UNIVERSITY OF WATERLOO

Faculty of Engineering
Department of Electrical and Computer Engineering

SELECTION AND ANALYSIS OF ASSOCIATIVE DATA STRUCTURES

Harris Corporation
Toronto, ON

Prepared by
Paul Roukema
ID XXXXXXXX
userid pwroukem
1A

September 19, 2010

26 Gretna Green,
London, ON
N6G 1X1

September 19, 2010

Professor Sachdev, Chair
Electrical and Computer Engineering,
University of Waterloo,
Waterloo, ON
N2L 3G1

Dear Professor Sachdev:

This report, entitled "Selection and Analysis of Associative Data Structures", was prepared as my 1A Work Report for Harris Corporation. This report is in fulfillment of the course WKRPT 100. The purpose of this report is to evaluate associative data structures in C++ for use in a feature enhancement.

Harris Corporation is a global communications and IT services company serving the government and commercial markets. The Broadcast Communications division, whose Command Control Systems team, managed by Leonardo Araujo, I worked for, creates hardware and software solutions to manage a wide range of broadcast hardware and software.

I would like to thank Mr. Leonardo Araujo for allowing me to work on this feature and for providing feedback on the required functionality. I would also like to thank Mr. Hashem Moustafa for helping me determine how to implement the feature by suggesting where the additional required information could be found. Lastly I wish to thank Mr. Simon Law for creating the uw-wkrpt Latex document class which was used to typeset this report.

I hereby confirm that I have received no further help other than what is mentioned above in writing this report. I also confirm that this report has not been previously submitted for academic credit at this or any other academic institution.

Yours sincerely,

_____

Paul Roukema,
ID XXXXXXXX

Contributions

At Harris Corporation, I worked on the Monitoring & Control Systems team as part of the Research and Development group in the Broadcast Communications Division. The team was fairly small, consisting of 12 people. My manager was Leonardo Araujo.

The Monitoring & Control Systems group is responsible for the development and maintenance of software and hardware products that are used to determine the operational parameters of the company's other products, and to report their status to operators. The products we develop include a desktop application called *Navigator* and a hardware control panel called *Nucleus*, as well as some of the firmware for other product's control and resource cards.

The great majority of my four months at Harris were spent helping the team prepare a new version of Navigator for release. This release was focused on integrating control and configuration for the company's router products more tightly with Navigator. Most of my work was focused on fixing defects in the software, although I was responsible for the implementation of a small number of features as well.

The majority of the features I implemented were straightforward and were completed without the need for any major analysis, however one feature posed a more significant challenge. The initial implementation of this feature went smoothly, however analysis showed that performance was lacking. Further investigation showed that this was due to the presence of large numbers of inter-process procedure calls inside of an inner loop. Due to the usage scope of these calls and the availability of all data required for this functionality within the other process it was then possible to move this functionality into the other process for an approximately hundredfold performance improvement. The removal of inter-process calls also played a part in a major optimization that I performed, where I was able to remove a database access from a recursive method, saving almost a thousand calls and several seconds.

One task that I was assigned involved developing a new feature for one of our internal tools used as part of the build process. This tool is used to compile information from interface definition files into database tables for inclusion with Navigator. I was tasked with providing the capability to ensure that only products that were released and supported in Navigator were include in these tables, both in order to reduce the size of these tables as well as to protect information about unreleased products.

This report is focused on selecting the map data structure implementation for use in implementing this feature.

Overall my work at Harris contributed to one of the company's primary initiatives, known as Harris ONE, which focuses on tighter integration and interoperability between product lines. This was supported by the software release I worked on, which was, as previously mentioned, focused on router control integration. In addition, as most of my work centered around fixing bugs, I helped to improve general product quality. I also worked on a number of minor projects to improve our software development processes.

Summary

The purpose of this report is to examine associative data structures for use in an enhancement to a C++ program used during the build process of Harris Broadcast's Navigator application. This report examines the Standard Template Library's `std::map`, Technical Report 1's `std::tr1::unordered_map` and the nonstandard `stdext::hash_map`. This report is intended for a reader who has some knowledge of data structures and performance analysis.

The major sections of this report are candidates, approach and data. Candidates discusses the data structures being considered, along with their theoretical performance and scaling. Approach discusses the testing methods used to gather real world data. Data examines the quantitative performance of the data structures and compares it to the theoretical performance. Test results are analyzed here to determine the causes of any anomalies observed in the results. The last section of the analysis is Standardization, Portability and Flexibility, which looks at some of the usability and portability issues with the options.

The major conclusions of the report are that `hash_map` is the best data structure for the feature enhancement. `hash_map` had the best performance both when inserting and when performing lookups. It has some weaknesses with regards to portability, but these are not particularly important for the program being examined. From a performance standpoint `unordered_map` is the second best option and `std::map` is the third best.

The major recommendations of the report is that `hash_map` be used to implement the feature enhancements. In addition it is recommended that should the portability concerns regarding `hash_map` make its use undesirable, `unordered_map` be used in its place.

Table of Contents

List of Figures

List of Tables

1 Introduction

Harris Corporation's Broadcast Communications division is a worldwide supplier of broadcast equipment and software. It serves customers in over 150 countries, providing highly integrated solutions to meet the needs of broadcasters worldwide[1]. Harris has a history of innovation, providing over 70 technical firsts in the broadcast industry. The purpose of this report is to identify a solution to an information leak detected in part of the build process for the CCS Navigator product. In this section, background on the product and problem being worked on is presented, in addition to information on the scope and outline of the report.

1.1 Background

The CCS Navigator application is used to control and configure the hardware devices produced by Harris, including video and audio processing hardware, routers, encoders, servers and multiviewers, as well as third party products. Harris devices are primarily supported using a Harris developed protocol called CCS-P, which allows for real time monitoring and control of products[2]. Third party products can be supported using SNMP. In the case of SNMP, the interface provided by a product is described by a Management Information Base (MIB) stored in a file [3].

Although the CCS-P protocol lacks a direct analog to the concept of a MIB file and device interfaces can be discovered dynamically, interface definitions are still created and stored. These interface definitions are stored as XML files which are processed during CCS Navigator's build process into an internal representation. Because the interface definitions are stored in the source control system, even during development and before product release, this step is the source of a potential information leak about unreleased products, as well as a source of unnecessary additional file size when distributing Navigator.

1.2 Scope

The purpose of this report is to determine the appropriate associative data structure to use in order to efficiently exclude information about unreleased products from the CCS-P interface descriptions shipped with the Harris CCS Navigator application.

The basic approach to the problem is largely dictated by the available data and the desired output subset, however the algorithms and data structures used could make a significant difference in the speed of the process. The primary quantitative concerns for this data structure are insertion and lookup behavior with regard to time and the number of entries. Memory usage is not a major concern as the number of elements to be stored is relatively low in comparison to size of the main memory in modern computers. Qualitatively, the ease of integration and long term future of the data structures is also important. Due to the platform being targeted and the existing body of code that must be integrated with, this report will only examine classes and libraries that integrate easily with Microsoft's development tools. Also, although there is some potential for performance gains though these measures, tuning features such as maximum load factor and initial size will be left at their defaults in order to present a level playing field.

1.3 Outline

This report will contain one primary section examining the available data structures, divided into several subsections. The first subsection will introduce the data structures and their overall operation. The second subsection will discuss the methods used to gather the data, as well as the precise setup used to gather the data. The third subsection will present the results of the testing, discussing the relative performance of the data structures and examining how they scale. The last subsection will examine standardization and usability issues with each data structure.

Conclusions about the relative performance of the data structures will then be summarized along with conclusions about the portability of the options. Finally recommendations as to which data structure should be used will be made.

2 Data Structure Analysis

2.1 Candidates

The general operational mode of the program being written is to read through an XML file defining the Catalog used in the Navigator application to facilitate offline

configuration, parsing out the device identifiers for each entry. This data, along with the name of the device is to be stored. The program will then work though a directory tree containing interface definition files, parsing through each one and outputting its contents into Navigator's interface definition format if the device identifier was seen in the catalog file. In addition the program will keep track of the interface definitions it has skipped, as well as those that it has missed in order to provide a report upon completing execution.

In order to efficiently implement this operational flow, it was determined that a map or dictionary abstract data type was appropriate. Examining the available libraries identified three primary candidates for use as the concrete implementation of the data type. All of the options provide both the Unique Associative Container concept [4] and the Pair Associative Container concept [5]. The first and most obvious option is the C++ Standard Template Library's `std::map` [6]. This class is widely used and well tested, being the standard choice for a associative data structure in C++. `std::map` provides reasonable worst case runtime guarantees of $O(\log N)$ for both insertions and lookups with regard to the number of entries in the map [7].

The second option to be examined is the implementation of `stdext::hash_map` provided in Microsoft's C++ base libraries [8]. It should be noted that this differs from the `std::hash_map` defined by the SGI implementation of the STL. Although they implement the same general interface, they have incompatible template arguments for specifying the hash and compare functions. This class is not formally specified as part of the STL, but is able to provide improved performance in the average case, running in constant time, although worst case performance is now $O(N)$.

The third option is `std::tr1::unordered_map` class provided as part of the Technical Report 1 (TR1) extensions to the standard C++ library [9]. Although this class is not yet a formal part of the standard library, an implementations are provided with some recent development environments, such as Microsoft Visual Studio 2008 and as part of the Boost C++ library [10]. As part of TR1, `unordered_map` is slated to become part of the actual C++ standard in the near future. This has potential maintainability benefits in the long term.

2.2 Approach

In order to gather quantitative information about the relative performance of the data structures involved, it was decided to write a custom program in C++ to provide relevant, empirical data about the performance characteristics of the data structures under a realistic workload for the usage scenario. This program gathers data about two primary test scenarios. The first is the time required to perform a varying number of insertions into each data structure. The second is the time required to execute a fixed number of lookups on each data structure for each size of data structure in the previous test.

Test runs are executed based on the total number of inserts to attempt. Assuming this is an even number, exactly half of these will create a new map entry containing the number of the insert modulo one half the total number of inserts in a `std::wstring`. Once the inserts are complete, the program attempts 10,000 sequential lookups on the map. Timing data is collected separately for each stage using the Windows `QueryPerformanceCounter` API. On the system used to perform these tests, this API provides a granularity of approximately 70 ns, sufficient to produce accurate results given the observed execution times. The program performs all of the tests 101 times each, dropping the first result in order to ensure that all necessary code and data segments have had the opportunity to be cached. The remaining runs have their execution times collected and averaged to provide more accurate results.

The results shown here were collected on a Lenovo Thinkpad X61, with an Intel Core 2 Duo T8300 processor and 4 GiB of RAM. Test programs were compiled using the Microsoft C++ Compiler, version 15.00.30729.01 and the libraries used are those provided by Visual Studio 2008. All optimization flags and options were left at their default settings. The operating system used was Microsoft Windows Vista Business x64 Service Pack 1. Despite the use of a 64-bit operating system, the binaries created were targeted for the x86 architecture, as this is representative of the environment used in production.

2.3 Data

Test runs were performed for five different numbers of insertions and map sizes. The results of the tests are summarized in Table 1. The insertion counts tested were 1000,

2000, 3000, 8000, and 16000 entries. Insertion times are for the full number of inserts shown in the left hand column. Lookup times are for 10,000 lookups on the map created by the insertions with the number of entries actually created shown in the parentheses. Timing results have been rounded to the nearest microsecond in order to make analysis cleaner and to help ensure that the resolution of the timer does not introduce errors.

Table 1: Execution times for benchmarking runs in microseconds

Insertions	unordered_map		std::map		hash_map	
(Unique)	Insert	Lookup	Insert	Lookup	Insert	Lookup
1000 (500)	1277	1433	1302	3477	1036	775
2000 (1000)	2516	1888	2674	3508	1999	917
4000 (2000)	5034	1983	5571	4379	4149	1444
8000 (4000)	12236	1775	11519	4589	8380	855
16000 (8000)	22560	2261	23554	5268	16624	1152

2.3.1 Insertion

Insertion times show a fairly linear progression in all cases, as seen in Figure 1. This fits well with the predicted performance of the two hash based maps, but does not match with the expected scaling of `std::map`, which would be expected to demonstrate logarithmic scaling in the time required for each insertion. The other major deviation from the expected performance is an unusually large increase in execution time for `unordered_map` when making 8000 insertions.

Both of these deviations are fairly minor overall and may be caused by certain aspects of the data structure's memory behavior. Since `unordered_map` is hash based, during the insertion process, as the number of entries increases it is occasionally necessary to increase the size of the hash table. Thus helps to avoid having to high a load factor on the data structure, which would lead to degraded performance. The process of increasing the size of the hash table will take additional time beyond the insertion time. In the case of `std::map` the run times may not be displaying the expected logarithmic worst case insertion times due to the nature of the inserted strings. The inserted strings will be mostly in ascending order according to the standard lexicographical compare used when inserting items into the map.

Overall `hash_map` clearly operates significantly faster that the other two options. Quantitatively it performs more than 35% faster than `unordered_map`. It also demonstrates nearly perfect linear scaling in runtime as the number of elements increases, indicating that insertions times are constant as expected. From the perspective of insertion performance `hash_map` is the best option.

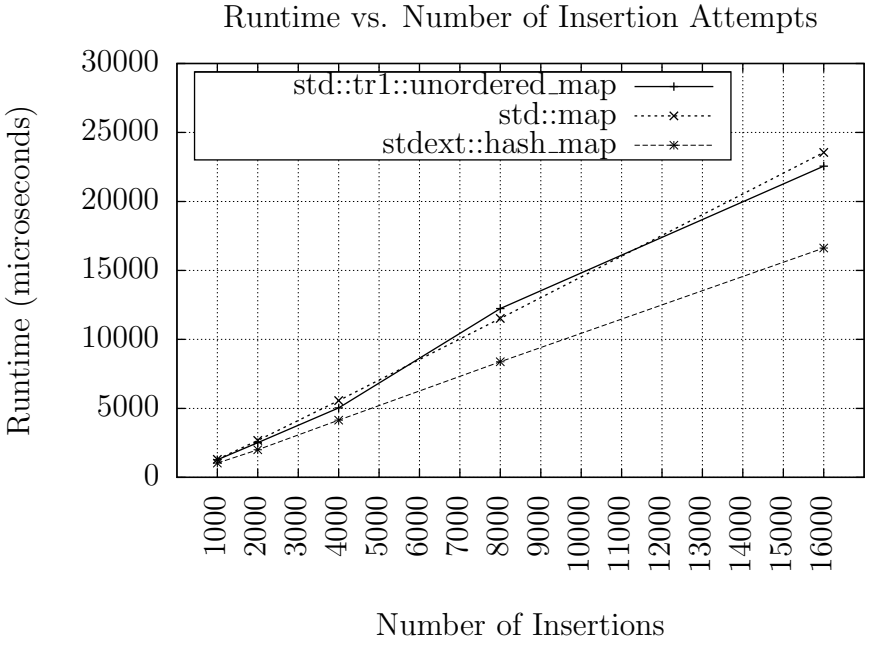

Figure 1: Execution time for map insertions, 50% duplicates

2.3.2 Lookup

Lookup times provide a clearer picture of their performance since the number of lookups could be held constant, allowing easier analysis of the runtime scaling. As can be seen in Figure 2, both of the hash based maps demonstrate a loose trend of slowly increasing execution times with larger map sizes. In comparison, `std::map` shows a clear monotonic trend of increasing execution times.

The stronger initial performance of `std::map` for map sizes under 1000 is likely due to caching effects as the smaller maps are likely able to fit mostly within the 32 KiB L1 cache of the processor of the computer used. Even accounting for the apparent performance decrease that the this causes, the scaling effects can still be seen as the run time increases in a regular manner. The performances of `unordered_map`

and `hash_map` do not initially appear to demonstrate the constant average lookup time that is suggested by their reference pages. Closer examination suggests that the lookup times are remaining nearly constant, with performance variations being caused by varying load factors on the hash tables.

The above theory about the cause of lookup performance variations is supported by the decreased execution time for `unordered_map` at 4000 entries, which corresponds the increased execution time observed when inserting 8000 entries. As discussed above, the slower insert times could correspond to hash table resize which would result in a reduced load factor. Reducing the load factor on the hash table would result in fewer elements per bucket and less time spent searching linearly through the bucket contents, resulting the observed faster lookups.

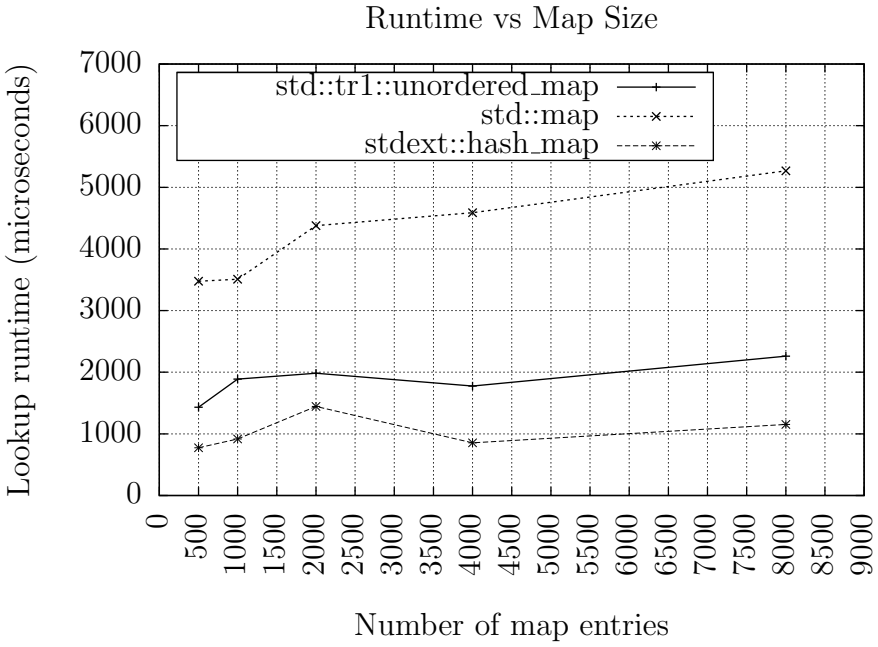

Figure 2: Execution time for 10,000 map lookups

Overall there is a clear ordering of the lookup performance between the different containers, with `std::map` clearly demonstrating inferior performance and scaling. Comparing the scaling results between `unordered_map` and `hash_map` shows that both demonstrate remarkably similar scaling patterns. Despite the similarity in the patterns `hash_map` clearly offers better performance, in many cases operating at least twice as fast as `unordered_map`.

2.4 Standardization, Portability and Flexibility

The three associative containers being examined have differing origins and levels of standardization. These differences make some of the options easier to use and support in the long term, as well as making them more practical for future use with multiple tool sets. This is important as the development tools in use are always evolving as newer versions and technologies become available.

`std::map` is the most straightforward option, originating with the classical C++ Standard Template Library as developed by SGI and HP [11]. As a result of its origin, this class is fully specified and modern libraries, supported by compilers which support all required features, implement it uniformly. This uniformity of implementation as well as its origin with the STL mean that this is the most portable option.

`unordered_map` originates with C++ Technical Report 1, whose proposals are slated for inclusion in the next official C++ specification, known as C++0X. This positions `unordered_map` as the second best option in terms of standardization. In terms of portability, implementations are already available from multiple sources, including Microsoft [12] and the Boost C++ libraries [10]. The availability of this option though Boost makes integration on platforms lacking native support a relatively simple matter as well. The flexibility of this option is somewhat better than that of `std::map` as it does not require a comparison operator or `std::less<T>` template specialization to exist for the key data type, relying instead on a hash function and an equality operator. This can be advantageous if the key objects are difficult or expensive to impose an ordering on.

`hash_map` is an extension to the STL and is not a formally standardized component of it [8]. As a result there are some inconsistencies in the implementations. The lack of standardization also means that its availability may be nonexistent on some platforms. One of the major inconsistencies is in the template parameters used to specify the data structure. In many implementations the template for `hash_map` is `hash_map<Key, Data, HashFcn, EqualKey, Alloc>` which allows easy specification of alternate hash functions and uses an equality comparison [13]. This template is incompatible with the template parameters used by the implementation provided by Microsoft, which uses `hash_map<Key, Type, Traits=hash_compare<Key, less<Key> >, Allocator>` [8] where `hash_compare` implements `operator()` in unary form as the hash function and in binary form as a comparison. This interface is more

difficult to work with than the interface shown above and also requires that an ordering be imposed on the key objects. The non-standard nature of this class along with the major inconsistencies in interface between implementations make `hash_map` the worst option in terms of standardization and portability.

3 Conclusions

From the analysis in section 2 it was determined that `hash_map` provided the best overall performance. It is followed by `unordered_map` and lastly `std::map`.

Section 2.3.1 discussed insertion performance and showed that `hash_map` had the best performance followed by `unordered_map` in most cases and then `std::map`. This means that `hash_map` is the best option in terms of insertion performance.

Section 2.3.2 looked at lookup performance and showed that `hash_map` had the best performance on lookups well. `unordered_map` again performed second best followed by `std::map`. This means that `hash_map` is also the best option in terms of insertion performance.

Section 2.4 discussed the origin and availability of the options. In this area, `std::map` is the best option as it is fully standardized as part of the C++ standard library. `unordered_map` is the second best option here since it is on track to become a part of the C++ standard library. `hash_map` is the worst option due to variations in implementation and lack of standardization.

Overall, the performance demonstrated by `hash_map`, performing the best in the two quantitative tests and worst in terms of standardization and portability make it the best option. This is especially true since portability is not a major concern for the target application, limiting the importance of that factor.

4 Recommendations

The testing and analysis performed in this report show that `hash_map` has the best overall performance. `hash_map` showed better results than the other options in both of the performance tests. `hash_map` is available and supported fully on the current development tools in use and is also available on the next version of the development

tools, indicating that it can continue to be used in the future. Based on these results it is recommended that `hash_map` be used as the data structure for the implementation of the unreleased devices exclusion feature. This data structure is likely to cause the smallest overhead in the execution of the tool being modified. Although it is considered unlikely, should it be required that the tool become portable for use with alternate development tools or should `hash_map` be removed from the libraries at some point in the future, it is recommended that `unordered_map` be used in order to replace `hash_map`. Both data structures showed similar lookup performance and `unordered_map` has better portability and standardization.

Glossary

API - Application programming interface, a collection of functions and data structures used to access operating system or library functionality.

C++0X - The working name for the next version of the C++ programming language, which will include TR1 along with lower level language changes.

L1 Cache - The fastest part of a computer's memory hierarchy. Consists of fast static RAM on the processor die.

RAM - Random Access Memory, the primary storage location for a computer's active data.

SNMP - Simple Network Management Protocol, a network protocol used to monitor and in some cases configure networked devices.

STL - The C++ Standard Template Library, a software library for C++ build using generic programming to provide containers, iterators and algorithms.

TR1 - ISO/IEC Draft Technical Report 19768, also known as Technical Report 1, is a document describing a number of extensions to the C++ Standard Library, including the `unordered_map` class discussed in this report.

References

[1] Harris Corporation. "Broadcast overview / backgrounder," [online]. Available from: http://www.broadcast.harris.com/overview/background.asp [cited 21 April 2009].

[2] Harris Corporation. "CCS Navigator," [online]. Available from: http://www.broadcast.harris.com/product_portfolio/product_details.asp?sku=CCS_Navigator [cited 21 April 2009].

[3] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser, "Introduction to version 2 of the internet-standard network management framework," RFC 1441, Internet Engineering Task Force, Apr. 1993. Available from: http://www.rfc-editor.org/rfc/rfc1441.txt.

[4] *Unique Associative Container*. Available from: http://www.sgi.com/tech/stl/UniqueAssociativeContainer.html.

[5] *Pair Associative Container*. Available from: http://www.sgi.com/tech/stl/PairAssociativeContainer.html.

[6] Silicon Graphics Inc., *map<Key, Data, Compare, Alloc>*. Available from: http://www.sgi.com/tech/stl/Map.html [cited 4 May 2009].

[7] Caldera International Inc., *std::map Performance*, October 2002. Available from: http://uw713doc.sco.com/en/SDK_c++/_Perform_map.html [cited 4 May 2009].

[8] Microsoft Corp., *hash_map Class*. Available from: http://msdn.microsoft.com/en-us/library/0d462wfh.aspx [cited 5 May 2009].

[9] M. Austern, "Draft Technical Report on C++ Library Extensions," ISO/IEC DTR 19768, C++ Technical Commitee, June 2005.

[10] James Daniel, *Chapter 25. Boost.Unordered*, May 2009. Available from: http://www.boost.org/doc/libs/1_39_0/doc/html/unordered.html [cited 6 May 2009].

[11] A. Stepanov and M. Lee, "The standard template library," Technical Report 95-11, HP Labratories, November 1995.

[12] Microsoft Corp., *unordered_map Class*. Available from: http://msdn.microsoft.com/en-us/library/bb982522.aspx [cited 5 May 2009].

[13] Silicon Graphics Inc., *hash_map<Key, Data, HashFcn, EqualKey, Alloc>*. Available from: http://www.sgi.com/tech/stl/hash_map.html [cited 4 May 2009].