Self-Adaptive Containers: Building Resource-Efficient Applications with Low Programmer Overhead

Wei-Chih Huang and William J. Knottenbelt
Department of Computing, Imperial College London,
South Kensington Campus, London, SW7 2AZ, United Kingdom
{wei-chih.huang11, wjk}@imperial.ac.uk

***Abstract*—Despite advances in operating system resource management and the availability of standardised container libraries, developing scalable high-capacity applications remains a non-trivial endeavour. Naïve implementations of fundamental algorithms often rapidly exhaust system resources under heavy load. Resolving this via manual refactoring is usually possible but requires significant programmer effort, an effort which often has to be repeated in order to meet the resource constraints encountered in each different execution environment.**

This paper proposes a library of self-adaptive containers which provide a ready route to developing scalable applications with low programmer overhead. Given an execution environment, the library flexibly adapts its use of data structures in an effort to meet programmer-specified service level objectives. The library features a mechanism for tighter functionality specification than that provided by standard container libraries. This enables greater scope for efficiency optimisations, including the exploitation of probabilistic data structures and out-of-core storage.

We have demonstrated the capabilities of the proposed library through a prototype implementation in C++. We show that when a Breadth First Search explicit state space exploration algorithm is executed, using the proposed library reduces insertion time by 68.5%, search time by 86.1%, and primary memory usage by 90.1% compared with the Standard Template Library.

***Index Terms*—Self-Adaptive Systems, Containers, Standard Template Library, Probabilistic Data Structures**

I. Introduction

The variety of system environments in which software may be expected to execute is exploding (e.g. servers, laptops, tablets, smartphones). Adapting software to the different resource constraints and performability (performance and dependability [1]) requirements of software executing in each environment is a high-overhead operation requiring substantial programmer expertise and effort. Further, scaling software to be able to support large input sizes can require months or years of programmer time and frequently results in the refactoring of the majority of program code. By contrast, the idea of the present paper is to propose a framework for "intelligent" software which adapts at run-time to the resource constraints of its environment, as well as automatically scaling up to handle large input sizes while respecting performability requirements. The method of developing such software should be as close to that of developing ordinary software as possible, in order to reduce required expertise and programmer effort.

A key motivation for our work is the observation that similar techniques end up being adopted across apparently very different application domains in order to boost software performance and scalability. For example, consider the problem of explicit state space exploration. This is a key preliminary step in the model checking or performance analysis of concurrent systems and is based around a breadth-first search core. Over three decades, the gradual adoption of probabilistic data structures and algorithms [2–7], out-of-core storage [6, 8–11] and parallelism [6, 7, 12–16] have driven supported capacities from $\sim 10^5$ states to $\sim 10^{10}$ states [6]. A fundamental problem in the field of bioinformatics which has seen a similar evolution over the last 20 years or so is that of assembling genomic sequences from a set of overlapping reads. First proposed in 1995, de Bruijn graphs [17] were incorporated into the Euler [18] and Velvet [19] DNA assembly software packages, respectively. However, their memory requirements prohibit their use with large genomes. Subsequent adoption of parallelism [20–23], out-of-core methods [22, 24, 25] and very recently probabilistic techniques [26–28] have seen commodity computers increase their practical assembly capability from organisms with $\sim 10^6$ base pairs (e.g. simple viruses) to organisms with $\sim 10^9$ base pairs (e.g. humans), the latter requiring just 5.7GB RAM and around 23 hours wall clock time. Deeper investigation into the adopted techniques across both domains reveals a lot of commonality e.g. the probabilistic improvements universally involve the adoption of some variant of a Bloom filter [15, 26–28].

In addition, while most past research concerning software reuse has addressed development time [29–31], it has been observed that run-time issues warrant attention [32]. Many applications have seen the need to equip software with the ability to self-reconfigure at execution time [33, 34]. This need to self-adapt may complicate the development process because programmers have to adjust applications to meet different service level objectives (SLOs), which may in turn depend on both the execution environment and the class of application.

In this paper, we present a container library that allows programmers to build resource-aware software that adjusts according to its execution environment. Each data structure is associated with desired SLOs specified in the standard WSLA [35] format. These can specify desired per operation response

time, reliability and maximum primary memory consumption for each container, which then adapts autonomously at run time according to its current capacity and required functionality. Where appropriate, the library deploys probabilistic data structures in order to dramatically reduce memory consumption and boost performance. It also stores data in out-of-core memory (secondary memory) when the in-core memory (RAM) limit of the container is reached.

To encourage adoption of our library, and to lower programmer overhead, we have designed our library to provide similar functionality as is provided by existing container libraries such as the C++ Standard Template Library (STL) [36], albeit with a reduced number of template classes, most notably the `ICollection` and `IKeyValue` classes. As will be shown in the case study, this allows straightforward adaptation of existing code.

This paper yields the following three contributions:

- We propose a resource-aware container library which dynamically adjusts the underlying data structure associated with each container at execution time in order to satisfy resource constraints and SLOs specified in WSLA.
- Our library makes use of probabilistic data structures where appropriate and dynamically transfers data onto out-of-core memory when the in-core memory limit of a container is reached.
- By including the major member functions associated with containers, the library provides a straightforward way to adapt naïve implementations of fundamental algorithms so they they become scalable and suitable for deployment in multiple system environments.

The remainder of this paper is organised as follows. Section II presents relevant background and related work. Section III describes the detailed design of our library while Section IV illustrates its use of probabilistic data structures. Experimental results are shown in Section V. Finally, the conclusion and future work are summarised in Section VI.

II. Background and Related Work

A. Autonomic Computing

Autonomic computing, introduced by IBM in 2001 [37], refers to a computing model which enables any system adopting it to self-manage through an adaptive process. The process [38] consists of three steps: observation, analysis, and adaptation. Firstly, the observation step is responsible for monitoring system status and recording required information such as response time and memory consumption. After that, the analysis step carries out the tasks of analysing data and selecting operations – only if any constraint is violated. If none of the constraints is broken, the process returns to the observation step; otherwise, the analysis step is followed by the adaptation step, where the selected operations are executed. When the final step of adaptation is completed, the whole process is repeated.

B. Containers

Containers are collections of objects of the same data type. Because they are usually implemented as template classes, they can store a variety of built-in and user-defined data types. There are three categories of STL containers, namely sequence containers, associative containers, and container adaptors [36]. The sequence containers, consisting of `deque` (double-ended queue), `list`, and `vector`, maintain an ordered set of elements, and allow developers to specify the position of an inserted element. The associative containers, comprised by `set`, `multiset`, `map`, and `multimap`, maintain inserted elements in some predefined order. Container adaptors are interfaces which provide functionalities on top of an underlying container. For example, a stack, whose default underlying container is a `deque`, is a container adaptor which provides LIFO-based operations.

C. Bloom Filters

Bloom filters [39] were invented by Burton Howard Bloom in 1970. They can reduce the time of inserting and determining the presence of given elements in a collection by way of an m-bit array B (initialised with zeros) and k hash functions $h_1, h_2, \ldots, h_k$ with range $\{0, 1, \ldots, m-1\}$. Item i is inserted by setting bits $h_1(i), h_2(i), \ldots, h_k(i)$ in the bit array. When searching for element j, it is supposed to exist in the collection if and only if $\prod_{i=1}^{k} B[h_i(j)] = 1$. The primary benefit of using Bloom filters is memory efficiency (since the elements themselves are not explicitly stored). The primary drawback is the possibility of false positives arising from hash collisions. Fortunately the latter can be quantified and kept arbitrarily small [15]. Indeed, observing that Bloom filters with very low false positive probability are typically very sparsely populated, further memory savings arise by storing the position of marked bits rather than a dense bit array. Sparse Bloom filters, as developed by e.g. [15] implement this idea and allocate memory dynamically rather than statically. A secondary drawback is the inability to delete elements or to support multisets; this is straightforward to overcome using integer counters rather than bits, resulting in a *counting* Bloom filter. The latter have memory-efficient implementations (see e.g. [40, 41]), now made available in open-source libraries (e.g. [42]).

D. Web Service Level Agreements

Web Service Level Agreement (WSLA) [43] is an XML-based standard which allows service providers and customers to specify service targets. A WSLA instance contains three sections, namely *Parties*, *Service Description*, and *Obligations*. *Parties* defines the involved participants, who are divided into signatory parties and supporting parties. *Service Description* describes related information about a service, including any number of *SLAParameters* and *Metrics*. *SLAParameters* are used to define observable parameters of any service, such as response time and throughput. Each *SLAParameter* is mapped to a *Metric*, which defines how to measure the value of the *SLAParameter*. The final section, *Obligations*, specifies service level objectives and their action guarantees. The service level

objectives indicate desired service levels via comparison with their corresponding *SLAParameters*. When a service level objective is not met, its corresponding action guarantee is intended to result in a particular notification or control activity.

E. Related Work and Context

Self-adaptive software is a vast research area. Interested readers are directed to [33, 44] for two surveys of its evolution and current challenges. In terms of the classification presented in [33], we note that our proposed technique can be classed as *strong adaptation* through the replacement, addition or deletion of entities (in our case data structures) that have the same interface but different non-functional characteristics [45], the overarching goal being QoS management. Furthermore, both [33] and [44] mention that policies and QoS management are important factors of self-adaptive software which drive the adaptions of software. Although QoS requirements can be written in many formats, our library applies WSLA to clearly define these in a standards-conformant manner.

In terms of efforts to develop scalable container libraries, there have been several attempts, most of which have sought to exploit either parallelism, e.g. the Parallel Standard Template Library [46] and STAPL [47], or out-of-core techniques, e.g. STXXL [48]. In contrast to the present work, however, the programmer overhead, required expertise and configuration effort involved in using these libraries are relatively high, there is no ability to specify SLOs on containers and probabilistic data structures are rarely deployed.

III. Library Design and Implementation

The library in this paper is developed with reference to the concepts of *containers*, which store collections of objects while hiding their implementation details from software developers, and the concept of *autonomic computing*, which enables our library to monitor and alter its consumption of system resources. Figure 1 shows the architecture of the library, which consists of two major components: the Application Programming Interface, and the Self-adaptive Unit.

A. Application Programming Interface

The Application Programming Interface (API) provides software developers with two container template classes intended to provide most of the functionalities of STL containers: `ICollection`, which subsumes the *vector*, *list*, *set*, *stack*, and *queue* classes, and `IKeyValue` which subsumes the *map* class. These classes have methods of two kinds: *configuration* interfaces and *operation* interfaces. Configuration interfaces act as the means through which resource constraints, functionality requirements and service level objectives are imparted to the library while operation interfaces are responsible for manipulating containers managed by the library.

1) Configuration Interfaces: The constructors of our container classes are used to set the range of functionality that should be provided by a particular container as well as the service level objectives applicable to that container and (optionally) the frequency with which the SLO compliance should be checked.

TABLE I
Definitions of operation descriptors

Operation descriptor		Definition
OP_INSERT		Insertion
OP_ERASE		Deletion
OP_FIND		Find (retrival)
OP_SEARCH		Search (existence)
OP_INDEX		Direct index-based access
OP_ITERATOR		Iterator support
OP_FRONT	OP_INSERT_FRONT	Front insertion
	OP_ERASE_FRONT	Front deletion
OP_BACK	OP_INSERT_BACK	Back insertion
	OP_ERASE_BACK	Back deletion

The corresponding constructors are:

`ICollection<`*T*`>(`*op_desc*, *SLO_file*[, *freq*]`)`

and

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

where

- *op_desc* describes a required set of container functionalities (so-called operation descriptors). This recognises that it is rarely the case that every container instance will utilise its full set of potential functionality, allowing for more efficient underlying data structures. The definitions of all currently supported operation descriptors are listed in Table I, and the corresponding functionalities are shown in Table II. To elegantly achieve commonly-desired combinations of functionalities, combined operation descriptors are provided, as shown in Table III.
- *SLO_file* specifies a path to an XML file containing a description of the SLOs relevant to the container in WSLA format. SLOs can relate to response time, reliability or primary memory consumption and are described in a configuration file in WSLA format. The library infers the priority of each SLO according to their order of appearance (those appearing earlier are assigned higher priority). Where MeasurementURIs are required to specify the target of measurement operations, we use the Uniform Resource Name (URN) scheme to describe these as follows:
  urn:*ContainerClass*:*ResourceName*:*OperationDescriptor*
  ContainerClass may be `ICollection` or `IKeyValue`. *ResourceName* specifies the name of target resource. The available resource names are listed in Table IV. The *OperationDescriptor* is used to provide further information of *ResourceName*. For example, *ResponseTime* can be related to insertion time, search time, or deletion time, which can be specified through OP_INSERT, OP_SEARCH, and OP_ERASE, respectively.
- *freq* is an optional parameter specifying the frequency with which adaptation actions are carried out. This may be subsequently updated via the `setAdaptationFrequency` control interface.

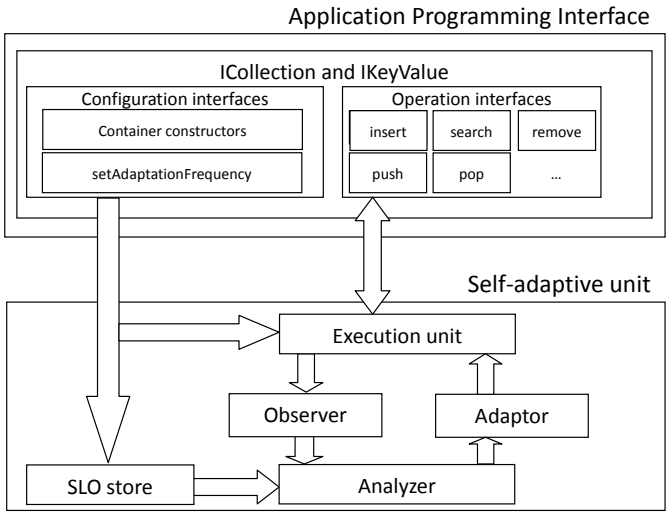

Fig. 1. The architecture of the library

TABLE II
MEMBER FUNCTIONS AND INVOLVED OPERATION DESCRIPTORS

ICollection<T> method	**IKeyValue<K,V> method**	**Involved operation descriptor**
insert(const T& x)	insert(const std::pair<K, V>T& x)	OP_INSERT
insert(iterator position, const T& x)	insert(iterator position, const std::pair<K, V>& x)	OP_INSERT | OP_ITERATOR
erase(const T& x)	erase(const K& x)	OP_ERASE
find(const T& x)	find(const K& x)	OP_FIND | OP_ITERATOR
search(const T& x)	search(const K& x)	OP_SEARCH
begin(), end()	begin(), end()	OP_ITERATOR
operator[]	operator[]	OP_INDEX
push_front()		OP_INSERT_FRONT
push_back()		OP_INSERT_BACK
pop_back()		OP_ERASE_BACK
pop_front()		OP_ERASE_FRONT

TABLE III
COMBINED OPERATION DESCRIPTORS

Data type	**Representative descriptor**	**Involved operation descriptors**
List	OP_LIST	OP_INSERT | OP_ERASE | OP_SEARCH | OP_ITERATOR | OP_BACK | OP_FRONT
Vector	OP_VECTOR	OP_INSERT | OP_ERASE | OP_SEARCH | OP_ITERATOR | OP_BACK | OP_INDEX
Set	OP_SET	OP_INSERT | OP_ERASE | OP_SEARCH | OP_ITERATOR
Stack	OP_STACK	OP_INSERT_FRONT | OP_ERASE_FRONT
Queue	OP_QUEUE	OP_INSERT_BACK | OP_ERASE_FRONT

TABLE IV
POSSIBLE VALUES OF *ResourceName*

Resource Name	**Definition**
RAM	Primary memory consumption
ResponseTime	The response time of a certain operation
Reliability	The container's reliability

2) Operation Interfaces: The Operation interfaces in our library provide commonly used operations such as `insert`, `search`, `remove`, `push`, and `pop`.

B. Self-adaptive Unit

The Self-adaptive unit maintains the currently used data structure and decides if it should be adjusted. It consists of an SLO store, an Execution unit, an Observer, an Analyzer, and an Adaptor.

1) SLO Store: The SLO store holds all service level objectives laid down by the configuration interfaces. These objectives include per operation response times (insertion time, search time, and deletion time), maximum primary memory usage, and reliability, which for probabilistic data structures is defined as the probability that every inserted element is mapped to a unique key [15]. For response times, soft requirements based on percentiles can be indicated, which means a certain percentage of response times can be above a response time target without violating the SLO.

2) Execution Unit: The Execution unit accepts container manipulation commands given via the API. If a command is compatible with the functionality that the target container should provide (as declared via the configuration interfaces), it is applied to the underlying data structure currently selected by the Analyzer for that container. Otherwise the command is rejected and an appropriate exception is thrown.

3) Observer: The Observer monitors per operation response times and computes memory consumption according to current container capacity, and, where appropriate, (i.e. for probabilistic data structures) reliability.

4) Analyzer: The Analyzer is a decision maker, which periodically determines if the library needs to adjust its data structures. The frequency of its activation is controlled through the `setAdaptationFrequency` configuration interface. When activated the Analyzer compares the Observer's profile data against the expectations of the SLOs in the SLO store. If the results indicated the violation of an SLO, the Analyzer determines if an adaptation could address it. Of course it may not be possible to meet some subset (or any) of the SLOs within resource constraints. Our library recognises this and therefore makes no guarantees that SLOs will be satisfied. In addition, actions taken to address one violated SLO may result in the subsequent violation of another (for example enforcing restrictions on memory consumption by applying out-of-core techniques may result in unacceptably large response times).

We resolve the ambiguity of which SLOs the library should attempt to satisfy by requiring each SLO to be assigned a distinct priority according to application and execution context. The adaptation mechanism addresses each of the SLOs in priority order, beginning with the highest. If the SLO being addressed is currently satisfied, no action is taken. If the SLO is violated, then an adaptation is invoked, provided that (a) the adaptation is expected to result in either the satisfaction of the SLO or a reduction in the degree of violation of the SLO and (b) the adaptation is not expected to result in the violation of a currently-satisfied SLO of higher priority.

5) Adaptor: The Adaptor executes adaptations that are expected to improve container compliance with its SLOs, as identified by the Analyzer.

There are three kinds of adaptations which can be made, according to the nature of the violated SLO and subject to the priority restrictions mentioned above. 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, 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 a set 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).

IV. Probabilistic Container Data Structures

An improved sparse Bloom filter is one of the many data structures adopted in our library. Such an underlying data structure may be appropriate given a container that does not require iterator-based functionality and which has reliability requirements less than 100%. It utilises a forest of AVL trees, whose number can be dynamically adjusted, in order to store hash keys of items. In contrast with many container libraries, users need not generate the hash keys themselves, since we make use of the CityHash [49] function library to do so. CityHash is capable of generating 32, 64, 128 and 256 bit hash keys from arbitrary data according to reliability requirements. This is adequate to provide search, insert and delete functionality on containers like sets. For containers where multiplicity of items is important (e.g. in *multisets*) a sparse counting Bloom filter [40, 41] is used to provide the necessary functionality.

V. Case Study

The case study chosen is the breadth-first-search (BFS) core of an explicit state-space exploration algorithm (commonly employed in the domains of model checking [50] and performance analysis [15]). A naïve implementation of the BFS algorithm is shown in Figure 2. Figure 3 displays the modified program adopting our library; note that it differs from the naïve program only in terms of the container declarations (one for the queue of unexplored states and one for the table of explored states). To evaluate the library's self-adaptive ability, the following SLOs were specified on the table of explored states (the variable *explored* in Fig. 3):

1) 90% of insertion times should be less than 1000ns, and 85% of search times should be less than 1200ns.
2) Reliability should be higher than 0.99.
3) Memory consumption should be no more than 7.5GB

```
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();
        foreach (State s' in G.succ(next)) {
            if (!explored.search(s')) {
                unexplored.push(s');
                explored.insert(s');
            }
        }
    }
}
```

Fig. 2. The naïve algorithm

```
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();
        foreach (State s' in G.succ(next)) {
            if (!explored.search(s')) {
                unexplored.push(s');
                explored.insert(s');
            }
        }
    }
}
```

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

The above SLOs were input to our library in the format of WSLA in a configuration file "ExploredSLOs.xml", the contents of which is shown in the Appendix in Listing 1. In a similar manner, an SLO requiring the primary memory consumption of the unexplored state queue (the variable *unexplored* in Fig. 3) to remain below 8MB was input via a configuration file "UnexploredSLOs.xml". As can be seen in Fig. 3, the value of `AdaptationFrequency` is 100, i.e. the Analyzer is activated every 100 operations. The influence of different values of `AdaptationFrequency` on response time is shown in Table V, which indicates that increasing the value of `AdaptationFrequency` could reduce response times. However, when its value reaches 1000, the response times rise due to the postponement of adaptations.

In order to evaluate the performance, the memory consumption and the reliability of our library, the algorithm was executed using a STL set, an AVL tree, a standard Bloom filter, and our library. In addition, the SLOs were put in different orders of priority, yielding six possible SLO sequences so as to observe our library's behaviour under different SLO priorities.

A. Comparison with Conventional Containers

In order to evaluate the efficacy of our library, the algorithm was executed using several conventional containers (an AVL tree, a STL set and a standard Bloom filter) and our library. Their insertion and search times, and primary memory consumptions were then compared. The results of insertion and search times are shown in Figures 4 and 5, which illustrate that our library yields better performance than conventional data structures. The two figures also show that our library's insertion time and search time rose suddenly at some points. That was because our library needed to adjust its data structures to satisfy the SLOs.

Figure 6 depicts the relationship between the average search time and the number of stored elements, while Figure 7 exhibits the memory consumptions of the AVL tree, the STL set, the Bloom filter, and our library. Our library used an order of magnitude less memory space than the AVL tree and the STL set. It was not as memory efficient as a standard Bloom filter, although its reliability was considerably higher.

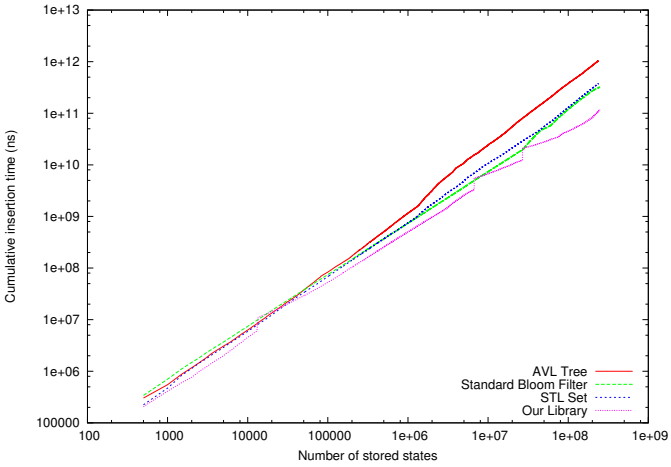

Fig. 4. Cumulative insertion times

B. Influence of SLO Priority

The results of insertion and search times are shown in Figures 8 and 9, which illustrate that when the given SLOs indicated that performance had higher priority over memory consumption, our library would expend considerably less execution time. The two figures also show that when the memory consumption was highest in order of priority, it would cost our library more in terms of execution time, because of frequent access to out-of-core memory.

The library's memory consumption conditions under the six priority orders is depicted in Figure 10. When memory consumption has the highest priority (MemPerRel or MemRelPer), the consumed memory space is the least. On the other hand, when memory consumption was lowest in order of priority,

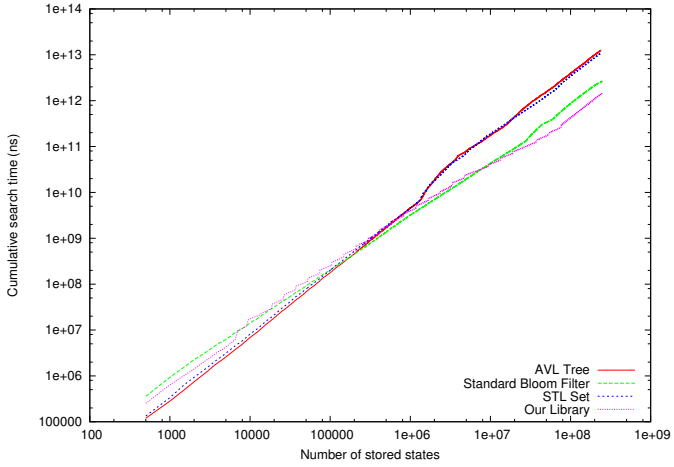

Fig. 5. Cumulative search times

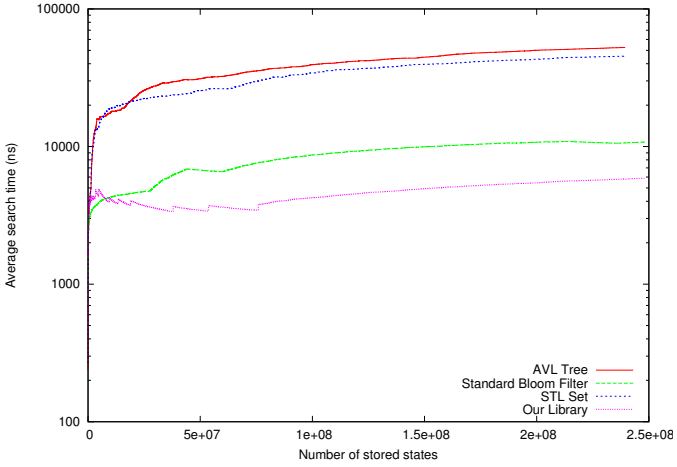

Fig. 6. Average search times

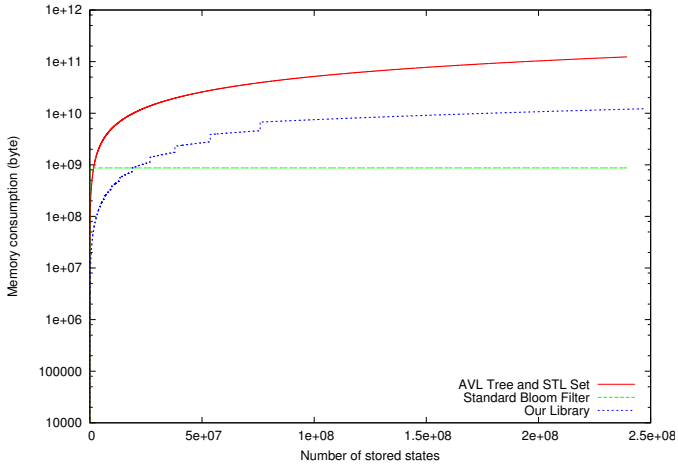

Fig. 7. Memory consumption

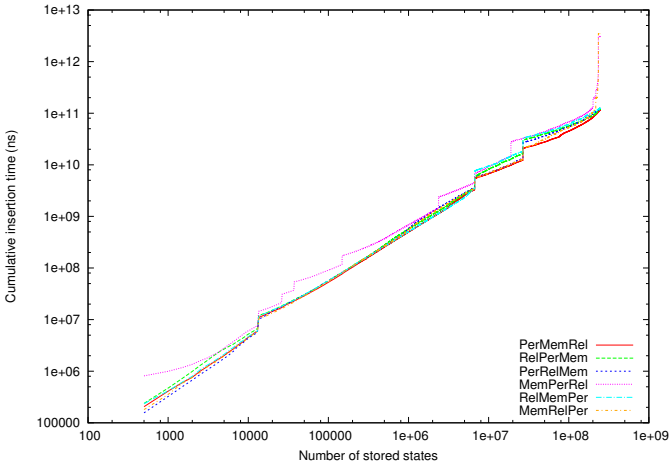

Fig. 8. Cumulative insertion times under different SLO priorities

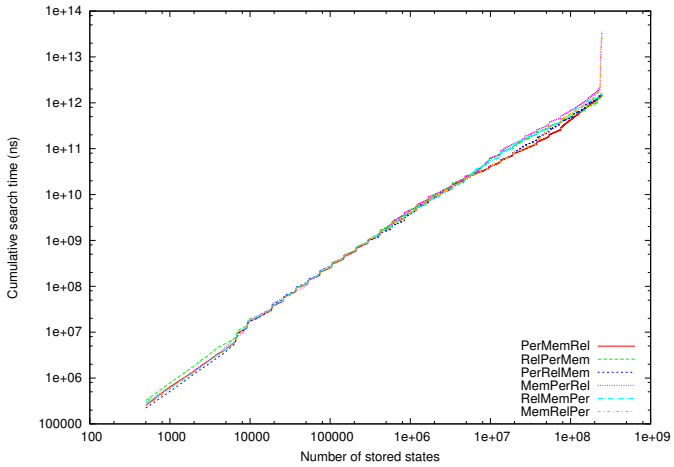

Fig. 9. Cumulative search times under different SLO priorities

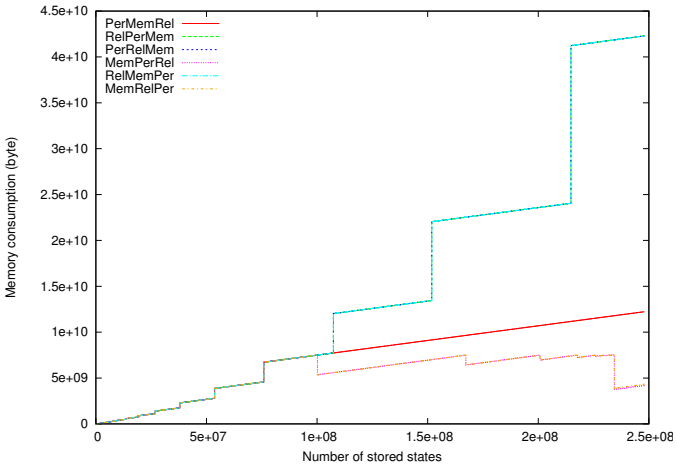

Fig. 10. Memory consumption under different SLO priorities

TABLE V
THE INFLUENCE OF VARIOUS VALUES OF ADAPTATIONFREQUENCY ON RESPONSE TIME

	AdaptationFrequency			
	1	10	100	1000
Insertion time (ns)	$3.26591 * 10^{11}$	$1.53390 * 10^{11}$	$1.02284 * 10^{11}$	$1.40904 * 10^{11}$
Search time (ns)	$1.62012 * 10^{12}$	$1.51335 * 10^{12}$	$1.50460 * 10^{12}$	$1.63597 * 10^{12}$

more memory space was consumed so as to boost performance and the reliability. This figure also indicates that when memory consumption reached its limit, MemPerRel and MemRelPer saved memory by reducing the number of AVL trees used. When the number of AVL trees could not be reduced, an out-of-core technique was activated to store states on disk. Additionally, notice that PerRelMem increased the number of AVL trees when the states' number was approximately 100 million while PerMemRel did not. That was because in the latter case memory consumption has higher priority than reliability. As a result, when the actual reliability was lower than the required reliability, PerMemRel would not enhance reliability to protect the memory quota.

The change in our library's reliability is shown in Fig. 11, which reveals that when reliability had the highest priority (RelMemPer and RelPerMem), the library would adapt its data structures to maintain a desirable reliability – over 0.99. By contrast, if reliability was the lowest in order of priority, the library's reliability would deteriorate as stored states increased.

C. Exploiting Out-of-core Storage

As mentioned, the variable *unexplored* in Fig. 3 was assigned an SLO indicating a maximum quota of primary memory of 8 MB. The actual memory consumptions of a queue adopting a naïve implementation and using our library, respectively, are shown in Figure 12. The queue adopting *ICollection* consumed a mere 8 MB primary memory space, which reduced primary memory consumption when compared with a naïve queue implementation by 97%.

VI. CONCLUSION AND FUTURE WORK

In this paper we have introduced a container library whose instances have the ability to adapt to various system environments and offered load at run time. Containers automatically deploy probabilistic and out-of-core techniques in an effort to meet SLOs defined on per operation response time, primary memory usage and reliability. Programmer overhead is kept low in terms of migrating existing code and moving software from one execution environment to another can be as simple as redefining the SLOs to be appropriate to the resource constraints of a new environment.

So far, we have proposed a framework for the use and deployment of self-adaptive containers and implemented part of the required functionalities. Besides completing the remaining functionalities, we could further reduce programmers' overheads of developing resource-aware software by developing tools capable of automatically scanning existing code to ascertain what subset of container functionality is being used, and hence work towards the automated transformation of existing code that uses container functionality into modified code that exploits our library.

REFERENCES

[1] J. F. Meyer, "On evaluating the performability of degradable computing systems," *IEEE Trans. Computers*, vol. 29, no. 8, pp. 720–731, 1980.

[2] G. J. Holzmann, "An improved protocol reachability analysis technique," *Software Practice and Experience*, vol. 18, no. 2, pp. 137–161, 1988.

[3] P. Wolper and D. Leroy, "Reliable hashing without collision detection," in *CAV*, 1993, pp. 59–70.

[4] U. Stern and D. L. Dill, "Improved probabilistic verification by hash compaction," in *CHARME*, 1995, pp. 206–224.

[5] B. Haverkort, A. Bell, and H. Bohnenkamp, "On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets," in *Proc. 8th International Conference on Petri Nets and Performance Models*, 1999, pp. 12–21.

[6] B. Bingham, J. Bingham, F. M. de Paula, J. Erickson, G. Singh, and M. Reitblatt, "Industrial strength distributed explicit state model checking," in *Proc. 9th International Workshop on Parallel and Distributed Methods in Verification*, 2010, pp. 28–36.

[7] R. T. Saad, S. D. Zilio, and B. Berthomieu, "A general lock-free algorithm for parallel state space construction," in *Proc. 9th International Workshop on Parallel and Distributed Methods in Verification*, 2010, pp. 8–16.

[8] D. D. Deavours and W. H. Sanders, "An efficient disk-based tool for solving very large Markov models," in *Proc. 9th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, 1997, pp. 58–71.

[9] W. Knottenbelt and P. Harrison, "Distributed disk-based solution techniques for large Markov models," in *Proc. 3rd International Workshop on the Numerical Solution of Markov Chains (NSMC '99)*, Sep. 1999, pp. 58–75.

[10] A. Bell and B. Haverkort, "Serial and parallel out-of-core solution of linear systems arising from Generalised Stochastic Petri Nets," in *Proc. High Performance Computing*, 2001, pp. 181–200.

[11] M. Z. Kwiatkowska and R. Mehmood, "Out-of-core solution of large linear systems of equations arising from stochastic modelling," in *Proc. 2nd Intl. Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification*, 2002, pp. 135–151.

[12] S. C. Allmaier and G. Horton, "Parallel shared-memory state-space exploration in stochastic modeling," in *Proc. IRREGULAR 1997*, 1997, pp. 207–218.

[13] S. Caselli, G. Conte, and P. Marenzoni, "Parallel State Space Exploration for GSPN Models," in *Proc. Application and Theory of Petri Nets*, vol. 935. Springer-Verlag, Berlin, 1995, pp. 181–200.

[14] G. Ciardo, J. Gluckman, and D. Nicol, "Distributed state-space generation of discrete-state stochastic models," *INFORMS J. of Computing*, vol. 10, pp. 82–93, 1996.

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

[16] S. Edelkamp and D. Sulewski, "Efficient explicit-state model checking on general purpose graphics processors," in *Proc. 17th International SPIN Conference on Model Checking Software*, 2010, pp. 106–123.

[17] R. M. Idury and M. S. Waterman, "A new algorithm for DNA sequence assembly," *Journal of Computational Biology*, vol. 2, pp. 291–306, 1995.

[18] P. A. Pevzner, H. Tang, and M. S. Waterman, "An Eulerian path approach to DNA fragment assembly," *Proc. Natl. Acad. Sci. USA*, vol. 98, no. 17, pp. 9748–9753, Aug. 2001.

[19] D. R. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read

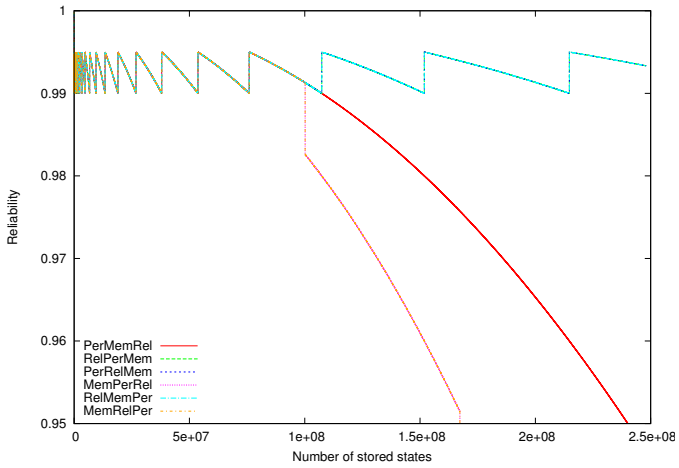

Fig. 11. Reliability under different SLO priorities

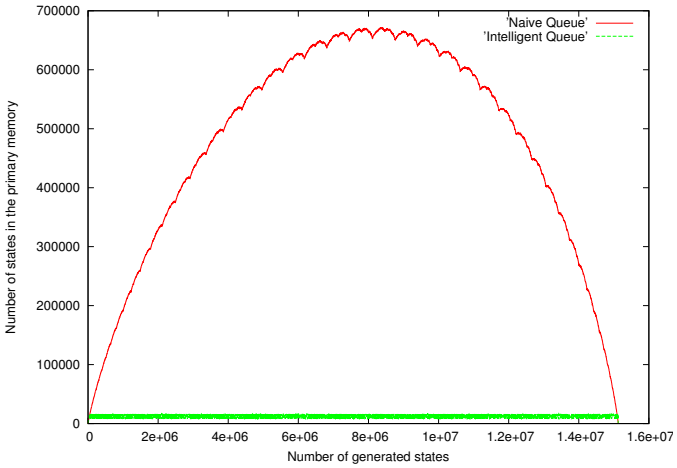

Fig. 12. Memory consumptions of naïve queue and intelligent queue

assembly using de Bruijn graphs," *Genome Research*, vol. 18, no. 5, pp. 821–829, 2008.

[20] J. Butler *et al.*, "ALLPATHS: De novo assembly of whole-genome shotgun microreads," *Genome Research*, vol. 18, no. 5, pp. 810–820, 2008.

[21] B. G. Jackson, M. Regennitter, X. Yang, P. S. Schnable, and S. Aluru, "Parallel de novo assembly of large genomes from high-throughput short reads." in *24th International Parallel and Distributed Processing Symposium (IPDPS)*, 2010, pp. 1–10.

[22] V. Kundeti, S. Rajasekaran, H. Dinh, M. Vaughn, and V. Thapar, "Efficient parallel and out of core algorithms for constructing large bi-directed de Bruijn graphs," *BMC Bioinformatics*, vol. 11, p. 560, 2010.

[23] Y. Liu, B. Schmidt, and D. L. Maskell, "Parallelized short read assembly of large genomes using de Bruijn graphs." *BMC Bioinformatics*, vol. 12, p. 354, 2011.

[24] Y. Li, P. Kamousi, F. Han, S. Yang, X. Yan, and S. Suri, "Memory efficient de Bruijn graph construction," *CoRR*, vol. abs/1207.3532, 2012.

[25] J. J. Cook and C. B. Zilles, "Characterizing and optimizing the memory footprint of de novo short read DNA sequence assembly." in *ISPASS*, 2009, pp. 143–152.

[26] P. Melsted and J. K. Pritchard, "Efficient counting of k-mers in DNA sequences using a Bloom filter." *BMC Bioinformatics*, vol. 12, p. 333, 2011.

[27] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown, "Scaling metagenome sequence assembly with probabilistic de Bruijn graphs," *Proceedings of the National Academy of Sciences*, vol. 109, no. 33, pp. 13 272–13 277, 2012.

[28] R. Chikhi and G. Rizk, "Space-efficient and exact de Bruijn graph representation based on a Bloom filter," *Algorithms in Bioinformatics*, vol. 7534 of Lecture Notes in Computer Science, pp. 236–248, 2012.

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

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

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

[32] W. Frakes and K. Kang, "Software reuse research: Status and future," *IEEE Transactions on Software Engineering*, vol. 31, pp. 529–536, July 2005.

[33] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.

[34] B. H. Cheng *et al.*, "Software engineering for self-adaptive systems," B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26.

[35] 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, p. 2003, 2003.

[36] 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.

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

[38] M. Rohr, S. Giesecke, W. Hasselbring, M. Hiel, W.-J. van den Heuvel, and H. Weigand, "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.

[39] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, pp. 422–426, July 1970.

[40] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting Bloom filters," in *LNCS 4168: Proc. ESA 2006*, 2006, pp. 684–695.

[41] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The variable-increment counting Bloom filter," in *Proc. Infocom 2012*, 2012.

[42] J. Hines *et al.*, "dablooms: An open source, scalable counting Bloom filter library," 2012. [Online]. Available: http://word.bitly.com/post/28558800777/dablooms-an-open-source-scalable-counting-bloom

[43] H. Ludwig, A. Keller, A. Dan, R. P. King, and R. Franck, *Web service level agreement (WSLA) language specification*, IBM Corporation Std., 2003.

[44] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing - degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, no. 3, 2008.

[45] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *Computer*, vol. 37, no. 7, pp. 56–64, Jul. 2004.

[46] E. Johnson and D. Gannon, "HPC++: Experiments with the Parallel Standard Template Library," in *Proc. ICS '97*, 1997, pp. 124–131.

[47] G. Tanase *et al.*, "The STAPL parallel container framework," in *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP)*, 2011.

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

[49] G. Pike and J. Alakuijala, "The CityHash family of hash functions," 2010. [Online]. Available: http://code.google.com/p/cityhash/

[50] O. Grumberg, E. Clarke, and D. Peled, *Model Checking*. MIT Press, Cambridge, 2000.

APPENDIX

Listing 1. The SLO configuration file of *explored*

```
<?xml version='1.0' ?>
<SLA xmlns="http://www.ibm.com/wsla" xmlns:xsi=
        "http://www.w3.org/2001/XMLSchema-instance">

 <Parties>
  <ServiceProvider />
  <ServiceConsumer />
 </Parties>

 <ServiceDefinition name='SampleService'>

  <Operation name='insert'>
   <SLAParameter name="InsertTimeRatio" unit="Percent">
    <Metric> InsertTimeRatio_Metric </Metric>
   </SLAParameter>
   <Metric name="InsertTimeRatio_Metric" unit="Percent">
    <Source>ServiceProvider</Source>
    <Function xsi:type="PercentageLessThanThreshold">
     <Metric> InsertTime_Metric </Metric>
     <Value> <LongScalar> 1000 </LongScalar> </Value>
    </Function>
   </Metric>
   <Metric name="InsertTime_Metric" unit="ns">
    <Source>ServiceProvider</Source>
    <MeasurementDirective xsi:type="ResponseTime">
     <MeasurementURI>
      urn:ICollection.ResponseTime.OP_INSERT
        </MeasurementURI>
    </MeasurementDirective>
   </Metric>
  </Operation>

  <Operation name='search'>
   <SLAParameter name="SearchTimeRatio" unit="Percent">
    <Metric> SearchTimeRatio_Metric </Metric>
   </SLAParameter>
   <Metric name="SearchTimeRatio_Metric" unit="Percent">
    <Source>ServiceProvider</Source>
    <Function xsi:type="PercentageLessThanThreshold">
     <Metric> SearchTime_Metric </Metric>
     <Value> <LongScalar> 1200 </LongScalar> </Value>
    </Function>
   </Metric>
   <Metric name="SearchTime_Metric" unit="ns">
    <Source>ServiceProvider</Source>
    <MeasurementDirective xsi:type="ResponseTime">
     <MeasurementURI>
         urn:ICollection.ResponseTime.OP_SEARCH
        </MeasurementURI>
    </MeasurementDirective>
   </Metric>
  </Operation>

  <Operation name='Reliability'>
   <SLAParameter name="CurrentReliability" unit="">
    <Metric> CurrentReliability_Metric </Metric>
   </SLAParameter>
   <Metric name="CurrentReliability_Metric" unit="">
    <Source>ServiceProvider</Source>
    <MeasurementDirective xsi:type="wsla:Gauge">
     <MeasurementURI>
      urn:ICollection.Reliability.OP_Reliability
     </MeasurementURI>
    </MeasurementDirective>
   </Metric>
  </Operation>

  <Operation name='RAM'>
   <SLAParameter name="RAMSIZE" unit="GB">
    <Metric> RAMSIZE_Metric </Metric>
   </SLAParameter>
   <Metric name="RAMSIZE_Metric" unit="GB">
    <Source>ServiceProvider</Source>
    <MeasurementDirective xsi:type="wsla:Gauge">
     <MeasurementURI>
      urn:ICollection.RAM.OP_RAM
     </MeasurementURI>
    </MeasurementDirective>
   </Metric>
  </Operation>

 </ServiceDefinition>

 <Obligations>

  <ServiceLevelObjective name="InsertTimeSLO">
   <Obliged>service_provider</Obliged>
    <Validity>
     <Start>2013-01-01T14:00:00</Start>
     <End>2014-01-01T14:00:00</End>
    </Validity>
   <Expression>
     <Predicate xsi:type="GreaterEqual">
      <SLAParameter>
       InsertTimeRatio
      </SLAParameter>
      <Value> 0.9 </Value>
     </Predicate>
   </Expression>
   <EvaluationEvent>NewValue</EvaluationEvent>
  </ServiceLevelObjective>

  <ServiceLevelObjective name="SearchTimeSLO">
   <Obliged>service_provider</Obliged>
    <Validity>
     <Start>2013-01-01T14:00:00</Start>
     <End>2014-01-01T14:00:00</End>
    </Validity>
   <Expression>
     <Predicate xsi:type="GreaterEqual">
      <SLAParameter>
       SearchTimeRatio
      </SLAParameter>
      <Value> 0.85 </Value>
     </Predicate>
   </Expression>
   <EvaluationEvent>NewValue</EvaluationEvent>
  </ServiceLevelObjective>

  <ServiceLevelObjective name="ReliabilitySLO">
   <Obliged>service_provider</Obliged>
    <Validity>
     <Start>2013-01-01T14:00:00</Start>
     <End>2014-01-01T14:00:00</End>
    </Validity>
   <Expression>
     <Predicate xsi:type="GreaterEqual">
      <SLAParameter>
       CurrentReliability
      </SLAParameter>
      <Value> 0.99 </Value>
     </Predicate>
   </Expression>
   <EvaluationEvent>NewValue</EvaluationEvent>
  </ServiceLevelObjective>

  <ServiceLevelObjective name="RAMSIZESLO">
   <Obliged>service_provider</Obliged>
    <Validity>
     <Start>2013-01-01T14:00:00</Start>
     <End>2014-01-01T14:00:00</End>
    </Validity>
   <Expression>
     <Predicate xsi:type="LessEqual">
      <SLAParameter>
       RAMSIZE
      </SLAParameter>
      <Value> 7.5 </Value>
     </Predicate>
   </Expression>
   <EvaluationEvent>NewValue</EvaluationEvent>
  </ServiceLevelObjective>

 </Obligations>

</SLA>
```