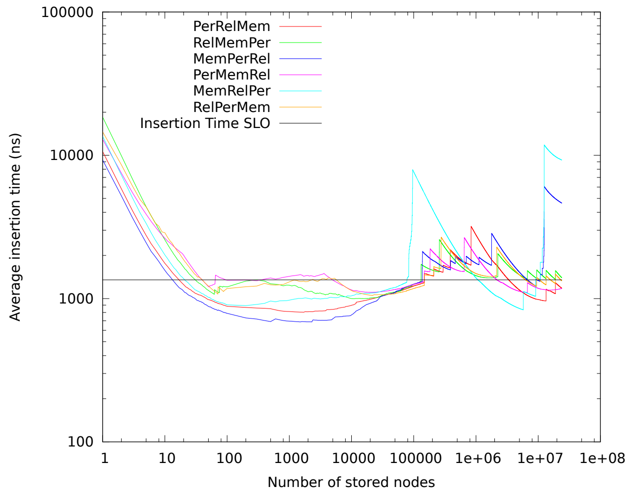

Figure 7. Average insertion times under different SLO priorities

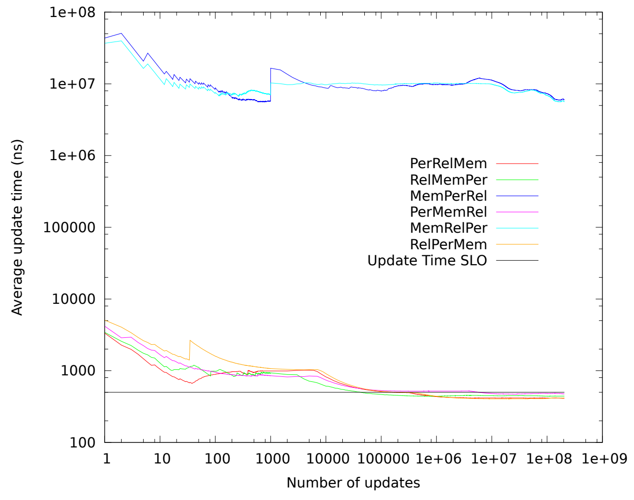

Figure 8. Average update times under different SLO priorities

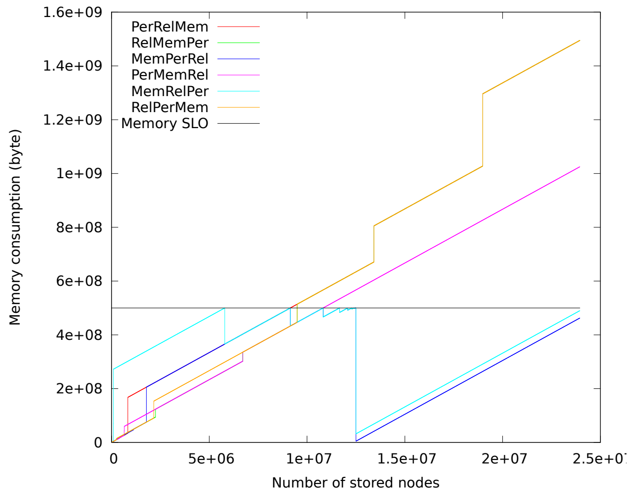

Figure 9. Memory consumption under different SLO priorities

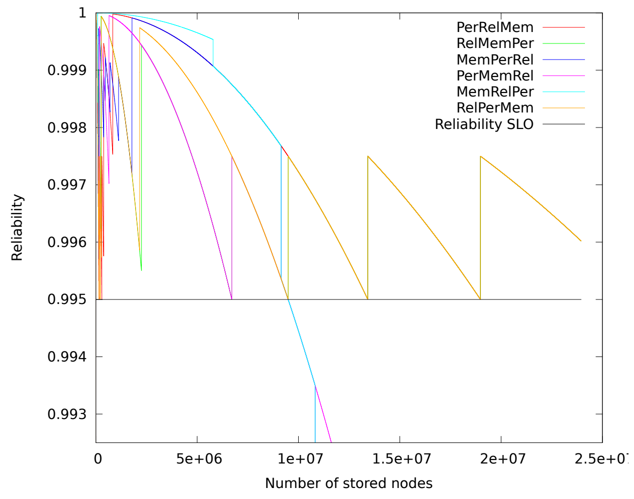

Figure 10. Reliability under different SLO priorities

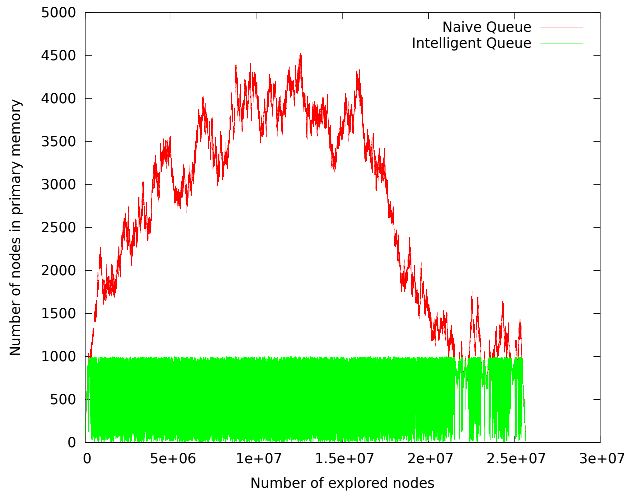

Figure 11. Memory consumptions of the naïve priority queue and the intelligent queue

memory consumption. When memory consumption is lowest in order of priority, more memory space is consumed to enhance performance or reliability. By contrast, when memory consumption has the highest priority (MemPerRel and MemRelPer), the consumed memory space is the least. This figure also indicates that when MemPerRel and MemRelPer reach the memory limit, our library, whose currently-used data structure is an improved sparse Bloom filter, reduces the number of AVL trees to save memory space. Once the number of AVL trees cannot be reduced, out-of-core storage is activated.

The variation in our library's reliability is depicted in Figure 10. As can be seen, when reliability has the highest priority (RelPerMem and RelMemPer), the reliability is kept at a desirable level – over 0.995. According to the two rules of our self-adaptive mechanism, when reliability is the highest in order of priority, it can be boosted without consideration of the side effects – most notably the increase in memory consumption. As a result, RelPerMem and RelMemPer rebound several times when reliability is equal to or lower than 0.995. But when reliability is lower in order of priority, the library's reliability descends as the number of inserted elements increases. Take MemRelPer for example. Memory consumption has higher priority than reliability, which implies that reliability cannot be improved once the memory limit is reached. Further, the reliability sharply deteriorates after adaptation actions which reduce the number of AVL trees are taken. While PerRelMem keeps reliability over 0.995, PerMemRel does not enhance reliability when the number of inserted elements is approximately one million. That is because for PerMemRel's memory consumption has higher priority than reliability. Consequently, when the current reliability is lower than the desired reliability, PerMemRel does not enhance reliability so as to prevent consumed memory exceeding the memory quota.

C. *Out-of-core Storage for Priority Queue*

As mentioned, the memory limit of the variable, `PQ`, was 300 KB. The primary memory consumptions using our library and the STL's *priority_queue* are shown in Figure 11. It indicates that our library consumed 300 KB, which was a mere 78% of the memory space consumed by the STL's *priority_queue*.

VII. Conclusion and Future Work

In this paper, we have broadened the functionalities of our self-adaptive container library, which now supports the fundamental container class of key-value stores and enhances our previously-developed container class with the functionalities of priority queues. The new functionalities can dynamically exploit probabilistic data structures and out-of-core storage in an effort to meet different QoS requirements. Their efficacy has been proven in a case study, which illustrates how a naïve implementation of an algorithm utilising our library becomes scalable and resource-efficient by swift library-driven adaptations which successfully maintain programmer-specified Service Level Objectives in order of priority. Simultaneously, programmer overhead is kept low in terms of adapting software to a new environment. This can be easily achieved by means of redefining SLOs which are suitable for the resource constraints of a new environment.

So far, the library's self-adaptive mechanism follows a strict order of priority, which can be further extended to multi-objective optimisation methods such as a weighted product or a weighted sum of multiple SLOs. Another future direction of the library entails the cooperation with other container frameworks. Integrating them into our library can increase flexibility of the library in terms of its ability to meet SLOs in resource-constraint environments.

References

[1] A. Hervieu, B. Baudry, and A. Gotlieb, "Managing execution environment variability during software testing: an industrial experience," in Proceedings of the International Conference on Testing Software and Systems (ICTSS), 2012, pp. 24–38.

[2] 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, ser. ICPE '12, 2012, pp. 75–84.

[3] I. Boutsis and V. Kalogeraki, "Radar: Adaptive rate allocation in distributed stream processing systems under bursty workloads," in SRDS, October 2012, pp. 285–290.

[4] R. Weiss, K. Krogmann, Z. Durdik, J. Stammel, B. Klatt, and H. Koziolek, "Sustainability guidelines for long-living software systems," in Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM), ser. ICSM '12, September 2012, pp. 517–526.

[5] J. Greenfield and K. Short, Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools. John Wiley and Sons, 2002.

[6] A. Mili, R. Mili, and R. Mittermeir, "A survey of software reuse libraries," Annals Software Eng., vol. 5, 1998, pp. 349–414.

[7] P. Clements and L. Northrop, Software Product Lines: Practices and Patterns. Addison-Wesley, 2002.

[8] E. Gamma, R. Helm, J. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[9] 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, 2013, pp. 123–132.

[10] 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, 2003, pp. 57–81.

[11] 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/.

[12] M. Rohr et al., "A classification scheme for self-adaptation research," in Proc. International Conference on Self-Organization and Autonomous Systems In Computing and Communications (SOAS'2006), September 2006, p. 5.

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

[14] IBM. Autonomic computing toolkit. [Online]. Available: http://www.ibm.com/developerworks/autonomic/r3/overview.html [retrieved: April, 2005]

[15] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills, and Y. Diao, "Able: A toolkit for building multiagent autonomic systems," IBM Syst. J., vol. 41, no. 3, Jul. 2002, pp. 350–371.

[16] G. E. Kaiser, J. J. Parekh, P. Gross, and G. Valetto, "Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems," in Active Middleware Services, 2003, pp. 22–31.

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

[18] A. Acharya, M. Ranganathan, and J. Saltz, "Sumatra: A language for resource-aware mobile programs," in Mobile Object Systems. Springer-Verlag, 1997, pp. 111–130.

[19] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, ser. SOSP '11, 2011, pp. 1–13.

[20] D. R. Musser, G. J. Derge, and A. Saini, STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library. Boston, Mass. Addison-Wesley, 2001.

[21] G. DeCandia et al., "Dynamo: Amazon's highly available key-value store," in Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, 2007, pp. 205–220.

[22] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, 2012, pp. 53–64.

[23] B. Fitzpatrick, "Distributed caching with memcached," Linux J., no. 124, Aug. 2004, p. 5.

[24] J. Petrovic, "Using memcached for data distribution in industrial environment." in ICONS, 2008, pp. 368–372.

[25] Linkedin. Project Voldemort. [Online]. Available: http://www.project-voldemort.com/voldemort/ [retrieved: January, 2014]

[26] M. Bruggmann. Sparkey. [Online]. Available: https://github.com/spotify/sparkey-java [retrieved: March, 2014]

[27] Google. leveldb. [Online]. Available: http://code.google.com/p/leveldb/ [retrieved: December, 2013]

[28] M. Majkowski. Ydb. [Online]. Available: http://code.google.com/p/ydb/ [retrieved: October, 2010]

[29] Cassandra. Apache Cassandra. [Online]. Available: http://cassandra.apache.org/ [retrieved: February, 2014]

[30] Basho. Riak. [Online]. Available: http://basho.com/riak/ [retrieved: February, 2014]

[31] F. Labs. Tokyo Cabinet. [Online]. Available: http://fallabs.com/tokyocabinet/ [retrieved: August, 2012]

[32] Aerospike. Aerospike. [Online]. Available: http://www.aerospike.com/ [retrieved: February, 2014]

[33] W. Knottenbelt, "Performance analysis of large Markov models," Ph.D. dissertation, Imperial College of Science, Technology and Medicine, February 2000.

[34] Chiang et al., "External-memory graph algorithms," in Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 1995, pp. 139–149.

[35] U. Meyer, P. Sanders, and J. F. Sibeyn, Eds., Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002], ser. Lecture Notes in Computer Science, vol. 2625. Springer, 2003.

[36] N. R. Zeh, "I/O-efficient algorithms for shortest path related problems," Ph.D. dissertation, Carleton University, April 2002.

[37] R. Dementiev, L. Kettner, and P. Sanders, "STXXL: Standard Template Library for XXL data sets," Software: Practice and Experience, Aug 2007.

[38] D. Delling, P. Sanders, D. Schultes, and D. Wagner, "Engineering route planning algorithms," in Algorithmics of Large and Complex Networks. Lecture Notes in Computer Science. Springer, 2009.

[39] U. Brandes, "A faster algorithm for betweenness centrality," Journal of Mathematical Sociology, vol. 25, 2001, pp. 163–177.

[40] 9th DIMACS Implementation Challenge. Shortest paths. [Online]. Available: http://www.dis.uniroma1.it/challenge9/download.shtml [retrieved: June, 2010]