Imperial College London
Department of Computing

Self-Adaptive Containers: A Novel Framework for Building Scalable QoS-Aware Software with Low Programmer Overhead

Wei-Chih Huang

Submitted in part fulfillment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London
March 2015

Abstract

As the number of execution environments increases dramatically, ever-changing non-functional requirements often lead to the challenge of frequent code refactoring. Despite help of traditional software engineering techniques, adapting software to meet each execution environment and application context remains a non-trivial endeavour. Manually reimplementing software possibly takes months or years of programmer effort and requires high levels of expertise. Furthermore, to build software for different execution environments often results in either a small code base which cannot guarantee Quality of Service or a large manually-optimised code base which is difficult to maintain.

This thesis presents a novel self-adaptive container framework which can dynamically adjust its resource usage in an effort to meet resource constraints and scalability requirements. Each container instance is associated with programmer-specified Service Level Objectives with respect to performance, reliability, and primary memory use. To prevent ambiguity among multiple Service Level Objectives, each of them is specified in the format of standard Web Service Level Agreement. This framework features tighter functionality specification than that of standard container frameworks, which enables greater scope for efficiency optimisations, including the exploitation of probabilistic data structures, out-of-core storage, parallelism, and cloud storage. These techniques are utilised in a low-cost way through the integration of third-party libraries, which also enable our framework to provide a wider class of Service Level Objectives. In addition, to reduce the time of learning how to use the framework, its interfaces are designed to be close to those of standardised libraries.

The framework has been implemented in C++ and utilised in two case studies centred on explicit state-space exploration adopting a breadth-first search algorithm, and route planning adopting a Dijkstra's shortest path algorithm. In order to illustrate the framework's viability and capability, various Service Level Objectives are assigned. Furthermore, the implementation of our framework is utilised to explore approximately 240 million states in the first case study and to find the shortest path of a graph representing the USA road network, containing approximately 24 million nodes and 58 million arcs. The experimental results show that the framework is capable of dynamically adjusting its resource usage according to assigned Service Level Objectives and dealing with large-scale data. At the same time, the programmer overhead is kept low in terms of the degree to which code is modified.

Acknowledgements

I would like to thank:

- My supervisor, Dr. William Knottenbelt, for teaching me how to be a qualified researcher and inspiring me to think and explore cutting-edge techniques
- My wife, Yaru, for keeping me company, taking care of me, helping me deal with trivia, and giving me the most precious gift, Ian
- My parents, who give me full support to fulfil my dream

Dedication

This thesis is dedicated to my family. Without them, I won't have the courage to leave my comfort zone and to explore other possibilities in my life.

‘You must be shapeless, formless, like water. When you pour water in a cup, it becomes the cup. When you pour water in a bottle, it becomes the bottle. When you pour water in a teapot, it becomes the teapot. Water can drip and it can crash. Become like water my friend.’
Bruce Lee, Actor (1940 –1973)

Contents

List of Tables

List of Figures

Chapter 1

Introduction

'Our dilemma is that we hate change and love it at the same time; what we really want is for things to remain the same but get better.'

Sydney J. Harris, American Journalist (1914 –1986)

1.1 Motivation

Modern software industries are faced with the explosion in number of execution environments (e.g. tablet, smartphone, laptop, server, etc.) and application contexts (any information that can characterise the situation of an application). When software is ported to a new execution environment, its functionality requirements, which describe what software is supposed to do, can be easily satisfied via cross-platform tools e.g. Marmalade (Marmalade, 2014), Titanium (Appcelerator, 2014), Unity (Unity Technologies, 2015). However, these tools cannot automatically fulfil non-functional requirements, which define how software is supposed to be. Consequently, software reimplementation remains a non-trivial job.

Applications in each potential execution environment may be assigned different non-functional requirements (also known as Quality of Service requirements) in order to provide high quality user experience. Consider Figure 1.1, which exhibits the importance of three common Quality of Service (QoS) parameters on different application contexts and execution environments. As can be seen, when games are executed on game consoles, their performance is expected at a high level in an effort to meet players' expectation, which may result in high consumption of memory and electricity power. By contrast, if games are operated on smartphones, lower performance can be allowed. That is because high performance may rapidly exhaust limited memory space and battery power. A similar situation can also be observed when web browsers run on smartphones, servers and game consoles. Web browsers are frequently-used applications on smartphones, which leads to the demand of high performance with low memory consumption. The same demand is not required when they are executed on servers and game consoles because these two execution environments can supply sufficient memory space and are not expected to be the main means of browsing web pages. As a result, QoS requirements vary according to the target execution environment and application context.

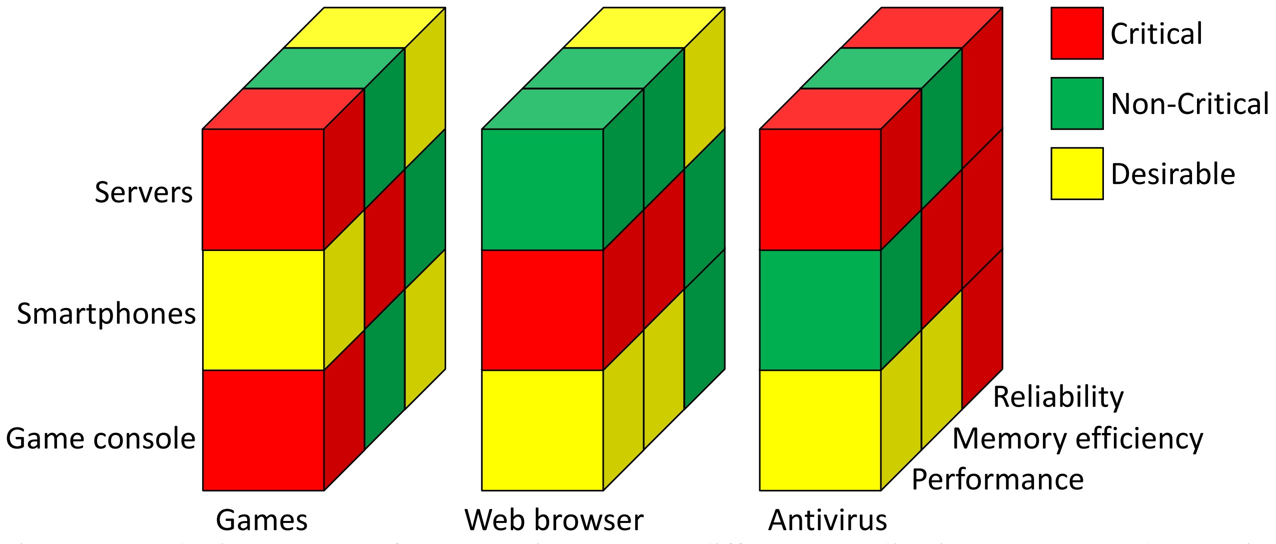


Figure 1.1: The importance of QoS requirements on difference application contexts and execution environments

Adapting software to each execution environment and application context is not a trivial job, which may involve months or years of programmer effort and require high levels of expertise. Even if sound software engineering techniques are adopted to maximise software reuse, frequent and time-consuming software reimplementation remains an unsolved challenge. To address this, we propose a novel self-adaptive container framework which dynamically adapts to the QoS and scalability requirements of its current execution environment.

Our work is inspired by the discovery that similar techniques are reinvented for scalability, robustness, and performance improvement across apparently different application domains. Table 1.1 lists five application domains utilising the techniques of out-of-core storage, parallelism, and probabilistic data structures to deal with massive input data. The first application domain, explicit state-space exploration, is a main approach to performance verification of model-based concurrent systems. The major issue regarding this domain is the large number of states, which causes the shortage of primary memory and performance deterioration. Subsequent adoption of these techniques has carried the supported states forward (from $\sim 10^5$ states to $\sim 10^{10}$ states (Bingham et al., 2010)).

The second application domain, DNA sequence assembly, refers to DNA sequence aligning and merging in order to reconstruct the original sequence. To effectively assemble DNA fragments, Idury and Waterman (Idury & Waterman, 1995) introduce de Bruijn graphs, whose directed edges represent sequence reads or fragments of fixed size. It is then implemented in Euler (Pevzner, Tang, & Waterman, 2001, 17) and Velvet (Zerbino & Birney, 2008) software packages. However, the prohibitive memory consumption of the two packages restricts the capacity to small genomes. To solve this issue, many research teams make use of the similar techniques displayed in Table 1.1, which drive the assembly capacity from organisms with $\sim 10^6$ base pairs (e.g. simple virus) to organisms with $\sim 10^9$ base pairs (e.g. humans) (Chikhi & Rizk, 2012). Specifically, when *impatiens* genome (containing approximately 300 million reads) is assembled, Velvet requires 20 GB memory space. By contrast, Minia only consumes 1.2 GB (Kleftogiannis, Kalnis, & Bajic, 2013).

The third application domain, route planning, refers to the computation of the optimal route involving several stopovers between two geographical locations. It has been widely used in GPS systems, whose memory architecture consists of faster but limited primary memory and sufficient but slower out-of-core memory (e.g. flash memory, memory card). Such architecture may rapidly exhaust primary memory when a large-scale map is input. Through the adoption of parallelism, out-of-core storage, and probabilistic data structures seen in Table 1.1, performance is considerably boosted and primary memory consumption is reduced. For example, the query time of computing a route containing 30 million nodes is improved from 329 seconds to 42 seconds. Simultaneously, memory consumption is reduced from 3735 MB (Goldberg & Werneck, 2005b) to 548 MB (P. Sanders, Schultes, & Vetter, 2008).

Application Domains	**Applied Techniques**		
	Out-of-core Storage	**Probabilistic Data Structures**	**Parallelism**
Explicit state-space exploration	(Deavours & W. H. Sanders, 1998) (Knottenbelt & Harrison, 1999) (Kwiatkowska & Mehmood, 2002) (Bingham et al., 2010)	(Holzmann, 1988) (Wolper & Leroy, 1993) (Stern & Dill, 1995) (Knottenbelt, 2000) (Haverkort, Bell, & Bohnenkamp, 1999) (Bingham et al., 2010) (Saad, Zilio, & Berthomieu, 2010)	(Caselli, Conte, & Marenzoni, 1995) (Allmaier & Horton, 1997) (Ciardo, Gluckman, & Nicol, 1998) (Knottenbelt & Harrison, 1999) (Edelkamp & Sulewski, 2010) (Bingham et al., 2010) (Saad, Zilio, & Berthomieu, 2010)
DNA sequence assembly	(Cook & Zilles, 2009) (Kundeti, Rajasekaran, Dinh, Vaughn, & Thapar, 2010) (Y. Li et al., 2012)	(Melsted & Pritchard, 2011) (Pell et al., 2012) (Chikhi & Rizk, 2012)	(Butler et al., 2008) (B. G. Jackson, Regennitter, Yang, Schnable, & Aluru, 2010) (Kundeti, Rajasekaran, Dinh, Vaughn, & Thapar, 2010) (Y. Liu, Schmidt, & Maskell, 2011)
Route planning / motion planning	(T. Li, Yang, & Lian, 2012) (Edelkamp & Schrödl, 2000) (Goldberg & Werneck, 2005a)	(Jiang, Ji, Wang, Zhu, & Cheng, 2014) (Wewetzer, Scheuermann, Lübke, & Mauve, 2009) (Chi, Ning, Lang, & Yuan, 2009)	(Witkowski, 1983) (Gudaitis, Lamont, & Terzuoli, 1995) (Delling, Katz, & Pajor, 2012)
Visualisation	(Chiang & Silva, 1999) (Cignoni, Montani, Rocchini, & Scopigno, 2003) (Vo, Silva, Scheidegger, & Pascucci, 2012)		(Upson et al., 1989) (Abram & Treinish, 1995) (Meredith, Ahern, Pugmire, & Sisneros, 2012)
Similarity search	(Gionis, Indyk, & Motwani, 1999) (Fogaras & Rácz, 2005) (Wang & K. Liu, 2012)	(Krishnamurthy et al., 2007) (Nie, Hua, Feng, Li, & Sun, 2014) (Zhao, Tang, & Ye, 2012)	(Berchtold, Böhm, Braunmüller, Keim, & Kriegel, 1997) (Teodoro, Valle, Mariano, Torres, & Meira, 2011) (Alabduljalil, Tang, & Yang, 2013)

Table 1.1: The application domains adopting out-of-core storage, probabilistic data structures, and parallelism

The fourth application domain, visualisation, refers to a technique which makes use of images, diagrams, or animations to express data. When massive data is displayed, it has to be separated into two parts (i.e. one part of data is stored in primary memory and the other part of data is stored in disk) in an effort to reduce primary memory use. However, this may cause considerable performance drop. A stockpile of research adopts parallelism and efficient out-of-core algorithms to boost process time. Among them, Vo et al. lessen the time of visualising the Happy Buddha statue (Levoy, 2013), which consists of approximately 0.5 million vertices and one million triangles, from 71 seconds to 0.4 second (Vo, Silva, Scheidegger, & Pascucci, 2012).

The final application domain, similarity search, represents the finding of the closet pairs in a data library. Its application entails multimedia, chemistry, and biology. Similar to the above-mentioned application domains, a large-sized data library may cause either performance decline or primary memory shortage. The similar techniques (i.e. out-of-core storage, parallelism, and probabilistic data structures) are also adopted to solve this issue. Take protein search on a 20 giga-base pairs dataset for example. Zhao et al. exploit a probabilistic data structure, which reduces memory consumption from 8 GB to 2 GB and response time from 3 weeks to 5 hours (Zhao, Tang, & Ye, 2012).

The above-mentioned application domains reveal a lot of commonality. First, Bloom filters or their variants are commonly adopted to reduce primary memory use. Second, the adopted techniques are implemented and maintained by individual research teams. Finally, the problems they try to solve are all related to how to efficiently manipulate large-scale data. This is the reason why this thesis focuses on containers, which encapsulate the ways of storing data and manipulating it. In other words, data is efficiently stored and manipulated by our containers. Furthermore, the selection of a proper container data structure is frequently the bottleneck of QoS (Isensee, 2014). As a result, we design and implement self-adaptive containers that can automatically choose a data structure when software starts to be executed and dynamically change it to meet QoS requirements.

Through the utilisation of self-adaptive containers, our vision is to build a single code base which satisfies functional requirements and adapt at run time to meet varying QoS requirements in different execution environments. As can be seen in Figure 1.2, traditionally, functional requirements and QoS requirements are hard-coded in programs. Since QoS requirements are mixed with code and functional requirements, the modification of QoS requirements may lead to software reimplementation. By contrast, as can be seen in Figure 1.3, the concern of QoS requirements is separated from programs and QoS requirements of target environments are dynamically achieved by our framework. This brings the following advantages to programmers, managers, and end-users. For programmers, the tedium of reimplementing software to meet various execution environments, application contexts, and QoS requirements is prevented through the self-adaptive mechanism, which will take care of peripheral concerns (Lalanda, McCann, & Diaconescu, 2013). As a result, programmers can put their effort on algorithm implementations. For managers, the effort to build robust and scalable software and the requirement for very skilled programmers are considerably reduced (Kramer & Magee, 2007). Furthermore, software becomes easily managed owing to a single code base of software which can satisfy varying QoS requirements. For end-users, the Quality of Service they experience is consistent with the expectation of Service Level Objectives on different environments.

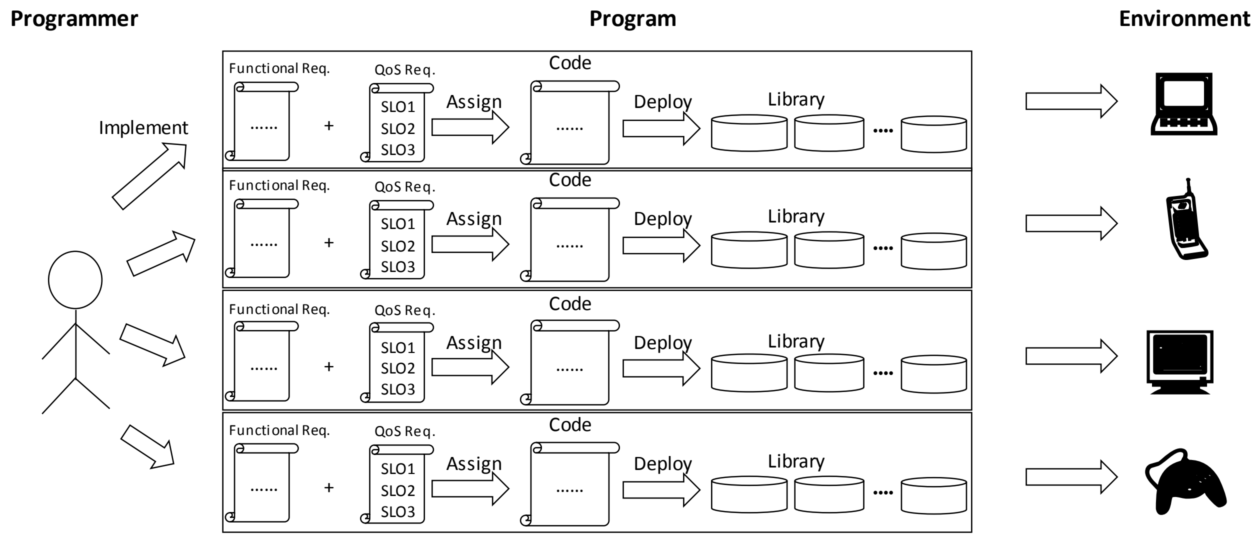


Figure 1.2: The conventional way to develop software for different execution environments

1.2 Objectives

The aims and objectives of this thesis are

- To offer a container framework which separates out QoS concerns from programs and dynamically adjusts its underlying data structures so as to prevent frequent software reimplementation.

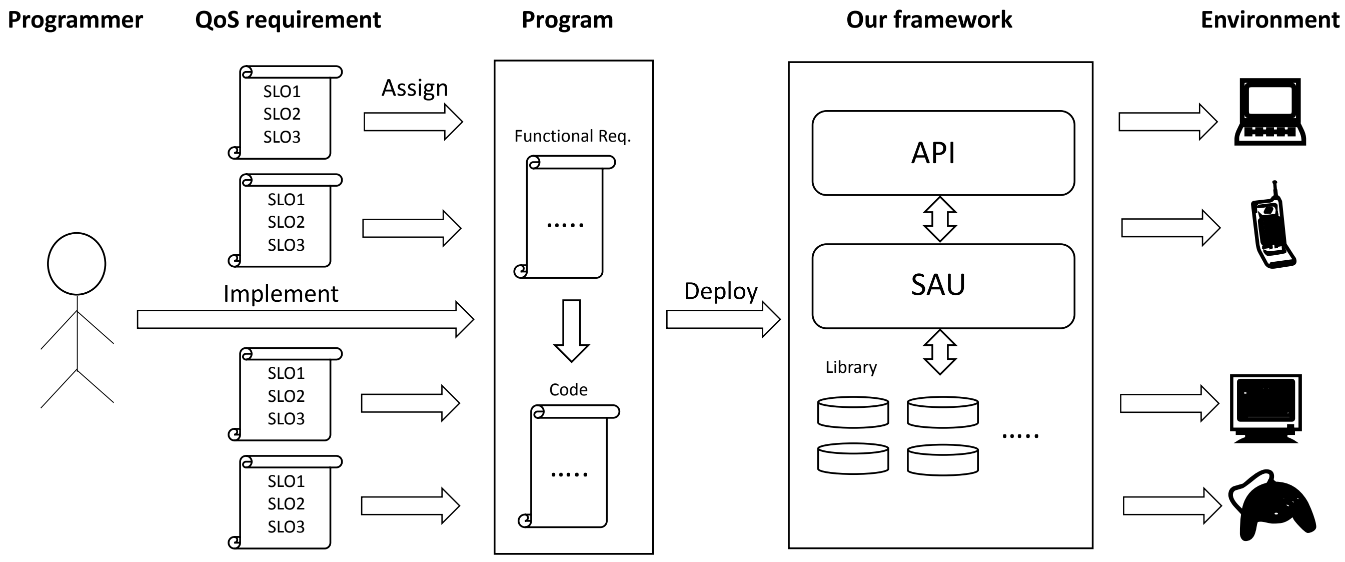


Figure 1.3: Our vision to build single-version software for all execution environments

- To feature tighter functionality specification in an effort to provide greater scope for efficiency optimisation, including the techniques of parallelism, probabilistic data structures, and out-of-core storage.
- To provide a mechanism for specifying Service Level Objectives.
- To exploit third-party container libraries in order to broaden the class of Service Level Objectives at low cost.
- To investigate alternatives to disk-based out-of-core memory space.
- To devise a means of providing a straightforward and low-overhead way of satisfying different QoS requirements and migrating existing code from one execution environment to another.

1.3 Contributions

This thesis presents a container framework which dynamically adapts its data structures in order to meet Service Level Objectives (SLOs) and which features tighter functionality specification for greater scope of efficiency optimisations. This thesis yields the following specific contributions:

- The self-adaptive container framework is designed and implemented in C++. Its underlying data structures are automatically selected and dynamically changed by a classical self-adaptive cycle composed of an Observer, an Analyser, and an Adaptor according to specified functionalities and Service Level Objectives in terms of performance, primary memory use, and reliability.
- The framework supplies operation descriptors through which a suitable data structure (e.g. FIFO queue, priority queue, trees, probabilistic data structures, out-of-core data structures) can be chosen at run time. For probabilistic data structures, an improved sparse Bloom filter, whose performance, memory use, and reliability can be dynamically adjusted, is implemented.
- It accepts programmer-specified Service Level Objectives in the format of Web Service Level Agreement.
- It contains an abstract layer to integrate third-party libraries and APIs including CityHash for generating uniformly-distributed hash keys, STXXL for out-of-core storage, Intel Threading Building Blocks for parallelism, and WebStor for cloud storage.
- It can dynamically transfer data from and to cloud storage (Amazon Simple Storage Service) via the utilisation of WebStor when memory-related SLOs are violated.
- The framework is applied to the domains of state-space exploration adopting a breadth-first search algorithm and route planning adopting a Dijkstra's shortest path algorithm. The former algorithm is used to explore approximately 240 million states, and the latter is assigned approximately 23 million nodes and 58 million edges. The results suggest that software deploying the framework can dynamically adjust itself to satisfy specified SLOs. Simultaneously, existing code only needs to modify declaration of container variables.

1.4 Thesis Outline

The remainder of this thesis is organised as follows.

Chapter 2 introduces the two fundamental concepts of this framework, autonomic computing and containers. We then present QoS specification languages including Web Server Level Agreements, Web Service Agreement specification, SLAng, Web Service Offerings Language, and Performance Trees. After the description and comparison of the QoS specification languages, Bloom filters and their variants are presented. Cloud storage is then introduced. Next, we compare the capability of well-developed container libraries from the perspectives of out-of-core storage, parallelism, probabilistic data structures, and self-adaptation. Finally, we discuss related contexts which involve language extensions and reference models for building self-adaptive systems as well as techniques of dynamically changing data structures.

Chapter 3 presents the novel self-adaptive container framework from the functionality of its three major components to the responsibilities of units in each major component. We then describe the core of our framework, self-adaptive mechanism, in detail. The whole mechanism is formed by SLO metrics, a self-adaptive cycle, and adaptation actions. Finally, a prototype is implemented and applied to a case study centred on explicit state-space exploration, adopting a breadth-first search algorithm, in order to show the viability, capability, and scalability of the framework.

Chapter 4 illustrates the extensions of our framework's functionality. The functionality of the previously developed prototype is broadened through the implementation of key-value stores and priority queues. The former have been supported by many programming languages and adopted in many industries managing large-scale data, and the latter are widely exploited by scheduling and networking. In addition, the enhanced prototype is applied to a new case study centred on route planning, adopting a Dijkstra's shortest path algorithm. The experimental results suggest that the extended functionalities support the dynamic adjustment of QoS with low programmer overhead.

Chapter 5 presents the way to cooperate with other container libraries and cloud storage. The purpose of cooperation with other libraries is to achieve out-of-core storage and parallelism at low cost and to provide a broader class of Service Level Objectives. The container libraries we choose to fulfil the functionality of out-of-core storage and parallelism are STXXL and Intel Threading Building Blocks, respectively. We then describe how to effectively utilise cloud storage, which has become important in recent years. The dynamic deployment of cloud storage can supply alternative external memory when other memory spaces are not available and prevent programmers from reimplementing software when cloud storage services change. The case studies utilised in Chapter 3 and Chapter 4 are investigated again to show the impact on performance and memory consumption when our framework dynamically activates these libraries and cloud storage.

Chapter 6 concludes this thesis with a summary of our achievement, a discussion of applications, and the directions of future research.

Appendix A shows how to express Service Level Objectives in the format of Web Service Level Agreement.

1.5 Publications and Statement of Originality

I declare that this thesis was composed by myself, and that the work it presents is my own, unless stated otherwise.

The following publications arose from the work carried out during my PhD:

- **8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2013)** (Huang & Knottenbelt, 2013) presents a self-adaptive container framework which can dynamically change the underlying data structures so as to meet programmer-specified Service Level Objectives expressed in the format of Web Service Level Agreement. This paper also illustrates the dynamic exploitation of out-of-core storage and probabilistic data structures. Furthermore, a case study centred on explicit state-space exploration is investigated to show the feasibility, scalability, and capability. The experimental results reveal that our framework is capable of boosting performance and reducing memory consumption based on assigned SLOs. The work presented in Chapter 3 is based on this paper.

- **6th International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE 2014)** (Huang & Knottenbelt, 2014b) extends our paper published in SEAMS 2013. This paper broadens the functionality of the previously-developed prototype in terms of the support of priority queues and key-value stores. In addition, a new case study, route planning utilising a Dijkstra's shortest path algorithm, is investigated in order to exhibit the ability of the enhanced prototype. Similarly, the experimental results suggest that the framework yields better performance and consume less memory space compared to the STL containers. In addition, the framework can dynamically adjust the underlying data structures and deploy the techniques of out-of-core storage and probabilistic data structures. Simultaneously, code modification is restricted to declaration of container variables. The content of Chapter 4 is based on this paper.

- **Chapter 5 in Handbook of Research on Emerging Advancements and Technologies in Software Engineering** (Huang & Knottenbelt, 2014a) surveys many application domains dealing with large-scale data, entailing explicit state-space exploration, route planning, DNA sequence assembly, visualisation, and similarity search, as well as related research areas including self-adaptive systems, QoS specification languages, probabilistic data structures, and third-party libraries supporting the techniques of out-of-core storage, parallelism, probabilistic data structures, and self-adaptation. It also discusses some limitations of the framework and possible solutions to overcome them. The material presented in Chapter 2 is based on this work.

- **11th IEEE International Conference on Autonomic and Trusted Computing (ATC 2014)** (Huang & Knottenbelt, 2014c) integrates cloud storage and extends the framework's interoperability through the cooperation with third-party libraries. Cooperating with these libraries enables our framework to explore a low-cost way of supporting out-of-core storage and parallelism and to provide a broader class of Service Level Objectives, especially those related to performance and memory consumption. The integration of cloud storage enables our framework to offer alternative out-of-core memory and prevent implementation of data access methods for different cloud storage services. The material featured in Chapter 5 is based on this paper.

Chapter 2

Background and Related Context

‘Life is neither static nor unchanging. With no individuality, there can be no change, no adaptation and, in an inherently changing world, any species unable to adapt is also doomed’

Jean M. Auel, American Writer

2.1 Introduction

This chapter introduces the background knowledge we adopt to build the self-adaptive container framework. We first describe autonomic computing, which aims to reduce rapidly growing software complexity. An autonomic computing system refers to a system with the ability of self-management. Such system can be built through a variety of methodologies such as feedback control loops, programming language extensions, component-based software engineering, and external self-adaptive architectures. We will review the most commonly-used feedback control loop, MAPE-K (Kephart & Chess, 2003; IBM, 2003), and classify our methodology through the taxonomies presented in the following surveys: Huebscher and McCann, 2008, Salehie and Tahvildari, 2009, Khalid, Haye, Khan, and Shamail, 2009, Macías-Escrivá, Haber, del Toro, and Hernandez, 2013, and Krupitzer, Roth, VanSyckel, Schiele, and Becker, 2014.

Next, we present three standard container libraries, Standard Template Library (Josuttis, 2012), Java Collections Framework (Watt & D. Brown, 2001), and Python, all of which are implemented as template classes to store a variety of data types, hide development details from programmers, and reduce development time in terms of the reuse of existing code.

Our framework accepts programmer-specified Service Level Objectives, which should ideally be defined rigorously. We, therefore, survey five QoS specification languages: Web Service Level Agreement (Keller & Ludwig, 2003), Web Service Agreement Specification (Andrieux et al., 2005), SLAng (Lamanna, Skene, & Emmerich, 2003), Web Service Offerings Language (Tosic, Patel, & Pagurek, 2002), and Performance Trees (Suto, Bradley, & Knottenbelt, 2006), and compare them in order to find a QoS specification language which is suitable for our framework.

Probabilistic data structures are an effective means which is capable of considerably saving primary memory. As a result, our framework may adopt probabilistic data structures as the underlying data structure when sub-100% reliability requirements and limited functionalities are allowed. We will review the most widely-used probabilistic data structure: Bloom filters (Bloom, 1970), followed by their variants, counting Bloom filters (Fan, Cao, Almeida, & Broder, 2000), compressed Bloom filters (Mitzenmacher, 2001), scalable Bloom filters (Almeida, Baquero, Preguiça, & Hutchison, 2007), and sparse Bloom filters (Knottenbelt, 2000).

We will present an introduction to cloud storage, including private, public, hybrid, and personal cloud storage, and challenges to implement software supporting cloud storage. This chapter will also discuss how to overcome these challenges through the exploitation of our framework.

Next, we classify third-party libraries which enhance the functionality of Standard Template Library (STL) and Java Collections Framework (JCF) in an effort to deal with large-scale data. STL and JCF provide general-purpose containers, which suffer from serious performance deterioration and primary memory shortage when a large amount of data is input. Hence, many container libraries are implemented so as to boost performance and reduce memory consumption. Furthermore, some of them have the ability to self-optimise performance. We will analyse these libraries to identify their abilities, which enables our framework to properly cooperate with them and to provide wider classes of SLOs.

Finally, we conclude this chapter by reviewing existing language extensions and reference models utilised for building self-adaptive systems as well as the techniques of dynamically changing data structures.

2.2 Autonomic Computing

The term autonomic computing derives from the human autonomic nervous system, which can unconsciously control human bodies (e.g. heart rate, salivation, perspiration) and was proposed by IBM (Horn, 2001). It is aimed at reducing the complexity of a rapidly growing system via self-management. As a result, an autonomic computing system should exhibit the properties of self-configuration, self-healing, self-optimisation, and self-protection (so-called self-CHOP) (Murch, 2004), which are defined as follows:

- Self-configuration: The ability of a system to automatically adjust itself.
- Self-healing: The ability to detect, analyse, and repair faults.
- Self-optimisation: The ability to evaluate the current performance and attempt to keep improving it with respect to assigned requirements.
- Self-protection: The ability to be aware of potential threats in order to defend against them.

As shown in Figure 2.1, an autonomic computing system is a collection of autonomic elements each of which consists of autonomic managers and managed resources (Kephart & Chess, 2003). Managed resources may entail hardware resources (e.g. storage, CPU) or software resources (e.g. database). An autonomic manager is responsible for monitoring and controlling managed resources through sensors and effectors, respectively. Furthermore, an autonomic manager should contain a control loop to manage resources. The most well-known control loop is IBM's MAPE-K (Monitor, Analyse, Plan, Execute, and Knowledge) control loop (IBM, 2003; Kephart & Chess, 2003), where Monitoring component observes and captures data from managed resources via sensors, Analysing component performs data analysis in accordance with information reported from Monitoring component, Planning component receives change requests sent from Analysing component and selects a suitable action to adjust managed resources, Executing component conducts the actions suggested by Planning component through effectors, and Knowledge component stores data related to metrics, logs, symptoms, and policies shared by the other components. MAPE-K has been implemented in several projects such as Autonomic Toolkit (IBM, 2005), ABLE (Bigus, Schlosnagle, Pilgrim, Mills, & Diao, 2002), and Kinesthetics eXtreme (Kaiser, Parekh, Gross, & Valetto, 2003).

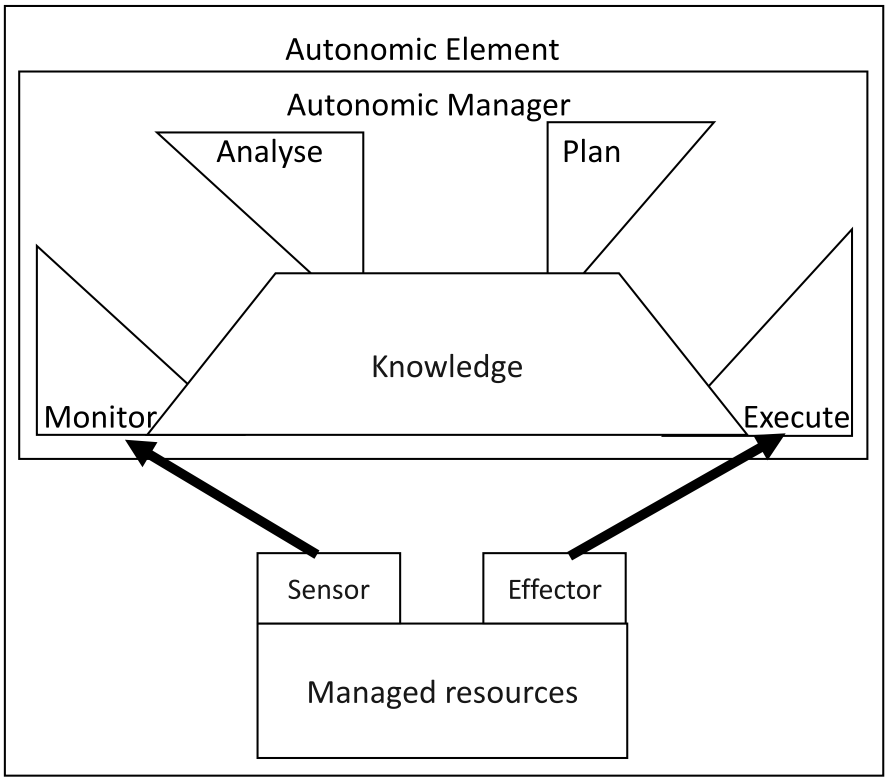


Figure 2.1: The architecture of autonomic elements (IBM, 2003; Kephart & Chess, 2003)

Several studies have been conducted to categorise autonomic computing systems. We will classify our framework through these studies. After the concept of autonomic computing was proposed, IBM introduced an Autonomic Computing Adoption Model (IBM, 2003), which divides systems into five levels according to the degree of autonomic capability. The first level is Basic level, which requires professional staff to manage systems. The second level is Managed level, where professional staff can utilise efficient ways to manage systems. In other words, management effort is significantly reduced. The third level is Predictive level, where system behaviour patterns are recognised for manually predicting suitable configurations and suggesting an action. The fourth level is Adaptive level, where human intervention is minimised due to the analysis and actions in response to environment changes being automatically taken by systems. The final level is Autonomic level, where systems dynamically adapt to satisfy Service Level Objectives without external intervention. In the context of IBM's model, software utilising our framework meets the definition of Autonomic level because it can adapt itself according to specified Service Level Objectives.

Huebscher and McCann's (Huebscher & McCann, 2008) define four elements of autonomicity: Support, Core, Autonomous, and Autonomic. The Support element refers to a work whose self-adaptation only focuses on part of its component or functionality. The Core element represents that the core application of an system is driven by a self-adaptive mechanism. In addition, systems falling into this category cannot accept high-level policies. The Autonomous element involves intelligence and agent-based technologies. Systems falling into this category can adapt themselves to handle failures but they do not measure performance or adjust themselves to achieve performative goals. The Autonomic element considers high-level objective specification (e.g. business goals, Service Level Agreements). Based on this classification, our framework falls into Autonomic category due to the consideration of Service Level Objectives.

Salehie and Tahvildari (Salehie & Tahvildari, 2009) propose a taxonomy to categorise research related to autonomic computing. In their point of view, self-adaptive systems can be divided into Primitive level, Major level, and General level, which can be seen in Figure 2.2, based on their self-* properties. Systems falling into the Primitive level have the ability to monitor environments and contexts but cannot manage themselves. Systems falling into the Major level embody a subset of self-CHOP properties. Systems categorised in the General level contain all self-* properties. Based on this hierarchy, a system is further classified by answering W5H1 (i.e. where, when, what, why, who, and how) questions, e.g. which part of the system can be changed? (*where*), when can the system be changed? (*when*), what resources can be adjusted? (*what*), why is self-adaptation needed? (*why*), who is responsible for system changes? (*who*), and how is the adaptation applied? (*how*), which help programmers develop self-adaptive mechanisms in the development phase and enable software to adapt itself in the operating phase. Self-adaptation in the operating phase is carried out by a cycle composed of Monitoring, Analysing, Planing, and Executing element. Monitoring element asks questions related to *where*, *when*, *what*. *When* an adaptation should be taken is asked by Analysing element, *what* resources have to be changed and *how* to change them are answered by Planning element. Finally, *how*, *when*, and *what* to change are asked by Executing element. According to this classification method, software adopting our framework falls into the Major level because it can automatically optimise and configure itself.

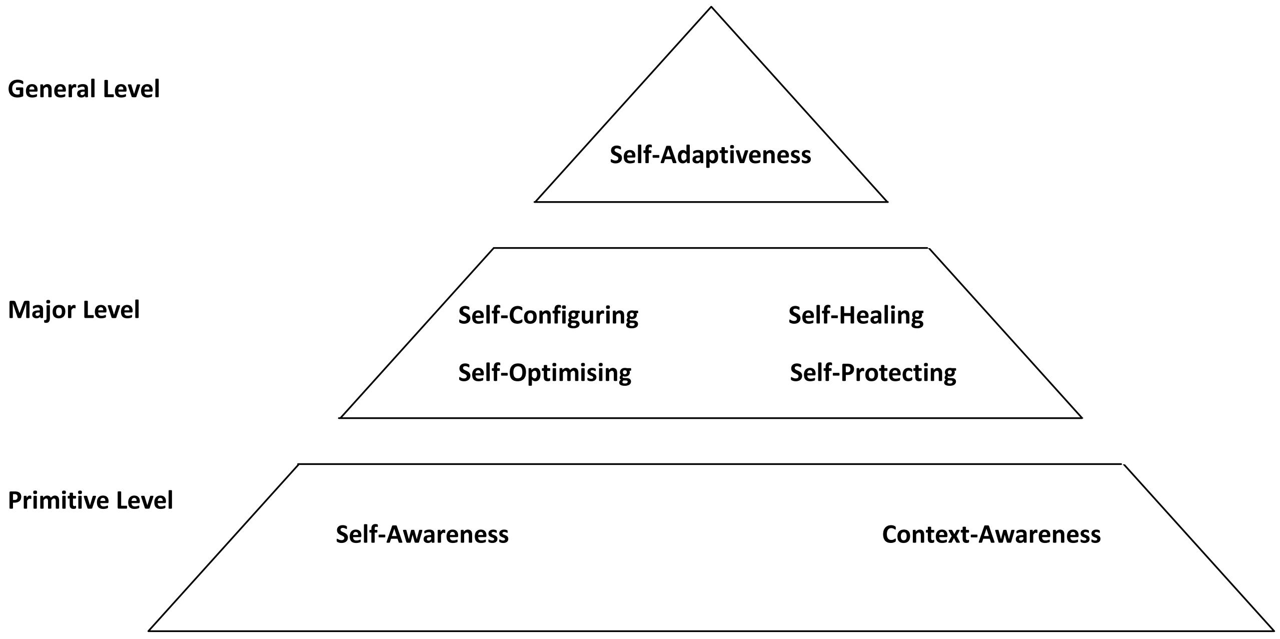


Figure 2.2: The hierarchy view of self-adaptive systems (Salehie & Tahvildari, 2009)

Khalid et al. (Khalid et al., 2009) review existing autonomic computing frameworks, architectures, infrastructures, and techniques. The existing methodologies are categorised into biologically-inspired, large-scale distributed, agent-based, component-based, technique-focused, service-oriented, and non-autonomous-system-specific frameworks or architectures. For biologically-inspired methodologies, their self-adaptive mechanisms mimic biological systems. Large-scale distributed methods focus on building large-scale distributed databases and systems which are able to manage themselves. Agent-based methodologies break systems into various agents each of which can manage itself and communicate with other agents. Component-based methodologies achieve self-adaptation by means of reconfiguration of components. Technique-focused methodologies utilise artificial intelligence and control theory to build self-adaptive systems. Service-oriented methodologies refer to the exploitation of self-adaptive mechanisms in service-based applications such as web services. Non-autonomous-system-specific methodologies represent the injection of autonomic mechanisms into existing non-autonomic systems. In addition to the review of existing methodologies, techniques achieving self-CHOP properties (e.g. hot swapping, machine learning, control theory, etc.) are identified as well. Based on this classification, our framework makes use of component-based methodologies since components (i.e. data structures) are dynamically changed to meet specified Service Level Objectives.

Macías-Escrivá et al. (Macías-Escrivá et al., 2013) survey recent research to identify research issues and methods of developing autonomic computing systems. They categorise existing methodologies into six main approaches, four global tools and methods, as well as three specific tools and methods. The six main approaches include external control mechanisms, component-based software engineering, model-driven approaches, nature-inspired engineering, multiagent systems, and feedback systems. The external control mechanisms refer to utilisation of the methodology which splits the component managing self-adaptation from existing systems. These mechanisms are suitable for software whose source code is missing. The component-based software engineering designs components which can be replaced at run time, utilises numerical metadata to find the most suitable component, and integrates caching techniques to reduce the time of finding candidate components. The model-driven approaches make use of abstract models for self-adaptation and then perform transformation from abstract models to code. The nature-inspired engineering, which refers to the biologically-inspired methodologies in Khalid et al.'s classification, derives from natural and biological systems. Multiagent systems, which utilise agent-based technologies, support self-adaptation through agents which can cooperate with other agents to complete their jobs. The feedback systems exploit control loops to fulfil the functionality of self-adaptation. Based on their classification, our framework exploits feedback loop approach because it adopts a control loop to monitor resources, analyse operation profile, and perform adaptations. In addition, we also adopt component-based software engineering to dynamically change components. The four global tools and methods, which are divided into models, simulation, architecture, and frameworks, are general-purpose methodologies for implementing whole self-adaptive systems. The three specific tools and methods, which are categorised into feedback control loops, decision-making, and requirements-engineering, only support parts of self-adaptation such as monitoring or planning.

In Krupitzer et al.'s work (Krupitzer et al., 2014), existing research is reviewed and a taxonomy is proposed. Their taxonomy, which is similar to Salehie and Tahvildari's classification methodology, comprises five dimensions, Time (corresponding to *When* question), Reason (corresponding to

Why question), Level (corresponding to *Where* question), Technique (corresponding to *What* question), and Adaptation Control (corresponding to *How* question). Since self-adaptation is performed by self-adaptive mechanisms, *Who* question is not asked. In addition, their survey classifies existing approaches through the adaptation logic, which is a process of controlling resources through monitoring, analysing, and adjusting. Based on the nature of the adaptation logic, approaches for building self-adaptive systems are model-based, architecture-based, reflection-based, programming paradigms, control theoretic, service-oriented, agent-based, nature-inspired, formal modelling and verification-based, learning-based, requirements engineering-based, or task-based approaches. Compared to Macías-Escrivá et al.'s survey, Krupitzer et al.'s work does not include external control mechanism, component-based software engineering, and feedback systems. Specifically, the external control mechanism is included in the architecture-based approaches. The component-based software engineering approaches are broken into architecture-based approaches and programming paradigms. The feedback systems fall into control theory. Furthermore, new categories, namely reflection-based approaches, programming paradigms, service-oriented approaches, formal modelling and verification-based approaches, are considered. The reflection-based approaches refer to software which has the ability to monitor (so-called introspection) and modify (so-called intercession) its architecture or behaviour at run time. The programming paradigms represent the development of self-adaptive systems via the utilisation of programming approaches, which are not originally designed for building of such systems. The service-oriented approaches make use of services, each of which is an autonomous unit designed for a specific task, to build self-adaptive systems. The formal modelling and verification-based approaches refer to the utilisation of formal methods to guarantee behavioural or structural correctness of self-adaptive systems. One approach adopting formal methods to achieve self-adaptation is presented by Aidarov, Ezhilchelvan, and Mitrani, 2013, who make use of queueing models in an effort to minimise energy consumption and maximise performance in service provisioning environments. Furthermore, one of the approaches exploiting formal methods to verify the behaviour of self-adaptation is proposed by Schaeffer-Filho, Lupu, Sloman, and Eisenbach, 2009. They introduce a formal model to verify policies and functionality of self-managed cells via the Alloy analyser (D. Jackson, 2002).

2.3 Containers

Containers are abstract data types whose instances hold a collection of objects. They are supported in many programming languages (e.g. C++, Java, Python) and implemented in various frameworks and libraries (e.g. Standard Template Library, Java Collections Framework, AS3Commons Collections Framework, and .NET's System.Collections) through the use of template classes in order to store different data types and supply either member functions or iterators to access stored objects. In this section, we will briefly introduce the most-commonly used container libraries, Standard Template Library (Josuttis, 2012), Java Collections Framework (Watt & D. Brown, 2001), and Python.

Standard Template Library (STL) is a C++ container library, whose supported containers are shown in Table 2.1. As can be seen, STL consists of sequence containers, associative containers, unordered associative containers and container adaptors. Sequence containers support the functionality of sequential access. Associative containers sort stored objects according to a predefined order (e.g. ascending order). Unordered associative containers utilise hash tables to store objects. Container adaptors are interfaces on top of sequence containers. In other words, sequence containers can be exploited to implement the functionality of a certain container adaptor. For example, the functionality of *queue* can be implemented through *deque* or *list*.

Containers in Java Collections Framework (JCF) are composed of interfaces and implementations. The separation between interfaces and implementations allows programmers to implement different methods for the same interface. Similar to STL, JCF supports generics, which enables the same container to store different data types without duplicate code. The interfaces provided by JCF are displayed in Table 2.2, which shows that JCF's containers are classified as Collection and Map. The former provides single-value containers, and the latter provides key-value containers.

Python is a programming language which aims to build more readable software with fewer lines of code compared to other languages such as C++. It provides three types of containers, which are exhibited in Table 2.3. The first type is the sequence type, which consists of *list*, *bytearray*, *str*, *bytes*, and *tuple*. The difference between *list* and *tuple* is the ability of modifying contents. Content modification is not allowed in *tuple*, which enables *tuple* to be used as keys in the mapping type. The

Sequence Container	
Name	**Description**
array	statically allocated continuous array
vector	dynamically allocated continuous array
deque	double-ended queue
forward_list	singly-linked list
list	doubly-linked list
Associative Container	
Name	**Description**
set	collection of unique keys, sorted according to predefined orders
map	collection of key-value pairs, sorted by unique keys according to predefined orders
multiset	collection of keys, sorted according to predefined orders
multimap	collection of pairs of keys and values, sorted by keys according to predefined orders
Unordered Associative Container	
Name	**Description**
unordered_set	collection of unique keys, sorted by hashes of keys
unordered_map	collection of key-value pairs, sorted by hashes of keys
unordered_multiset	collection of keys, sorted by hashes of keys
unordered_multimap	collection of key-value pairs, sorted by hashes of keys
Container Adaptors	
Name	**Description**
stack	LIFO functionality
queue	FIFO functionality
priority_queue	data is stored and manipulated according to priorities

Table 2.1: The containers supported by Standard Template Library

Collection		
Interface	**Implementation**	**Description**
Set	HashSet	Set implementation which stores objects in hash tables
	TreeSet	Set implementation which stores objects in red-black trees according to their values
	LinkedHashSet	Set implementation utilising hash tables where objects can be traversed through linked lists according to their insertion orders
List	ArrayList	Resizable array implementation
	LinkedList	Doubly-linked list
Queue	LinkedList	FIFO queue
Deque	LinkedList	Double-ended queue
Map		
Interface	**Implementation**	**Description**
Map	HashMap	Hash table storing key-value pairs and accepting null keys and values
	TreeMap	Red-black tree implementation where objects are sorted by their keys
	LinkedHashMap	Implementation combining hash tables and linked list, whereby elements can be traversed according to their insertion orders through the linked lists

Table 2.2: The containers supported by the Java Collections Framework

second type is the mapping type (i.e. key-value stores), which contains *dist*. Elements stored in *dist* can have different types, but all of them are unique. In other words, if a duplicate element is inserted, the earlier stored element will be replaced. The final type is the set type, which is composed of *set* and *frozenset*. They are similar containers, but elements in *frozenset* cannot be modified.

2.4 QoS Specification Languages

The concept of Quality of Service originates from networking and has now been applied in many domains to identify the responsibility of involved parties, properties of services, desired levels of these properties, and consequences if the levels are not met. To clearly specify QoS, involved parties should sign a Service Level Agreements (SLA) contract. An SLA entails properties of services, metrics of these properties (i.e. Service Level Indicators), desired levels of these properties (i.e. Service Level Objectives), and consequences when objectives are violated. SLAs are frequently confused with SLOs. Hence, we use the following description given by John Wilkes to explain the difference between an SLA and an SLO:

Sequence Type	
Name	**Description**
list	A list which can store mixed data types
bytearray	A sequence of bytes whose content can be modified
str	A character string whose content cannot be modified after it is declared
bytes	A sequence of byte whose content cannot be modified after it is declared
tuple	Similar to list but its content cannot be modified
Mapping Type	
Name	**Description**
dict	An associative array which stores pairs of key and values
Set type	
Name	**Description**
set	A collection of elements which does not have duplicate elements and can contain mixed data types
frozenset	set whose content cannot be modified

Table 2.3: The containers supported by Python

‘Why do people keep talking about "SLA violations"? That makes no sense: SLA stands for Service Level Agreement – i.e., an agreement, or contract, that includes a service level, not just a specific agreed-to-service level – the latter is much better called an SLO, or Service Level Objective. (The SLA adds things like how much you will pay for obtaining that level of service, or penalties if the provider doesn’t do so.)’

John Wilkes, Principal Software Engineer, Google

Many QoS specification languages have been proposed to help involved parties formally define their services. This section will present five well-known QoS specification languages: Web Service Level Agreement (Keller & Ludwig, 2003), Web Service Agreement Specification (Andrieux et al., 2005), SLAng (Lamanna et al., 2003), Web Service Offerings Language (Tosic et al., 2002), and Performance Trees (Suto et al., 2006).

Web Service Level Agreement (WSLA) (Keller & Ludwig, 2003) is an XML-based language used for monitoring and measuring QoS parameters and negotiating Service Level Agreements. The XML schema and structure of WSLA are shown in Figure 2.3. As can be seen, WSLA consists of a Parties section, a ServiceDescription section, and an Obligations section. The Parties section involves groups divided into signatory parties and supporting parties. Signatory parties (i.e. service providers and service customers) are supposed to sign the SLA. Supporting parties supply signatory parties with measurement and condition evaluation services. The ServiceDefinition section specifies properties of various services. Each property is mapped to an *SLAParameter* and a *metric*. A *metric* may specify either a way to measure a source through the definition of *MeasurementDirective* or a way to compute the metric through a *Function*. The Obligations section defines Service Level Objectives and actions to guarantee them. A Service Level Objective is a commitment of the maintenance of a certain service level for a period of time. It specifies the target property and service level via a *SLAParameter* and a *Predicate* (i.e. greater than, equal, less than, etc.), respectively. An action guarantee defines activities (e.g. a particular notification or control activity) to be taken when its corresponding Service Level Objective is not met.

```
<xsd:complexType name="WSLAType">
   <xsd:sequence>
      <xsd:element name="Parties" type="wsla:PartiesType"/>
      <xsd:element name="ServiceDefinition"
                  type="wsla:ServiceDefinitionType"
                  maxOccurs="unbounded"/>
      <xsd:element name="Obligations" type="wsla:ObligationsType"/>
   </xsd:sequence>
</xsd:complexType>

<xsd:element name="SLA" type="wsla:WSLAType"/>
```

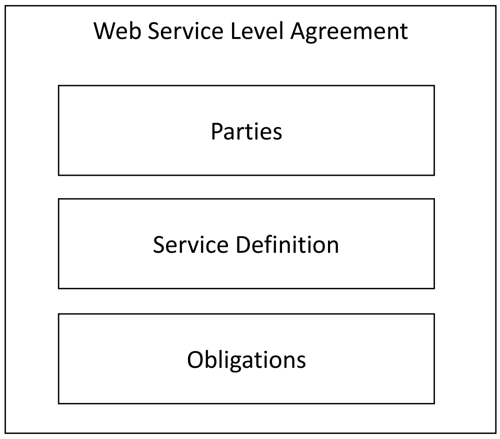


Figure 2.3: The XML schema and structure of Web Service Level Agreement (Ludwig, Keller, Dan, King, & Franck, 2003)

Web Service Agreement specification (WS-Agreement) (Andrieux et al., 2005), whose architecture and XML schema are shown in Figure 2.4, is an XML schema whose objective is to create guarantee terms between service providers and service customers during service provisioning. It comprises an optional Name section, a Context section, and a Terms section. The Context section, which is similar to the Parties section of WSLA, presents the involved participants and the lifetime of the agreement. The Terms section, consisting of Service Terms and Guarantee Terms, defines a commitment to the levels of services. The Service Terms express the target services via *Service Descriptions*, the references of the target services via *Service References*, and the measurable properties of the services via *Service Properties*. Each Guarantee Term contains an obligated party, the scope this guarantee applies to, any number of Service Level Objectives, and business values associated with the service (e.g. the importance of Service Level Objectives).

```
<wsag:Agreement AgreementId="xs:string">
  <wsag:Name>
    xs:string
  </wsag:Name> ?
  <wsag:AgreementContext>
    wsag:AgreementContextType
  </wsag:AgreementContext>
  <wsag:Terms>
    wsag:TermCompositorType
  </wsag:Terms>
</wsag:Agreement>
```

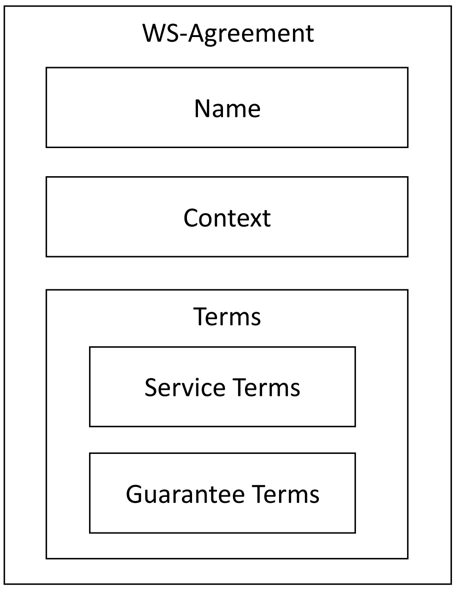


Figure 2.4: The XML schema and structure of Web Service Agreement (Andrieux et al., 2005)

SLAng (Lamanna et al., 2003) is also an XML-based language for describing QoS properties of SLAs. As can be seen in Figure 2.5, it specifies four vertical SLAs and three horizontal SLAs. These SLAs are identified through the involved parties (e.g. Application, Web Service, Component, Container, Storage, and Network). Each SLA is composed of an ID (the identification of the SLA), a server (the responsibilities of the server), a client (the responsibilities of the client), and a mutual (the responsibilities of both server and client). The relationships between the type of SLAs and involved parties are shown in Table 2.4.

SLA Type	**Server**	**Client**
Application	Web Services or Applications	Components
Hosting	Containers	Components
Persistence	Storage	Containers
Communication	Network	Containers
Service	Components	Web Services
Container	Containers	Containers
Networking	Network	Network

Table 2.4: The relationships between the involved parties and the types of Service Level Agreements in SLAng

The major functionality of SLAng is similar to other SLA specification languages, but it claims three differences. First, it can be applied not only to web services but also to domains such as Internet Service Provision, Application Service Provision, and Storage Service Provision. Second, SLAng is designed with practicality and monitorability in mind, such that constraints can only be placed on activities that can be observed by contracting parties. Third, the formally defined semantics of SLAng, which can check the consistency of SLAs, focus on service and client behaviour.

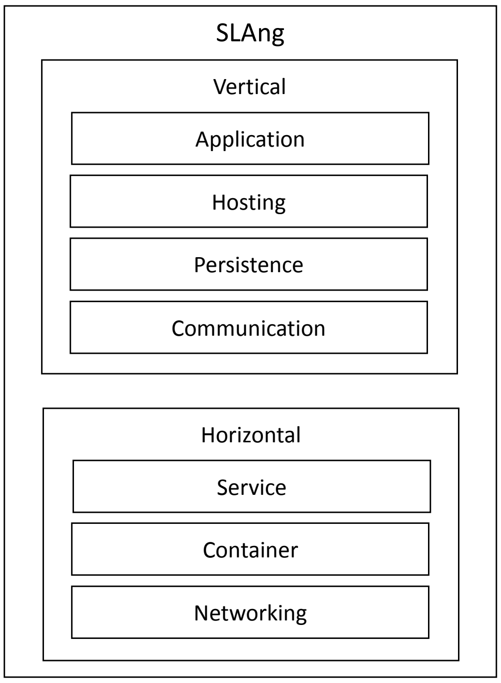


Figure 2.5: The structure of SLAng (Lamanna, Skene, & Emmerich, 2003)

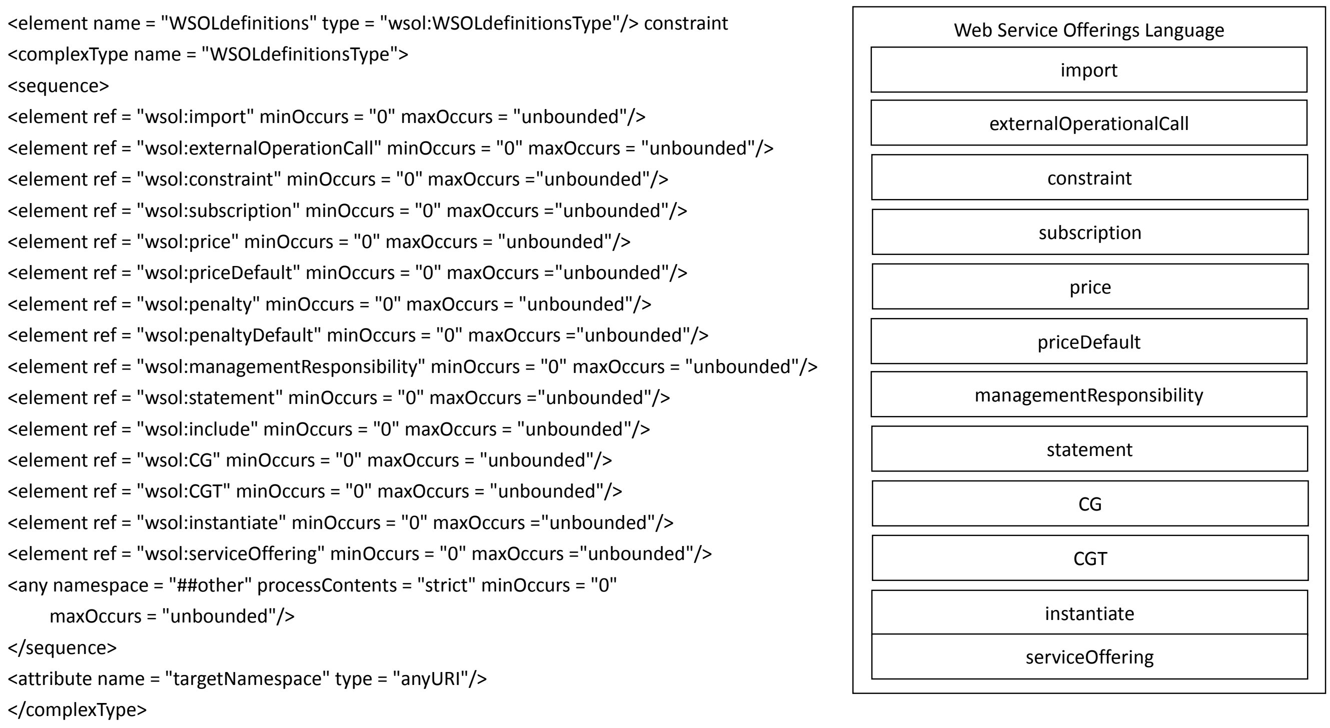
Web Service Offerings Language (WSOL) (Tosic et al., 2002) is an XML-based specification language for service orderings, whose XML schema and structure are depicted in Figure 2.6. As can be seen, there are 12 elements, *import*, *externalOperationCall*, *constraint*, *subscription*, *price*, *priceDefault*, *penalty*, *penaltyDefault*, *managementResponsibility*, *statement*, *include*, *CG*, *CGT*, *instantiate*, and *serviceOffering*, which are described as follows:

- *import* is used to include any number of WSOL files.
- *externalOperationCall* lays down the format of external services, including input parameters, return values, and service names.
- *constraint* specifies any number of functional constraints (e.g. pre-conditions, post-conditions, and future-conditions), non-functional (i.e. QoS) constraints, and access rights.
- *subscription* specifies the duration and price of a subscription paid by a service customer to a service provider.

- *price* defines names and domains of particular services.
- *priceDefault* defines default prices of particular services.
- *penalty* refers to the money paid due to violation of SLAs.
- *penaltyDefault* refers to the default amount of money paid due to violation of SLAs.
- *managementResponsibility* draws up the responsibilities among involved parties.
- *statement* defines schemas for corresponding statements (i.e. the external operation call statement, the subscription statement, the price statement, the price default statement, the management responsibility statement, the include statement, the instantiate statement).
- *include* is used to include other constructs such as *constraint*.
- *CG* represents Constraint Group, which assembles constraints into groups.
- *CGT* is similar to *CG* but contains additional parameters.
- *instantiate* specifies the instantiate relationship between a *CGT* and a *CG*.
- *serviceOffering* defines the format of service offerings.

Compared with the above-mentioned QoS specification languages, Performance Trees (Suto et al., 2006) are an alternative which provides a more intuitive and graphical way of defining QoS parameters with respect to performance. A performance query is transformed into a tree structure consisting of nodes and connecting arcs. There are two types of nodes: operation nodes and value nodes. The former represent performance-related functions, and the latter are used to store literal information associated with the performance query (e.g. set of states, actions, numerical/Boolean constants). Figure 2.7 shows an example of how to express a performance query, the expected amount of time expended from one state to another, in the form of a performance tree. In addition, the structure of Performance Trees can be generated and analysed via the use of *PIPE2* (Dingle, Knottenbelt, & Suto, 2009) software tool.

In summary, WSLA concentrates on web service interactions. It helps involved parties to arrange resources at deployment time and to monitor Service Level Objectives and to detect violations at run



```
<element name = "WSOLdefinitions" type = "wsol:WSOLdefinitionsType"/> constraint
<complexType name = "WSOLdefinitionsType">
<sequence>
<element ref = "wsol:import" minOccurs = "0" maxOccurs = "unbounded"/>
<element ref = "wsol:externalOperationCall" minOccurs = "0" maxOccurs = "unbounded"/>
<element ref = "wsol:constraint" minOccurs = "0" maxOccurs ="unbounded"/>
<element ref = "wsol:subscription" minOccurs = "0" maxOccurs ="unbounded"/>
<element ref = "wsol:price" minOccurs = "0" maxOccurs = "unbounded"/>
<element ref = "wsol:priceDefault" minOccurs = "0" maxOccurs ="unbounded"/>
<element ref = "wsol:penalty" minOccurs = "0" maxOccurs = "unbounded"/>
<element ref = "wsol:penaltyDefault" minOccurs = "0" maxOccurs ="unbounded"/>
<element ref = "wsol:managementResponsibility" minOccurs = "0" maxOccurs = "unbounded"/>
<element ref = "wsol:statement" minOccurs = "0" maxOccurs ="unbounded"/>
<element ref = "wsol:include" minOccurs = "0" maxOccurs = "unbounded"/>
<element ref = "wsol:CG" minOccurs = "0" maxOccurs = "unbounded"/>
<element ref = "wsol:CGT" minOccurs = "0" maxOccurs = "unbounded"/>
<element ref = "wsol:instantiate" minOccurs = "0" maxOccurs ="unbounded"/>
<element ref = "wsol:serviceOffering" minOccurs = "0" maxOccurs ="unbounded"/>
<any namespace = "##other" processContents = "strict" minOccurs = "0"
     maxOccurs = "unbounded"/>
</sequence>
<attribute name = "targetNamespace" type = "anyURI"/>
</complexType>
```

Figure 2.6: The XML schema and structure of Web Service Offerings Language (Tosic, Patel, & Pagurek, 2002)

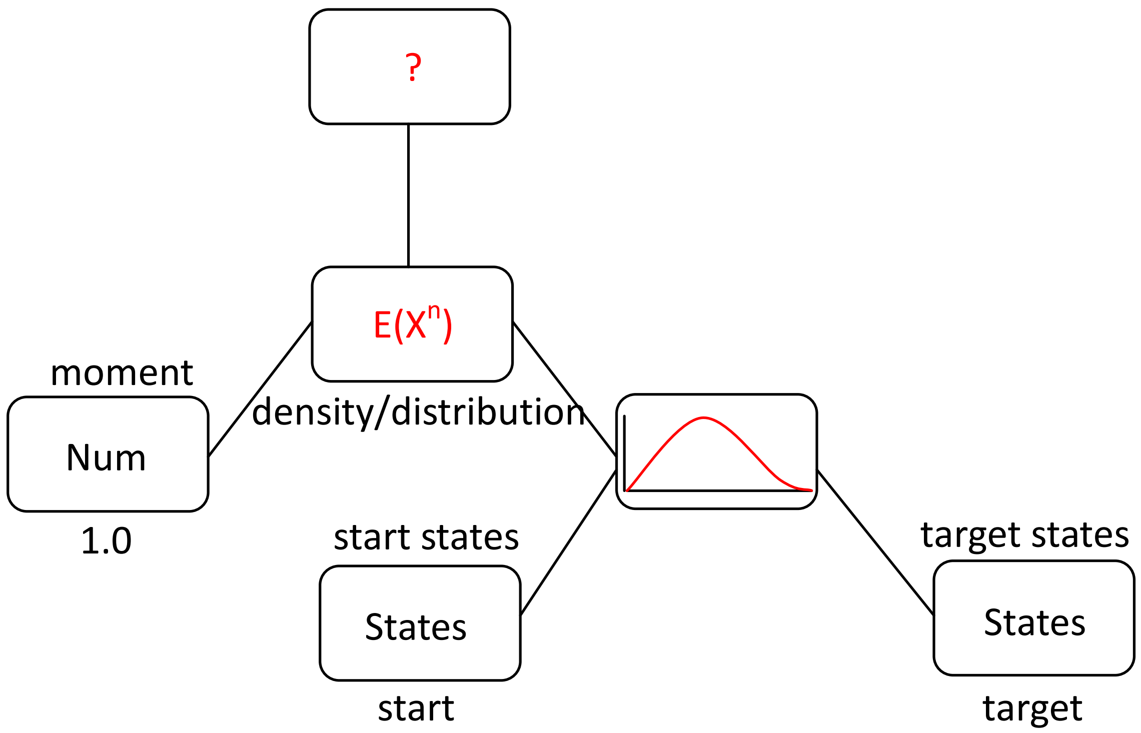


Figure 2.7: An example of how to express a performance query in the form of a performance tree (Suto, Bradley, & Knottenbelt, 2006)

time. SLAng concentrates on both web service interaction and SLA specification for hosting service provisioning and communication service provisioning. WS-Agreement concentrates on definitions of protocols which express providers' capability, create Service Level Agreements, and monitor their compliance at run time. WSOL focuses on constraint specification and management information. Performance Trees concentrates on graphical specification of performance queries. Our framework chooses WSLA to specify QoS requirements for three reasons. First, WSLA is an XML-based standard, which makes QoS requirements more easily parsed and validated. Second, it can define desired resources, specify how to measure them, and compare the target resources with SLOs. Third, it does not have to define hard QoS guarantees because our framework attempts to fulfil QoS requirements but does not guarantee that they can be satisfied 100% of the time.

2.5 Probabilistic Data Structures

Probabilistic data structures refer to data structures which contain probabilistic factors, which may lead to wrong answers when an element is tested for membership. The most commonly-used probabilistic data structures are Bloom filters (Bloom, 1970), which provide insertion as well as test-for-membership operations and have been applied to Google's Bigtable (Chang et al., 2008) and Chrome (Geravanda & Ahmadib, 2013) to reduce disk access and represent URL blacklists, respectively. They are constructed by k hash functions (h_1, $h_2, \dots, h_k$ with range $[0, m-1]$) and an m-bit storage array (B) whose initial values are 0. With insertion of an element i, the positions of $h_1(i)$, $h_2(i), \dots, h_k(i)$ in the m-bit array are set to 1. When membership of an element j is tested, the element is considered to exist if and only if $\prod_{i=1}^{k} B[h_i(j)] = 1$. Because Bloom filters only make use of m-bit arrays, their memory consumption is not related to the number of inserted elements, which saves a huge amount of memory space. However, the main disadvantage of Bloom filters is the lack of deletion operations. Furthermore, as we have mentioned, Bloom filters may return wrong answers (so-called false positives), whose probability is approximately $(1 - e^{-kn/m})^k$ where n is the number of stored elements. The equation implies another two drawbacks of Bloom filters. First, they will become unreliable when too many elements are inserted. Second, their memory space has to be pre-allocated to achieve a desirable probability of false positives. However, it is difficult to predict how many elements will be inserted, which may waste memory space when the size of bit arrays is not properly configured. These issues trigger the invention of counting Bloom filters (Fan et al., 2000), scalable Bloom filters (Almeida et al., 2007), compressed Bloom filters (Mitzenmacher, 2001), and sparse Bloom filters (Knottenbelt, 2000).

Counting Bloom filters, which are proposed by Fan et al., 2000, aim to solve the issue of lacking deletion operations. They replace the m-bit arrays with an array (C) with m positions, each of which contains a counter initially set to zero. With insertion of an element i, 1 is added to the values of $C[h_1(i)], C[h_2(i)], \ldots, C[h_k(i)]$. Deletions are performed by deducting 1 from the values of $C[h_1(i)]$, $C[h_2(i)]$, ..., $C[h_k(i)]$. Although counting Bloom filters support deletion operations, they may also result in false negatives, which may cause more serious results than false positives. Hence Guo et al. attempt to find the root cause of this issue and propose a variant to minimise the probability of false negatives (Guo, Liu, Li, & Yang, 2010). In addition, counting Bloom filters consume more memory than standard Bloom filters do. As a result, several variants (Bonomi, Mitzenmacher, Panigrahy, Singh, & Varghese, 2006; Rottenstreich, Kanizo, & Keslassy, 2014) are invented to save memory.

Compressed Bloom filters (Mitzenmacher, 2001) compress standard Bloom filters' bit arrays in order to reduce transmission size across networks. From the perspective of networking, Bloom filters are not only data stored in local memory but messages transmitted across networks. The size of Bloom filters, therefore, becomes a critical issue. When Bloom filters are treated as data, the optimal number of hash functions should be computed to achieve a desirable probability of false positives. The formula, $(m/n)\mathrm{ln}2$, is utilised to get an optimal value of k. However, this optimal number of hash functions makes the bit arrays less effective (Adler, Chakrabarti, Mitzenmacher, & Rasmussen, 1995). To benefit from compression, compression ratio should be taken into consideration. Hence the probability of false positives is changed to

$$f = (1-p)^{\frac{-z\mathrm{ln}p}{nH(p)}} \tag{2.1}$$

where n is the number of stored elements, z is the compressed size, p is the probability with which each bit in the bit array is 1, and

$$H(p) = -p \log_2 p - (1-p)\log_2(1-p). \tag{2.2}$$

Scalable Bloom filters (Almeida et al., 2007) aim to prevent the need for allocating a fixed amount of memory space in advance. Each scalable Bloom filter consists of a list of Bloom filters (BF_0, BF_1, …, BF_i with h_1, h_2, …, h_i hash functions, respectively). When the maximum probability of false positives is met, a new Bloom filter is created and inserted into the list in accordance with the following rule. The false positive of the new Bloom filter should be reduced to $P_i * r$ where P_i is the probability of B_i's false positive and r is the tightening ratio with $0 < r < 1$. As a result, if the current number of Bloom filters is l, their maximum false positive rates are P_0, $P_0 * r$, …, $P_0 * r^{l-1}$, respectively. Their compound false positive rate is $1 - \prod_{i=0}^{l-1}(1 - P_0 r^i)$. When a test-for-membership operation is invoked, every existing Bloom filter will be checked. If any filter returns a positive result, the operation will also give a positive result.

To improve Bloom filters' reliability and prevent the need of memory preallocation, Knottenbelt (Knottenbelt, 2000) introduces sparse Bloom filters, which utilise two independent hash functions and a forest of AVL trees. The first hash function determines which AVL tree should be used to store elements, and the second hash function generates hash keys. With the insertion of an element i, it is input to the first hash function to decide which AVL tree it should be inserted into and then input to the second hash function to obtain the corresponding hash key. Finally, the key is inserted into the corresponding AVL tree. In this thesis, we will further improve sparse Bloom filters to dynamically adjust the number of AVL trees and choose the improved sparse Bloom filters as one potential data structure when reliability requirements are less than 100% and where functionality specification allows.

2.6 Cloud Storage

Cloud storage refers to a data storage model where data is stored in distributed storage devices and accessible at any place. This data storage provides high accessibility and protection of data backup.

Despite these benefits, cloud storage highly depends on and consumes network bandwidth. Consequently, if the internet connection is unstable or slow, the stored data may not be accessible. This barrier is possible to be overcome as network bandwidth is broadened and connectivity becomes ubiquitous.

Cloud storage can be categorised into public cloud storage, private cloud storage, and hybrid cloud storage. The first type of cloud storage, public cloud storage, represents that data owned by a company or an individual is stored in and managed by a hosting company e.g. Amazon, Google, Dropbox, Microsoft. Benefits of public cloud storage include low data maintenance, low storage costs, and high scalability. Since data is managed by storage providers, purchasing storage hardware is not needed. Additionally, storage customers can easily scale up and down the storage space they need. However, public cloud storage is not suitable for the data which is confidential due to security concern or frequently updated due to performance concern. Another concern of public cloud storage is the lack of standards among different storage providers, which increases the difficulty of changing storage providers. As can be seen in Table 2.5, features and prices vary among storage providers. The variety of public cloud storage services complicates the implementation of transferring data between storage providers and customers. To ease this difficulty, our framework supplies an abstract layer which enables programmers to easily integrate software APIs supporting data transfer of different storage providers.

The second type of cloud storage, private cloud storage (also called internal cloud storage), is similar to public cloud storage in that the aim is to provide scalable, flexible, and location-independent data storage; however, stored data is not publicly accessible. Data stored in private cloud storage is kept behind the firewall and can be accessed only from the local network or by some secure means over the Internet. This enhances data security and performance and shifts the responsibility of data management from storage providers to data owners. Recent years have seen the popularity of such cloud storage implemented through Network-Attached Storage (NAS) devices. NAS provides easy data backup, file sharing, and remote access control. Furthermore, it enables data owners to have full access control of their data and to easily scale up storage space. Many hardware manufacturers have introduced their NAS devices, e.g. Western Digital's My Cloud, Seagate's Central, D-Link's ShareCenter, and NetGear's ReadyNAS.

	Amazon Cloud Drive	**Google Drive**	**Dropbox**	**Microsoft OneDrive**
File size restriction	2GB	10 GB	300 MB via browsers, no limit via app	10 GB
Free storage	5 GB	15 GB	2 GB	15 GB
Price per year (price in March 2015)	20 GB : £6	100 GB : £15	1 TB : £79	100 GB : £15
File sharing	Yes	Yes	Yes	Yes
File synchronisation	Yes	Yes	Yes	Yes
Desktop app	Windows MAC	Windows MAC	Windows Mac Linux	Windows MAC
Mobile app	Android iOS	Android iOS	Android iOS Blackberry OS Fire OS	Android iOS Windows
Web GUI	Yes	Yes	Yes	Yes

Table 2.5: The comparison of cloud storage services

The third type of cloud storage, hybrid cloud storage, is a combination of public cloud storage and private cloud storage. Confidential and frequently updated data is stored in private cloud storage. Non-confidential and rarely-updated data is stored in public cloud storage. The separation of data allows hybrid cloud storage to minimise data transfer latency between storage providers and customers and to increase security compared to public cloud storage.

2.7 Third-Party Container Libraries

As the scale of data rapidly increases, many container libraries have been invented to boost performance and reduce primary memory use via the techniques of out-of-core storage, probabilistic data structures, parallelism, and self-adaptation. This section will briefly introduce and categorise them according to the appearance order in Table 2.6.

- Standard Template Library for Extra Large Data Sets (STXXL) is a library implementing out-of-core containers and algorithms for processing massive input data.

- bloom is a container library supporting the functionality of standard Bloom filters and the technique of compressing bit arrays.
- Parallel Standard Template Library (Parallel STL) is implemented via high-performance C++ and provides distributed containers and parallel algorithms.
- Transparent Parallel I/O Environment (TPIE) supplies external algorithms and data structures in order to reduce the effort of implementing out-of-core storage.
- dablooms is an implementation of the combination of counting Bloom filters and scalable Bloom filters.
- Standard Adaptive Parallel Library (STAPL) is a C++ library developed at Texas A&M University for supporting parallelism in application development.
- Library of Efficient Data Types and Algorithms to Secondary Memory (LEDA-SM) is an extension of LEDA (Mehlhorn & Näher, 1995), which provides external data structures and algorithms for manipulating large input data.
- Intel Threading Building Blocks (TBB) is a library providing concurrent containers and algorithms. We will give more descriptions of Intel TBB in Chapter 5.3.
- Multi-Core Standard Template Library (MCSTL) is an STL-comparable parallel algorithm library which is capable of utilising multiprocessors or multicore of a processor with shared memory.
- Persistent Standard Template Library (Persistent STL) intends to replace STL's containers with those that can efficiently access data in external memory.
- Smart Data Structures are a group of parallel data structures which can dynamically improve their performance through a machine learning approach.
- Java Access to Generic Underlying Architectural Resources (Jaguar) provides external objects, which are stored outside of Java heap.

- Parallel Java 2 Library (PJ2) is a collection of Java interfaces and middlewares which support parallel programming.
- Guava is a Java-based library which is composed of several Google's core libraries. Furthermore, it supports the functionality of Bloom filters.
- Oracle Berkeley DB is a library embedded in a direct persistence layer, which enables fast object serialisation and deserialisation.
- Joafip is a Java library which manages persistent data in file systems.

Table 2.6 categorises them according to their enhanced functionality. As can be seen, STXXL, TPIE, LEDA-SM, and Persistent STL are C++-based implementations for out-of-core storage, and Jaguar, Oracle Berkeley DB, and Joafip are Java-based out-of-core storage implementations. bloom, dablooms, and Guava implement the functionality of Bloom filters. Parallel STL, STAPL, Intel TBB, Smart Data Structures, and PJ2 provide parallel containers. Furthermore, Smart Data Structures are capable of dynamically adjusting themselves to boost performance. Except for Smart Data Structures, most of these libraries are not able to be aware of QoS at run time. Even if Smart Data Structures attempt to automatically improve performance, their application area is restricted to distributed environments. However, these well-developed libraries can provide our framework with fundamental bases when the techniques of out-of-core storage, probabilistic data structures, and parallelism are required to be implemented. Through dynamic deployment of these libraries, our framework can exploit out-of-core storage and parallelism at low cost and provide a broader class of Service Level Objectives.

2.8 Related Contexts

This section will review the studies which make use of language extensions and reference models for building self-adaptive systems. In addition, the technique of dynamically changing data structures across different applications is described as well.

		Enhanced Functionality			
	Language	**Out-of-core Storage**	**Probabilistic Data Structures**	**Parallelism**	**Self-adaptation**
STXXL (Dementiev, Kettner, & Sanders, 2005)	C++	✓		✓	
bloom (Partow, 2000)	C++		✓		
Parallel STL (Johnson & Gannon, 1997)	C++			✓	
TPIE (Vengroff, 1994)	C++	✓			
dablooms (Hines, 2013)	C		✓		
STAPL (Buss et al., 2010)	C++			✓	
LEDA-SM (Crauser & Mehlhorn, 1999)	C++	✓			
Intel TBB (Intel, 2014)	C++			✓	
MCSTL (Singler, Sanders, & Putze, 2007)	C++			✓	
Persistent STL (Gschwind, 2001)	C++	✓			
Smart Data Structures (Eastep, Wingate, & Agarwal, 2011)	C++			✓	✓
Jaguar (Welsh & Culler, 2000)	Java	✓			
PJ2 (Kaminsky, 2014)	Java			✓	
Guava (Andreou & Bourrillion, 2014)	Java		✓		
Oracle Berkeley DB (Oracle, 2014)	Java	✓			
Joafip (Peuvrier, 2012)	Java	✓			

Table 2.6: The third-party libraries supporting the techniques of out-of-core storage, probabilistic data structures, parallelism, and self-adaptation

2.8.1 Language Extensions for Building Self-Adaptive Systems

Many programming paradigms e.g. meta programming (Abrahams & Gurtovoy, 2004), component-based programming (Heineman & Councill, 2001), aspect-oriented programming (Kiczales et al., 1997), generative programming (Czarnecki & Eisenecker, 2000), adaptive programming (Gouda & Herman, 1991), context-oriented programming (Hirschfeld, Costanza, & Nierstrasz, 2008) have been adopted for building self-adaptive systems. These paradigms are solidified through programming languages or language extensions, which will be described below.

Meta programming is the act of developing programs that manipulate other programs (including themselves) as input data (e.g. compilers, interpreters, lex, yacc, etc). To modify themselves, meta programs should have the ability of self-evaluation, which is called *reflection*. Because *reflection* supplies a way to evaluate and modify a program, self-adaptive mechanisms can perform adaptations through it. Some programming languages (e.g. C#, Lisp, Ruby, etc.) are equipped with this ability, but some programming languages (e.g. C++, Java) do not support this feature. As a result, their extensions are implemented to achieve this ability. An example of such extensions is Iguana/J (Dowling, Schäfer, Cahill, Haraszti, & Redmond, 1999), which is a Java extension for dynamically modifying programs.

Component-based programming (Heineman & Councill, 2001) divides software into individual and independent components to maximise software reuse. Since functionalities of a system are implemented in different components, self-adaptations are achieved by the way of dynamic replacements of components. Such actions are controlled by a component which determines when to trigger adaptations according to a pre-defined logic. Among language extensions using component-based programming to build self-adaptive systems is Julia (Bruneton, Coupaye, Leclercq, Quéma, & Stefani, 2006), which is a Java implementation of the component-based programming model.

Aspect-oriented programming (Kiczales et al., 1997) separates independent concerns in software. Each concern is developed and then composed to establish the whole software. Languages supporting aspect-oriented programming provide the specification of join points where program flows can be redirected to another concerns. The self-adaptation, therefore, can be accomplished through the transfer of concerns. An example of a programming language which conducts self-adaptation via this technique is JAsCo (Suvée, Vanderperren, & Jonckers, 2003), which is a Java-based aspect-oriented programming language.

Generative programming (Czarnecki & Eisenecker, 2000) refers to automatic generation of software through reusable components and configuration knowledge. In Nierstrasz et al.'s work (Nierstrasz, Denker, & Renggli, 2009), Reflexity, a platform for dynamic adaptation of software, and Diesel (Fowler, 2005), a tool for transforming domain specific languages into code, are utilised to build self-adaptive systems.

Adaptive programming (Gouda & Herman, 1991) is the act of building software which is capable of dynamically carrying out adaptations in order to respond to changes in input (e.g. environments, goals). One adaptive programming language is A^2BL (Simpkins, Bhat, Isbell, & Mateas, 2008), an Adaptive Behaviour Language , whose adaptivity is achieved through the exploitation of reinforcement learning (Sutton & Barto, 1998).

Context-oriented programming (Hirschfeld et al., 2008) refers to the method of expressing behavioural variation based on contexts. Since it is designed for performing run-time variations, its is suitable for the development of self-adaptive systems. Many programming languages extensions, e.g. Context/J (Hirschfeld et al., 2008), ContextErlang (Salvaneschi, Ghezzi, & Pradella, 2012), ContextL (Costanza & Hirschfeld, 2005), have been implemented to support context-oriented programming.

In addition to the above-mentioned language extensions, Stevens, Parsons, and King present an autonomic container which is capable of run-time configuration and testing to validate adaptations (Stevens, Parsons, & King, 2007). In their work, a prototype for the functionality of *stack* is implemented. As can be seen in Figure 2.8, it consists of an *ACApplication*, a *SysController*, and an *Autonomic Container*. The *ACApplication* refers to those applications utilising autonomic containers. The *SysController* makes use of threads to execute *ACApplications* and autonomic containers in parallel. The autonomic container includes a stack, a *SelfConfigAM*, which is used to configure the underlying data structure (i.e. the stack), and a *SelfTestManager*, which is responsible for monitoring the test process, analysing the test results, and deciding if the adaptation performed by the *SelfConfigAM* is allowed.

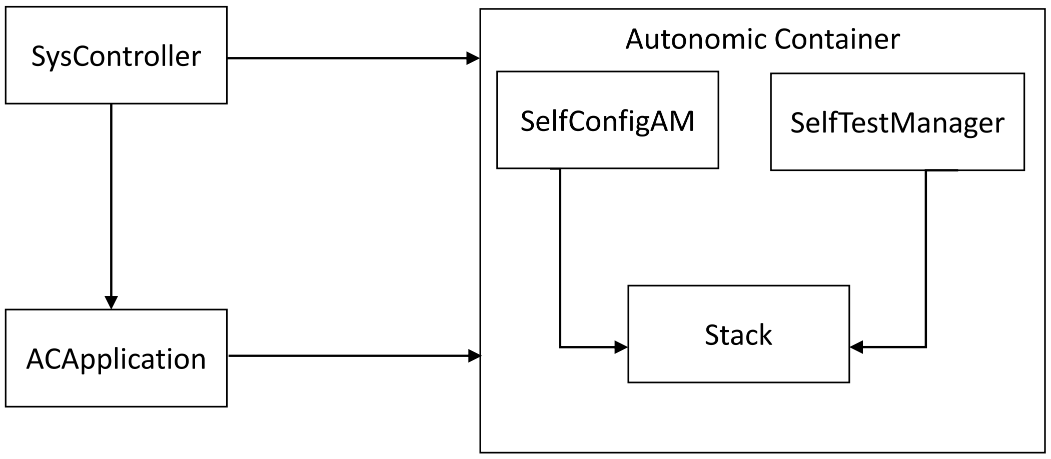


Figure 2.8: The architecture of the self-testing autonomic container (Stevens, Parsons, & King, 2007)

In summary, these language extensions are able to perform self-adaptations, but compared with our framework, they require higher programmer overhead in terms of substantial modification of existing code. Furthermore, they do not provide a mechanism for specifying Service Level Objectives.

2.8.2 Reference Models for Building Self-Adaptive Systems

This subsection will introduce some well-known models for building self-adaptive systems. Kramer and Magee (Kramer & Magee, 2007) present a three layer reference model which is inspired by robots. As can be seen in Figure 2.9, the bottom layer is the Component Control layer, which comprises components used to accomplish tasks. This layer is able to monitor components in an effort to report their status to the upper layer and to dynamically create, destroy and communicate with them in order to perform adaptations. Furthermore, it contains a feedback loop for low-level self-adjustment (e.g. reconfiguration of parameters). When the current configuration cannot meet a certain situation, this layer will inform the upper layer of this. The middle layer is the Change Management layer, which plans adaptations to deal with the situation reported from the lower layer (i.e. Component Control layer) or to achieve a new objective given by the upper layer. To quickly respond to state changes, this layer should activate predefined plans. If a plan is missing, this layer will request the upper layer to lay down the plan. In addition, when the upper layer lays down a new objective, a new plan will also be given. The highest layer is the Goal Management layer, which generates change plans for the requests sent from the lower layer and for new goals.

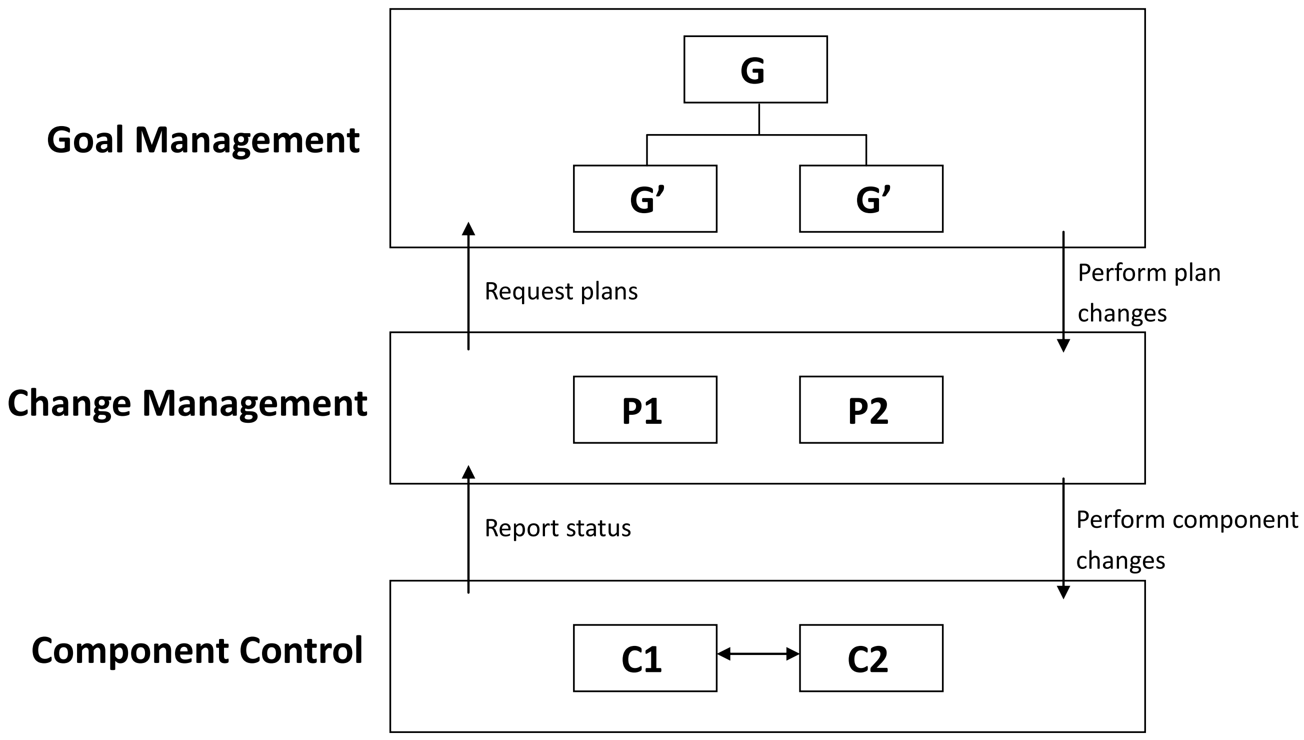


Figure 2.9: Kramer and Magee’s three-layer reference model (Kramer & Magee, 2007)

MADAM (Floch et al., 2006) is a middleware which can detect context changes of mobile systems (e.g. resources, battery), analyse these changes in order to select suitable adaptations, and execute adaptations. Figure 2.10 displays its architecture, which makes use of *Context Model*, *Framework Architecture Model*, *Instance Architecture Model*, *Context Manager*, *Adaptation Manager*, and *Configurator* to achieve the purpose of self-adaptations. The *Context Model* describes related information of contexts. The *Framework Architecture Model* stores information of alternative components to help the *Adaptation Manager* decide how to replace components. The *Instance Architecture Model* is used to evaluate adaptations and to reconfigure applications. The self-adaptive mechanism of MADAM is formed by the *Context Manager*, the *Adaptation Manager*, and the *Configurator*. While the *Context Manager* detects context changes, the *Adaptation Manager* will be notified of these changed and make use of the *Framework Architecture Model* and the *Instance Framework Model* to select a proper adaptation which can satisfy assigned objectives. The *Configurator* is then invoked to perform the adaptation.

Rainbow (Garlan, Cheng, Huang, Schmerl, & Steenkiste, 2004) is an architecture-based self-adaptive system framework, which implements an external self-adaptive mechanism. Rainbow’s framework

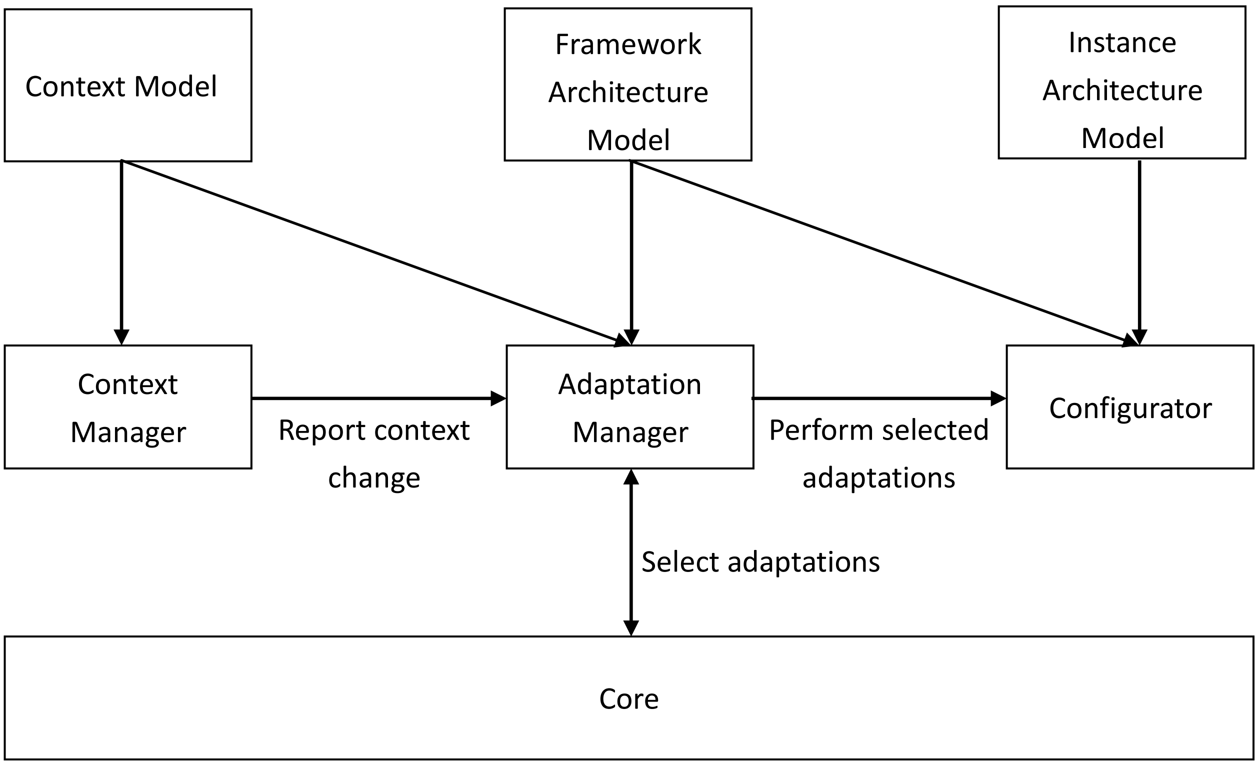


Figure 2.10: The architecture of MADAM (Floch et al., 2006)

can be divided into system-specific adaptation knowledge and adaptation infrastructure. System-specific adaptation knowledge provides target systems with operational models in an effort to ensure that adaptation infrastructure behaves as expected. The operation models involve resource constraints, adaptation strategies, and information related to target systems (e.g. component types and properties). As can be seen in Figure 2.11, the adaptation infrastructure is divided into system layer, architecture layer, and translation layer. At the system layer, Rainbow implements a set of system APIs (e.g. probes, effectors, resource recovery), which are used to monitor and measure system states, perform adaptations, and find new resources. At the architecture layer, the decision of performing an adaptation is made according to system states detected by probes. Furthermore, the adaptation is conducted via effectors. At the translation layer, information is translated between the system layer and the architecture layer.

dynamicTao (Kon et al., 2000) is a reflective middleware which supports dynamic reconfiguration for component-based systems. The structure of dynamicTao is shown in Figure 2.12. As can be seen, *Persistent Repository* is used to manage categories consisting of a collection of components stored in local file systems. A *Network Broker* forwards reconfiguration requests from the network to the

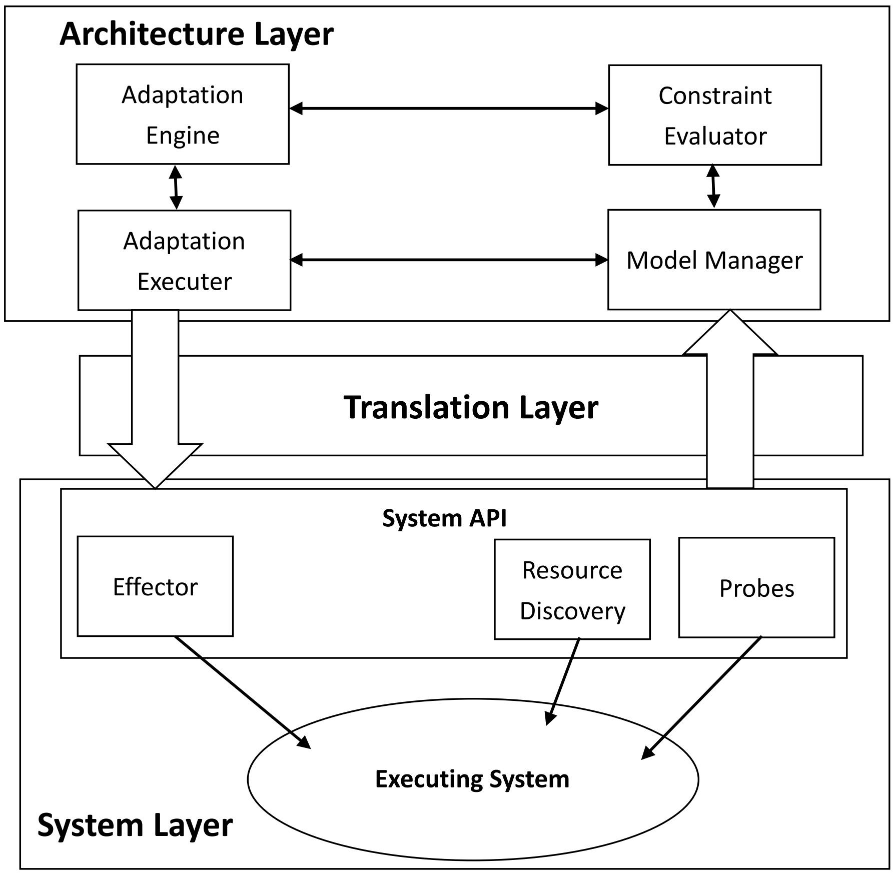


Figure 2.11: The adaptation infrastructure of Rainbow (Garlan, Cheng, Huang, Schmerl, & Steenkiste, 2004)

Dynamic Service Configurator, which supports dynamic component configuration and contains a *DomainConfigurator*. It controls so-called servants (server-side applications) and the *TaoConfigurator*, where implementations of strategies (e.g. concurrency, security, scheduling, and monitoring) can be registered. These implementations can be loaded and replaced at run time, which enables dynamic satisfaction of various constraints. Although dynamicTao can change behaviour of execution environment at run time, adaptation decisions still rely on administrators.

Zanshin (Souza, 2012), whose architecture is shown in Figure 2.13, is a requirements-based self-adaptive system framework, which should be applied to systems with the ability to record state changes of requirements (so-called instrumented systems). Based on this assumption, Zanshin can identify awareness requirements, which refer to the success or failures of other requirements, and evolution requirements, which identify desired evolutions of other requirements. When changes in requirement states are detected by the *Monitor* component, the *Adapt* component is triggered if an adaptation is required. The adaptation decision is based on requirements describing desired strategies

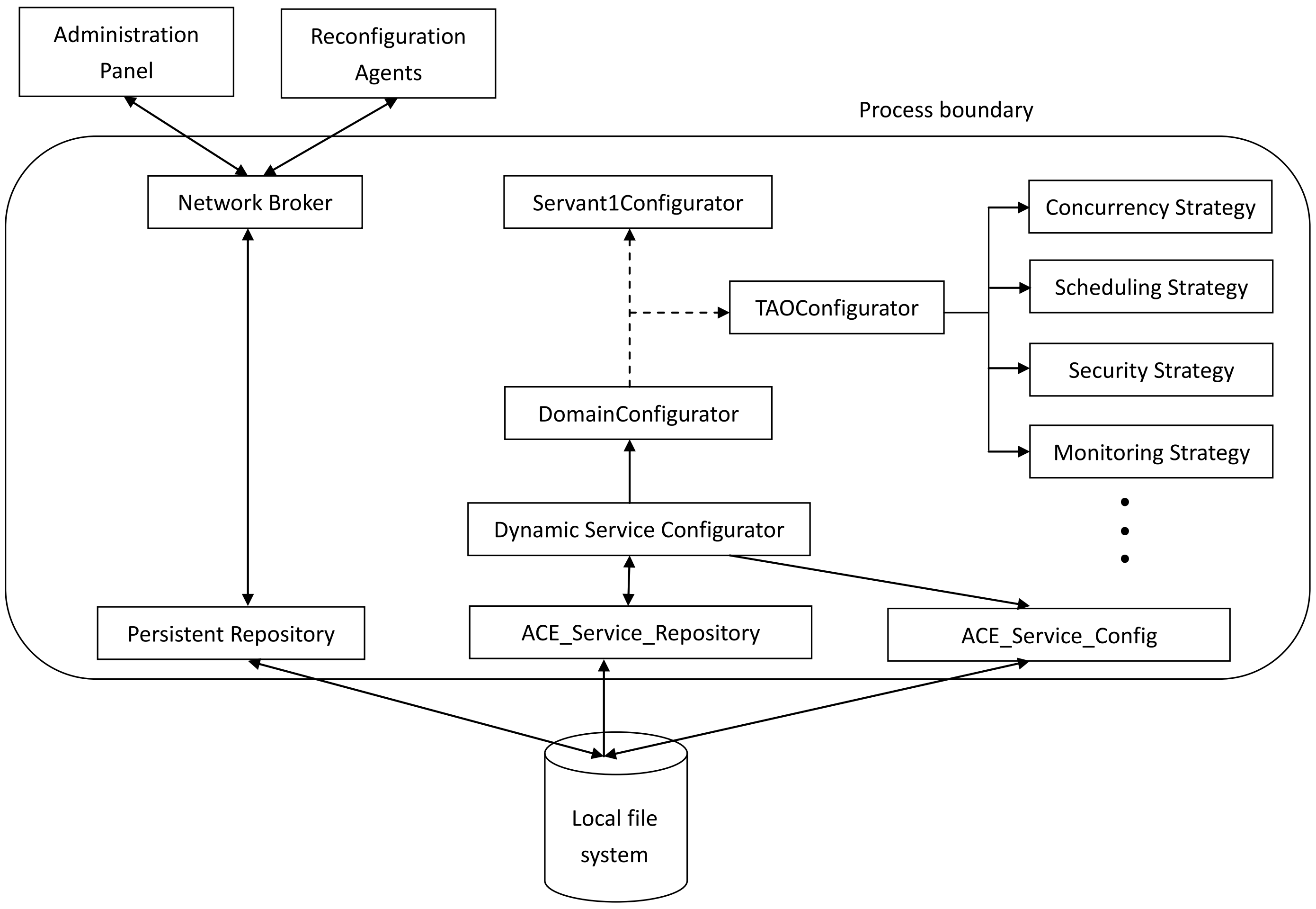


Figure 2.12: The architecture of dynamicTao (Kon et al., 2000)

and is selected by the Event-Condition-Action-based adaptation component and the qualitative component. The ECA-based component is responsible for choosing a suitable adaptation strategy and the qualia component is responsible for executing the strategy.

Unity (Tesauro et al., 2004) is an architecture which attempts to enable distributed computing systems to manage themselves through the interaction of autonomous agents (also called autonomic elements). These elements entail an application manager, a resource arbiter, OSCounters, a registry, a policy repository, and sentinels, each of which manages its own resources, performs self-adaptations, and communicates with other elements via standard web service interfaces (e.g. OSGA). The application manager performs the following four tasks: management of the environment, acquisition of sufficient resources for achieving system goals, interaction with other elements, and evaluation of the impact of changes on resources. The resource arbiter is responsible for finding out the optimal resource usage

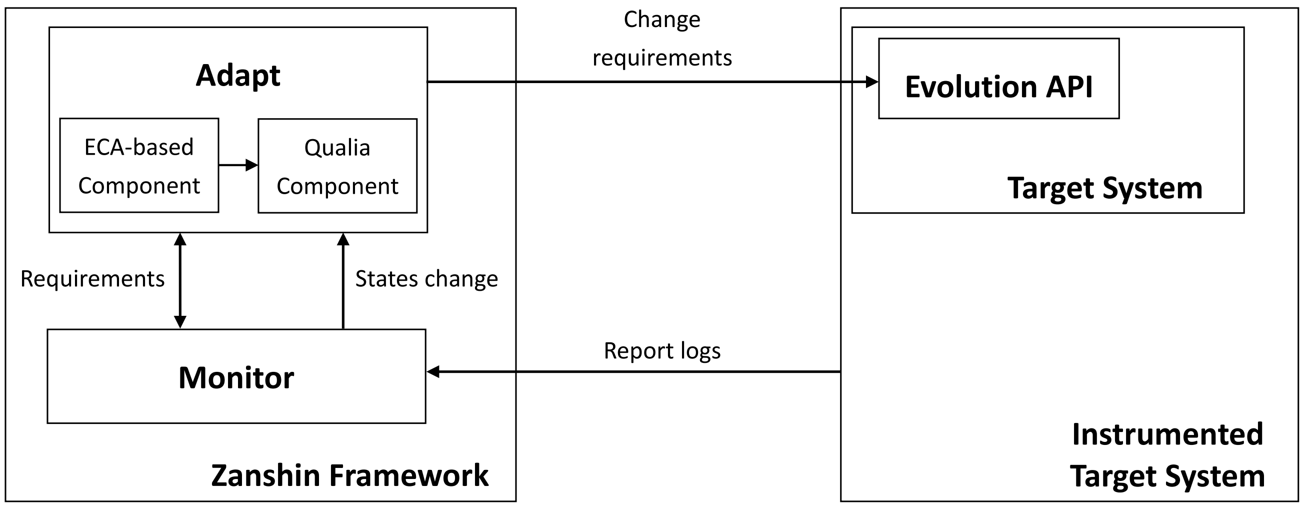


Figure 2.13: The architecture of Zanshin (Souza, 2012)

for the whole system. The OSCounters, which are host computers supporting autonomic elements, activate services or other autonomic elements when a request is received. The registry is a platform through which an autonomic element can ascertain the element with which it wants to communicate. The policy repository supplies interfaces via which high-level policies can be laid down. The sentinels are responsible for monitoring services. When the above-mentioned elements are initialised, they will locate required elements by means of the registry. The first two elements to start are OSCounters and the registry. The resource arbiter is then triggered to decide which elements should be activated and to contact with OSCounters. Next, the policy repository and sentinels are activated and registered in the registry. After that, the arbiter can be registered, locate registered repositories and sentinels, and communicate with a sentinel to monitor all repositories. The application manager contacts the arbiter to allocate required resources. Since Unity is a flexible architecture, it can be implemented for different applications such as self-healing clusters and self-optimising data centres.

Aura (Garlan, Siewiorek, Smailagic, & Steenkiste, 2002) is a task-based self-adaptive system which is designed to reduce human intervention at run time. Its architecture is exhibited in Figure 2.14. A conventional system is inserted into two layers in order to achieve self-tuning. One layer is between Linux kernel and applications. This layer enables the system to monitor and adjust resources. The other layer (so-called Prism) is between applications and users. Prism comprises service suppliers, a task manager, a context observer, and an environment manager. The service suppliers are responsible for providing required services to complete users' tasks. The task manager transforms users' tasks into various services and records them. The context observer detects context changes and notifies the task manager of these changes. The environment manager carries out the commands of resource monitoring and adjustment given by the task manager.

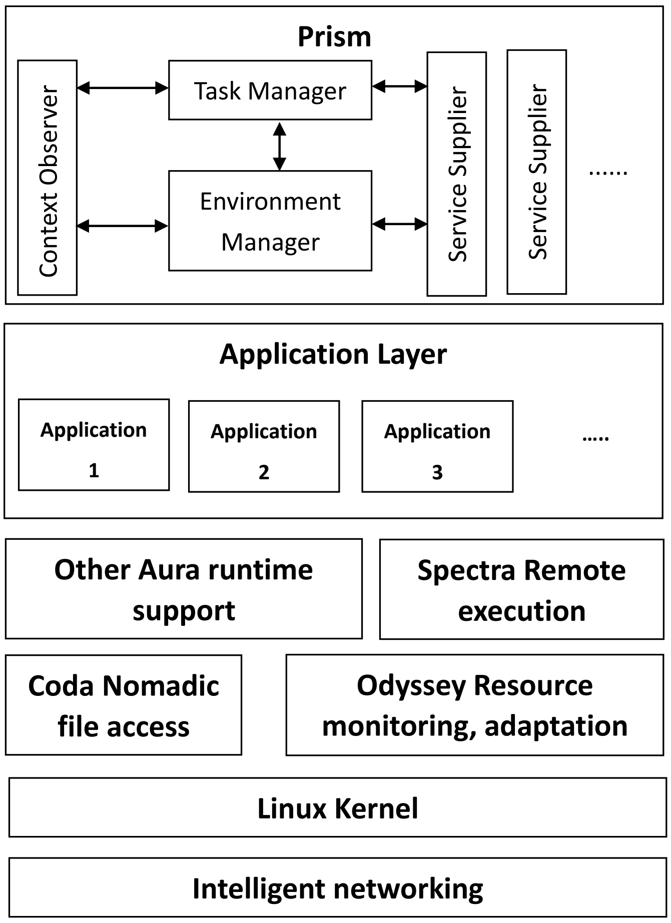


Figure 2.14: The architecture of Aura (Garlan, Siewiorek, Smailagic, & Steenkiste, 2002)

TOTA (Tuples On The Air) (Mamei & Zambonelli, 2009), whose architecture is shown in Figure 2.15, is a middleware and programming approach for building self-adaptive software in pervasive and mobile environments. A tuple is defined as a structured set of data elements. Applications can coordinate through exchange of tuples. As can be seen in Figure 2.15, TOTA comprises *TOTA API*, *TOTA Engine*, and *Event interface*. *TOTA API* is an interface that allows applications to inject and retrieve tuples and to lay down events in *Event Interface*, which reports events received from *TOTA Engine*. *TOTA Engine* is responsible for managing tuples. The management of tuples includes the actions of injecting, receiving, and updating tuples. In addition, the engine embodies local tuples, which are used to trace tuples reaching other nodes.

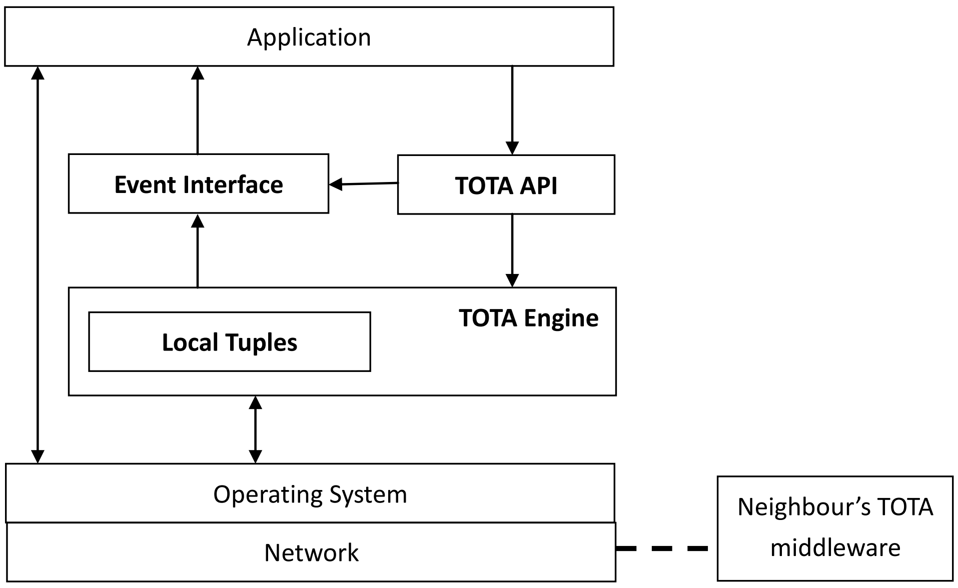


Figure 2.15: The architecture of TOTA (Mamei & Zambonelli, 2009)

2.8.3 Dynamic Deployment of Data Structures

Among approaches which automatically change internal data structures to adjust resource usage are SILT (Lim, Fan, Andersen, & Kaminsky, 2011), OSKI (Vuduc, Demmel, & Yelick, 2005), and Kusum et al.'s work (Kusum, Neamtiu, & Gupta, 2015). 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. OSKI, which is a collection of low-level primitives, provides automatically-tuned computational kernels as well as a mechanism for selecting a data structure and code transformations. Kusum et al. present an approach which dynamically changes data structures (e.g. change between Adjacency List and Adjacency Matrix) so as to boost performance and memory efficiency of graph applications. Although the above-mentioned approaches can efficiently utilise resources, they only focus on memory use and performance. Other QoS metrics are not taken into account. Furthermore, there is no mechanism for specifying Service Level Objectives, which leads to difficulties in adapting software to meet different QoS requirements.

In addition to the internal transformation of data structures, Abbasi et al. (Abbasi, Wolf, Schwan, Eisenhauer, & Hilton, 2004) present a middleware infrastructure, *XChange*, to exchange data between different applications. This infrastructure enables applications to dynamically provide their own data filters and transformation methods. The former can select required data, and the latter converts data into required formats. For example, when Application A wants to exchange data with Application B, it needs to first register its data format through an API provided by *XChange* before data is sent. Similarly, Application B needs to register its data format as well. Additionally, it needs to specify data transformation methods. As a result, *XChange* can dynamically perform data exchange and transformation between applications.

Chapter 3

A Novel Self-Adaptive Container Framework

‘There is at least one point in the history of any company when you have to change dramatically to rise to the next level of performance. Miss that moment - and you start to decline.’

Andy Grove, Former CEO of Intel

3.1 Introduction

This chapter presents a novel self-adaptive container framework which intends to reduce the frequency of software reimplementation when application contexts and execution environments change. To achieve this, the framework is equipped with a self-adaptive mechanism, which dynamically changes underlying data structures when specified Service Level Objectives are violated.

The diversity of QoS requirements in different execution environments and application contexts has led to frequent code refactoring. Manually adapting software may require months or years of programmer effort and a high level of expertise. To solve this issue, software should have “intelligence” so as to automatically satisfy QoS requirements of different execution environments. This chapter will present a novel container framework which endows software with the ability of dynamically satisfying QoS requirements with low programmer overhead. This is achieved through a self-adaptive mechanism embedded in the framework that is capable of monitoring and measuring SLOs in terms of response time, memory consumption, and reliability, comparing operation profiles with specified SLOs, planing adaptations based on the effect of the adopted actions, and executing them so as to achieve specified Service Level Objectives. The measurable QoS parameters are specified through a standard QoS specification language, WSLA, which clearly defines the metrics of QoS parameters and how to measure them. In addition, our mechanism deals with each Service Level Objective according to their priorities, which prevents conflicts when multiple SLOs are assigned.

Our framework focuses on containers because their underlying data structures are critical to software's non-functional behaviour (e.g. performance, memory use, and reliability). This implies that the shift from one data structure (DS1) to another (DS2) may enable software to satisfy a non-functional requirement which is violated when DS1 is adopted. Through dynamic changes of data structures, our framework can automatically satisfy specified Service Level Objectives without human intervention. In addition, this framework features tighter functionality specification, which allows greater scope of efficiency optimisations, including the techniques of probabilistic data structures, offload storage, and parallelism.

To prove the framework's feasibility, we have implemented a prototype in C++, which supports a single-value container exploiting probabilistic data structures and out-of-core storage. This prototype supplies insertion and search operations as well as the functionality of FIFO queues. It is then applied to explicit state-space exploration adopting a breadth-first search algorithm. The evaluation of the framework is conducted through the comparison with conventional containers' performance and memory consumption. Furthermore, the framework is assigned multiple SLOs with different priority orderings in order to observe the behaviour and influence of priority orderings.

The remainder of this chapter is organised as follows. Section 3.2 presents the architecture of the self-adaptive container framework and describes each component in the framework. After Section 3.3 introduces the self-adaptive mechanism, an improved probabilistic data structure is proposed in Section 3.4. Section 3.5 describes a prototype implementation of the framework supporting out-of-core storage and the improved probabilistic data structure. A case study is investigated in Section 3.6 to demonstrate the capability and viability of the framework. Section 3.7 concludes this chapter.

3.2 The Design of the Novel Self-Adaptive Container Framework

The design of our self-adaptive container framework references the concepts of containers and autonomic computing. The former hides complex implementation details of data manipulation from programmers and prevents duplicate code for different data types, and the latter enables our framework to dynamically adjust resource usage to meet SLOs. The architecture of the framework is shown in Figure 3.1, which comprises three major components: the Application Programming Interface, the Self-Adaptive Unit, and Third-Party Libraries.

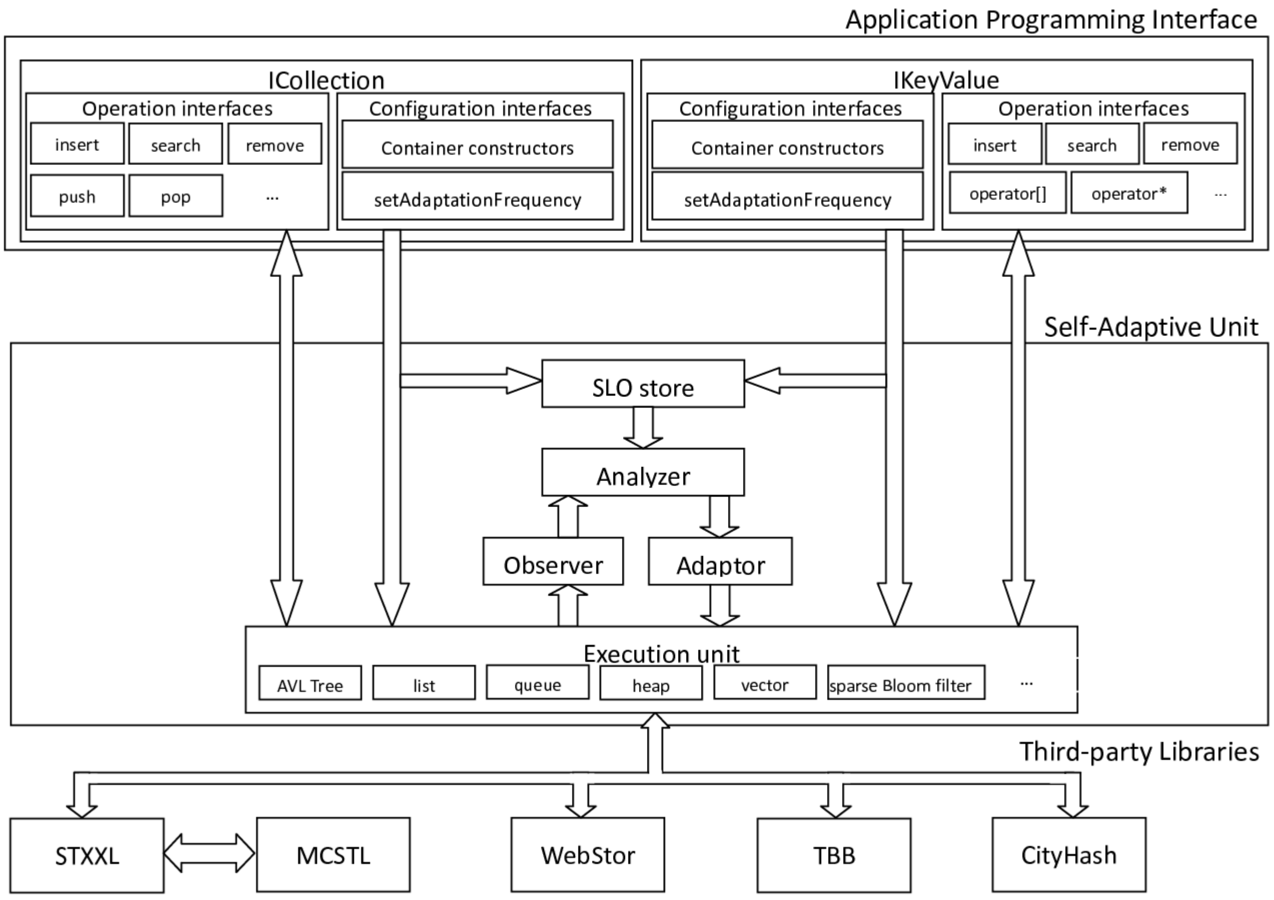


Figure 3.1: The self-adaptive container framework architecture

3.2.1 Application Programming Interface

The Application Programming Interface (API) is a collection of interfaces through which programmers can perform desired operations and control the framework. It contains two template classes (i.e.

`ICollection` and `IKeyValue`) covering most functionalities of the Standard Template Library. In particular `ICollection` is a single-value container, whose functionality subsumes STL's *list*, *vector*, *queue*, *stack*, *deque*, *set*, *priority_queue*, *multiset*, *unordered_set*, and *unordered_multiset*. `IKeyValue` implements the functionality of key-value stores, which subsumes STL's *map*, *multimap*, *unordered_map*, and *unordered_multimap*. A detailed description of key-value stores will be presented in Section 4.2. The member functions of `ICollection` and `IKeyValue` can be separated into operation interfaces and configuration interfaces. The operation interfaces are a set of commonly-used operations (e.g. insert, search, remove, etc.). The configuration interfaces (i.e. `setAdaptationFrequency` and container constructors) provide a way of controlling the self-adaptive mechanism, which acts according to parameters of the configuration interfaces. For container constructors, their usage is

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

and

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

where

- *T* is the stored data type for the single-value container.
- *Compare* is a binary predicate, which compares two objects with the same type and returns a boolean value. Its default value is *less*, i.e. less-than operator.
- *K* is the data type of key-value stores' keys.
- *V* is the data type of key-value stores' values.
- *op_desc* describes a required set of container functionality (so-called operation descriptors). This recognizes that it is rarely the case that every container instance will utilise its full set of potential functionality, allowing for the deployment of more efficient underlying data structures. The definitions of all currently-supported operation descriptors are listed in Table 3.1 and the involved operation descriptors of all member functions are listed in Table 3.2. To easily

configure a desired set of functionality, combined operation descriptors are provided, as shown in Table 3.3.

- *SLO_file* specifies a path of an XML file which contains a description of desired SLOs in WSLA format. SLOs can relate to response time, primary memory use, or reliability. To clearly define desired QoS parameters and the corresponding operations, the Uniform Resource Name (URN) scheme is utilised in the *MeasurementURI* section (see Section 2.4). Its usage is described as follows:

  urn:*ContainerClass*:*QoSMetric*:*OperationDescriptor*

  ContainerClass is either `ICollection` or `IKeyValue`, which specifies the target container. *QoSMetric* indicates the target QoS metrics. The available QoS metrics are listed in Table 3.4. *OperationDescriptor* specifies corresponding operations of QoS parameters. For example, response time can be related to insertion time, search time, or deletion time, which can be specified through OP_INSERT, OP_SEARCH, and OP_ERASE, respectively. An example of how to express desired SLOs in WSLA format is shown in Appendix A.

- *Freq* is an optional parameter defining the frequency with which the self-adaptive mechanism is activated. This parameter may also be subsequently updated via `setAdaptationFrequency`.

Operation descriptor		**Definition**
OP_INSERT		Insertion
OP_ERASE		Deletion
OP_FIND		Find (retrieval)
OP_SEARCH		Search (existence)
OP_INDEX		Direct index-based access
OP_MULTI		Allowance of duplicate elements
OP_UORDER		Unordered storage
OP_ITERATOR		Iterator support
OP_PRIORITY		Prioritised operation
OP_FRONT	OP_INSERT_FRONT	Front insertion
	OP_ERASE_FRONT	Front deletion
OP_BACK	OP_INSERT_BACK	Back insertion
	OP_ERASE_BACK	Back deletion

Table 3.1: Definitions of Operation Descriptors

ICollection<T> member functions	
Function Name	Involved Operation Descriptor
insert(const T& x)	OP_INSERT
insert(iterator position, const T& x)	OP_INSERT | OP_ITERATOR
erase(const T& x)	OP_ERASE
find(const T& x)	OP_FIND | OP_ITERATOR
search(const T& x)	OP_SEARCH
begin(), end()	OP_ITERATOR
operator[]	OP_INDEX
push()	OP_INSERT_FRONT or OP_INSERT_BACK
push_front()	OP_INSERT_FRONT
push_back()	OP_INSERT_BACK
pop	OP_ERASE_BACK or OP_ERASE_FRONT
pop_back()	OP_ERASE_BACK
pop_front()	OP_ERASE_FRONT
IKeyValue<K,V> member functions	
Function Name	Involved Operation Descriptor
insert(const pair<K, V>& x)	OP_INSERT
insert(iterator position, const pair<K, V>& x)	OP_INSERT | OP_ITERATOR
erase(const K& x)	OP_ERASE
find(const K& x)	OP_FIND | OP_ITERATOR
search(const K& x)	OP_SEARCH
begin(), end()	OP_ITERATOR
operator[]	OP_INDEX

Table 3.2: Member functions provided by our framework and involved 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
MultiSet	OP_MULTISET	OP_INSERT | OP_ERASE | OP_SEARCH | OP_ITERATOR | OP_MULTI
Unordered Set	OP_UORDERSET	OP_INSERT | OP_ERASE | OP_SEARCH | OP_ITERATOR | OP_UORDER
Unordered MultiSet	OP_UORDERMULTISET	OP_INSERT | OP_ERASE | OP_SEARCH | OP_ITERATOR | OP_UORDER | OP_MULTI
Stack	OP_STACK	OP_INSERT_FRONT | OP_ERASE_FRONT
Queue	OP_QUEUE	OP_INSERT_BACK | OP_ERASE_FRONT
Priority Queue	OP_PQUEUE	OP_INSERT | OP_ERASE_FRONT | OP_PRIORITY
Map	OP_MAP	OP_INSERT | OP_ERASE | OP_SEARCH | OP_ITERATOR | OP_INDEX
MultiMap	OP_MULTIMAP	OP_INSERT | OP_ERASE | OP_SEARCH | OP_ITERATOR | OP_INDEX | OP_MULTI
Unordered Map	OP_UORDERMAP	OP_INSERT | OP_ERASE | OP_SEARCH | OP_ITERATOR | OP_INDEX | OP_UORDER
Unordered MultiMap	OP_UORDERMULTIMAP	OP_INSERT | OP_ERASE | OP_SEARCH | OP_ITERATOR | OP_INDEX | OP_UORDER | OP_MULTI

Table 3.3: Combined Operation Descriptors for frequently used functionality

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

Table 3.4: Currently supported QoS metrics

3.2.2 Self-Adaptive Unit

The Self-Adaptive Unit is the core of the framework, which carries out operations given through the API and activates the self-adaptive mechanism to perform adaptations. It consists of an SLO store, an Execution unit, an Observer, an Analyser, and an Adaptor. The SLO store holds all Service Level Objectives retrieved from the XML file specified via the configuration interfaces. The Execution unit carries out operations laid down through the operation interfaces. If an operation is compatible with the functionalities that the target container should provide (as declared via the configuration interfaces), it is applied to the currently-selected data structure. Otherwise the operation is rejected and an appropriate exception is thrown. The Observer monitors the Execution unit to measure per operation response time, compute primary memory consumption, and reliability (when the underlying data structure is a probabilistic data structure). The Analyser is a decision maker which is periodically activated to determine if the underlying data structure needs to be adjusted. The frequency of the activation can be subsequently updated through the `setAdaptationFrequency` configuration interface. The Adaptor performs adaptation actions that are expected to improve container compliance with its SLOs. The Observer, Analyser, and Adaptor form the self-adaptive cycle, which will be described in detail in Section 3.3.2.

3.2.3 Third-Party Libraries

Third-Party Libraries are a set of libraries which provide a low-cost and robust way of implementing the techniques of hashing, parallelism, out-of-core storage, and cloud storage. In our prototype, we make use of CityHash (Pike & Alakuijala, 2013) for hashing, Intel Threading Building Blocks (TBB) (Intel, 2014) for parallelism, STXXL (Dementiev, Kettner, & Sanders, 2005) and MCSTL (Singler, Sanders, & Putze, 2007) for out-of-core storage, and WebStor (OblakSoft, 2014) for cloud storage. The use of CityHash will be described in Section 3.4 and that of the other libraries will be described in Chapter 5.

3.3 Self-Adaptive Mechanism

This section will describe the operation of the self-adaptive mechanism. First, SLO metrics are defined to specify QoS parameters with respect to response time, primary memory use, and reliability. Second, a self-adaptive cycle is developed to dynamically manage specified QoS parameters via adaptations. Finally, the effects of adopted adaptations are analysed.

3.3.1 SLO Metric

The SLO metrics involve 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 (Knottenbelt & Harrison, 1999). 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.

3.3.2 Self-Adaptive Cycle

The framework adopts the classical self-adaptive cycle (Rohr et al., 2006), which consists of the Observer, the Analyser, and the Adaptor. The cycle starts from the Observer, which records per operation response time, computes primary memory use, and where appropriate reliability. The Analyser is then activated to compare the Observer's profile data with SLOs and to decide if an adaptation is required. The decision-making process depends on which adaptation strategy is adopted. If the goal of adaptations is to find the maximum sum of multiple objectives's importances, the weighted sum method should be adopted. If the goal of adaptations is to find the maximum product of multiple objectives's importances, the weighted product method should be deployed. In our prototype implementation, the decision-making process adopts the strategy of strict ordering, which ensures that objectives with higher importance are addressed first and not affected by less important objectives. We adopt this strategy for two reasons. First of all, we observe that many execution environments naturally impose a priority ordering on the SLOs. Second the decision-making process should expend as little time as possible. Strict ordering enables our framework to decide if an adaptation is required in a short time. To satisfy this strategy, the following two rules are applied repeatedly in priority ordering once for each SLO:

A. The adaptation action will result in either the satisfaction of the SLO or a reduction in the degree to which the SLOs are violated.

B. The adaptation action is not expected to result in the violation of a currently-satisfied SLO of higher priority.

The purpose of the first rule aims to solve the situation where some subset (or any) of the SLOs cannot be met within resource constraints. Our framework, therefore, does not guarantee that all SLOs will be satisfied. The purpose of the second rule is to prevent adaptation actions taken to address violated SLOs from violating another SLO (for example the deployment of out-of-core storage may result in unacceptably large response times). As a result, each SLO is assigned a distinct priority according to the SLO's declaration sequence in the configuration file. The priority ordering decides the sequence where the Analyser addresses SLOs. If the SLO being addressed is satisfied, no adaptation action is necessary. If the SLO is violated, the Adaptor is called in for an adaptation.

3.3.3 Adaptation Actions

The Adaptor may perform three kinds of adaptation actions in accordance with the nature of the violated SLO and its priority. If it is performance-related (e.g. an SLO related to insertion, search, or deletion response time), then gains may be had from subdividing the underlying data structures. This may result in two side effects. The first side effect is the increase in memory consumption. For example, if tree data structures are adopted, the underlying trees are subdivided into shorter trees via rearrangement of elements in the original trees. This subdivision can reduce time to locate an element. However, primary memory consumption is also increased because the framework needs to store a greater number of trees compared to the tree number before the adaptation. The second side effect is the improvement in reliability when a probabilistic data structure is used. The reason why the reliability is increased will be explained in Section 3.4. 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 and where appropriate reliability), or, should reliability and functionality requirements allow, moving to a probabilistic data structure (If iterator-based functionality is required, out-of-core storage should be utilised). 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. This will cause the side effects of improving performance and increasing memory use.

3.4 The Utilisation of Probabilistic Data Structures

Our framework makes use of an improved sparse Bloom filter, whose structure is depicted in Figure 3.2, as one of the many adopted data structures. Such a data structure comprises a forest of AVL trees and utilises two hash functions. To generate uniformly distributed hash keys, CityHash (Pike & Alakuijala, 2013) function library is utilised. CityHash is capable of generating 32, 64, 128 and 256 bit hash keys from arbitrary data according to reliability requirements and supplying independent hash functions by giving different seeds. This is adequate to provide search, insertion and deletion functionality on containers. For containers where multiplicity of items is important (e.g. in multisets or multimaps) sparse counting Bloom filters (Bonomi et al., 2006; Rottenstreich et al., 2014) are used to provide the necessary functionality. Compared to original sparse Bloom filters, we enable the improved sparse Bloom filter to dynamically change its number of AVL trees. As can be seen in in Figure 3.2, the number of AVL trees is 2^k where k is the number of AVL trees. When an element i is inserted, it is first input to the primary hash function to determine which AVL tree (target tree) it should be inserted into. The secondary hash function is then utilised to generate the second hash key. Both hash keys will be stored in a tree node of the target tree. The primary hash key should be stored because it will be reused to determine the new position of this node when adaptations are performed. When an element j is searched, it is first input to the primary and the secondary hash function to get hash keys, i.e. $PHF(j)$ and $SHF(j)$. $PHF(j)$ is then utilised to obtain the tree where this element might be stored. The tree is searched to see if a node whose stored secondary hash key equals $SHF(j)$. If the sparse Bloom filter is adopted by `ICollection`, only hash keys are stored. On the other hand, if it is adopted by `IKeyValue`, each node in AVL trees stores both hash keys and corresponding values.

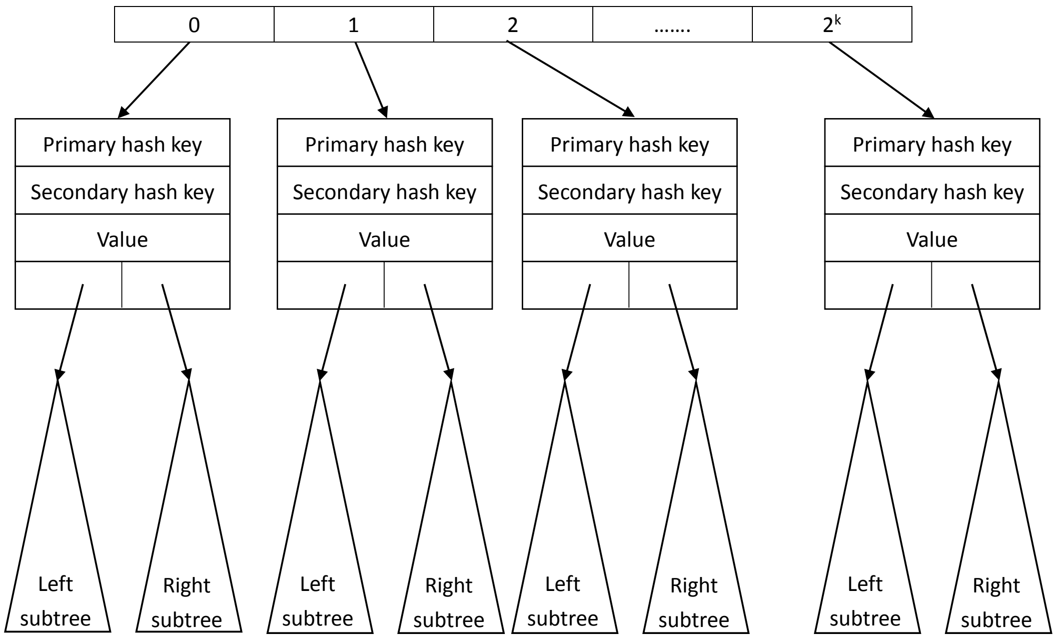


Figure 3.2: The structure of the improved sparse Bloom filter

The improved sparse Bloom filter can be applied to a container according to the following scenarios First, the reliability requirement is less than 100%. Second, if iterator-based operation descriptors are specified, memory-related SLOs should have higher priority over performance-related SLOs. That is because this scenario requires support of out-of-core storage. To compute the current reliability (every inserted element is mapped to a unique key), the following formula (Knottenbelt, 2000) is utilised:

$$Reliability = 1 - \frac{n^2}{N_{AVL}2^{b+1}} \tag{3.1}$$

where n is the number of inserted elements, N_{AVL} is the number of utilised AVL trees, and b is the number of bits used to represent a hash key. For example, if the number of currently-stored elements is $10\,000$, the number of AVL trees is $1\,024$, the size of each hash key is 32 bits, the reliability will be 0.99999999. Additionally, this formula indicates that the increase of AVL trees can also improve reliability.

3.5 A Prototype Implementation of the Self-Adaptive Container Framework

In this section, we will describe a prototype of the self-adaptive container framework which is implemented in C++. The prototype includes a template class (i.e. `ICollection`) with member functions of *insert*, *search*, *push*, *pop*, and *empty*, a self-adaptive unit (i.e. an Observer, an Analyser, an Adaptor, and an SLO store), an improved sparse Bloom filter, and a FIFO queue. To utilise this prototype, a header file (i.e. *ICollection.h*) should be included, and an instance of `ICollection` should be declared. After the instance is declared, our framework will choose an initial data structure based on assigned SLOs and operation descriptors. Hence, if OP_INSERT and OP_SEARCH are specified, either a tree data structure (the reliability requirement is 100%) or a modified sparse Bloom filter (the reliability requirement is less than 100%) will be selected. If OP_QUEUE (i.e.OP_PUSH_BACK and OP_POP_FRONT) is specified, an array will be exploited. The size of the array depends on the assigned memory constraints.

After the initial data structure is chosen, the framework begins to accept operations laid down via `ICollection`'s operation interfaces. Once an SLO is violated, a corresponding adaptation action is initiated. As can be seen in Figure 3.2, if the currently-used data structure is a sparse Bloom filter, it will be adjusted according to the violated SLO. The violation of performance-related and reliability-related SLOs will cause the increase in the number of AVL trees. The violation of memory-related SLOs may lead to one of the following two adaptations. If the reduction of AVL trees' number can satisfy memory-related SLOs, the number of AVL trees will be reduced. Otherwise, out-of-core storage will be activated. This activation converts all AVL trees into a sorted array before it is moved to out-of-core memory, which enables external binary search. Furthermore, some index nodes are kept to reduce the number of I/Os.

Our implementation of FIFO queues only focuses on memory-related SLOs because the supported operations cannot be further improved and contents of elements should be stored. Hence, the design of our FIFO queue is shown in Figure 3.3. Our implementation contains a head block and a tail block, both of which are stored in primary memory, and numbers of body blocks, which are stored in out-of-core memory. When a push operation is invoked, the tail block will be checked. If it has enough memory space to store the element, this element will be pushed into either the head block or the tail block. If the tail block is full, elements stored in the tail block will be moved to out-of-core storage and the element is pushed into the tail block. When a pop operation is invoked, the head block is checked to see if it is empty. If the head block is not empty, the first element will be removed. Otherwise, the first body block will be moved to the head block, and the first element is then removed.

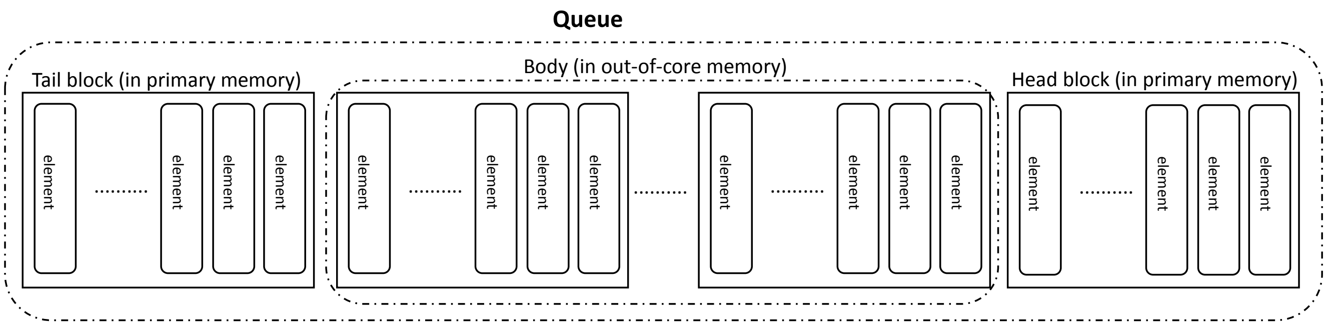

Queue
Tail block (in primary memory)
Body (in out-of-core memory)
Head block (in primary memory)
element

Figure 3.3: The structure of our FIFO queue

3.6 Case Study

In this section, we investigate an application centred on explicit state-space exploration, exploiting a breadth-first search (BFS) algorithm to explore approximately 240 million states, to illustrate the framework's viability, scalability, and capability. This application is commonly employed in numerous domains including model checking (Clarke, Grumberg, & Peled, 1999) and performance analysis of concurrent systems (Knottenbelt & Harrison, 1999). A naïve implementation of the BFS algorithm and the same algorithm adopting our framework are shown in Figure 3.4. As can be seen, the only difference of the two programs is the container declarations (one for the queue of unexplored states, *unexplored*, and one for the table of explored states, *explored*). To evaluate the framework's behaviour and ability, the following SLOs are assigned in *ExploredSLOs.xml*, whose path is indicated in *explored*'s second parameter (i.e. *SLO_file*):

1. 90% of insertion times should be less than 1000 ns, and 85% of search times should be less than 1200 ns.

```
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 3.4: The naïve BFS (left) and the BFS adopting our framework (right)

2. Reliability should be higher than 0.99.

3. Memory consumption should be no more than 7.5 GB.

The full content of the XML file is shown in Appendix A. As can be seen, the XML file contains four targets with respect to insertion time, search time, reliability, and primary memory consumption. Each objective is specified by an *SLAParameter* and corresponding metrics. For response times (insertion time and search time), the percentages of response times which are less than the desired response times should be measured. Take insertion time for example. Its final *Metric* specifies the measurement of insertion time (see Line 23 –30). After the insertion time is measured, its value will then be compared with 1 000 to obtain the percentage of insertion time which is less than 1 000 ns (Line 16 – 22). For memory and reliability, soft requirements are not provided. As a result, their metrics only define where to retrieve values. All values obtained in the *ServiceDefinition* section are compared with the objectives defined in the *ServiceLevelObjective* section (Line 86 –154). Take reliability for example. Its SLO specifies that the value of the *SLAParameter*, *CurrentReliability*, should be greater than (i.e. *GreaterEqual*) 0.99. Through this file, our framework can acquire all information related to SLOs, including their metrics, objectives, and priorities. Similarly, the SLO for the unexplored state queue, which requires the primary memory consumption lower than 40 MB, is specified in *UnexploredSLOs.xml*. Figure 3.4 also illustrates that the value of *AdaptationFrequency* is 100, which means that the Analyser is activated every 100 operations. The influence of different frequencies is shown in Table 3.5. As can be seen, increasing the values of *AdaptationFrequency* can reduce both insertion time and search time, but when its value reaches 1 000, insertion time and search time rise owing to the delay of adaptation actions.

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

Table 3.5: The influence of various values of *AdaptationFrequency* on cumulative response time

3.6.1 Comparison with Conventional Containers

Figures 3.5 and 3.6 depict the average insertion and search time consumed by an STL set, an AVL tree, a standard Bloom filter, and our framework under the priority ordering of performance, reliability, and memory in one run. The x-axes in the two figures represent the cumulative numbers of operations, and the y-axes refer to average response time required for one operation. As can be seen, our framework yields better performance than conventional data structures. Specifically, it reduces cumulative insertion time by 74.9% and cumulative search time by 86.2% compared to the STL set. The two figures also show that the framework has occasional sharp rises in insertion time and search time, which represent that self-adaptations are conducted in order to protect QoS. In addition, when the x-values are small, our framework consumes less insertion time but more search time compared to the STL set. That is because when x-value is small, the STL set needs more time to be initialised but expends less time on comparing a small number of stored elements. By contrast, our framework expends less initialisation time but consumes more time for generating hash keys (because the currently-used data structure is a sparse Bloom filter).

The memory consumption of the STL set, the AVL tree, the Bloom filter, and our framework is displayed in Figure 3.7. Our framework consumes merely 10% of memory space required by the STL set and the AVL tree. Although its memory consumption is not as efficient as the Bloom filter, its reliability is considerably higher. In addition, because the assigned SLOs specify that performance

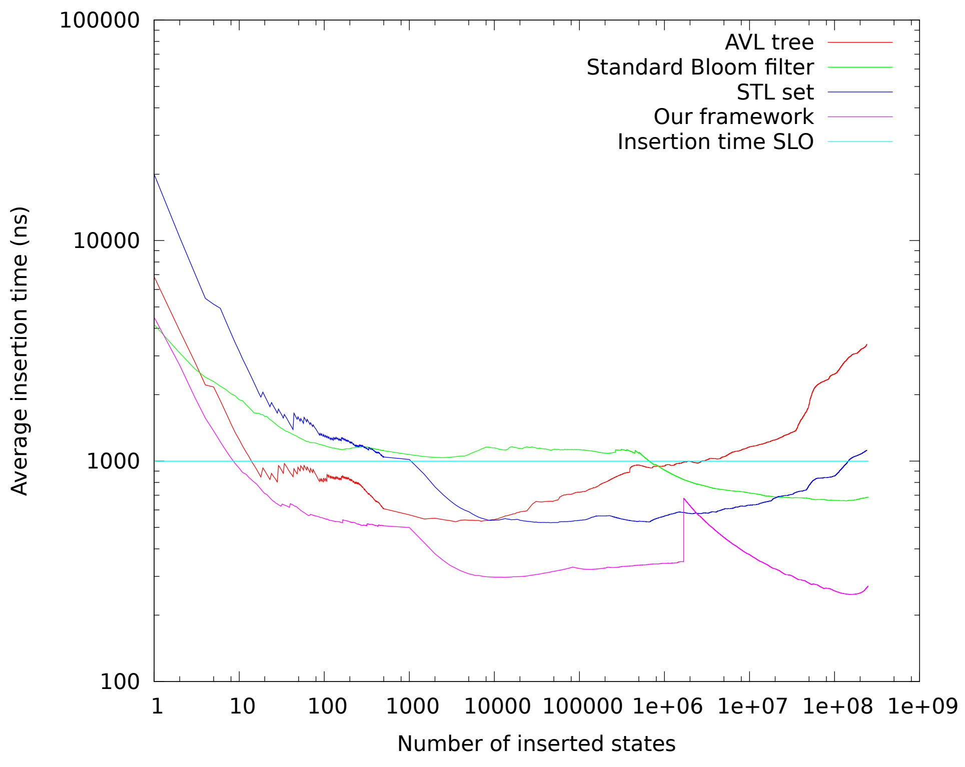


Figure 3.5: The average insertion time of *explored* adopting conventional containers and our framework

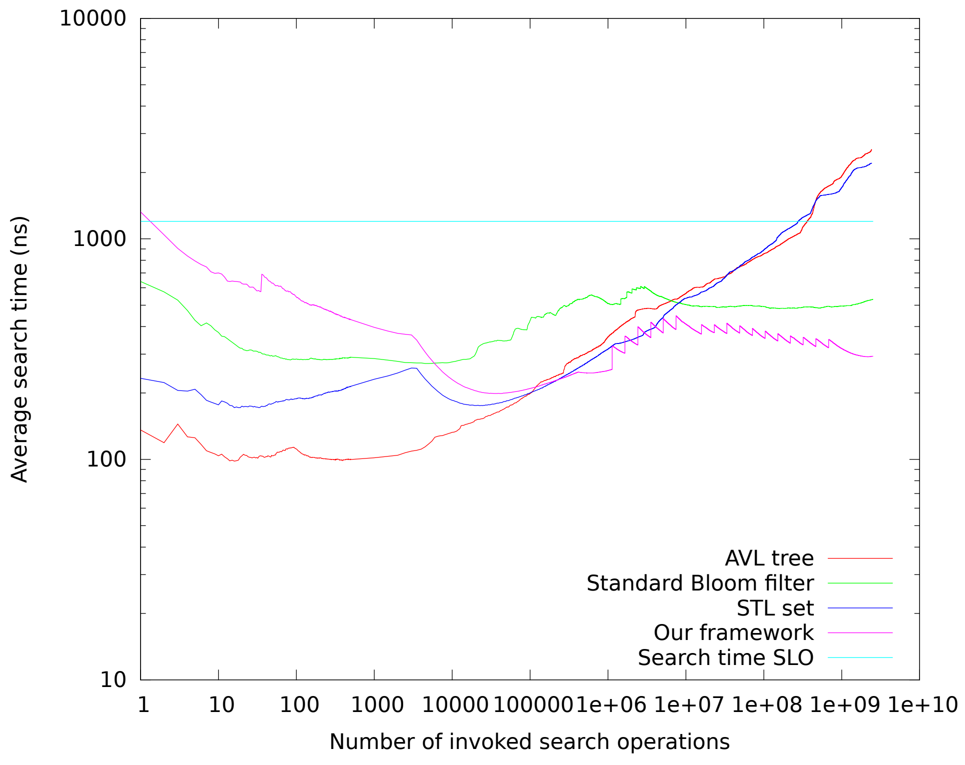


Figure 3.6: The average search time of *explored* adopting conventional containers and our framework

and reliability has higher priority than memory, our framework violates the memory-related SLO to protect performance or reliability.

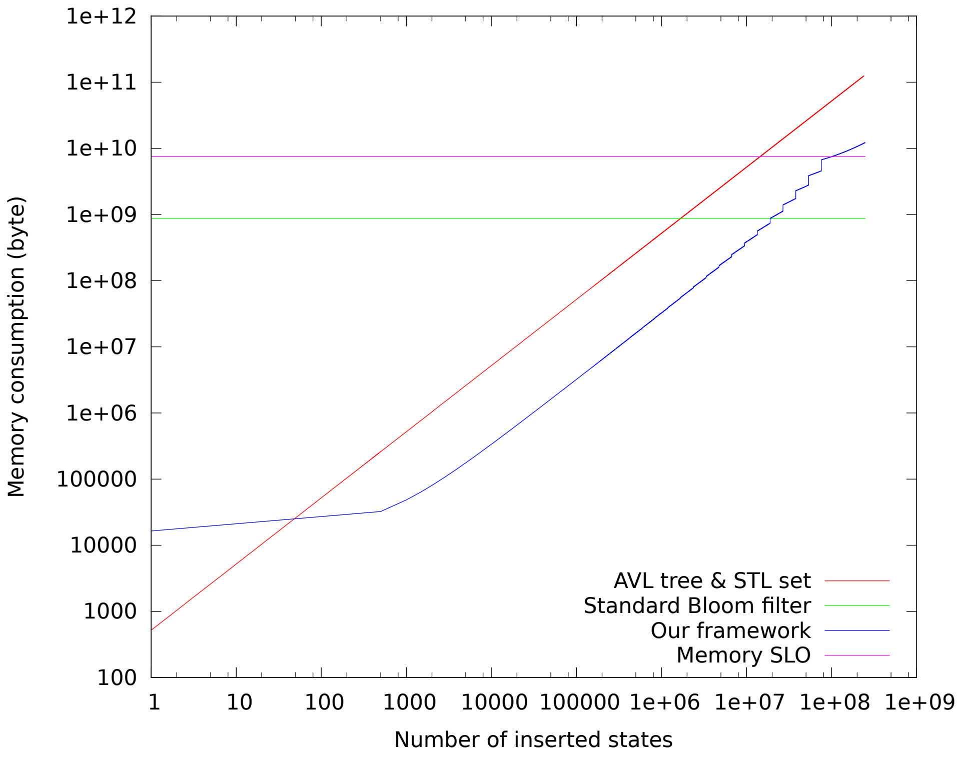


Figure 3.7: The memory consumption of *explored* adopting conventional containers and our framework

3.6.2 Influence of SLO Priority Ordering

The self-adaptive mechanism equipped in the framework addresses SLOs according to the specified priority ordering, which may lead to different behaviour. Figures 3.8 and 3.9 depict the average insertion time and average search time under the six priority orderings (i.e. PerMemRel, PerRelMem, MemPerRel, MemRelPer, RelPerMem, and RelMemPer), where Per represents performance-related SLOs, Mem represents memory-related SLOs, and Rel represents reliability-related SLOs. As can be seen, the framework expends less insertion time and search time when the given SLOs specify that performance has higher priority over memory consumption. By contrast, if memory consumption is the highest in order of priority (i.e. MemPerRel or MemRelPer), the insertion time and search time will significantly rise. That is because out-of-core memory is frequently accessed.

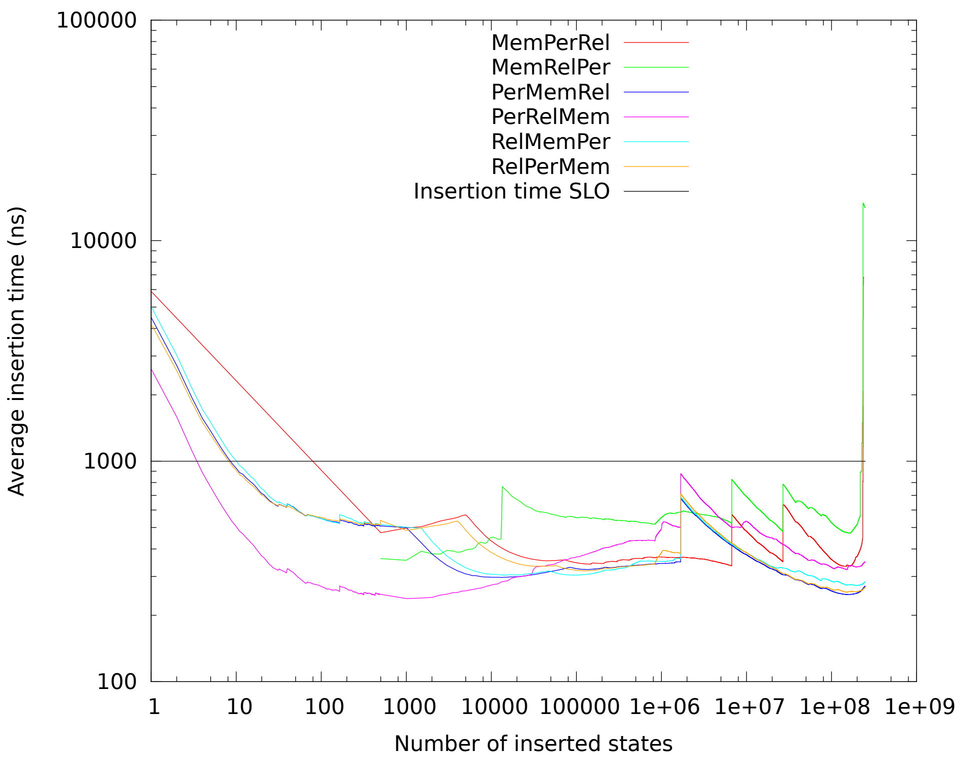


Figure 3.8: The average insertion time under the six priority orderings

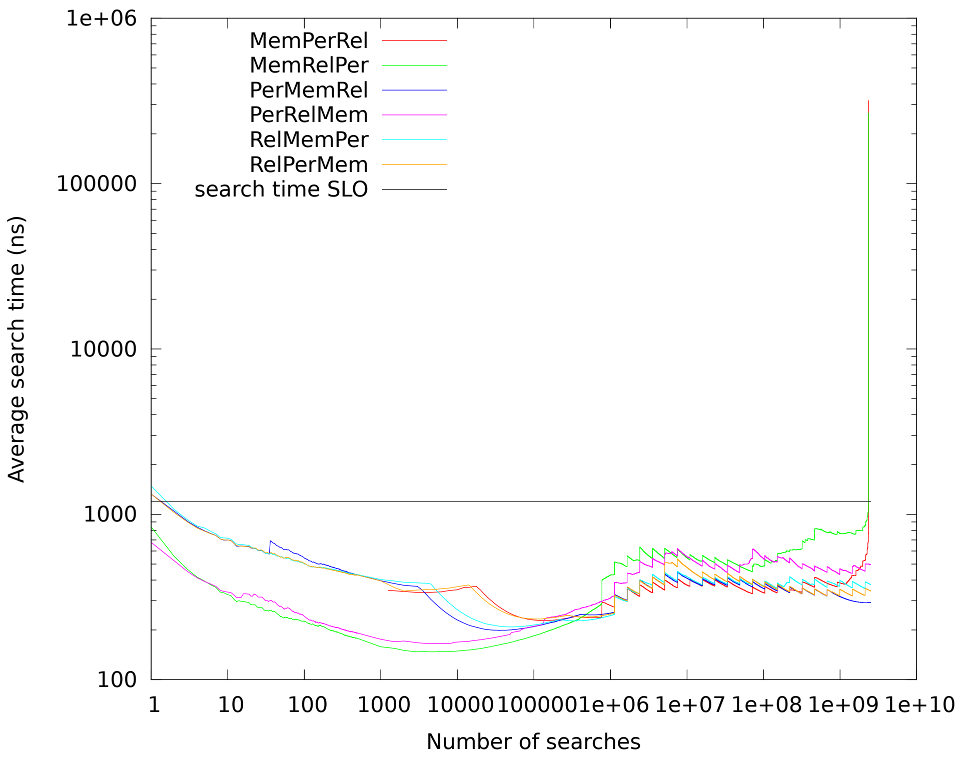


Figure 3.9: The average search time under the six priority orderings

Figure 3.10 exhibits memory consumption under the six priority orderings. It illustrates that when memory has the highest priority, the consumed memory space is the least. By contrast, when memory consumption is the lowest in order of priority, the framework occupies more memory space so as to boost performance or reliability. This figure also indicates that when the memory limit is reached, MemPerRel and MemRelPer attempt to reduce the size of the currently-used sparse Bloom filter in an effort to save memory. The reduction is achieved through diminishing the number of AVL trees. Once the number of AVL trees cannot be reduced, out-of-core storage is activated.

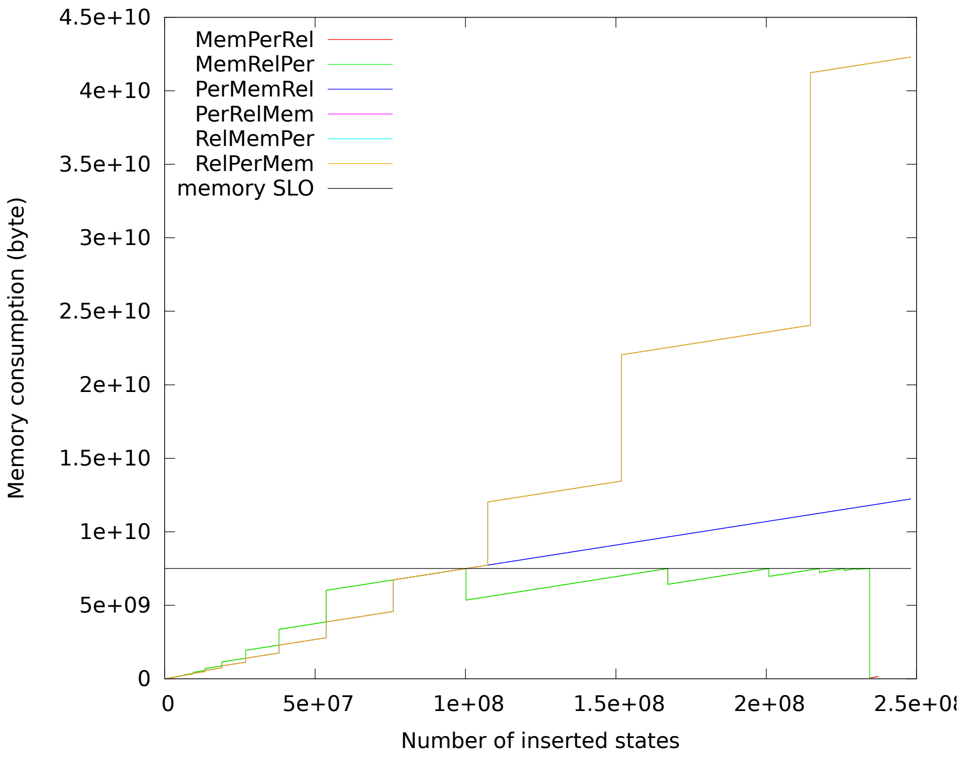


Figure 3.10: The memory consumption under the six priority orderings

Figure 3.11 depicts reliability variation among the six priority orderings. As can be seen, when reliability has higher priority over performance and memory consumption (RelPerMem and RelMemPer), the framework adapts its underlying data structure to maintain desirable reliability – over 0.99. However, when the priority of reliability is the lowest, the reliability may decline as the number of stored states increases. In addition, notice that PerRelMem boosts the reliability when the number of stored states is approximately 100 million while PerMemRel does not. That is because in the latter case memory consumption has higher priority than reliability. Hence, when the actual reliability is lower than the desired reliability, PerMemRel will not enhance reliability to protect the memory quota.

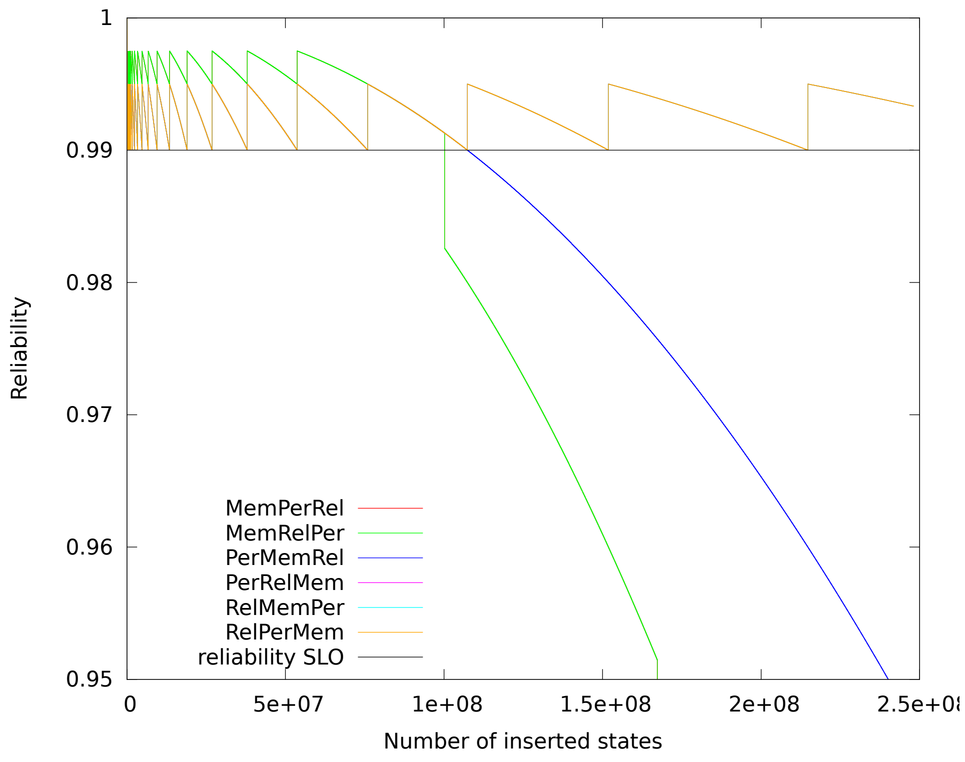


Figure 3.11: The reliability under the six priority orderings

3.6.3 Exploiting Out-of-core Storage

As can be seen in Figure 3.4, the variable *unexplored* adopting our framework is assigned a memory constraint, 40 MB. The memory consumption of this variable adopting an STL queue and our framework is depicted in Figure 3.12. It illustrates that adopting out framework keeps memory consumption under 40 MB. That is because our framework only allocates 40-MB memory space. When this space is full, the body of the queue is moved to out-of-core memory.

3.7 Conclusion

This chapter has discussed the design and implementation of a self-adaptive container framework with an embedded self-adaptive mechanism. This mechanism monitors managed resources, periodically analyses operation profiles, plans adaptation actions, and executes them. During the analysis phase, the assigned SLOs are retrieved to compare with the operation profiles reported from the monitoring phase, and various adaptation actions are evaluated to choose a proper action capable of satisfying

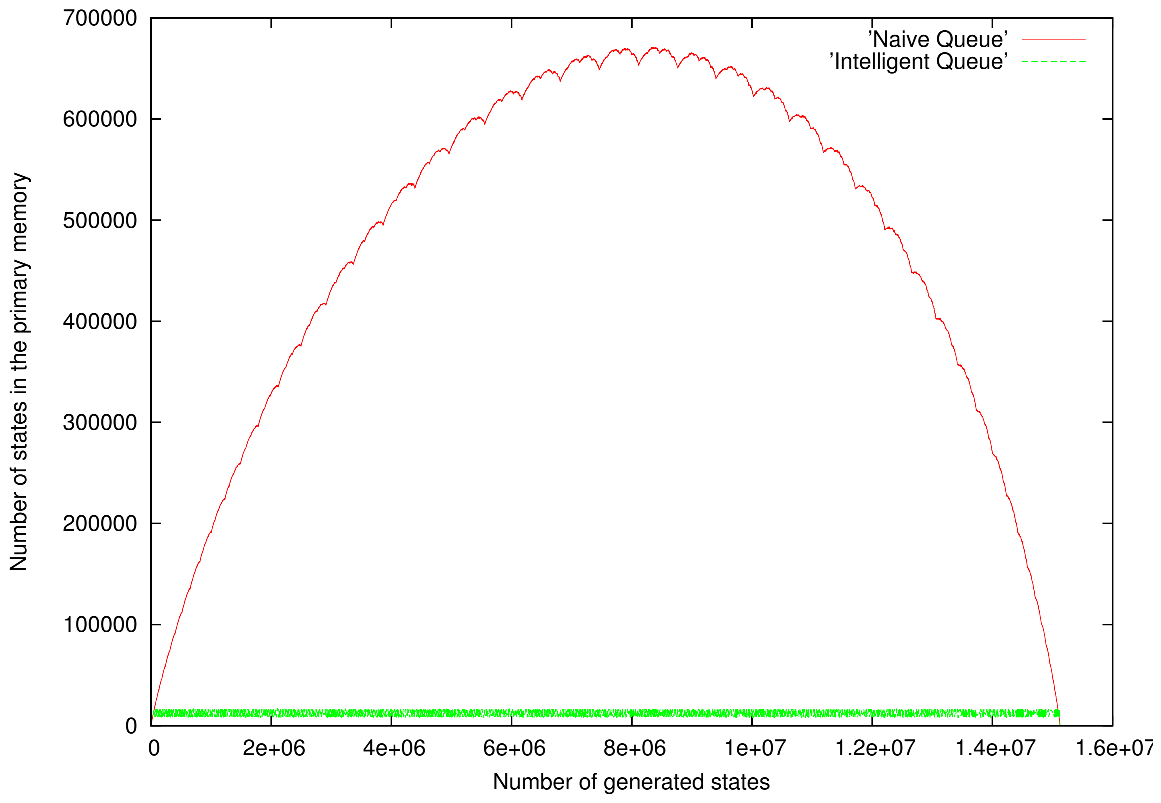


Figure 3.12: The memory consumption of *unexplored* adopting STL's queue (naïve queue) and our framework (intelligent queue)

violated SLOs. This mechanism enables software to achieve a variety of QoS without reimplementation. In addition, compared with conventional containers, our framework yields better performance, consumes less primary memory, and provides higher reliability. By adjusting the SLO priority ordering, our framework exhibits different behaviour. If memory has the highest priority, out-of-core storage is activated to save primary memory. If performance is the highest in order of priority, the underlying data structure is subdivided to boost performance. If reliability is the highest in order of priority, the underlying data structure is also subdivided to increase reliability.

To show our framework's viability and capability, a prototype has been implemented and applied to a case study centred on explicit-space exploration. The experimental results reveal that cumulative insertion time is reduced by 74.9%, cumulative search time is reduced by 89.2%, and memory consumption is reduced by 90% compared to an STL *set*. Furthermore, by means of deployment of out-of-core storage, the memory consumption of FIFO queues supported by our framework is merely 1% of memory space consumed by an STL *queue*. At the same time, programmer overhead is kept low in terms of the degree to which code is modified.

The framework has been implemented and proved as a concept. However, the limited functionality restricts its applications. In the next chapter, we will extend its functionality through the implementation of two important functionalities: key-value stores and priority queues. In addition, a new case study will be investigated to show that our framework can be deployed in different application areas.

Chapter 4

Functionality Extension and Further Case Study

‘To improve is to change; to be perfect is to change often.’

Winston Churchill, English Statesman (1874-1965)

4.1 Introduction

In the previous chapter, we presented a self-adaptive container framework for preventing frequent code refactoring and implemented a prototype with limited functionalities. This prototype was applied to a fundamental algorithm so as to prove the framework’s feasibility and to show its applicability. This chapter will extend our framework’s functionality through the implementation of a key-value store container and priority queues in the previously-developed container. Key-value stores are a form of data storage. They are supported in many programming languages such as C++, Java, and Python. In recent years, due to the explosion of data capacity, key-value stores are adopted to solve the issues resulting from relational databases. Traditionally, database systems make use of a structured way to store data, which works well when the number of stored data is thousands. However, this data storage method restricts the system to a single server. Current database systems are required to store terabytes of data, which causes relational databases serious problems. Compared to relational databases, key-value store databases only store unique keys and collections of values, which enables stored data to be easily distributed to different servers.

The other data structure now supported in our framework, priority queue, is a queue which always processes the element with the highest priority. It is widely adopted in network management. For example, in many network devices such as routers, some applications (e.g. VoIP or IPTV) can be specified priorities to ensure QoS. The specified priority is recorded in each outgoing packet, which is then sent to a priority queue. Through the utilisation of the priority queue, packets are transmitted according to their priorities.

Since our framework acts according to specified Service Level Objectives, the instance adopting the newly-supported functionalities of `ICollection` or `IKeyValue` is associated with a configuration file following the format of WSLA. In addition, an enhanced prototype supporting out-of-core storage and probabilistic data structures in both container classes is implemented. This prototype is applied to a case study centred on route planning, adopting a Dijkstra's shortest path algorithm. To evaluate these new functionalities from the perspectives of applicability and scalability, we first input a graph representing the USA road network and then compare our framework with conventional containers. Next, different SLO priority orderings are assigned so as to observe the behaviour of our framework when the new functionalities are exploited. Finally, a memory limit is assigned to a priority queue adopting our framework in an effort to illustrate the dynamic activation of out-of-core storage. The experimental results show that our framework delivers better performance and expends less memory than conventional containers do. Furthermore, the framework automatically adopts different adaptation actions based on assigned priority orderings, and the implementation of priority queues can transfer data from/to out-of-core memory at run time in order to meet memory constraints.

The remainder of this chapter is organised as follows. While Section 4.2 gives a full introduction of key value stores and how we implement them, Section 4.3 describes applications of priority queues and detailed implementation. A case study is investigated in Section 4.4. Section 4.5 concludes this chapter.

4.2 Key-Value Stores Design and Implementation

Key-value stores represent data stored in pairs of keys and values. In the 1990s, STL began to support the functionality of key-value stores via *map*. After that, Java and Python also implemented key-value stores (e.g. *Map* in Java and *dict* in Python) as well. In the early 2009, the concept of key-value stores commenced being adopted in the field of databases in order to deal with large-scale data. Traditionally, database developers make use of relational databases to store data, which can be managed via SQL. This mechanism works well when all data can be stored and manipulated in a single server. However, as workload increases, stored data has to be distributed to multiple servers, which may violate the properties of Atomicity, Consistency, Isolation, and Durability (ACID) in relational database systems. As a result, key-value store databases are proposed to provide a flexible mechanism for dealing with large-scale data. Many industries managing large-scale data e.g. Amazon (DeCandia et al., 2007), Facebook (Atikoglu, Xu, Frachtenberg, Jiang, & Paleczny, 2012), Twitter (Fitzpatrick, 2004; Petrovic, 2008) have adopted key-value stores in the form of NoSQL databases e.g. Cassandra (Apache, 2014), Riak (Basho, 2014), Tokyo Cabinet (FAL Labs, 2012), Aerospike (Aerospike, 2014). In addition, many libraries e.g. sparkey (Bruggmann, 2014), LevelDB (Google, 2013), YDB (Majkowski, 2010) can be used to implement in-memory key-value store databases.

Our framework implements the functionality of key-value stores in `IKeyValue`, which chooses either a tree data structure (e.g. AVL tree or red black tree) or a modified sparse Bloom filter depending on specified operation descriptors and Service Level Objectives. If iterator-based operations are required and the reliability requirement is 100%, a tree data structure is selected. Otherwise, a modified sparse Bloom filter is chosen.

As operations are performed through `IKeyValue`, the self-adaptive mechanism keeps measuring per operation response time, computing memory use, and where appropriate calculating reliability. If any of them violates assigned SLOs, an adaptation action discussed in Section 3.3.3 may be performed to satisfy the violated SLO. The subdivision of the underlying data structure can improve either performance or reliability. However, when out-of-core storage is activated to meet a memory limit, the functionality of the direct access operator (i.e. *operator[]*) will fail. The direct access operator may be used either as a lvalue, which appears on the left-hand side of an assignment expression, or as a rvalue, which appears on the right-hand side of an assignment expression. Hence, the return type of the direct access operator should be the reference of the stored object. When elements are stored in primary memory, *operator[]* can directly return the stored object. When elements are stored in out-of-core memory, *operator[]* cannot return the reference. To solve this problem, we design a proxy class which overloads the assignment operator (i.e. *operator=*) and the cast operator (i.e. *operator()*), whose code is shown as follows:

```
template<class K,class V>
class ProxyClass
{
      ProxyClass<K, V>& operator= (const V& rhs) {
      {
         // for lvalue
      }
      operator V() {
      {
         // for rvalue
      }
};
```

When *operator[]* is used as a rvalue, it invokes the overloaded *operator()* of the proxy class, which retrieves the mapped value from out-of-core memory. When *operator[]* is used as a lvalue, it calls the overloaded *operator=*, which writes the new value to the mapped value in external memory. In addition, the return type of *operator[]* is converted from the reference of the mapped value to the reference of a proxy instance.

4.3 Priority Queue

Priority queues are a data structure which is frequently adopted in operating systems, discrete event simulation, pathfinding algorithms, and data compression. A commonly-seen application of priority queues is the process management of an operating system. Through priority queues, operating systems can execute processes according to priorities of processes. This ensures that processes with higher priorities do not need to wait for processes with lower priorities. Additionally, the same technique can also be used in the interrupt handling, which permits operating systems to deal with interrupt requests according to their priorities. The propose of discrete event simulation is to simulate a series of events such as traffic. These events are assigned time specifying when they should be activated and then are pushed into a priority queue, which enables the simulation to easily obtain the next event that has to be triggered. Pathfinding algorithms (e.g. Dijkstra's shortest path algorithm and A* search algorithm) make use of priority queues to store unexplored routes. One popular technique of data compression is called Huffman coding. This exploits priority queues to construct a tree transforming characters into bits. The above-mentioned applications have shown that priority queues are an important data structure, especially when events/tasks have to be processed in accordance with their importances.

The functionality of priority queues is implemented in `ICollection` using heap as the underlying data structure. Heaps provide push operations, pop operations, and top operations, which always return the element with the highest priority. In order to compare priorities of different elements, `ICollection` accepts a custom comparison operator as an optional template parameter, whose default value is the less-than operator (*less*). As can be seen in Figure 4.1, our implementation of priority queues is composed of a data heap and an index heap. Before out-of-core storage is activated, push, pop, and top operations are performed in the data heap. Hence, the index heap is empty. When memory limits are met, out-of-core storage is activated. After this, push operations still manipulate the data heap. When primary memory limits are reached, the data heap is transformed into an array sorted according to priorities. This allows fast retrieval of the element with the next higher priority. The first element in the array is inserted into the index heap. The sorted array is then moved to out-of-core memory. After this, the data heap becomes empty. The above-mentioned actions may be performed several times to maintain primary memory use at a desirable level. Pop and top operations manipulate either the data heap or the index heap according to the priorities of the two heaps' roots. For top operations, the priorities of the two roots are compared to decide which element should be returned. For pop operations, if the data heap's root has higher priority, it will be removed. If the index heap's root has higher priority, it will be removed and the next element, which is stored in the file where the original root is stored, is inserted into the index heap. For example, if element l is removed, element m will be inserted into the index heap.

Our implementation does not consider performance-related and reliability-related adaptations for two

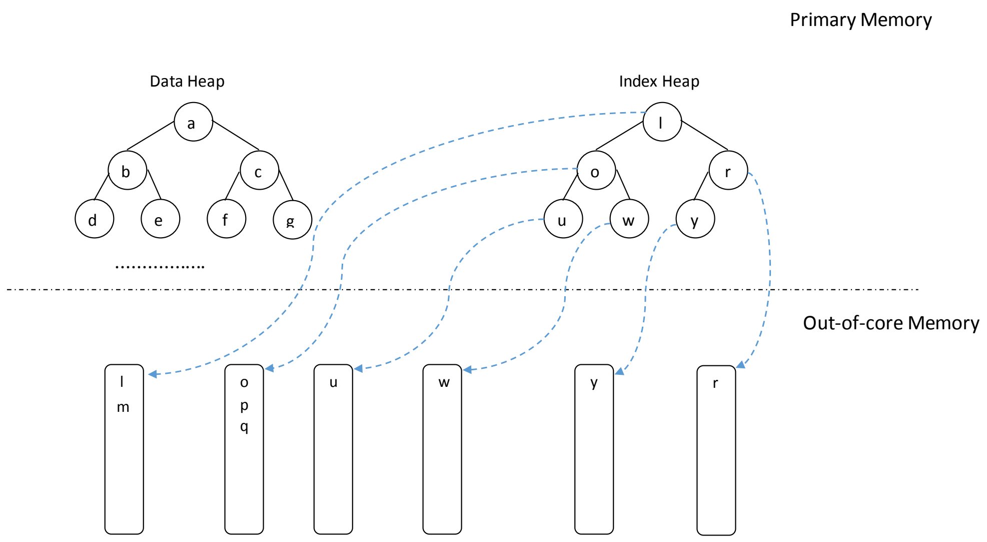


Figure 4.1: The underlying data structures of the priority queue

reasons. First, when priority queues are adopted, contents of stored elements need to be retrieved, which implies that stored elements cannot be converted into hash keys. Hence, the reliability requirement has to be 100%. Second, top operations, push operations, and pop operations have reached their optimal time complexity. This means that subdividing the underlying data structure does not improve performance. As a result, our implementation only focuses on memory-related SLOs.

4.4 Case Study

This section illustrates the capability of key-value stores (via `IKeyValue`) and priority queues (via `ICollection`) through a route-planning case study adopting a Dijkstra's shortest path algorithm. The programs adopting the STL and our framework are shown in Figure 4.2. As can be seen, the algorithm makes use of two container variables. One is for a table which stores the shortest distances from a given random node to all the other nodes (*Distance*) and the other one is for a priority queue whose first element is always the node with the shortest distance (*PQ*). The only difference between the two programs is container declarations. To evaluate the framework's effectiveness and scalability, a graph depicting the USA road network (DIMACS, 2006), which consists of 23 million nodes and 58 million edges, and the following SLOs are input .

For *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 consumption should be no more than 500 MB.

and for *PQ*, its primary memory use should be less than 300 KB.

The performance and memory consumption are then compared, using the STL's containers and our framework. Next, *Distance*'s SLOs are assigned in different sequences so as to observe the impact of priority orderings. Finally, an STL's *priority_queue* and our framework are utilised to compare memory variation of *PQ*.

```
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]));
            }
        }
    }
}
```

```
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 4.2: The naïve Dijkstra's shortest path algorithm (left) and the same algorithm adopting our framework (right)

4.4.1 Comparison with Conventional Containers

The average insertion time and average update time expended by an STL map and our framework under the priority ordering of performance, reliability, and primary memory use are displayed in Figures 4.3 and 4.4. As can be seen in Figure 4.3, the average insertion time consumed by our framework is slightly higher than that of the map. That is because the framework performs adaptation actions, which can be observed in sudden rises of insertion time, in order to boost performance or reliability. Although the adaptation actions add time to insertion operations, they successfully enable our framework to achieve the insertion-time SLO and the update-time SLO. In addition, Figure 4.3 shows that when x-value (cumulative number of invoked insertion operations) is small our framework expends more insertion time. That is because our framework requires more time to initialise its first stored element. By contrast, Figure 4.4 indicates that when x-value (cumulative number of invoked update operations) is small our framework yields better performance. That is because all data has been stored in the STL map and our framework when updated operations are invoked. As a result, the STL map needs more time to locate target elements.

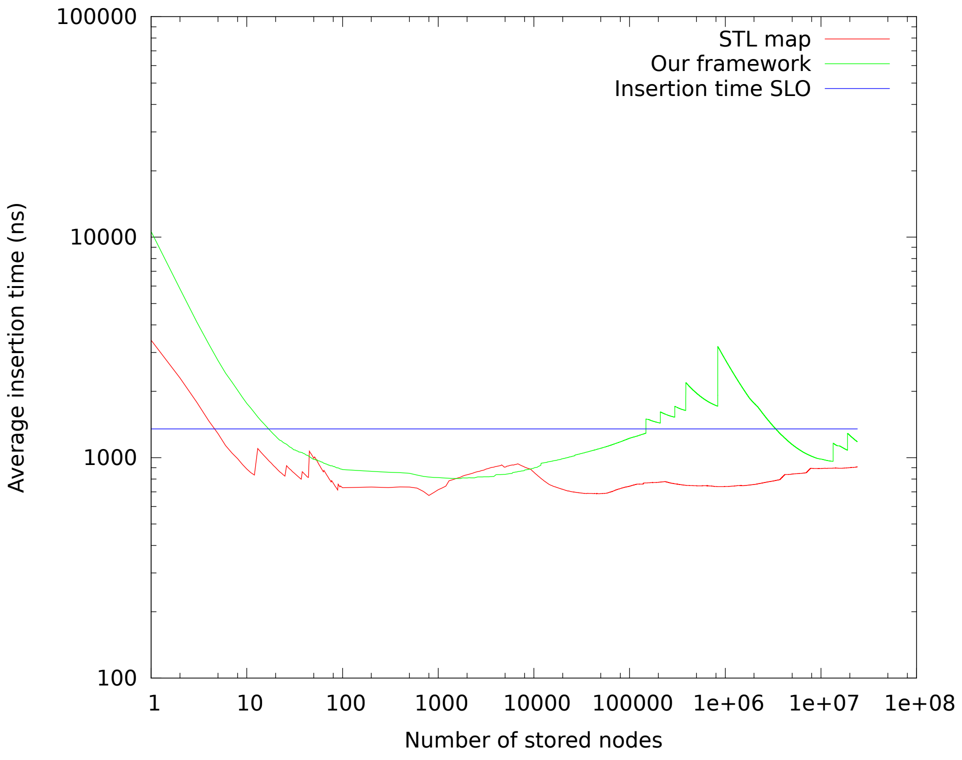


Figure 4.3: The average insertion time of *Distance* adopting the STL map and our framework

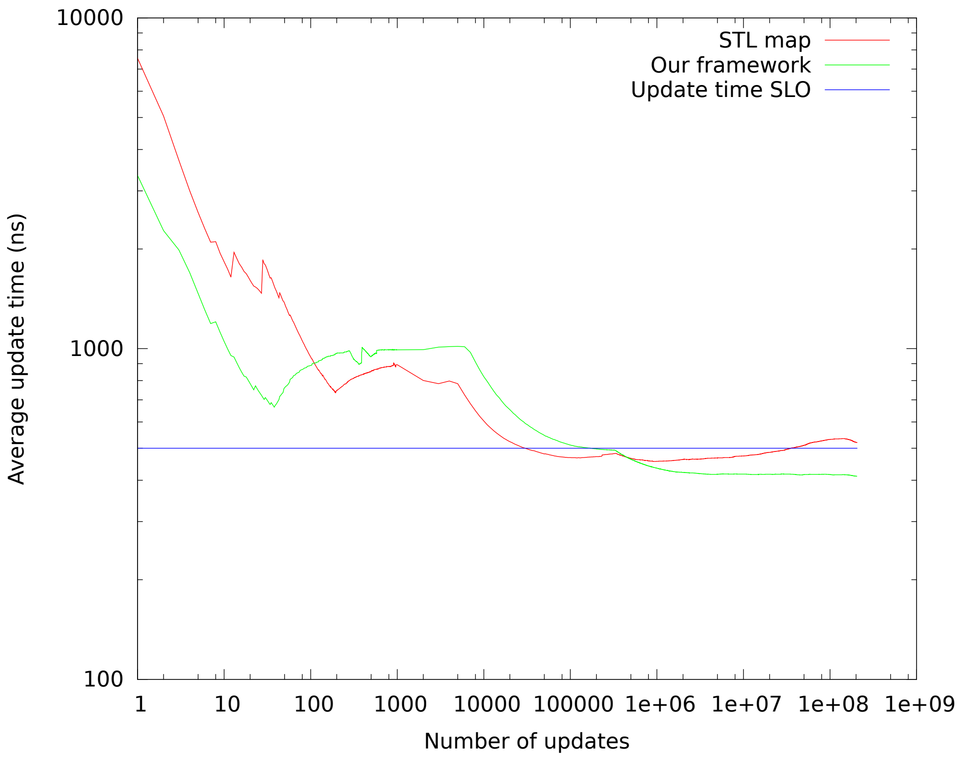


Figure 4.4: The average update time of *Distance* adopting the STL map and our framework

Figure 4.5 depicts the memory consumption of the STL map and our framework. It obviously shows that our framework uses considerably less memory space than the map. Furthermore, the memory limit is violated because performance and reliability have higher priority than primary memory use.

4.4.2 Influence of SLO Priority

Figures 4.6 and 4.7 display the average insertion time and update time under different priority orderings. The two figures illustrate that when the performance-related SLOs have higher priority than the memory-related SLO, the framework expends less insertion time and update time. This phenomenon can be seen in the following priority orderings: PerMemRel, PerRelMem, and RelPerMem, which force the framework to boost performance even if the memory-related SLO is violated. By contrast, if the memory-related SLO is the highest in order of priority (i.e. MemPerRel and MemRelPer), the insertion time and update time will dramatically increase, which is caused by the access of slow out-of-core memory.

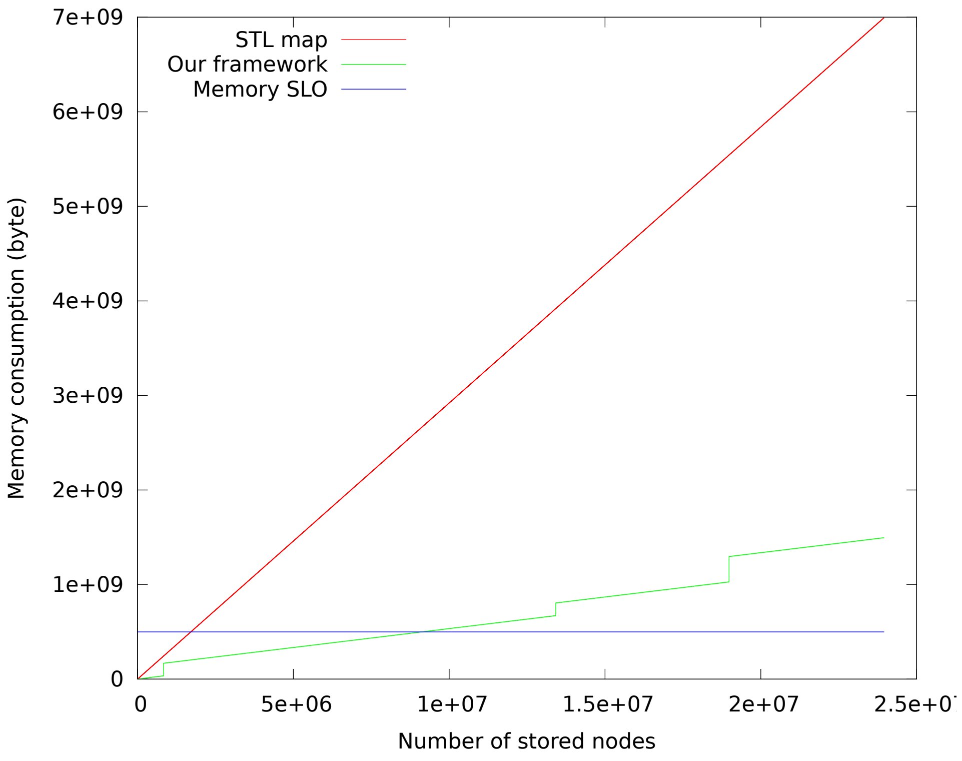


Figure 4.5: The memory consumption of *Distance* adopting the STL map and our framework

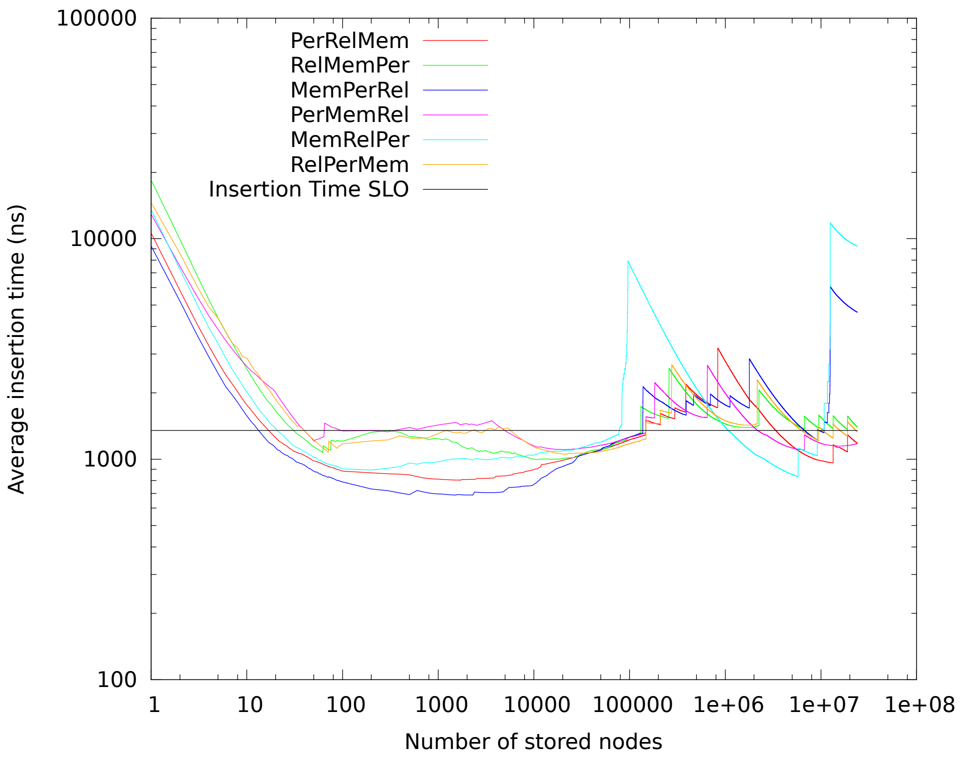


Figure 4.6: The average insertion time of our framework under the six priority orderings

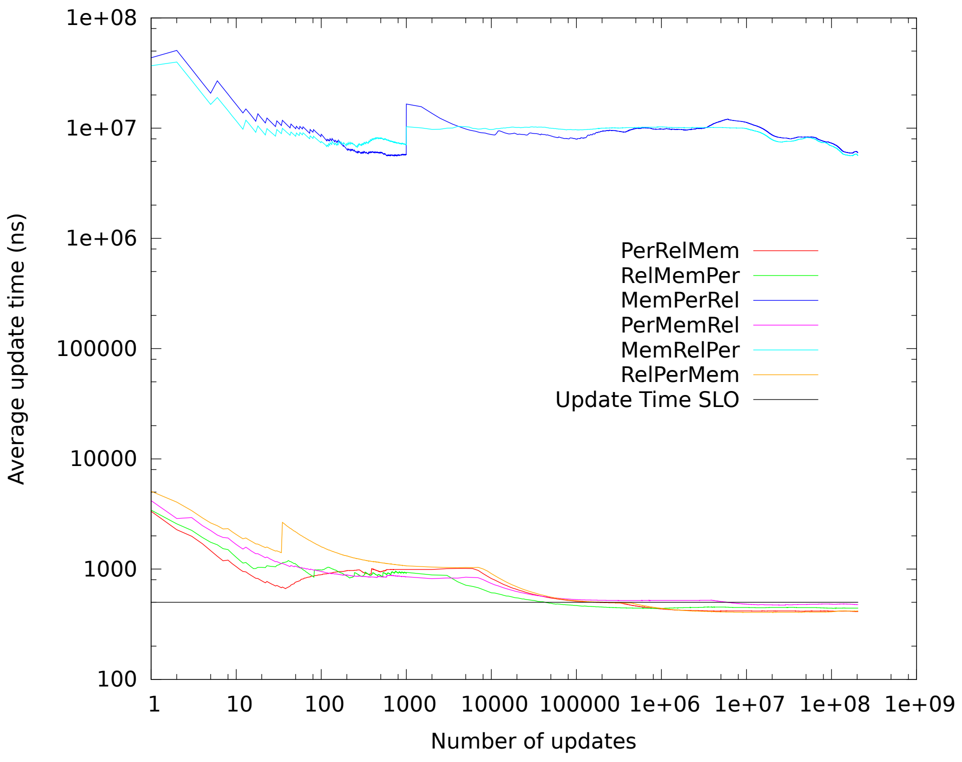


Figure 4.7: The average update time of our framework under the six priority orderings

The memory use of the six priority orderings is displayed in Figure 4.8, which shows two types of behaviour based on the priority of the memory-related SLO. First, when the priority of the memory-related SLO is lower than that of the other SLOs (i.e. PerRelMem, PerMemRel, RelPerMem, and RelMemPer), the framework consumes more memory space to boost performance or reliability. Second, when the memory-related SLO has the highest priority, the consumed memory space is the least. This figure also indicates that when MemPerRel and MemRelPer reach the memory limit, our framework, whose currently-used data structure is an improved sparse Bloom filter, begins to reduce the number of AVL trees to save memory space until its number is 1. After that, out-of-core storage is activated. Furthermore, PerMemRel and PerRelMem behave differently when the number of inserted nodes is approximately 10 million. PerRelMem has a sudden rise in memory consumption, but PerMemRel does not. For PerRelMem, its reliability has higher priority than memory. Hence, when reliability is lower than the desired level (i.e. 0.995), the framework will enhance it without consideration of memory use. But, for PerMemRel, its memory has higher priority than reliability. As a result, reliability can only be enhanced when memory use is lower than the memory constraint (i.e. 500 MB).

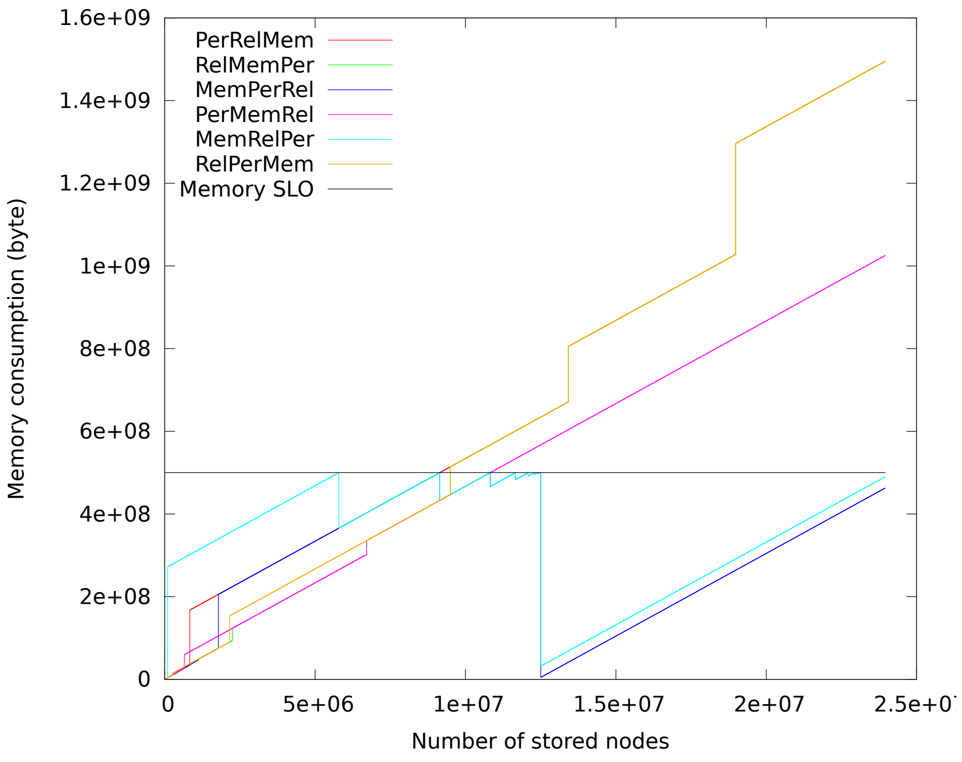


Figure 4.8: The memory consumption of our framework under the six priority orderings

The dynamic behaviour of our framework with respect to reliability when assigned the six priority sequences is shown in Figure 4.9. This shows that RelPerMem and RelMemPer rebound several times to keep the reliability at a desirable level (i.e. over 0.995). This figure also shows that when the reliability-related SLO is lower than the other SLOs, the framework's reliability may descend as the number of inserted nodes increases (i.e. MemRelPer, MemPerRel, PerMemRel). Take MemRelPer for example. Memory consumption has higher priority than reliability, which implies that reliability cannot be improved once the memory limit is reached. Furthermore, the reliability of MemRelPer sharply deteriorates after adaptation actions, which reduce the number of AVL trees, are taken.

4.4.3 Exploiting Out-of-core Storage

The memory limit of the variable, *PQ* seen in Figure 4.2, is 300 KB. The primary memory consumption using our framework and an STL's *priority_queue* are shown in Figure 4.10. It indicates that when the memory limit is met, out-of-core storage is activated to reduce primary memory consumption. In addition, adaptation actions are performed many times to protect the memory limit. Hence,

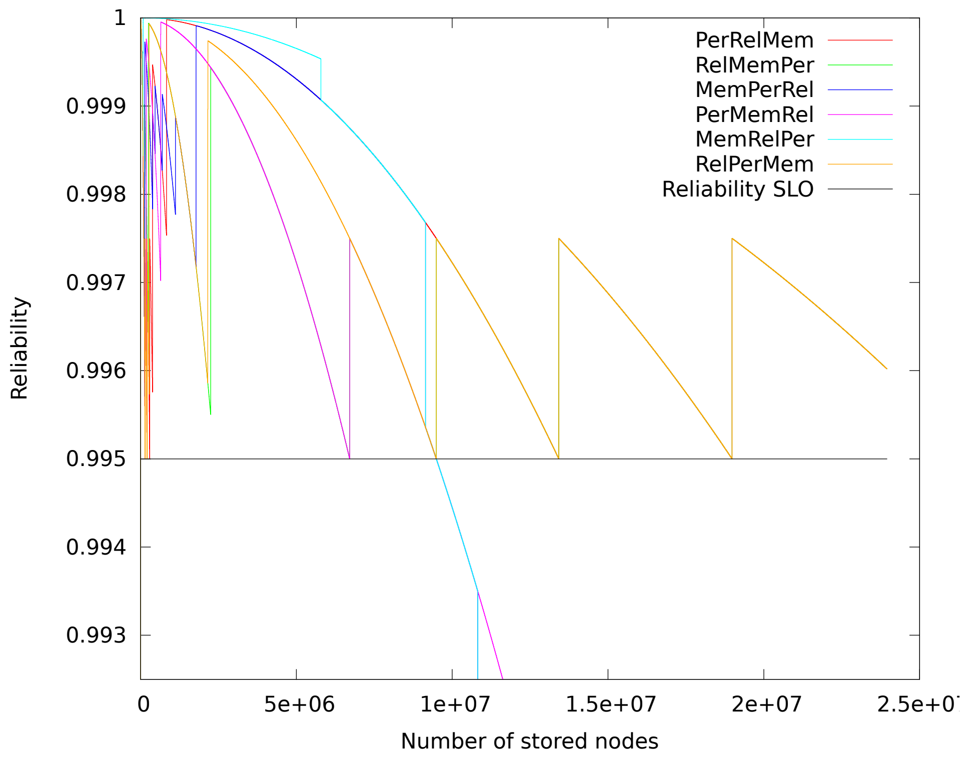


Figure 4.9: The reliability variation of our framework under the six priority orderings

our framework only consumes 300 KB, which is a mere 22% of the memory space consumed by the STL *priority_queue*.

4.5 Conclusion

The chapter has broadened the previously-developed prototype by means of the implementation of key-value stores and priority queues. The functionality of key-value stores is implemented in `IKeyValue` and that of priority queues is supported by `ICollection`. Both of them can dynamically exploit out-of-core storage and probabilistic data structures to satisfy specified SLOs. We also have presented how to support out-of-core storage in the new functionalities. For key-value stores, a proxy class overloading the cast operator and the assignment operator has been implemented so as to provide the direct access operator. For priority queues, to supply efficient out-of-core storage, the underlying data structures are divided into two heaps. One is for storing data in primary memory, and the other is for reducing the number of disk I/Os.

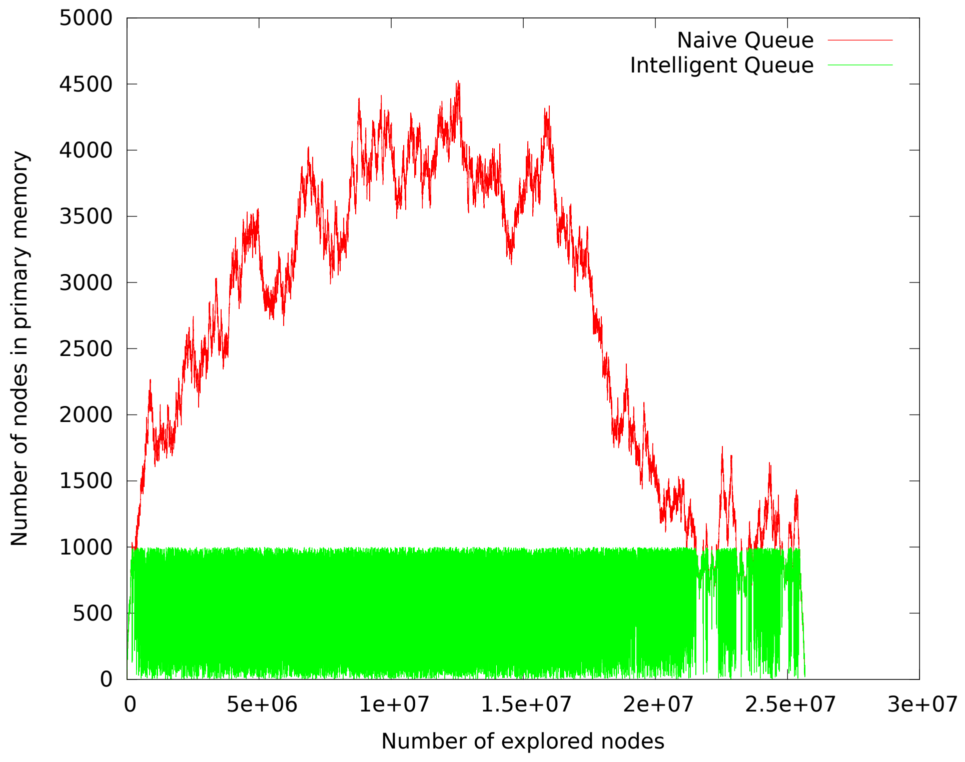


Figure 4.10: The memory consumption of *PQ* using the STL *priority_queue* (naïve queue) and our framework (intelligent queue)

The enhanced prototype has been evaluated by means of a case study centred on route planning, adopting a Dijkstra's shortest path algorithm with the input of a graph representing the USA road network. The experimental results suggest that our implementation provides better performance in terms of insertion time and update time and consumes less memory compared to an STL *map* and *priority_queue*. Furthermore, the framework exhibits different behaviour when Service Level Objectives are assigned in different priority orderings. The results also show that our framework implementation dynamically exploits out-of-core storage to reduce primary memory use.

Chapter 5

Interoperability Extensions and Cloud Integration

'Nature is a mutable cloud which is always and never the same.'

Ralph Waldo Emerson, American Essayist, Lecturer, and Poet (1803 –1882)

5.1 Introduction

This chapter will describe the cooperation between the framework and third-party container libraries. Through them, the framework can provide a wider class of Service Level Objectives and prevent the implementation of complex techniques (e.g. parallelism and out-of-core storage) from scratch. Furthermore, the integration of cloud storage is presented as well. By means of cloud storage, the framework is capable of providing alternative out-of-core memory and taking charge of the responsibility of moving data from and to cloud storage.

As described in Section 2.7, many research teams have proposed their container frameworks, which are efficient and well-developed. Although these libraries are designed as close to the standard libraries as possible, programmers still need to learn how to configure them. Take STXXL's *map* for example. Before its instance is declared, programmers have to configure the following definitions.

```
#define DATA_NODE_BLOCK_SIZE (4096)
#define DATA_LEAF_BLOCK_SIZE (4096)
template<class T>
struct Compare
{
      bool operator () (const T& a, const T& b) const ;
      static T max_value() ;
};
```

where the first two lines of code specify the sizes of nodes and *Compare* notifies STXXL of how to compare keys. This increases the complexity of exploiting STXXL. Hence, this chapter will show how our framework cooperates with these third-party containers in order to transfer the effort of configuring these containers from programmers to our self-adaptive mechanism, which decides when to trigger them and how to configure them on-the-fly. The third-party container libraries we choose to integrate are STXXL (Dementiev et al., 2005), which provides efficient out-of-core containers and algorithms, and Intel Threading Building Block (Intel, 2014), which supplies parallel containers and algorithms. The two container libraries allow our framework not only to support out-of-core storage and parallelism at low cost but to provide a wider class of Service Level Objectives, especially those related to performance and memory efficiency.

This chapter will also show that the framework is capable of exploiting cloud storage at run time. To the best of our knowledge, no library has the ability to do this. This ability implies that alternative out-of-core memory can be provided when local memory (i.e. RAM and disk) is not available, and data transfer from and to cloud storage is managed by our framework. As a result, programmers do not need to reimplement their software when cloud storage services change.

Similarly, the framework will be evaluated through two case studies centred on explicit state-space exploration and route planning. Through utilising STXXL, our framework can provide various out-of-core data manipulation in a low-cost way. Through deploying Intel TBB, performance in terms of insertion, search, update, push, and pop time is considerably boosted. By means of integrating cloud storage, our framework dynamically moves data from/to cloud storage so as to reduce primary memory use.

The remainder of this chapter is organised as follows. Section 5.2 and 5.3 introduce the cooperation of STXXL and Intel TBB, respectively. The integration of cloud storage is described in Section 5.4. Two case studies, explicit state-space exploration and route planning, are investigated in Section 5.5. Section 5.6 concludes this chapter.

5.2 The Integration of Out-of-core Container Frameworks

Our framework integrates STXXL (Dementiev et al., 2005) and MCSTL (Singler et al., 2007) to illustrate the ability of dynamically deploying third-party out-of-core containers. STXXL is a C++-based library which aims to deal with extra large data sets in out-of-core memory. Compared with other out-of-core libraries e.g. TPIE (Vengroff, 1994), LEDA-SM (Crauser & Mehlhorn, 1999), Persistent STL (Gschwind, 2001), STXXL supports parallel algorithms, which enable data to be simultaneously processed in different disks. Furthermore, the techniques of “pipelining” and “overlapping” are utilised to improve performance and resource utilisation, respectively. The containers supported by STXXL are described below.

- *vector* is an external data structure, which is divided into equal-sized blocks. Some blocks are kept in internal memory as cache, the storage requirement for which relies on the number of pages in cache (*CachePages*), the number of blocks in a page (*PageSize*), and the size of a block in bytes (*BlockSize*). Hence, the internal memory consumption is $CachePages \times PageSize \times BlockSize$.
- *stack* contains four types of stacks (e.g. *normal_stack*, *grow_shrink_stack*, *grow_shrink_stack2*, and *migrating_stack*). *normal_stack* is a general-purpose stack which keeps two pages as buffers. When one page is empty, a new page will be loaded from external memory. Similarly, when two pages are full, one page is written into external memory. As a result, its internal memory consumption is $2 \times PageSize \times BlockSize$. *grow_shrink_stack* is an external stack which is utilised by a series of push operations followed by a series of pop operations. This kind of stack grows to is maximum capacity and then shrinks. Due to this access pattern, *grow_shrink_stack* allows prefetching and buffered writing to boost performance. Because the

mechanism of data transfer from/to external memory is not changed, it consumes the same internal memory as *normal_stack* does. *grow_shrink_stack2* also aims to deal with the same access pattern as *grow_shrink_stack* but its buffers can be shared with other stacks. Its internal memory use, therefore, is *BlockSize* plus shared buffers. *migrating_stack* enables programmers to specify memory constraints over which data will be moved to external memory. Its internal memory consumption depends on where data is stored. Before data is transferred, its maximum memory use is equal to the memory constraint. After data is migrated, it only exploits some pointers to maintain status.

- *queue* makes use of two blocks of internal memory to hold head and tail blocks. Thus, the internal memory consumption is $2 \times BlockSize$. Our prototype also adopts the same way to implement out-of-core FIFO queues. Furthermore, STXXL's out-of-core queue supports prefetching and buffered writing, which enhances I/O performance.

- *deque* is an adaptor on top of *vector*. Hence, its maximum memory consumption is estimated to be $CachePages \times PageSize \times BlockSize$.

- *map* is an associative container which stores pairs of unique keys and values. Its underlying data structure is a B+-tree, which enables pre-retrieval of neighbour leaves because all leaf nodes are linked together. To boost performance, the root node as well as the most frequently used internal and leaf nodes are kept in primary memory. An important contribution of STXXL's *map* is that it is the first C++ library that provides I/O-efficient iterator-based search operations.

- *unordered_map* is a hash map. The main issue of an external hash map is its performance, which is seriously affected by the number of I/Os. This can be solved through the increase in buffered memory. Larger buffered memory results in better performance but leads to more internal memory consumption as well. As a result, the size of buffered memory should be assigned according to primary memory constraints.

- *priority_queue* is constructed by a sequence heap composed of R merge groups (G_1, G_2, …, G_R) where G_i contains up to k sorted sequences. To boost performance, it utilises three types of buffers. The first type of buffer is the group buffer, which stores the first m smallest (or largest) elements. The second type of buffer is the deletion buffer, which holds the smallest elements of the group buffers. The final type of buffer is the insertion priority queue, which keeps the newly inserted elements. These three types of buffers occupy primary memory, whose consumption is limited by a constructor parameter.

- *matrix* is an external container supporting matrix operations (e.g. addition, subtraction, multiplication, and transposition). It splits stored data into a variety of square submatrices, whose size equals the specified block size.
- *sorter* is a container which keeps inserted elements in a programmer-specified order. It features two phases of operations. In the first phase, elements are presorted when they are inserted (via *push*). Once the limit of primary memory is reached, they will be written into external memory. The second phase is activated when *sort* is invoked. In this phase, *push* is disabled, and iterator-based operations (e.g. *operator++*, *operator**) can be used (i.e. they cannot be utilised in the first phase). The memory limit of *sorter* relies on the specified block size.
- *sequence* is similar to STXXL's *deque* but does not support random access. It adopts the same implementation as STXXL's *queue*, which implies that its maximum primary memory consumption is also $2 \times BlockSize$.

MCSTL (Singler et al., 2007), which can cooperate with STXXL for the purpose of internal computation improvement, is an OpenMP-based (Dagum & Menon, 1998) algorithm library exploiting multiprocessors or multi-cores of a processor. Software adopting MCSTL is able to achieve performance improvement without any changes due to the common algorithm names shared by MCSTL and STL. In other words, all of the algorithms supported by MCSTL (e.g. *sort*, *random_shuffle*, *partition*, *merge*, *find*, *nth_element*, *partial_sum*, and *for_each*) can be found in STL. In spite of these advantages, MCSTL currently restricts adopted compilers (i.e. gcc and g++) to lower versions (lower than version 4.2).

Through experimentation with STXXL, it can be found that properly configuring STXXL is difficult. A correct configuration highly depends on its memory constraints. Furthermore, when to activate STXXL is important as well. If sufficient primary memory is available, the deployment of STXXL will cause serious performance decline. These issues are addressed by our framework, which activates STXXL only when primary memory limits are reached. Simultaneously, proper configuration, which is computed according to memory constraints, is assigned to STXXL. The automatic deployment allows programmers to skip the time of learning how to utilise STXXL and that of reimplementing software for different memory constraints.

5.3 Parallelism Integration

Parallelism is a commonly-used technique for boosting performance. However, its complexity (e.g. synchronisation) leads to high programmer effort and high levels of expertise. To reduce this, our framework cooperates with Intel Threading Building Blocks (TBB) (Intel, 2014) to provide parallel manipulation of stored data.

Intel TBB is a concurrent STL-like library which supplies parallel containers and algorithms in order to reduce the complexity of developing multi-threaded software. Furthermore, it has the ability to detect the number of CPU cores, which prevents reconfiguration when software runs on different execution environments; it adopts the technique of task stealing (Singh, Holt, Totsuka, Gupta, & Hennessy, 1995), which enables tasks to be dynamically reassigned to different CPU cores, so as to enhance core utilisation. The containers supplied by TBB include *concurrent_hash_map*, *concurrent_vector*, *concurrent_queue*, *concurrent_bounded_queue*, *concurrent_priority_queue*, *concurrent_unordered_set*, *concurrent_unordered_multiset*, *concurrent_unordered_map*, and *concurrent_unordered_multimap*. We will introduce these containers as follows:

- *concurrent_hash_map* is a hash map that stores pairs of keys and values. Each key in the map is unique and not sorted. When a custom hash function is required, it has to be encapsulated in a class with a copy constructor, a destructor, an *equal* function, which compares the equality of two keys, and a *hash* function, which should return the data type, *size_t*. As a result, depending on the execution environments where TBB is deployed, hash keys may contain 32 bits or 64 bits.

- *concurrent_vector* is a concurrent vector, which does not support *pop_back*. To concurrently and safely insert data, insertion operations are transformed into *push_back*, *grow_by*, and *grow_to_at_least*. Furthermore, when *size* is invoked, it may count the number of elements which are appended by the insertion functions.

- *concurrent_queue* is a concurrent queue without memory limits, and *concurrent_bounded_queue* is a concurrent queue with the limit of maximum capacity. Due to the limit of maximum memory use, *push* operations in *concurrent_bounded_queue* wait until the queue is not full. Similarly, *pop* operations in an empty queue wait until elements can be popped. Both containers do not provide *front* and *back* operations and transforms *pop* operations into *try_pop* operations for safety reasons. Compared to the STL's *pop*, *try_pop* returns a value representing if an element is successfully popped and requires a parameter for storing the popped element.

- *concurrent_priority_queue* is a priority queue allowing multiple threads to push and pop elements. Compared to the STL's *priority_queue*, *top* operations are not supported, and *pop* operations are converted into *try_pop*, whose behaviour is similar to *concurrent_queue*'s *try_pop*.

- *concurrent_unordered_set* and *concurrent_unordered_multiset* are similar to *unordered_set* and *unordered_multiset* of the STL, respectively but support thread-safe insertion and traversal operations. Because erasure operations are not concurrency safe, *erase* is converted into *unsafe_erase* to address this characteristic.

- *concurrent_unordered_map*, and *concurrent_unordered_multimap* are akin to *unordered_map* and *unordered_multimap* of the STL, respectively. As previously, erasure operations begin with *unsafe* to indicate that they are not concurrency safe.

Although TBB provides STL-like interfaces, some of their usage is changed (e.g. *pop* operations, *erase* operations), and some are removed (e.g. *top* operations, *front* operations). Furthermore, programmers have to change their code to meet memory constraints when execution environments change. The above-mentioned problems are solved by our framework, which automatically assigns memory limits to TBB according to specified memory-related SLO and supports the same interface names. For example, when the functionality of a FIFO queue is specified, programmers can still utilise *pop* operations, which are transformed into *try_pop* by our framework. In addition, the popped element which is protected by a semaphore is kept for supporting *top* operations.

5.4 Cloud Storage Integration

As described in Section 2.6, there are three types of cloud storage. Our framework adopts one public cloud storage service, Amazon Storage Service. Accessing this type of cloud storage involves establishing connections to cloud storage providers and transferring data via the Internet. These actions can be implemented though the use of third-party software APIs, e.g. WebStor (OblakSoft, 2014), Elasto (Disseldorp, 2014), JetS3t (Murty, 2014), lits3 (Farina, 2009), Dropbox-C (Python, 2014)). Our framework utilises WebStor to establish connections and transfer data from and to storage providers. 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 provide containers which employ cloud storage, three implementation challenges have to be considered. First, most cloud service providers offer free but limited post and get operations as well as capacity (e.g. Amazon S3 Free Tier provides 20 000 get requests, 2 000 put requests, and 5 GB of standard storage). The second challenge is the lack of search operation. Some cloud storage service providers do not support search operations, which causes malfunctions of some operations (e.g. *search*, *operator[]*). Consequently, the performance of searching for an element can involve downloading of the file that may contain the element and searching of the element on the file, which highly depends on the size of the downloaded file. The third challenge occurs when the value of an element kept in cloud storage should be updated. In the worst case, this involves downloading, updating, and uploading. The performance of the whole process also depends on the size of the file. To efficiently resolve these challenges, the number of put and get operations and cloud memory consumption should be reduced. Hence, when cloud storage is activated, data is split into different cloud space by our framework. This division cuts occupied memory in each space and reduces the numbers of push and pop operations since the file in each space is smaller.

5.5 Case Study

This section will apply the framework integrating STXXL, MCSTL, Intel TBB, and Amazon S3 to explicit state-space exploration adopting a breadth-first search algorithm and route planning adopting a Dijkstra's shortest path algorithm in order to observe the impact when these third-party containers and cloud storage are deployed. The two algorithms are shown in Figure 3.4 and 4.2, respectively. As can be seen, adopting our framework only needs to modify container declaration. In addition, to evaluate capability of the framework exploiting STXXL and Intel TBB, the BFS algorithm is assigned approximately 240 million states, and the Dijkstra's shortest path algorithm is given the USA road network, which contains 23 million nodes and 58 million edges. The framework uses 1 Gigabit Ethernet for connecting cloud storage.

5.5.1 The Automatic Deployment of an Out-of-core Container Library

This subsection will exhibit the influence of STXXL in terms of insertion time and search time. To trigger STXXL, the SLOs specified in Section 3.6 are assigned in the order of MemPerRel. Figures 5.1 and 5.2 show *explored*'s (as can be seen in Figure 3.4) average insertion time and average search time taken by an STXXL's *map*, our framework using baseline implementation (the method described in Section 3.5), and our framework utilizing STXXL and MCSTL. The two figures 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. They also illustrate that through the self-adaptive mechanism, STXXL is deployed when memory limits are reached, which prevents performance decline when primary memory is sufficient.

5.5.2 The Automatic Deployment of Parallelism

This subsection first applies our framework with the functionality of parallelism to the BFS algorithm and then to the Dijkstra's shortest path algorithm. Figures 5.3 and 5.4 exhibit the average insertion time and average search time expended by an STL *set* and our framework using TBB for the table of

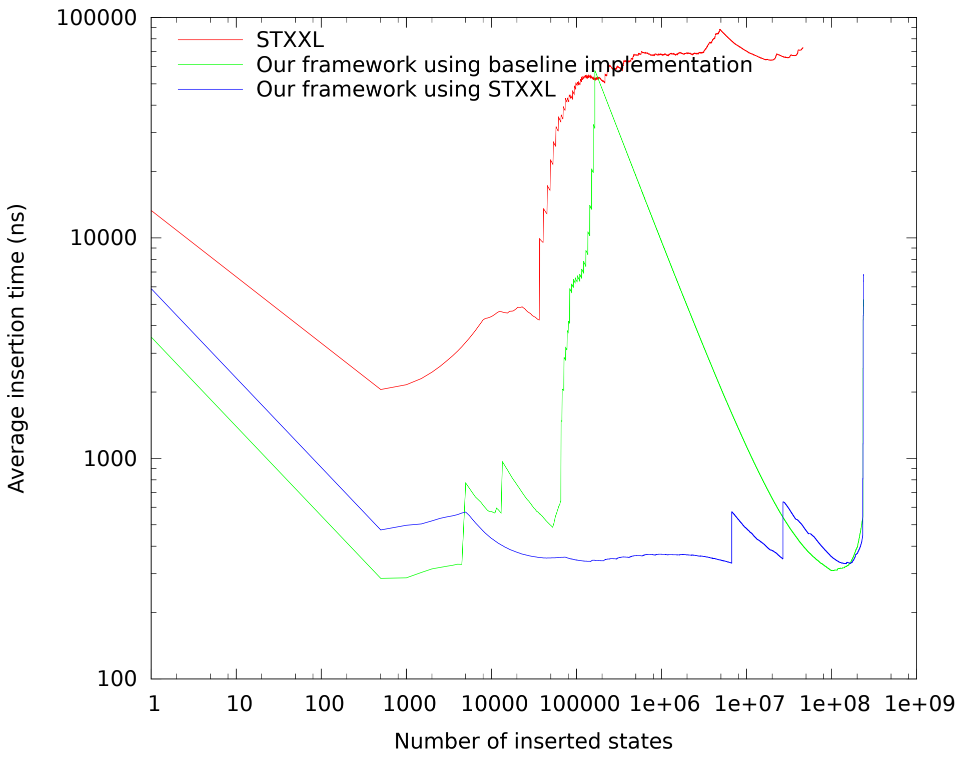


Figure 5.1: The average insertion time of *explored* adopting STXXL, our framework using baseline implementation, and our framework using STXXL

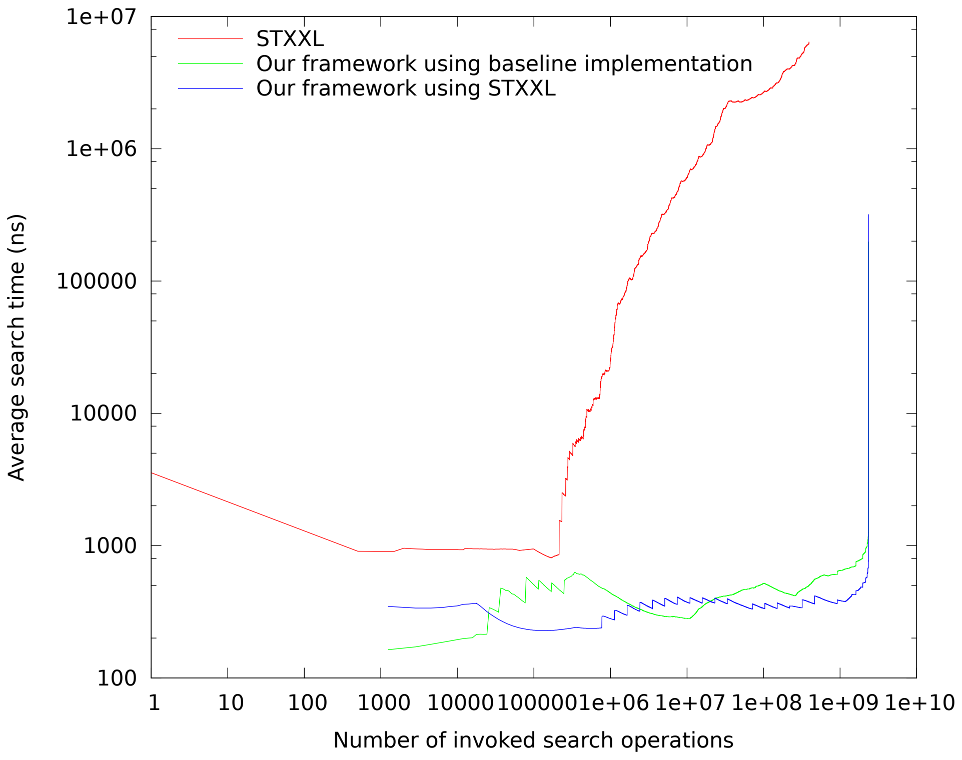


Figure 5.2: The average search time of *explored* adopting STXXL, our framework using baseline implementation, and our framework using STXXL

explored states. As can be seen, when our framework utilises TBB's *concurrent_hash_map* to store explored states, performance is improved. Specifically, cumulative insertion time is reduced by 76%, while cumulative search time is reduced by 86%.

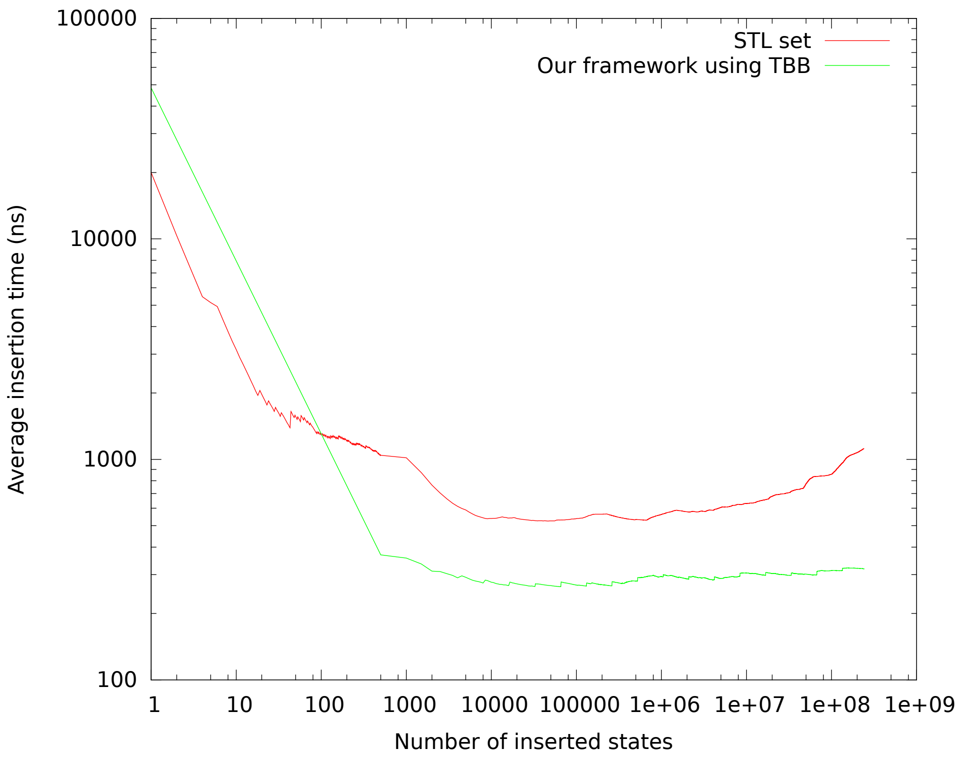


Figure 5.3: The average insertion time of the STL set and our framework using TBB

Figures 5.5 and 5.6 depict the average push and pop times of an STL *queue* and our framework for the queue of unexplored states. Similarly, our framework adopting TBB's *concurrent_queue* is faster than the STL *queue* (i.e. 54% reduction in cumulative push time and 77% reduction in cumulative pop time). In addition, the average push time considerably rises when the number of invoked states increases. This phenomenon results from the implementation of the STL *queue*. The STL *queue* is a container adaptor, whose default underlying container is *deque*. It utilises an array, each of whose positions points to a block for storing elements. When all blocks are full, the array will be resized to store more blocks. As a result, the average push time may rise as the number of stored elements increases.

In the second case study, route planning, our framework adopts TBB's *concurrent_unordered_map* for *Distance* and *concurrent_priority_queue* for *PQ* in Figure 4.2. The response times (insertion and

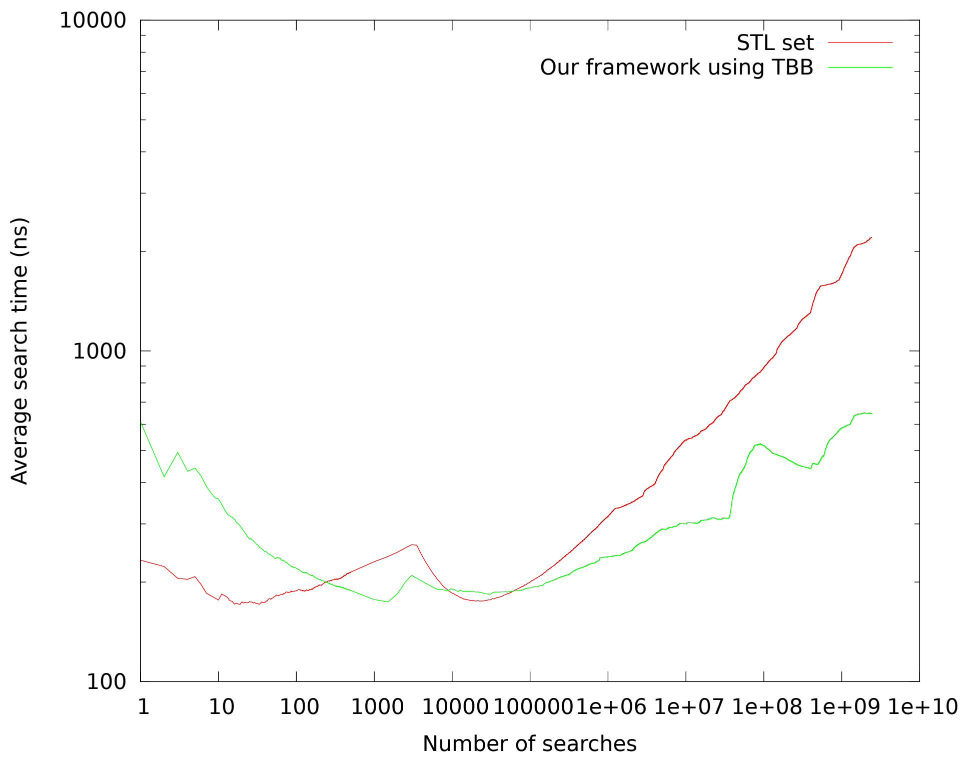


Figure 5.4: The average search time of the STL set and our framework using TBB

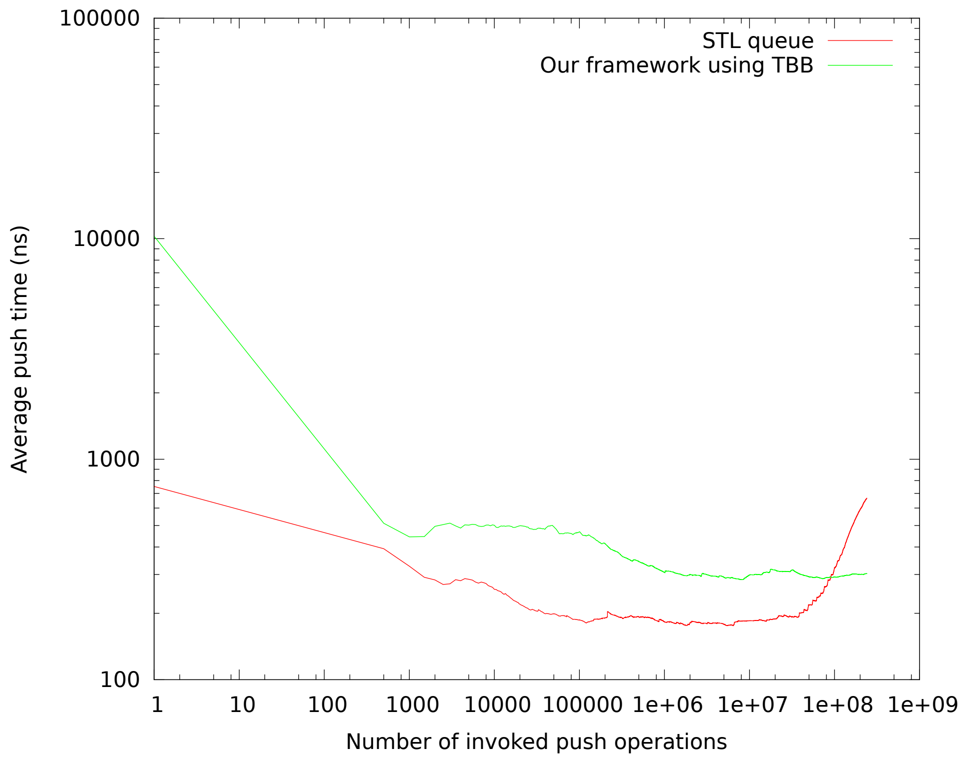


Figure 5.5: The average push time of the STL queue and our framework using TBB

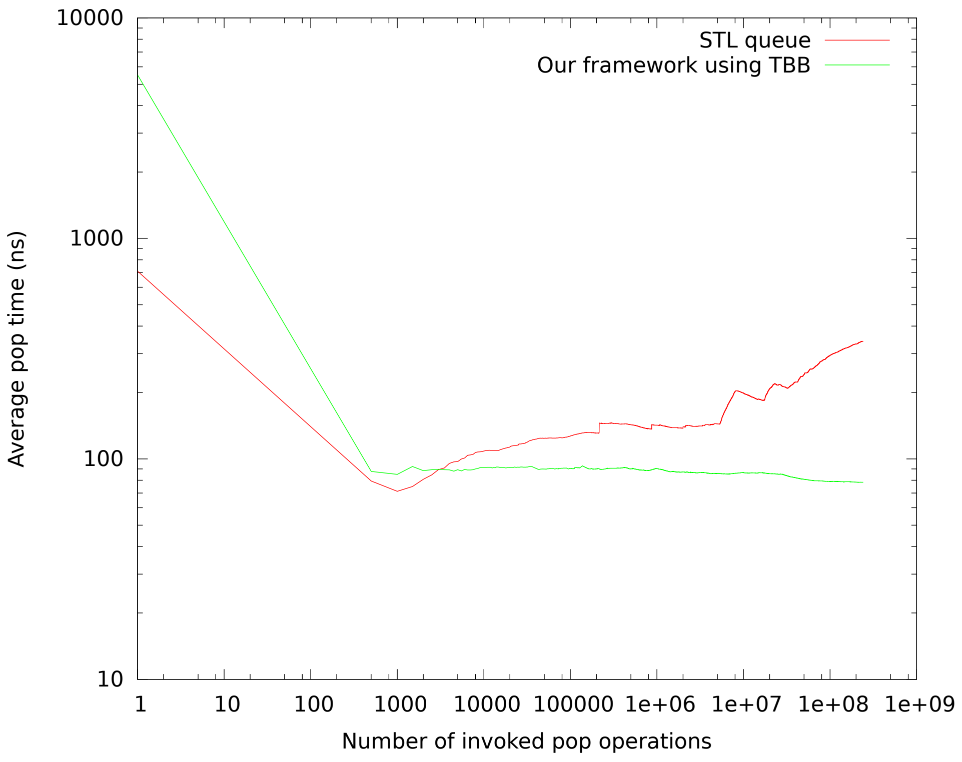


Figure 5.6: The average pop time of the STL queue and our framework using TBB

update) of *Distance* are displayed in Figures 5.7 and 5.8. As can be seen, our framework using TBB expends more insertion time than the map does, but it expends less update time. As a result, the overall response time (in terms of the sum of cumulative insertion time and cumulative update time) of our framework is reduced by 30.8% compared to the STL *map*.

Figures 5.9 and 5.10 depict the average push time and pop time of an STL *priority_queue* and our framework adopting TBB. As can be seen, cumulative push time is reduced by 9.7% and cumulative pop time is reduced by 24.8%.

5.5.3 The Automatic Deployment of Cloud Storage

This subsection will discuss the performance and memory impacts when cloud storage is deployed. It will also illustrate dynamic data transfer to and from the cloud storage. For the first case study, we modify the assigned SLOs due to the performance limitations of cloud storage. The new SLOs are assigned in the following order (MemPerRel):

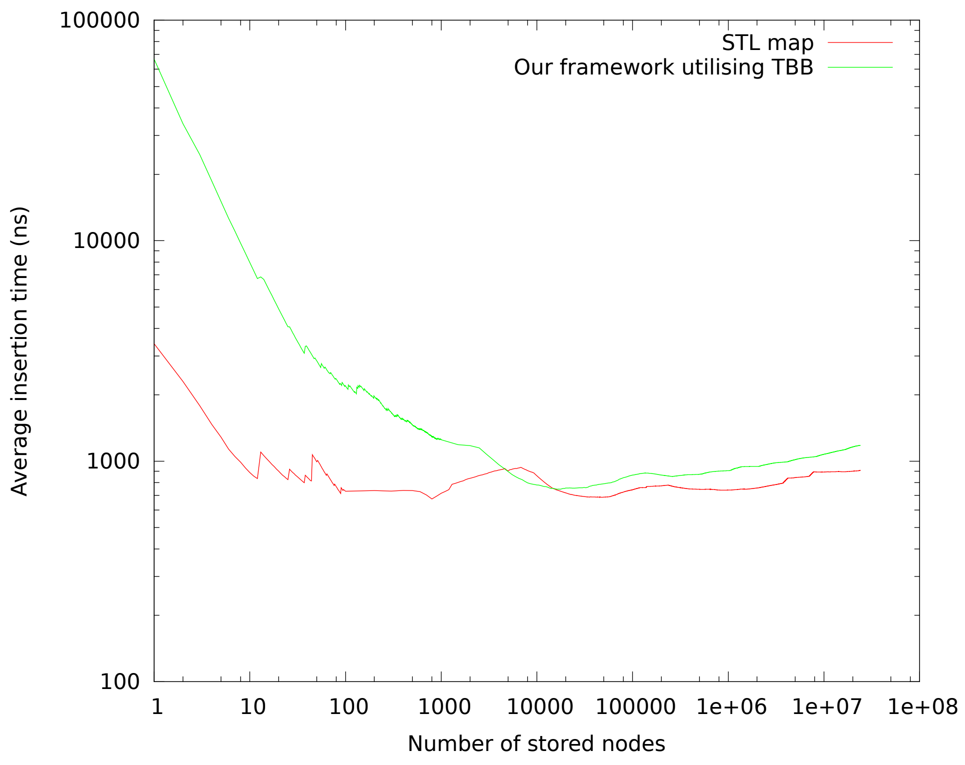


Figure 5.7: The average insertion time of STL map and our framework adopting TBB

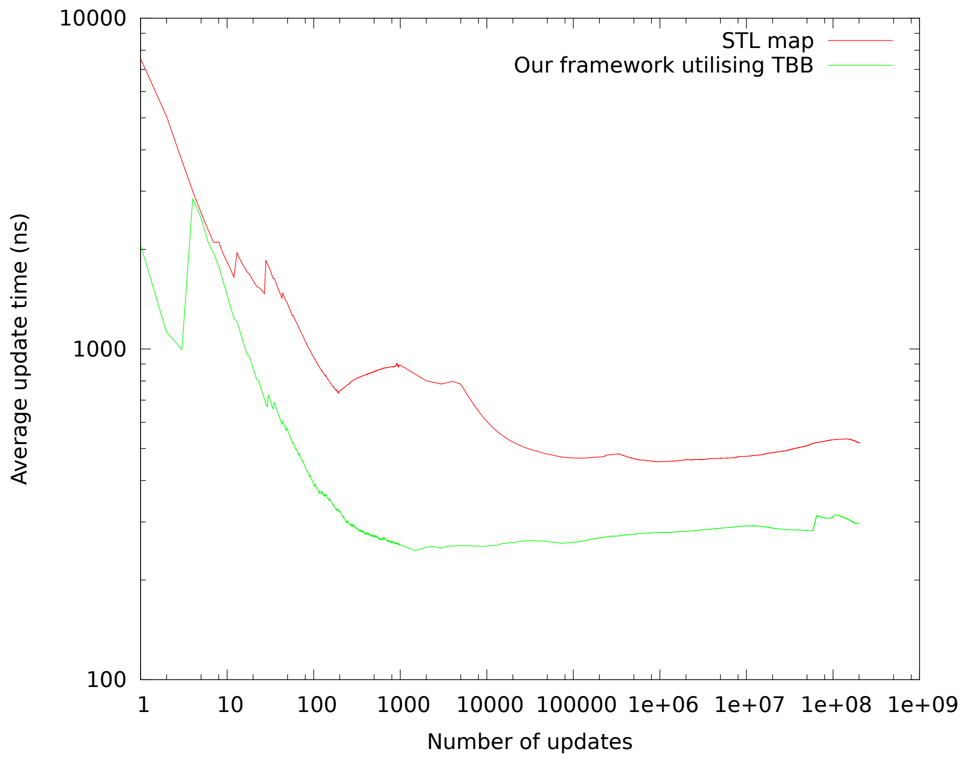


Figure 5.8: The average update time of STL map and our framework adopting TBB

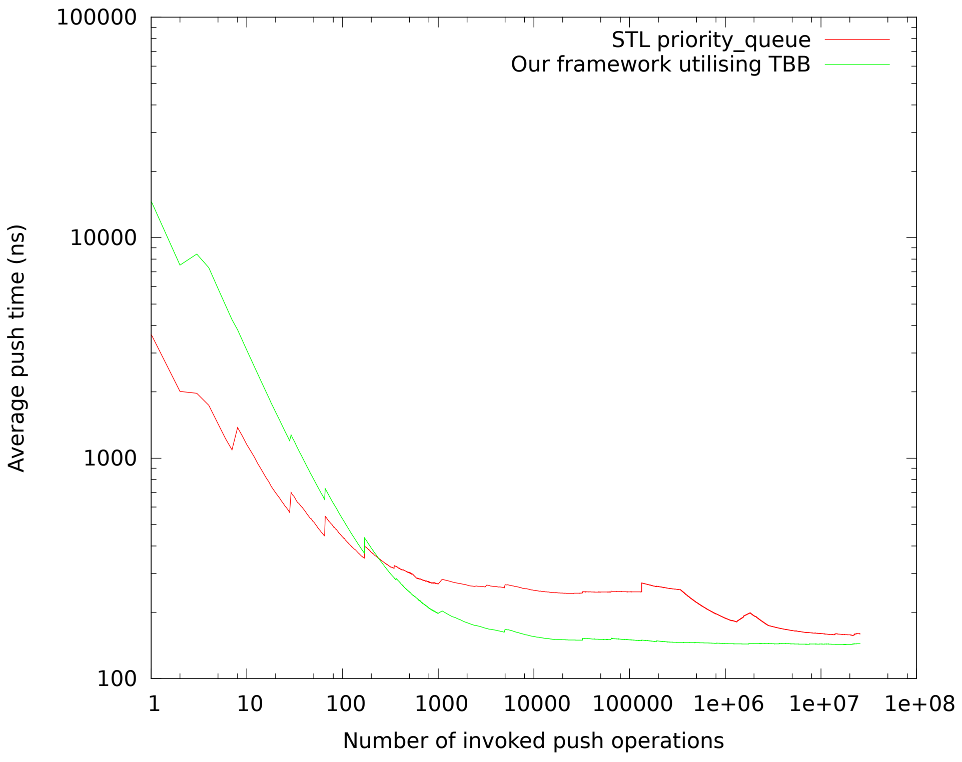


Figure 5.9: The average push time of STL *priority_queue* and our framework adopting TBB

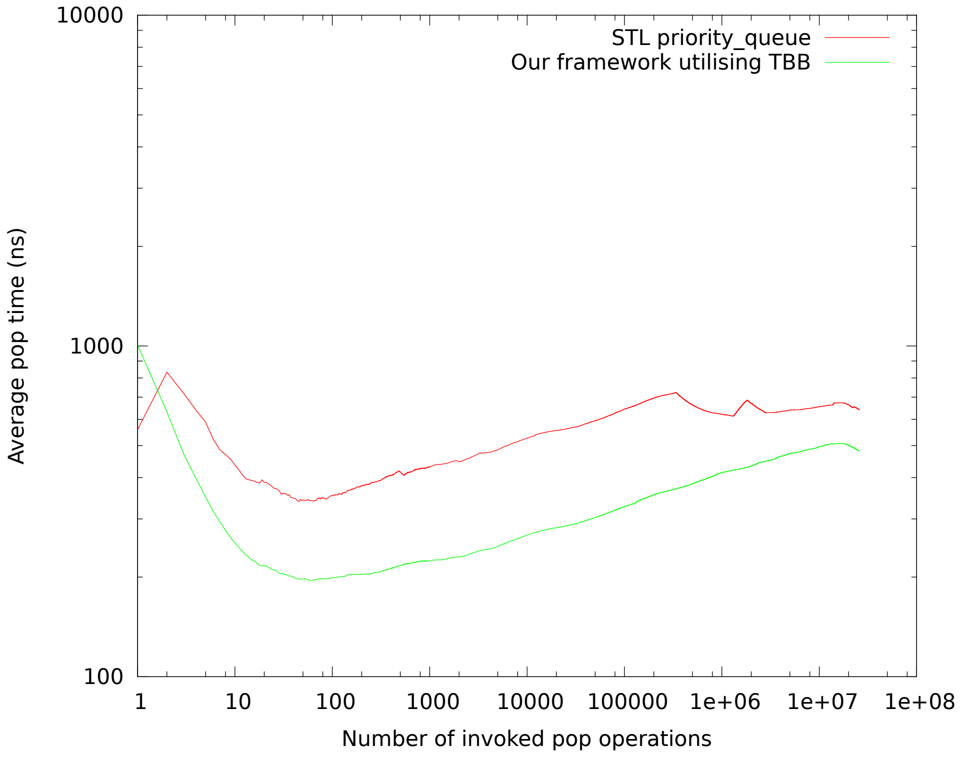


Figure 5.10: The average pop time of STL *priority_queue* and our framework adopting TBB

1. Memory consumption should be no more than 1 MB.
2. 90% of insertion times should be less than 400 ns, and 85% of search times should be less than 400 ns.
3. Reliability should be higher than 0.99.

Figures 5.11 and 5.12 display the average insertion time and search time expended by our framework exploiting out-of-core storage and cloud storage (i.e. Amazon S3). As can be seen, the performance of Amazon S3 is lower than that of out-of-core storage, which is in keeping with the typical situation where disk I/O performance is faster than network performance.

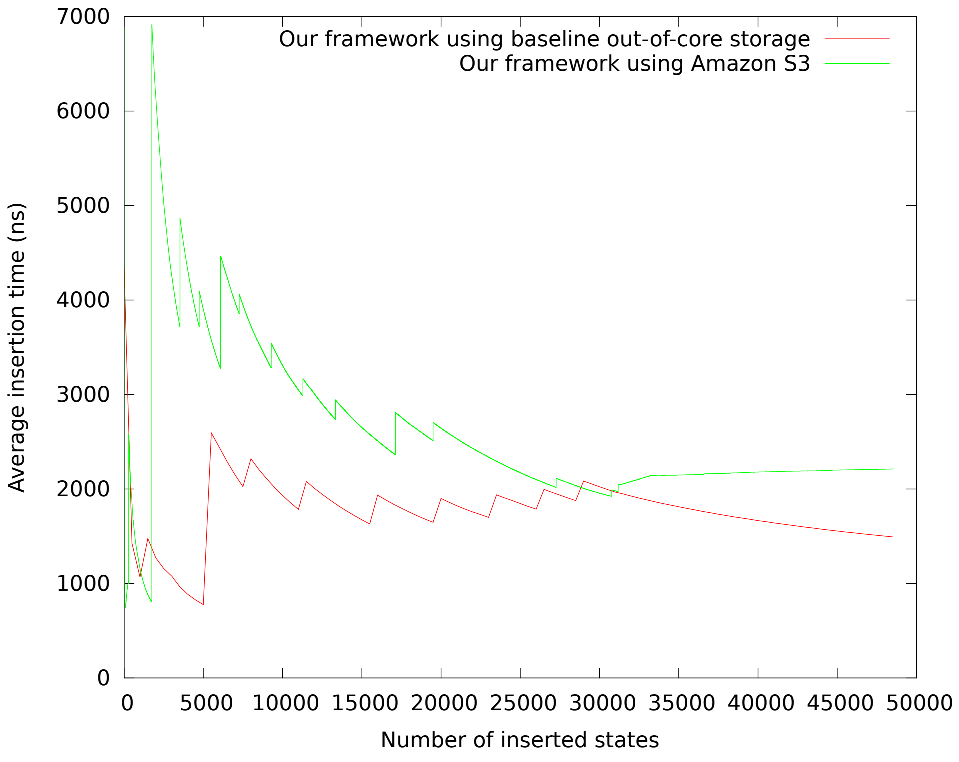


Figure 5.11: The average insertion time of our framework using out-of-core storage and Amazon S3

Figure 5.13 represents primary memory consumed by our framework adopting out-of-core storage and Amazon S3. It shows that when the number of stored states is approximately 31 000 our framework activates cloud storage in order to protect the memory-related SLO.

In the second case study, cloud storage is deployed by `IKeyValue`. For the same reason (performance limitation), the SLOs are modified and assigned in the following priority order (MemPerRel):

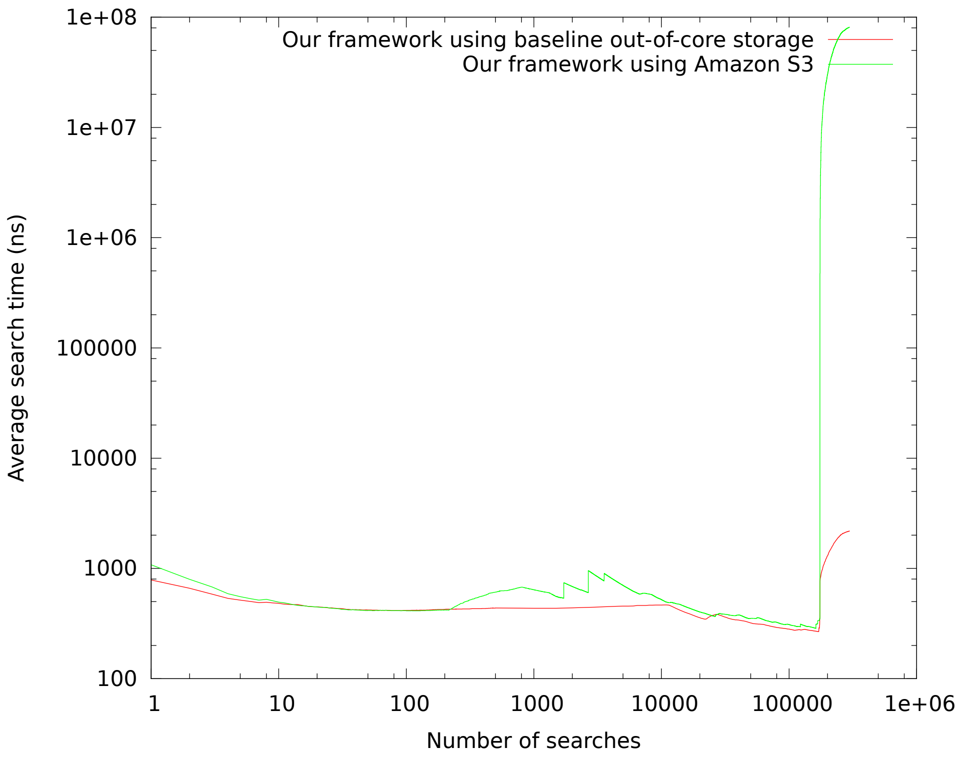


Figure 5.12: The average search time of our framework using out-of-core storage and Amazon S3

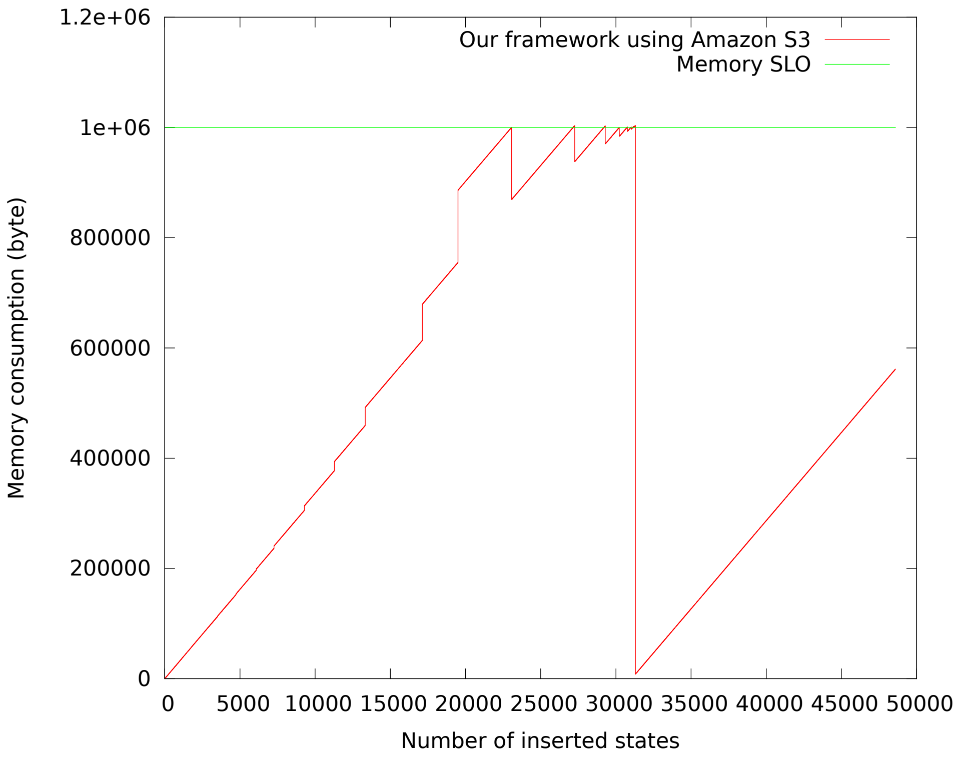


Figure 5.13: The memory consumption of our framework using Amazon S3

1. Memory consumption should be no more than 350 KB.

2. 95% of insertion times should be less than 100 ns, and 95% of update times should be less than 200 ns.

3. Reliability should be higher than 0.995.

Furthermore, the input data set is changed to a graph representing the road network of Washington DC, which contains approximately 10 000 nodes and 15 000 edges. The memory change of our framework utilising Amazon S3 is displayed in Figure 5.14, which exhibits that before the memory limit is met, the framework increase memory space to boost performance or reliability (i.e. the sudden rises in memory consumption). When the limit of *Distance*'s memory consumption (i.e. 350 KB) is reached, our framework will try to reduce memory use by shrinking the underlying data structures (i.e. reducing the number of AVL trees). Once the number of AVL trees is 1, cloud storage is activated (which occurs when the number of inserted nodes is approximately 8 800).

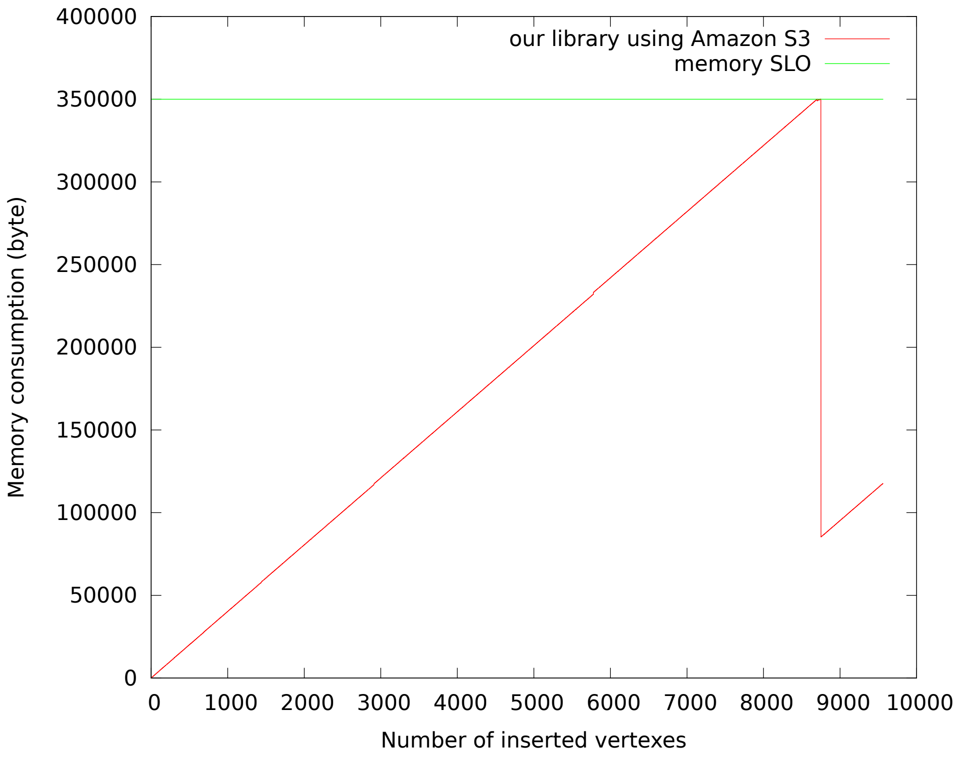


Figure 5.14: The memory consumption of our framework using Amazon S3 for route planning

5.6 Conclusion

This chapter has shown the interoperability with third-party container frameworks through the integration of STXXL and Intel TBB. The former supplies well-developed and efficient out-of-core containers, and the latter affords parallel containers. The adoption of both libraries enables our framework to satisfy a wider class of Service Level Objectives at low cost, especially those related to performance and memory consumption.

This chapter has also exhibited the ability of dynamically exploiting cloud storage. As a result, when local memory is not available, alternative memory can be utilised. In addition, the implementation detail of data transfer from and to cloud storage is hidden by our framework, which means that programmers do not need to modify their code for supporting various cloud storage.

We have adopted explicit state-space exploration and route planning to evaluate the framework's interoperability and capability. The results suggest that the interoperability is extended in terms of the automatic employment of third-party containers. Through these libraries, our framework considerably boosts performance, cuts primary memory consumption, and exploits out-of-core storage and parallelism at low cost. Furthermore, the implementation for accessing cloud storage is significantly simplified.

Chapter 6

Conclusion

‘The world is changing very fast. Big will not beat small anymore. It will be the fast beating the slow.’

Rupert Murdoch, Chairman and CEO, News Corp

6.1 Summary of Achievements

This dissertation has presented a novel self-adaptive container framework which aims to prevent frequent software reimplementation when execution environments, application contexts, QoS requirements, or scalability requirements change.

Traditionally, programmers make use of software engineering methodologies such as design patterns or standardised libraries to develop software in order to reduce implementation effort and levels of expertise. As the number of execution environments increases dramatically, these techniques cannot catch up with the speed of the increase in the diversity of execution environments, application contexts, QoS requirements, and scalability requirements, which leads to repeated code refactoring. To meet this challenge, software should have the ability to accept high-level objectives, monitor its resource usage, and automatically adjust it to satisfy assigned Service Level Objectives. Compared to our framework, the studies for building self-adaptive systems presented in Section 2.8 can dynamically manage resources but do not provide a mechanism for specifying Service Level Objectives based on a standard format. In addition, even if the third-party container libraries described in Section 2.7 are helpful in reducing resource consumption, they are incapable of finding a suitable configuration for the current environment. Our framework proposed in Chapter 3 is capable of identifying Service Level Objectives through the utilisation of WSLA, monitoring and analysing resource changes, and executing adaptation actions to satisfy assigned SLOs. Furthermore, it supplies tighter functionality specification compared to standard libraries. This enables the deployment of probabilistic data structures, out-of-core storage, parallelism, and cloud storage. A prototype for this framework with these techniques has been implemented and utilised in explicit state-space exploration to validate the feasibility and evaluate the capability and scalability. The results have shown that when approximately 240 million states are explored, the prototype delivers better performance and consumes less memory compared to conventional containers. In addition, the implementation exhibits different behaviour in response to the priority ordering of assigned SLOs.

The prototype of the self-adaptive container framework presented in Chapter 3 has been extended by means of the implementation of key-value stores and priority queues in an effort to provide widely-used functionality. Key-value store containers hold inserted elements formed by pairs of keys and mapped values. They were originally supported by programming languages (in-memory). In the recent decade, key-value stores have become a frequently-used technique in the field of databases so as to deal with large-scale data. Our framework supplies not only in-memory but persistent key-value stores through the exploitation of out-of-core storage. Furthermore, it can choose tree data structures or probabilistic data structures as the underlying data structure according to specified SLOs and desired functionalities. In addition to the support of key-value stores, the enhanced prototype has implemented another widely-adopted data structure, priority queues. Their applications may also need to process large amount of data. As a result, our implementation for priority queues supplies the automatic deployment of out-of-core storage that is triggered by memory-related SLOs. The prototype has been assessed via a case study centred on route planning, adopting a Dijkstra's shortest path algorithm. The results indicate that our framework provide faster key-value stores compared with STL's *map*. It can also dynamically perform adaptations in order to achieve various priority orderings of Service Level Objectives.

To satisfy a wider class of Service Level Objectives, the framework has integrated third-party container frameworks and cloud storage. As we have mentioned in Chapter 2.7, many research teams have developed libraries improving a certain capability (e.g. memory efficiency, performance, or reliability). However, to use them correctly, a suitable configuration should be given and an accurate activation time of these libraries should be determined. Hence, we extend the framework's interoperability in terms of the configuration and exploitation of these libraries on-the-fly. In addition, cloud storage has been integrated into our framework, which shifts the responsibility of transferring data from and to cloud storage from programmers to our framework and supplies alternative memory when local memory space (primary memory and out-of-core memory) is not available.

All of the above-mentioned contributions are achieved with low programmer overhead. Compared to our approach, other self-adaptive approaches need to modify the overall architecture and much of code. This increases the complexity of adopting self-adaptive techniques in existing systems. Utilising the framework to build self-adaptive software minimises the degree to which code is modified. Indeed, only two lines of code are changed in the two case studies.

6.2 Applications

This section will discuss the potential applications where our framework can be utilised.

Our framework intends to cover the functionality supported by the STL, which implies that applications implemented through the STL can also adopt our framework. Furthermore, as we have mentioned in Section 1.1, five application domains, explicit state-space exploration, route planning, DNA sequence assembly, visualisation, and similarity search, suffer from primary memory shortage and performance decline owing to large-scale data. Due to the exploitation of probabilistic data structure, out-of-core storage, and parallelism, our framework is particularly suitable for applications with large input data.

Other applications the framework can be applied to are those which are executed on multiple platforms (e.g. games, browsers, messaging software, antivirus, etc). The QoS requirements of these applications are highly affected by their execution environments and application contexts. Using our framework can prevent reimplementation when the same application is executed on different platforms. Furthermore, the satisfaction of different QoS requirements is simplified to the respecification of Service Level Objectives.

The framework is also suitable for applications that frequently access multiple cloud storage services. Take Rainbow Drive (Compal, 2014) for example. Rainbow Drive is an app which allows users to manage multiple accounts (i.e. it can access different cloud storage services). Although it can show information of each account, management still depends on manual operations. Furthermore, its developers need to implement access methods for all cloud storage services. All of these limitations can be solved by our framework, which monitors usage of each cloud storage, calculates its cost, and compares it with Service Level Objectives. If a certain cloud storage reaches its limit, an alternative cloud storage can be activated. Additionally, our framework can easily integrate other utility library supporting specific service providers, which removes the need for implementing explicit data transfer methods for new service providers.

Although our framework can be deployed in most applications, programmers cannot gain benefits from it in some particular situations, which are displayed in Table 6.1. First, our framework cannot improve performance when it is utilised by inefficient algorithms. Our framework can provide efficient way to manipulate data, but the overall performance highly depends on applied algorithms. Second, performance improvement and memory reduction supported by our framework result from the adopted techniques, which rely on desired functionalities. Hence, the more functionalities are specified, the less performance and memory efficiency are improved. Third, the framework intends to satisfy assigned SLOs via self-adaptations, which may result in useless adaptations when unrealistic SLOs are specified. Fourth, when the framework has adequate memory space, it can subdivide its underlying data structures to boost performance. By contrast, if memory space is limited, the above-mentioned actions cannot be taken. Fifth, the framework cannot exploit parallelism when software utilising it runs on a single-threaded CPU. Sixth, if 100% reliability is required, probabilistic data structures cannot be deployed. As a result, with the allowance of performance-related SLOs, the alternative way to reduce primary memory use is exploitation of out-of-core storage. Seventh, our framework can improve QoS of naïve algorithms but may not enhance that of algorithms which have been optimised. For example, if an algorithm has made use of out-of-core storage to reduce primary memory use, the adoption of our framework cannot save more primary memory. Finally, when applications run on environments incapable of connecting the internet, cloud storage cannot be exploited and treated as alternative out-of-core memory.

Ideal Scenario	Non-ideal Scenario
Fundamentally efficient algorithm	Fundamentally inefficient algorithm
Limited functionality specification	Full functionality specification
Realistic SLOs	Unrealistic SLOs
Adequate memory space	Limited memory space
Multi-threaded CPU	Single-threaded CPU
Sub-100% reliability requirement	100% reliability requirement
Naïve algorithms	Optimised algorithms
High-bandwidth Internet connection	No Internet connection

Table 6.1: The discussion of scenarios where programmers can or cannot benefit from our framework

6.3 Future Work

This section will indicate future research directions which can simplify the usage of the framework, enrich the self-adaptive mechanism, and extend the framework's capability.

Currently, the functionality provided by our framework is based on operation descriptors. When a self-adaptive container is declared, programmers have to know which operations are invoked in advance. This may cause the difficulty in maintaining software. For example, when the container needs to perform a new operation, the corresponding operation descriptor may not be added into its constructor. This kind of problem appears at run time but not compile time. To prevent this, our framework can provide a member function which automatically scans the scope of the container to decide which operation descriptors are required. The automatic assignment of operation descriptors not only avoids the maintenance issue but simplifies the usage of the framework.

The existing self-adaptive mechanism follows a strict order of priority, which can be broadened through the utilisation of other multi-objective optimisation methods. For example, the strict order of priority can be changed to a weighted product or a weighted sum of multiple Service Level Objects. This extension requires two modifications of the current methodology. First, the format of SLO specification has to accept the assignment of the optimisation method. Second, it also needs to be laid down more information related to assigned SLOs. For instance, if a weighted sum optimisation method is adopted, the weight of each SLO should be given to the framework.

Driven by energy price, the energy concern has become more and more important for computing systems. Energy-efficient software not only reduces the operation cost of systems but extends the uptime of some devices (e.g. mobile devices). However, it is not a trivial job to build such software. The challenges may include how to measure energy consumption of different applications, how to identify applications which result in energy waste, and how to improve energy efficiency. Manually overcoming these challenges is possible but restricts the benefits to small systems. As a result, self-adaptive software may be one workable solution. These challenges can be tackled by a self-adaptive mechanism with the ability to monitor resources (i.e. energy), analyse resource usage (i.e. find energy waste), and perform adaptations (i.e. improve energy efficiency).

Our framework has improved low-level complexity and resource usage of many applications. However, many performance bottlenecks result from inefficient higher-level algorithms. In other words, detection and modification of inefficient algorithms can considerably boost performance (Smaalders, 2006). The mechanism to self-detect and self-correct inefficient algorithm has not been addressed in the current work. One promising approach that might help relates to the automated detection and correction of performance anti-patterns. Compared to software design patterns (Gamma, Helm, Johnson, & Vlissides, 1995), which provide good solutions for software design problems, anti-patterns (W. J. Brown, Malveau, McCormick, & Mowbray, 1998) are templates for bad practices which are virtually guaranteed to lead to undesirable nonfunctional behaviour. If automated correction of performance anti-patterns were to become feasible in the future it would further push the frontiers of intelligent self-adaptive software.

Since our framework shows great interoperability with other third-party frameworks, it can benefit from the algorithms provided by other frameworks. Take STXXL and Intel TBB for example. Both of them do not only supply containers but also support out-of-core and parallel algorithms, respectively. This enables our framework to provide programmers with the corresponding algorithms whose data structures have been dynamically selected by the self-adaptive mechanism.

Finally, the integration of cloud storage allows our framework to deliver diverse combinations of cloud storage services. The features and prices of a cloud storage service relies on its service provider. Currently, there are at least ten companies supplying cloud storage services, which makes the choice difficult for both end-users and programmers. For end-users, they may make considerable effort select or transfer to a suitable service. For programmers, they may need to develop software for different service providers and platforms. These burdens can be relieved by our framework when it has sufficient knowledge to analyse end-users' requirements and the ability to monitor cloud storage usage and dynamically change it. To achieve this, a mechanism should be developed to specify cloud-related Service Level Objectives (e.g. maximum capacity, maximum expected cost, maximum number of push and pop operations, connectivity). Furthermore, those parameters related to these objectives should be measured by the self-adaptive mechanism. After the information is specified, the self-adaptive mechanism can dynamically exploit different cloud storage services.

6.4 Final Thoughts

Nowadays new execution environments are constantly emerging, e.g. smart watches arise wide attention in 2015. The varying QoS requirements of these environments cause software to be frequently refactored. As a result, we design the self-adaptive container framework for helping programmers reduce their effort of reimplementing software when execution environments change. Since the framework is implemented in C++ in this thesis (it can also be implemented in other programming languages), it should be able to replace the STL by the following steps. First, related header files (e.g. ICollection.h or IKeyValue.h) should be included. Second, container variables are changed to `ICollection` or `IKeyValue` according to their nature, i.e. use `ICollection` for single-value containers and `IKeyValue` for key-value store containers. Third, desired functionalities, Service Level Objectives, and adaptation frequency (optional) should be specified.

To build self-adaptive software, one critical issue is the time expended on adaptation (Floch et al., 2006). Our framework also takes this issue into consideration. When the underlying data structures need to be changed, the techniques of multi-threading are adopted to boost the performance of executing adaptation actions. Although we have considerably reduced the time of changing the underlying data structures, software still needs to suspend its current jobs to perform adaptations. This can be improved by utilising incremental adaptations. When adaptations are triggered, only part of stored data is changed to new data structures by threads. In addition, a semaphore is configured to protect the new data structures. After this, operations related to data modification are performed on the new data structures. The original data structures are kept for retrieval and will be destroyed when all data is moved to the new data structures.

Appendix A

An Expression of Service Level Objectives in WSLA format

Listing A.1: The SLO configuration file of *explored*

```xml

<?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>
```

Bibliography

Abbasi, H., Wolf, M., Schwan, K., Eisenhauer, G., & Hilton, A. (2004, September). Xchange: coupling parallel applications in a dynamic environment. In *Proceedings of 2004 IEEE International Conference on Cluster Computing* (pp. 471–480). San Diego, USA.

Abrahams, D. & Gurtovoy, A. (2004). *C++ template metaprogramming: concepts, tools, and techniques from boost and beyond (C++ in depth series)*. Addison-Wesley Professional.

Abram, G. & Treinish, L. (1995, November). An extended data-flow architecture for data analysis and visualization. In *Proceedings of the 6th Conference on Visualization* (pp. 263–270). Atlanta, USA.

Adler, M., Chakrabarti, S., Mitzenmacher, M., & Rasmussen, L. (1995). Parallel randomized load balancing. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing* (pp. 238–247). Las Vegas, USA.

Aerospike. (2014, February). Aerospike. Retrieved from http://www.aerospike.com/

Aidarov, K., Ezhilchelvan, P., & Mitrani, I. (2013). Energy-aware management of customer streams. *Electronic Notes in Theoretical Computer Science*, *296*, 199–210.

Alabduljalil, M. A., Tang, X., & Yang, T. (2013, February). Optimizing parallel algorithms for all pairs similarity search. In *Proceedings of the 6th ACM International Conference on Web Search and Data Mining* (pp. 203–212). Rome, Italy.

Allmaier, S. C. & Horton, G. (1997, June). Parallel shared-memory state-space exploration in stochastic modeling. In *Proceedings of the 4th International Symposium on Solving Irregularly Structured Problems in Parallel* (pp. 207–218). Paderborn, Germany.

Almeida, P. S., Baquero, C., Preguiça, N., & Hutchison, D. (2007). Scalable Bloom filters. *Information Processing Letters*, *101*(6), 255–261.

Andreou, D. & Bourrillion, K. (2014). Guava: Google core libraries for Java 1.6+. Retrieved from https://code.google.com/p/guava-libraries/

Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., ... Xu, M. (2005, September). *Web Services Agreement Specification (WS-Agreement)*. Global Grid Forum, Grid Resource Allocation Agreement Protocol (GRAAP) WG. Retrieved from http://www.ggf.org/Public_Comment_Docs/Documents/Oct-2005/WS-AgreementSpecificationDraft050920.pdf

Apache. (2014, February). Apache Cassandra. Retrieved from http://cassandra.apache.org/

Appcelerator. (2014). Titanium sdk. Retrieved from http://www.appcelerator.com/titanium/titanium-sdk/

Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., & Paleczny, M. (2012, June). 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* (pp. 53–64). London, UK.

Basho. (2014, February). Riak. Retrieved from http://basho.com/riak

Berchtold, S., Böhm, C., Braunmüller, B., Keim, D. A., & Kriegel, H.-P. (1997, May). Fast parallel similarity search in multimedia databases. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data* (pp. 1–12). Tucson, USA.

Bigus, J. P., Schlosnagle, D. A., Pilgrim, J. R., Mills, W. N., & Diao, Y. (2002, July). ABLE: a toolkit for building multiagent autonomic systems. *IBM System Journal*, *41*(3), 350–371.

Bingham, B., Bingham, J., de Paula, F. M., Erickson, J., Singh, G., & Reitblatt, M. (2010, October). Industrial strength distributed explicit state model checking. In *Proceedings of the 9th International Workshop on Parallel and Distributed Methods in Verification* (pp. 28–36). Enschede, Netherlands.

Bloom, B. H. (1970, July). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, *13*(7), 422–426.

Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., & Varghese, G. (2006, September). An improved construction for counting Bloom filters. In *Proceedings of the 14th Conference on Annual European Symposium - Volume 14* (pp. 684–695). Zurich, Switzerland.

Brown, W. J., Malveau, R. C., McCormick, H. W., III, & Mowbray, T. J. (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. New York, NY, USA: John Wiley & Sons, Inc.

Bruggmann, M. (2014, March). Sparkey. Retrieved from https://github.com/spotify/sparkey-java

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., & Stefani, J.-B. (2006, September). The FRACTAL component model and its support in Java: experiences with auto-adaptive and reconfigurable systems. *Software Practice & Experience*, *36*(11-12), 1257–1284.

Buss, A., Harshvardhan, Papadopoulos, I., Pearce, O., Smith, T., Tanase, G., ... Rauchwerger, L. (2010, May). STAPL: standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference* (14:1–14:10). Haifa, Israel.

Butler, J., MacCallum, I., Kleber, M., Shlyakhter, I. A., Belmonte, M. K., Lander, E. S., ... Jaffe, D. B. (2008). ALLPATHS: de novo assembly of whole-genome shotgun microreads. *Genome Research*, *18*(5), 810–820.

Caselli, S., Conte, G., & Marenzoni, P. (1995, June). Parallel state space exploration for GSPN models. In *Proceedings of the 16th International Conference on Application and Theory of Petri Nets* (pp. 181–200). Turin, Italy.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... Gruber, R. E. (2008, June). Bigtable: a distributed storage system for structured data. *ACM Transactions on Computer Systems*, *26*(2), 4:1–4:26.

Chi, J., Ning, Z., Lang, L., & Yuan, F. (2009, May). Research and application on Bloom filter in routing planning for indoor robot navigation system. In *Proceedings of the 2009 Pacific-Asia Conference on Circuits, Communications and Systems* (pp. 244–247). Chengdu, China.

Chiang, Y.-J. & Silva, C. T. (1999). External memory algorithms. In J. M. Abello & J. S. Vitter (Eds.), (Chap. External Memory Techniques for Isosurface Extraction in Scientific Visualization, pp. 247–277). American Mathematical Society.

Chikhi, R. & Rizk, G. (2012, September). Space-efficient and exact de Bruijn graph representation based on a Bloom filter. In *Proceedings of the 12th International Conference on Algorithms in Bioinformatics* (pp. 236–248). Ljubljana, Slovenia.

Ciardo, G., Gluckman, J., & Nicol, D. M. (1998). Distributed state space generation of discrete-state stochastic models. *INFORMS Journal on Computing*, *10*(1), 82–93.

Cignoni, P., Montani, C., Rocchini, C., & Scopigno, R. (2003, October). External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, *9*(4), 525–537.

Clarke, E. M., Jr., Grumberg, O., & Peled, D. A. (1999). *Model checking*. Cambridge, MA, USA: MIT Press.

Compal. (2014). Rainbow drive. Retrieved from http://www.compal.com/apps/rainbowdrive/

Cook, J. J. & Zilles, C. B. (2009, April). Characterizing and optimizing the memory footprint of de novo short read DNA sequence assembly. In *IEEE International Symposium on Performance Analysis of Systems and Software* (pp. 143–152). Boston, USA.

Costanza, P. & Hirschfeld, R. (2005). Language constructs for context-oriented programming: an overview of ContextL. In *Proceedings of the 2005 Symposium on Dynamic Languages* (pp. 1–10). DLS ’05. San Diego, USA.

Crauser, A. & Mehlhorn, K. (1999, July). LEDA-SM extending LEDA to secondary memory. In *Proceedings of the 3rd International Workshop on Algorithm Engineering* (pp. 228–242). London, UK.

Czarnecki, K. & Eisenecker, U. W. (2000). *Generative programming: methods, tools, and applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Dagum, L. & Menon, R. (1998, January). OpenMP: an industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, *5*(1), 46–55.

Deavours, D. D. & Sanders, W. H. (1998, June). An efficient disk-based tool for solving large Markov models. *Perform. Evaluation*, *33*(1), 67–84.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., … Vogels, W. (2007, October). Dynamo: Amazon’s highly available key-value store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles* (pp. 205–220). Stevenson, USA.

Delling, D., Katz, B., & Pajor, T. (2012, October). Parallel computation of best connections in public transportation networks. *Journal of Experimental Algorithmics*, *17*, 4.4:4.1–4.4:4.26.

Dementiev, R., Kettner, L., & Sanders, P. (2005, October). STXXL: standard template library for xxl data sets. In *Proceedings of the 13th Annual European Conference on Algorithms* (pp. 640–651). Palma de Mallorca, Spain.

DIMACS. (2006). 9th DIMACS implementation challenge - shortest paths. Retrieved from http://www.dis.uniroma1.it/challenge9/index.shtml

Dingle, N. J., Knottenbelt, W. J., & Suto, T. (2009, March). PIPE2: a tool for the performance evaluation of generalised stochastic Petri nets. *ACM SIGMETRICS Performance Evaluation Review*, *36*(4), 34–39.

Disseldorp, D. (2014). Elasto. Retrieved from https://code.google.com/p/elasto/

Dowling, J., Schäfer, T., Cahill, V., Haraszti, P., & Redmond, B. (1999, November). Using reflection to support dynamic adaptation of system software: a case study driven evaluation. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering* (pp. 169–188). Denver, USA.

Eastep, J., Wingate, D., & Agarwal, A. (2011, June). Smart data structures: an online machine learning approach to multicore data structures. In *Proceedings of the 8th acm international conference on autonomic computing* (pp. 11–20). Karlsruhe, Germany.

Edelkamp, S. & Schrödl, S. (2000, July). Localizing A*. In *Proceedings of the 7th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence* (pp. 885–890). Austin, USA.

Edelkamp, S. & Sulewski, D. (2010, September). Efficient explicit-state model checking on general purpose graphics processors. In *Proceedings of the 17th International SPIN Conference on Model Checking Software* (pp. 106–123). Enschede, The Netherlands.

FAL Labs. (2012, August). Tokyo Cabinet. Retrieved from http://fallabs.com/tokyocabinet/

Fan, L., Cao, P., Almeida, J., & Broder, A. Z. (2000, June). Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.* *8*(3), 281–293.

Farina, N. (2009). Lits3. Retrieved from https://code.google.com/p/lits3/

Fitzpatrick, B. (2004, August). Distributed caching with memcached. *Linux J.* (124), 5.

Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., & Gjorven, E. (2006, March). Using architecture models for runtime adaptability. *IEEE Software*, *23*(2), 62–70.

Fogaras, D. & Rácz, B. (2005, May). Scaling link-based similarity search. In *Proceedings of the 14th International Conference on World Wide Web* (pp. 641–650). Chiba, Japan.

Fowler, M. (2005, June). Language workbenches: the killer-app for domain- specific languages. Retrieved from http://www.martinfowler.com/articles/%20languageWorkbench.html

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., & Steenkiste, P. (2004, October). Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, *37*(10), 46–54.

Garlan, D., Siewiorek, D., Smailagic, A., & Steenkiste, P. (2002, April). Project Aura: toward distraction-free pervasive computing. *IEEE Pervasive Computing*, *1*(2), 22–31.

Geravanda, S. & Ahmadib, M. (2013). Bloom filter applications in network security: a state-of-the-art survey. *Computer Networks*, *57*(18), 4047–4064.

Gionis, A., Indyk, P., & Motwani, R. (1999, September). Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases* (pp. 518–529). Edinburgh, UK.

Goldberg, A. V. & Werneck, R. F. F. (2005a, January). Computing point-to-point shortest paths from external memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments and the 2nd Workshop on Analytic Algorithmics and Combinatorics* (pp. 26–40). Vancouver, Canada.

Goldberg, A. V. & Werneck, R. F. F. (2005b, January). Computing point-to-point shortest paths from external memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments and the 2nd Workshop on Analytic Algorithmics and Combinatorics* (pp. 26–40). Vancouver, Canada.

Google. (2013, December). LevelDB. Retrieved from http://code.google.com/p/leveldb/

Gouda, M. & Herman, T. (1991). Adaptive programming. *IEEE Transactions on Software Engineering*, *17*, 911–921.

Gschwind, T. (2001, January). PSTL: a C++ persistent standard template library. In *Proceedings of the 6th Conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 6* (pp. 11–11). San Antonio, USA.

Gudaitis, M. S., Lamont, G. B., & Terzuoli, A. J. (1995, February). Multicriteria vehicle route-planning using parallel A* search. In *Proceedings of the 1995 ACM Symposium on Applied Computing* (pp. 171–176). Nashville, USA.

Guo, D., Liu, Y., Li, X., & Yang, P. (2010, May). False negative problem of counting Bloom filter. *IEEE Transactions on Knowledge and Data Engineering*, *22*(5), 651–664.

Haverkort, B., Bell, A., & Bohnenkamp, H. (1999, September). On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets. In *Proceedings of the 8th International Workshop on Petri Nets and Performance Models* (pp. 12–21). Zaragoza, Spain.

Heineman, G. T. & Councill, W. T. (2001). Component-based software engineering. *Putting the Pieces Together, Addison-Westley*.

Hines, J. (2013). Dablooms - an open source, scalable, counting bloom filter library. Retrieved from https://github.com/bitly/dablooms

Hirschfeld, R., Costanza, P., & Nierstrasz, O. (2008). Context-oriented programming. *Journal of Object Technology*, *7*(3), 125–151.

Holzmann, G. J. (1988, February). An improved protocol reachability analysis technique. *Software Practice & Experience*, *18*(2), 137–161.

Horn, P. (2001). Autonomic computing: IBM's perspective on the state of information technology. Presented at AGENDA 2001, Socttsdale, Available via http://www.research.ibm.com/autonomic.

Huang, W.-C. & Knottenbelt, W. J. (2013, May). 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* (pp. 123–132). San Francisco, USA.

Huang, W.-C. & Knottenbelt, W. J. (2014a). Low-overhead development of scalable resource-efficient software systems. In W. K. I. Ghani & M. Ahmad (Eds.), *Handbook of research on emerging advancements and technologies in software engineering*. IGI Global.

Huang, W.-C. & Knottenbelt, W. J. (2014b, May). Self-adaptive containers: functionality extensions and further case study. In *Proceedings of the 6th International Conference on Adaptive and Self-Adaptive Systems and Applications* (pp. 92–98). Venice, Italy.

Huang, W.-C. & Knottenbelt, W. J. (2014c, December). Self-adaptive containers: interoperability extensions and cloud integration. In *Proceedings of the 11th IEEE International Conference on Autonomic and Trusted Computing*. Bali, Indonesia.

Huebscher, M. C. & McCann, J. A. (2008, August). A survey of autonomic computing&mdash;degrees, models, and applications. *ACM Computing Surveys*, *40*(3), 7:1–7:28.

IBM. (2003). *An architectural blueprint for autonomic computing*. IBM.

IBM. (2005). Autonomic computing toolkit. Retrieved from http://www.ibm.com/developerworks/autonomic/r3/overview.html

Idury, R. M. & Waterman, M. S. (1995). A new algorithm for DNA sequence assembly. *Journal of Computational Biology*, *2*, 291–306.

Intel. (2014). Threading building blocks. Retrieved from https://www.threadingbuildingblocks.org/

Isensee, P. (2014). C++ optimization strategies and techniques. Retrieved from http://www.tantalon.com/pete/cppopt/main.htm

Jackson, B. G., Regennitter, M., Yang, X., Schnable, P. S., & Aluru, S. (2010, April). Parallel de novo assembly of large genomes from high-throughput short reads. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing* (pp. 1–10). Atlanta, USA.

Jackson, D. (2002, April). Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodology*, *11*(2), 256–290.

Jiang, P., Ji, Y., Wang, X., Zhu, J., & Cheng, Y. (2014). Design of a multiple Bloom filter for distributed navigation routing. *Systems, Man, and Cybernetics: Systems, IEEE Transactions on*, *44*(2), 254–260.

Johnson, E. & Gannon, D. (1997, July). HPC++: experiments with the parallel standard template library. In *Proceedings of the 11th International Conference on Supercomputing* (pp. 124–131). Vienna, Austria.

Josuttis, N. M. (2012). *The C++ Standard Library: a tutorial and reference* (2nd). Addison-Wesley Professional.

Kaiser, G. E., Parekh, J. J., Gross, P., & Valetto, G. (2003, June). Kinesthetics eXtreme: an external infrastructure for monitoring distributed legacy systems. In *Active Middleware Services* (pp. 22–31).

Kaminsky, A. (2014). *BIG CPU, BIG DATA: solving the world's toughest computational problems with parallel computing*. Creative Commons.

Keller, A. & Ludwig, H. (2003, March). The WSLA framework: specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, *11*(1), 57–81.

Kephart, J. O. & Chess, D. M. (2003, January). The vision of autonomic computing. *Computer*, *36*(1), 41–50.

Khalid, A., Haye, M., Khan, M., & Shamail, S. (2009, April). Survey of frameworks, architectures and techniques in autonomic computing. In *Proceedings of the 5th International Conference on Autonomic and Autonomous Systems* (pp. 220–225). Valencia, Spain.

Kiczales, G., Lamping, J., A. Mendhekar, C. M., Lopes, C. V., Loingtier, J.-M., & Irwin, J. (1997, June). Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming* (pp. 220–242). Jyväskylä, Finland.

Kleftogiannis, D., Kalnis, P., & Bajic, V. B. (2013). Comparing memory-efficient genome assemblers on stand-alone and cloud infrastructures. *PloS ONE*, *8*(9), 1–11.

Knottenbelt, W. J. (2000). *Parallel performance analysis of large Markov models.* (Doctoral dissertation, Imperial College London (University of London)).

Knottenbelt, W. J. & Harrison, P. G. (1999). Distributed disk-based solution techniques for large Markov models. *Proceedings of the 3rd International Meeting on the Numerical Solution of Markov Chains*, *99*, 58–75.

Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhã, C., & Campbell, R. H. (2000, April). Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *IFIP/ACM International Conference on Distributed Systems Platforms* (pp. 121–143). New York, USA.

Kramer, J. & Magee, J. (2007, May). Self-managed systems: an architectural challenge. In *Proceedings of 2007 Future of Software Engineering* (pp. 259–268). Minneapolis, USA.

Krishnamurthy, P., Buhler, J., Chamberlain, R., Franklin, M., Gyang, K., Jacob, A., & Lancaster, J. (2007). Biosequence similarity search on the Mercury system. *Journal of VLSI Signal Processing*, *49*, 101–121.

Krupitzer, C., Roth, F. M., VanSyckel, S., Schiele, G., & Becker, C. (2014). A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, 33–42.

Kundeti, V., Rajasekaran, S., Dinh, H., Vaughn, M., & Thapar, V. (2010). Efficient parallel and out of core algorithms for constructing large bi-directed de Bruijn graphs. *BMC Bioinformatics*, *11*, 560.

Kusum, A., Neamtiu, I., & Gupta, R. (2015, January). Adapting graph application performance via alternate data structure representations. In *Proceedings of the 5th International Workshop on Adaptive Self-tuning Computing Systems*. Amsterdam, The Netherlands.

Kwiatkowska, M. Z. & Mehmood, R. (2002, July). Out-of-core solution of large linear systems of equations arising from stochastic modelling. In *Proceedings of the 2nd Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification* (pp. 135–151). Copenhagen, Denmark.

Lalanda, P., McCann, J. A., & Diaconescu, A. (2013). *Autonomic computing - principles, design and implementation.* Undergraduate Topics in Computer Science. Springer.

Lamanna, D. D., Skene, J., & Emmerich, W. (2003, May). SLAng: a language for defining service level agreements. In *Proceedings of the 9th IEEE Workshop on Future Trends of Distributed Computing Systems* (pp. 100–106). San Juan, Puerto Rico.

Levoy, M. (2013). The Stanford 3D scanning repository. Retrieved from http://www-graphics.stanford.edu/data/3Dscanrep/

Li, T., Yang, D., & Lian, X. (2012). Road crosses high locality sorting for navigation route planning. In *Recent advances in computer science and information engineering* (pp. 497–502). Springer.

Li, Y., Kamousi, P., Han, F., Yang, S., Yan, X., & Suri, S. (2012). Memory efficient de Bruijn graph construction. *CoRR*, *abs/1207.3532*.

Lim, H., Fan, B., Andersen, D. G., & Kaminsky, M. (2011, October). SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (pp. 1–13). Cascais, Portugal.

Liu, Y., Schmidt, B., & Maskell, D. L. (2011). Parallelized short read assembly of large genomes using de Bruijn graphs. *BMC Bioinformatics*, *12*, 354.

Ludwig, H., Keller, A., Dan, A., King, R. P., & Franck, R. (2003). Web service level agreement (WSLA) language specification. *IBM Corporation*, 815–824.

Macías-Escrivá, F. D., Haber, R., del Toro, R., & Hernandez, V. (2013). Self-adaptive systems: a survey of current approaches, research challenges and applications. *Expert Systems with Applications*, *40*(18), 7267–7279.

Majkowski, M. (2010, October). Ydb. Retrieved from http://code.google.com/p/ydb

Mamei, M. & Zambonelli, F. (2009). Programming pervasive and mobile computing applications: the TOTA approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *18*(4), 1–56.

Marmalade. (2014). Marmalade SDK. Retrieved from https://www.madewithmarmalade.com/

Mehlhorn, K. & Näher, S. (1995, January). LEDA: a platform for combinatorial and geometric computing. *Communications of the ACM*, *38*(1), 96–102.

Melsted, P. & Pritchard, J. K. (2011). Efficient counting of k-mers in DNA sequences using a Bloom filter. *BMC Bioinformatics*, *12*, 333.

Meredith, J. S., Ahern, S., Pugmire, D., & Sisneros, R. (2012, May). EAVL: the extreme-scale analysis and visualization library. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (pp. 21–30). Cagliari, Italy.

Mitzenmacher, M. (2001). Compressed Bloom filters. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing* (pp. 144–150). PODC '01. Newport, USA.

Murch, R. (2004). *Autonomic computing*. IBM Press.

Murty, J. (2014). Jets3t. Retrieved from http://www.jets3t.org/index.html

Nie, Z., Hua, Y., Feng, D., Li, Q., & Sun, Y. (2014, August). Efficient storage support for real-time near-duplicate video retrieval. In *Proceedings of the 14th International Conference on Algorithms and Architectures for Parallel Processing* (pp. 312–324). Dalian, China.

Nierstrasz, O., Denker, M., & Renggli, L. (2009). Model-centric, context-aware software adaptation. *Software Engineering for Self-Adaptive Systems*, *5525*, 128–145.

OblakSoft. (2014). WebStor: high-performancee API for cloud storage. Retrieved from http://www.oblaksoft.com/downloads/

Oracle. (2014). Oracle Berkeley DB Java edition. Retrieved from http://www.oracle.com/technetwork/database/berkeleydb/overview/index-093405.html

Partow, A. (2000). C++ Bloom filter library. Retrieved from https://libbloom.codeplex.com/

Pell, J., Hintze, A., Canino-Koning, R., Howe, A., Tiedje, J. M., & Brown, C. T. (2012). Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences*, *109*(33), 13272–13277.

Petrovic, J. (2008, April), In *Proceedings of the 3rd International Conference on Systems* (pp. 368–372). Cancun, Mexico.

Peuvrier, L. (2012). Joafip. Retrieved from http://joafip.sourceforge.net/

Pevzner, P. A., Tang, H., & Waterman, M. S. (2001, August). An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences of the United States of the America*, *98*(17), 9748–53.

Pike, G. & Alakuijala, J. (2013). The CityHash family of hash functions. Retrieved from https://code.google.com/p/cityhash/

Python, A. (2014). Dropbox-c. Retrieved from https://github.com/Dwii/Dropbox-C

Rohr, M., Giesecke, S., Hasselbring, W., Hiel, M., van den Heuvel, W.-J., & Weigand, H. (2006, September). A classification scheme for self-adaptation research. In *Proceedings of the International Conference on Self-Organization and Autonomous Systems in Computing and Communications (SOAS'2006)* (p. 5). Erfurt, Germany.

Rottenstreich, O., Kanizo, Y., & Keslassy, I. (2014). The variable-increment counting Bloom filter. *IEEE/ACM Trans. Netw. 22*(4), 1092–1105.

Saad, R. T., Zilio, S. D., & Berthomieu, B. (2010, October). A general lock-free algorithm for parallel state space construction. In *Proceedings of the 2010 9th International Workshop on Parallel and Distributed Methods in Verification, and 2nd International Workshop on High Performance Computational Systems Biology* (pp. 8–16). Enschede, Netherlands.

Salehie, M. & Tahvildari, L. (2009, May). Self-adaptive software: landscape and research challenges. *ACM Transactions on Autonomic and Adaptive Systems*, *4*(2), 14:1–14:42.

Salvaneschi, G., Ghezzi, C., & Pradella, M. (2012, March). ContextErlang: introducing context-oriented programming in the actor model. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development* (pp. 191–202). AOSD '12. Potsdam, Germany.

Sanders, P., Schultes, D., & Vetter, C. (2008, September). Mobile route planning. In *Proceedings of the 16th Annual European Symposium on Algorithms* (pp. 732–743). Karlsruhe, Germany.

Schaeffer-Filho, A., Lupu, E., Sloman, M., & Eisenbach, S. (2009, July). Verification of policy-based self-managed cell interactions using Alloy. In *Proceedings of 2009 IEEE International Symposium on Policies for Distributed Systems and Networks* (pp. 37–40). London, UK.

Simpkins, C., Bhat, S., Isbell, C., Jr., & Mateas, M. (2008, October). Towards adaptive programming: integrating reinforcement learning into a programming language. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (pp. 603–614). OOPSLA '08. Nashville, USA.

Singh, J. P., Holt, C., Totsuka, T., Gupta, A., & Hennessy, J. (1995, June). Load balancing and data locality in adaptive hierarchical n-body methods: barnes-hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, *27*(2), 118–141.

Singler, J., Sanders, P., & Putze, F. (2007, August). MCSTL: the multi-core standard template library. In *Proceedings of the 13th International Euro-Par Conference on Parallel Processing* (pp. 682–694). Rennes, France.

Smaalders, B. (2006, February). Performance anti-patterns. *Queue*, *4*(1), 44–50.

Souza, V. E. S. (2012, June). *Requirements-based software system adaptation* (Doctoral dissertation, University of Trento).

Stern, U. & Dill, D. L. (1995, October). Improved probabilistic verification by hash compaction. In *Proceedings of the IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods* (pp. 206–224). Frankfurt, Germany.

Stevens, R., Parsons, B., & King, T. M. (2007). A self-testing autonomic container. In *Proceedings of the 45th Annual Southeast Regional Conference* (pp. 1–6). Winston-Salem, USA.

Suto, T., Bradley, J. T., & Knottenbelt, W. J. (2006, September). Performance Trees: a new approach to quantitative performance specification. In *Proceedings of the 14th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (pp. 303–313). Monterey, USA.

Sutton, R. S. & Barto, A. G. (1998). *Introduction to reinforcement learning*. MIT Press.

Suvée, D., Vanderperren, W., & Jonckers, V. (2003, March). JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development* (pp. 21–29). Boston, USA.

Teodoro, G., Valle, E., Mariano, N., Torres, R., & Meira, W., Jr. (2011, October). Adaptive parallel approximate similarity search for responsive multimedia retrieval. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (pp. 495–504). Glasgow, UK.

Tesauro, G., Chess, D. M., Walsh, W. E., Das, R., Segal, A., Whalley, I., … White, S. R. (2004). A multi-agent systems approach to autonomic computing. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1* (pp. 464–471). New York, USA.

Tosic, V., Patel, K., & Pagurek, B. (2002). WSOL - web service offerings language. In *Revised papers from the international workshop on web services, e-business, and the semantic web* (pp. 57–67).

Unity Technologies. (2015). Unity. Retrieved from http://unity3d.com/unity/multiplatform

Upson, C., Faulhaber, T., Jr., Kamins, D., Laidlaw, D. H., Schlegel, D., Vroom, J., … van Dam, A. (1989, July). The application visualization system: a computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, *9*(4), 30–42.

Vengroff, D. E. (1994, July). A transparent parallel I/O environment. In *Proceedings of the 3rd DAGS Symposium on Parallel Computation* (pp. 117–134). Hanover, USA.

Vo, H. T., Silva, C. T., Scheidegger, L. F., & Pascucci, V. (2012). Simple and efficient mesh layout with space-filling curves. *Journal of Graphics, GPU, & Game Tools*, *16*(1), 25–39.

Vuduc, R., Demmel, J. W., & Yelick, K. A. (2005, June). OSKI: a library of automatically tuned sparse matrix kernels. In *Proceedings of SciDac 2005, Journal of Physics: Conference Series* (Vol. 16, pp. 521–530).

Wang, H. & Liu, K. (2012, August). User oriented trajectory similarity search. In *Proceedings of the ACM SIGKDD International Workshop on Urban Computing* (pp. 103–110). Beijing, China.

Watt, D. A. & Brown, D. (2001). *Java Collections: an introduction to abstract data types, data structures and algorithms* (1st). New York, NY, USA: John Wiley & Sons, Inc.

Welsh, M. & Culler, D. (2000). Jaguar: enabling efficient communication and I/O in Java. *Concurrency and Computation: Practice and Experience, Special Issue on Java for High-Performance Applications*, *12*(7), 519–538.

Wewetzer, C., Scheuermann, B., Lübke, A., & Mauve, M. (2009, October). Content registration in VANETs - saving bandwidth through node cooperation. In *Proceedings of the 3rd IEEE LCN Workshop on User MObility and VEhicular Networks* (pp. 661–668). Zurich, Switzerland.

Witkowski, C. M. (1983, August). A parallel processor algorithm for robot route planning. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence - Volume 2* (pp. 827–829). Karlsruhe, West Germany.

Wolper, P. & Leroy, D. (1993). Reliable hashing without collision detection. In *Proceedings of the 5th International Conference on Computer Aided Verification* (pp. 59–70). Elounda, Greece.

Zerbino, D. R. & Birney, E. (2008). Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, *18*(5), 821–829.

Zhao, Y., Tang, H., & Ye, Y. (2012). RAPSearch2: a fast and memory-efficient protein similarity search tool for next-generation sequencing data. *Bioinformatics*, *28*(1), 125–126.