NANYANG TECHNOLOGICAL UNIVERSITY

AS00-CE-F076

OPTIMISING KIRRKIRR: AN ELECTRONIC DICTIONARY BROWSER

Submitted in Partial Fulfilment of the Requirements
for the Degree of Bachelor of Applied Science
of the Nanyang Technological University

by

Sng Wee Jim

School of Applied Science
2000

FYP Supervisor: Dr. Nitin Indurkhya

ABSTRACT

This project was initiated to solve the performance problems of Kirrkirr (an electronic dictionary of English and Warlpiri, an Australian Aboriginal language), namely: slow start-up time and large memory usage. Revamping the underlying data and usage of the Extensible Query Language and Document Object Model lead to a smaller start-up index file and make possible the loading of the dictionary word in small chunks. This resulted in a very visible improvement in the start-up time, which was reduced from 7 minutes to 1 minute. The optimisation efforts also saw a reduction of memory requirement from 36Mbytes to 27MBytes. Adoption of better algorithm led to faster sorting time within the program and the installation process was shortened considerably by the archival of the 10000 definition files into a single archive. Apart from meeting the basic requirement, options were also explore to enhance the features and usability of the program. These include the addition of print modules that reduced the steps required to print out dictionary definitions; and the modification of the help system to take advantage of the user-friendly features of JavaHelp. Internationalization was also done to allow adaptation of Kirrkirr to other languages. As a result of the optimisations and enhancements made, Kirrkirr is now user-friendlier and its usage more enjoyable.

TABLE OF CONTENTS

Abstract i

CHAPTER 1: INTRODUCTION 1

1.1 Background Information 1
- 1.1.1 XML (Extensible Markup Language) 2
- 1.1.2 User Feedback 3

1.2 Objective and Scope 4

1.3 Main Accomplishments 4

1.4 Report Overview 5

CHAPTER 2: MAIN OPTIMISATIONS 6

2.1 Critique Of Old System 6
- 2.1.1 Extensive use of new keyword 6
- 2.1.2 Difference in processing speed of Human and Computer 7
- 2.1.3 Eager vs. Lazy Instantiation 7
- 2.1.4 Background Class Loading 8
- 2.1.5 Reuse and Recycle 9
- 2.1.6 Real Speed vs. Perceived Speed 10

2.2 Dictionary Representation 10
- 2.2.1 Redundancy in Old Data Structure 10
- 2.2.2 DOM (Document Object Model) 11
- 2.2.3 XML Path Language (XPath) 12
- 2.2.4 XQL (Extensible Query Language) 12
- 2.2.5 Eliminating Redundancy of Original Data Structure 16

2.3 Speeding up Retrieval of Dictionary Information 18

2.4 Dictionary Sorting 19

CHAPTER 3: OTHER OPTIMISATIONS 21

3.1 General Optimisation Techniques 21
- 3.1.1 80/20 Rule 22
- 3.1.2 Strength reduction 22
- 3.1.3 Common sub expression elimination 22
- 3.1.4 Code motion 23
- 3.1.5 Unrolling loops 23

3.2 Java Specific Optimisation 24
- 3.2.1 Using the compiler 24
- 3.2.2 Looping 25
- 3.2.3 Using Variable 27
- 3.2.4 Using Method 29
- 3.2.5 Using Operators 31
- 3.2.6 Typecasting 32
- 3.2.7 Instantiation 33
- 3.2.8 Throwing Exception 34

3.2.9 Using threads 35
3.2.10 Using String Buffer 36
3.2.11 Using Java Collections 36

CHAPTER 4: OTHER ENHANCEMENTS 38

4.1 Using JavaHelp API for the Help System 38

4.2 Installation 39

4.3 Print Function and Print Preview 39

4.4 Internationizization 41

CHAPTER 5: PERFORMANCE EVALUATION 42

5.1 Evaluation of Start-up Time 42

5.2 Evaluation of Creation of Index file 43

5.3 Dictionary Information Extraction 43

5.4 Evaluation of Memory Requirement 44

5.5 Evaluation of Sorting Time 44

CHAPTER 6: CONCLUSIONS AND FUTURE WORK 45

6.1 Future Work 45

6.2 Conclusions 46

CHAPTER 7: REFERENCES 47

LIST OF ILLUSTRATIONS

Figures

Figure 1: A Snapshot of Kirrkirr 2

Figure 2: Multithread vs. Single Thread Execution 9

Figure 3: Redundancy in Original Data Structure 11

Figure 4: A Sample DOM Tree 14

Figure 5: Faster Dictionary Information Retrieving Through Addition of A Cache. 19

Figure 6: Original Method of Sorting 19

Figure 7: New Method of Sorting 20

Figure 8: In-Built Navigation Features and Printing Option of Javahelp API 1.1 38

Figure 9: Using the Search Function in Javahelp API 1.1 38

Figure 10: Windows 9x Print Dialog & Page Setup 40

Figure 11: Print Preview Window In Solaris 40

Tables

Table 1: Normalised Time Taken to Perform Actions 6

Table 2: Main Differences Between SQL And XQL 14

Table 3: A Sample Example of the New Dictionary Data Structure 16

Table 4: Timings Justifying the Creation of A Start-Up Index 17

Table 5: Comparison of Merge Sort Against Quicksort 20

Table 6: Comparison of The Start-Up Times for Kirrkirr With Different Representations. 42

Table 7: Time Taken for Each Step of the Creation Of Start-Up Index (New Version). 43

Table 8: Summary of Time Taken for Dictionary Information Extraction 43

Table 9: Test Result for the Methods Of Sorting 44

Chapter 1: Introduction

1.1 Background Information

Kirrkirr was originally developed by Kevin Jansz [Jansz1998]. It was a bilingual electronic dictionary of English and Warlpiri (an Australian Aboriginal language) developed for a community of about three thousand people in Central Australia. The project's primary objective was in raising the literacy level of the Warlpiri community, regardless of their level of language competence and to make the learning process fun and enjoyable through its interactive nature and highly customisable interface. It consists of many interesting features not found in traditional electronic dictionary. These features are as follows:

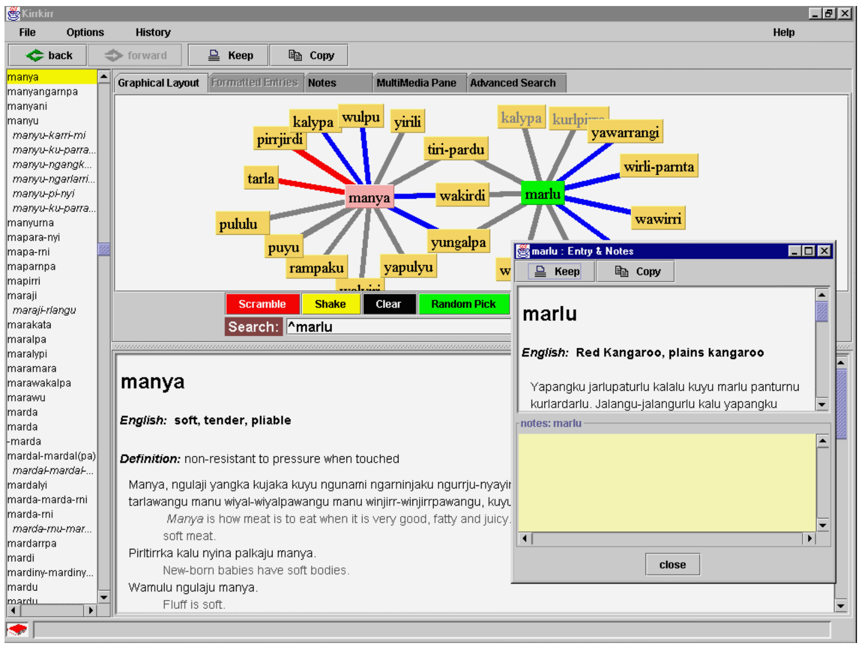
- Graph Layout: Rather than just looking up a word and returning the meaning of the word to the user, the program seek to recreate the learning experience of flipping through a real dictionary (seeing and learning new words in the process). To achieve this, the program provided the users with a visual animated representation (see Figure 1) of linkages of the word of interest to other words (synonym, antonym). This made the learning experience more effective, as the users can pick up more words in the process.

- Formatted Entries: Through the Formatted Entries tab, the user can access the entry of his selected word. Dictionary entries are formatted in the familiar HTML format, with sub-entries nicely laid out with different colours. The user can also click on a cross-referenced word and have the system jump to the entry for that word.

- Notes-Taking: To further facilitate learning, a note-taking feature was also included in the program. This enabled the user to make personal notes, simulating the real-life activity of taking notes during lectures. This was particularly useful if the user wanted to note down questions / doubts regarding the word which he intended to follow up with his Warlpiri tutor. The notes taken can also be saved in a user profile for revisions in future sessions.

- Multimedia: The user can hear the correct pronunciation of the word in Warlpiri through the multimedia tab. This feature greatly helped the user to acquire spoken Warlpiri.

- Search Function: The search function was included to enable the user to query the database. Advance search capabilities include searching using regular expressions, fuzzy spelling or just plain spelling. Fuzzy spelling was also included to help the users search for a word by how it sounds if they are unaware of its spelling. Common spelling mistakes were also tolerated by the program: in the occurrence of these mistakes, the program will auto-correct the spelling before performing the search.



manya

English: **soft, tender, pliable**

Definition: non-resistant to pressure when touched

Manya, ngulaji yangka kujaka kuyu ngunami ngarninjaku ngurrju-nyayir
tarlawangu manu wiyal-wiyalpawangu manu winjirr-winjirrpawangu, kuyu
Manya is how meat is to eat when it is very good, fatty and juicy.
soft meat.

Pirltirrka kalu nyina palkaju manya.
New-born babies have soft bodies.

Wamulu ngulaju manya.
Fluff is soft.

marlu

English: **Red Kangaroo, plains kangaroo**

Yapangku jarlupaturlu kalalu kuyu marlu panturnu kurlardarlu. Jalangu-jalangurlu kalu yapangku

Figure 1: A snapshot of Kirrkirr

1.1.1 XML (Extensible Markup Language)

The dictionary information was scripted and stored in XML. XML is a text-based markup language that is fast becoming the standard for data interchange on the Web. As with HTML, the data are identified using tags (identifiers enclosed in angle brackets: <...>). Collectively, the tags are known as "markup". An XML document is flexible and extensible, allowing new tags to be added without breaking an existing document structure. In addition, it is also possible to script in a wide variety of global languages, as XML support usage of Unicode.

HTML is used for formatting and displaying data while XML represents the contextual meaning of the data. For example, "<b>...</b>" in HTML tag means "display this data in bold font", whereas an XML tag acts like a field name in the database, putting a label on a piece of data that identifies it (e.g. < message >...</ message>).

In the same way that the field names are defined for a data structure, the XML tags provides the freedom to use tag name that make sense for a given application. For multiple applications to use the same XML data, they just have to agree on the tag names to be used.

```
<message to="you@yourAddress.com" from="me@myAddress.com"
         subject="XML Is Really Cool">
    <text>
       How many ways is XML cool? Let me count the ways...
    </text>
</message>
```

The ability for one tag to contain others gives XML its ability to represent hierarchical data structures. This is illustrated in the above example where the <text> tag is contained inside the <message>.

Java and XML technologies are complementary: Java provides the platform-independent, maintainable code that is needed to process application-independent XML data. The Java and XML partnership is here to stay and this is very evident in the latter being adopted by Sun in the JavaHelp API to script help files and in the new model for Java Persistency in future versions of JDK.

1.1.2 User Feedback

Kirrkirr was tested by potential users and the feedbacks received were encouraging. The young children who tried it were thrilled and enthusiastic. Several suggestions were made by the users on how the system may be improved. One prominent and general complaint was that the application was too slow and was taking up too much system resources to run. This is a serious problem as the majority of the users are young children; it will be unrealistic to expect these energetic kids to stay put in front of the computer waiting for the application to load slowly. It was exactly with this problem in mind that this final year project was initiated.

1.2 Objective and Scope

This project's primary objectives are to improve firstly the performance of Kirrkirr, in particular its start-up time, which is 7 min currently and the memory usage of the program, which stands at a huge 36 Mbytes (at start-up).

Porting the program to another programming language like C++, where the source code is compiled to machine code of the specific execution platform will neither be considered nor discussed here. Though this will result in a faster execution speed (as no Java bytecode have to be interpreted by the Java virtual machine), Java' s platform independence is sacrificed in the process. Instead the project will focus on the modification of the underlying data structure and algorithm used as it is believed that this will yield more tangible improvements. Optimisation techniques (both general and Java-specific) will also be investigated and implemented to complete the optimisation.

1.3 Main Accomplishments

The main objectives mentioned above were accomplished, notably:

- Start-up time - process sped up to 1min from 7 min.
- Memory footage - reduced from 36Mbytes at start-up to 27 Mbytes, a 25% improvement.
- Speed of sorting word list - improvement of over 175%.
- Installation - archived the large number of dictionary definition files (~10000) into a single file, which in turn resulted in a much faster installation.

Apart from meeting the main objectives, efforts were also channelled into improving the usability of the program. This includes:

- Additions of print function and print preview modules that simplify and reduce the steps needed to obtain a hardcopy of the definition file.

- Porting of the help system to use the JavaHelp API. This resulted in an easier-to-use interface for the user, who can now print out help topics by clicking the icon on the toolbar and search for desired help topics by keywords.
- Internationalisation of the system. This enabled different languages (for e.g. Warlpiri) to be used for the menus and labels within the program. This is particularly useful in adapting the program for users who are English-illiterate.

1.4 Report Overview

The report will be organised into the following chapters:

Chapter 2: Main Optimisations – discusses major work done to optimise the program. It includes sections on evaluation of the original system, designing of a new dictionary representation structure, creation of start-up index and dictionary sorting.

Chapter 3: Other Optimisations – describes general and Java-specific optimisation techniques used to improve program performance.

Chapter 4: Other Enhancements – covers the enhancements made to the program to tidy things up. Included are sections on the enhancing the help system, installation process and addition of the print function.

Chapter 5: Evaluations – evaluates the performance of the new system against the original. The tests were carried out on a Pentium-133 machine with 48 MB of RAM running JDK 1.2.1.

Chapter 6: Conclusions and Future Work – recommends things to be included in future versions of the program using some of the advance Java APIs and wraps up the report.

Chapter 2: Main Optimisations

2.1 Critique Of Old System

Users of the current system feedback that the system was slow and unresponsive when running the it on their Macintosh machines. The original programmer had developed the program on a fast Pentium II machine as such memory and CPU power probably did not poses such a serious problem. In this chapter, the current system will be reviewed and options explored to create a leaner and faster system.

2.1.1 Extensive use of new keyword

It is realised that the program uses the 'new' keyword extensively to instantiate objects. This proves to be costly in terms of CPU time as shown in Table 1 by Eckel [Eckel 1998]. The time has been normalised to provide a comparison independent of the type and speed of computer used.

Operation	Example	Normalised time
Local assignment	i = n;	1.0
Instance assignment	this.i = n;	1.2
int increment	i++;	1.5
byte increment	b++;	2.0
short increment	s++;	2.0
float increment	f++;	2.0
double increment	d++;	2.0
Empty loop	while(true) n++;	2.0
Ternary expression	(x<0) ? -x : x	2.2
Math call	Math.abs(x);	2.5
Array assignment	a[0] = n;	2.7
long increment	l++;	3.5
Method call	funct( );	5.9
throw and catch exception	try{ throw e; } catch(e){}	320
synchronized method call	synchMethod( );	570
New Object	new Object( );	980
New array	new int[10];	3100

"new" is an expensive operation

Table 1: Normalised time taken to perform actions

Evidently, unnecessary creation of new objects slows down the system considerably, as the "new" keyword is one of the most time-consuming primitive actions in the Java language. By not calling "new" unnecessarily, there will be fewer objects for the Java garbage collector to collect and the amount of virtual memory needed during run time will be reduced, giving way to fewer disk swapping.

2.1.2 Difference in processing speed of Human and Computer

After the system starts-up, the program will remain idle until some external events are triggered by the user (e.g. clicking on the tabbed panels, selection of a headword). The idling time is wasteful and it would be better if the computer (running in hundreds of megahertz, capable of processing millions of instructions per second) is used to perform some other useful task during this period.

2.1.3 Eager vs. Lazy Instantiation

In eager initialisation [Bishop1999], all resources are loaded at start-up. This wastes precious memory if the user does not use the resources in the session. With lazy instantiation, a program refrains from creating certain resources until they are first needed, freeing valuable memory space and CPU processing time. However, when the resources are first required, the user will have to wait while the resources are loaded.

The old system was found to be practising Eager Initialisation as it loads all tabbed panels in its constructor at start-up. This is time-consuming and wastes precious memory.

Lazy instantiation techniques consist of 2 categories: lazy class loading and lazy object creation. Lazy class loading refers to the loading of the classes into memory only when they are first referenced. This is taken care of by the Java runtime, which has built-in lazy loading for classes. Lazy object creation tries to delay object creation until the object is needed. Lazy object creation can reduce memory usage to a much greater extent than lazy class loading.

Putting lazy object creation to practice, the tabbed panels (like Search, and Formatted Entries) that are not needed immediately after start-up should not be instantiated in the constructor of the main class. Instead `JLabels` can be used as placeholders for these panels (as it is fast to create `JLabels`). The panels can be created when the user first needs to use them. This method frees up valuable resources to perform other tasks, resulting in faster system load-up. This may help reduce the start-up time, but the program will take a performance hit when the user first requests to use the panels that are not created previously. The waiting time is short for simple tabs like Notes and Multimedia, however complicated tabs like the Search panel will keep the user waiting for several seconds.

2.1.4 Background Class Loading

The performance hit mentioned in section 2.1.3 can be alleviated if the panel is already instantiated by the time the user requires it. However, this cannot be realised if we want to reduce the start-up time through lazy instantiation.

Fortunately, there is a reasonably good solution to the problem: to load the panels in a separate thread. It offers the benefit of parallelism, enabling the program to continue loading the panels while the user uses the program. In addition, multiple panels can be loaded concurrently. This is an interesting trade-off between a big, do-everything software package and a load-as-you-go system. It starts off as a bare-bones system but gradually grows.

As mentioned previously in 2.1.2, there is a significant amount of idling time when the program is waiting for user's input. If we could use the idling time to create the unloaded panels, there is a reasonably good chance that they will be fully instantiated by the time the user first requests for it. In the event that a panel has not been fully instantiated when needed, the user will still be able to enjoy reduced waiting time as part of the constructor of the panel has already been executed.

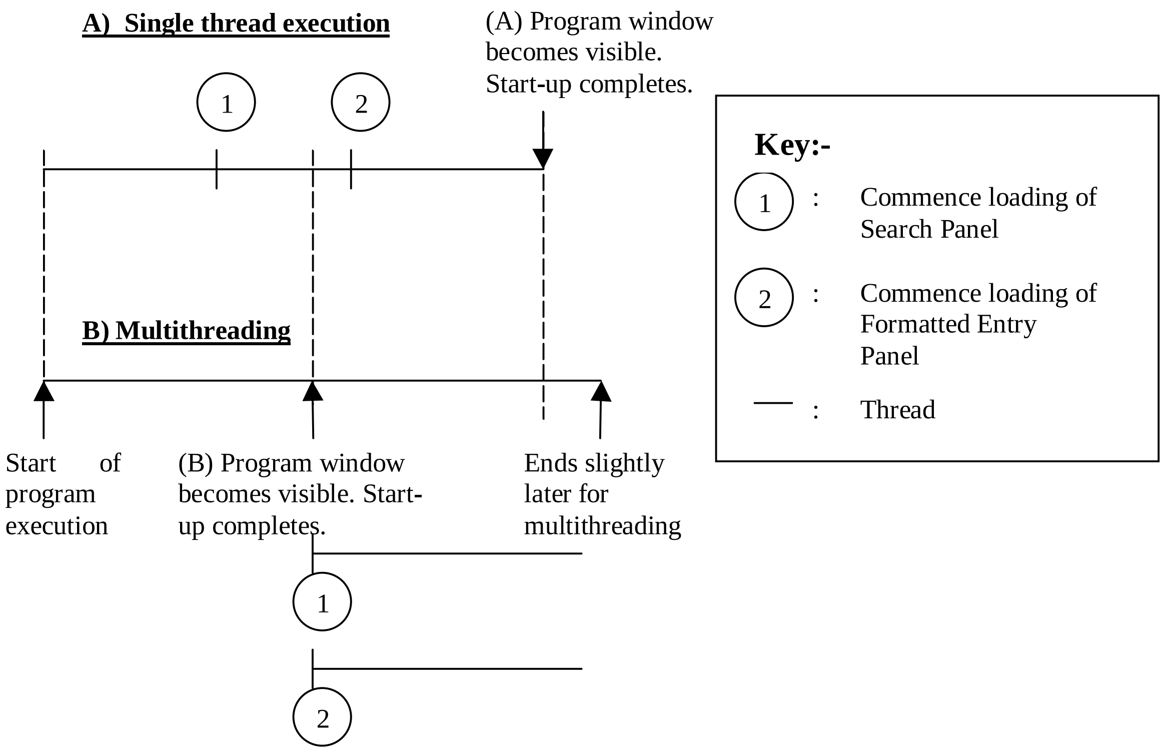

A) Single thread execution
(A) Program window becomes visible. Start-up completes.
1
2
B) Multithreading
Start of program execution
(B) Program window becomes visible. Start-up completes.
Ends slightly later for multithreading
1
2
Key:-
1 : Commence loading of Search Panel
2 : Commence loading of Formatted Entry Panel
: Thread

Figure 2: Multithread vs. single thread execution

As shown in Figure 2, the execution of the multithreaded method (for loading classes in the background) may take slightly longer than the single thread method to complete in a single processor environment due to the need for context switching among the threads. However, it does provide certain degree of parallelism to the user and has the advantage of a faster start-up.

2.1.5 Reuse and Recycle

CPU usage can be reduced through object instances reuse, as the need to create duplicate and transient instances is eliminated. For example, the tab panels should be shared between the top and bottom panes to avoid creation of duplicate objects, and to free up some memory which otherwise will be required to create the instances. The fact that the CPU can reuse an existing object faster than create a new copy of it [McManis1996] further advocates the adoption of this concept.

2.1.6 Real Speed vs. Perceived Speed

There are two types of speed associated with computers: real (machine) speed and (user) perceived speed [Bickford1997]. Of the two, the latter is of higher importance where user satisfaction is concerned. With such a concept in mind, it is advisable to perform tasks associated with visible work (GUI) first. While the user is busy absorbing what he sees on screen, other tasks may be performed in the background.

This point is of particular relevance to Kirrkirr, which needs to load a large number of dictionary words during start-up. Currently the time-consuming task of fetching all the words is performed before the creation of the user interface (thereby keeping the user waiting). Applying the above concept, the new system creates the GUI portion was created first and fetches a small amount of words to displayed to the user. This keeps him occupied while the rest of the words are being read from the harddisk. Although the time taken to perform the tasks remains more or less the same, the perceived speed will be greatly improved. Unfortunately, the current dictionary representation does not permit loading of the dictionary in small chunks, necessitating creation of a new dictionary representation (see 2.2.1.1 for discussion).

Another way to improve perceived speed is to provide feedback to users. This may take the form of a busy cursor or a progress bar. Feedback serves to pacifies the user and reduce the amount of anxiety caused by waiting.

2.2 Dictionary Representation

2.2.1 Redundancy in Old Data Structure

The current dictionary representation (an index file to be read in during start-up) contains redundancy. As illustrated in Figure 3, the headword is represented 2 (if the word does not have a sound and image file) to 4 times in the CTHashtable (a variation of the `Hashtable`). During the index file's creation (using Java serialization), back references (pointers) created for the duplicated headwords (instead of the repeated headwords) are stored. Although this reduced the size of the index file, the large number of back references stored still wasted storage space and together with the inclusion of information not needed at start-up, affected the time taken to load the dictionary (due to the large index file size).

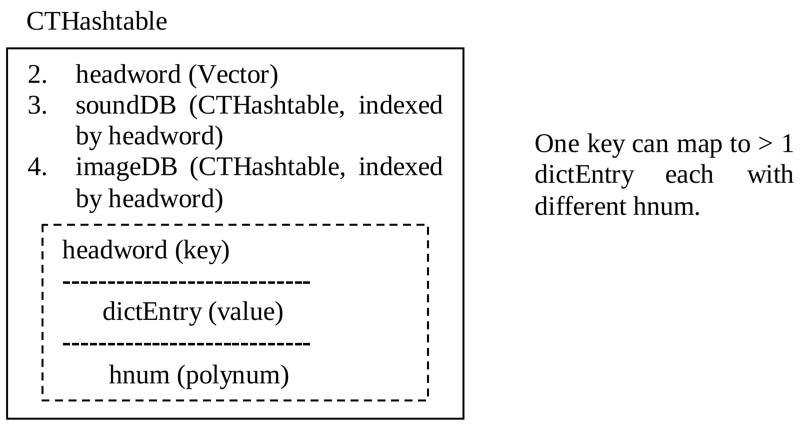

CTHashtable
2. headword (Vector)
3. soundDB (CTHashtable, indexed by headword)
4. imageDB (CTHashtable, indexed by headword)
headword (key)
dictEntry (value)
hnum (polynum)
One key can map to > 1 dictEntry each with different hnum.

Figure 3: Redundancy in Original Data Structure

2.2.1.1 New Data Structure

The new data structure attempts to reduce the size of files to be loaded at start-up (hence shortening start-up time). It makes use of the Document Object Model (see 2.2.2) and an Extensible Query Language (XQL, see 2.2.4) engine developed by GMD-IPSI of Germany. The following sections will elaborate on the concepts mentioned above, and explains why a start-up index has to be created to ensure faster start-up time.

2.2.2 DOM (Document Object Model)

XQL implementations typically operate on a model of the XML document known as the DOM (Document Object Model). The DOM is a platform-independent, programming-language-neutral application programming interface (API) for HTML and XML documents. Its core outlines a family of types that represent all the objects that make up an XML document: elements, attributes, entity references, comments, textual data and processing instructions. With that, it defines the logical structure of documents and the way a document is accessed and manipulated. (DOM specifies how XML documents are represented as objects, so that they may be used in object-oriented programs.)

Increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents. Nevertheless, XML presents this data as documents, and the DOM may be used to manage this data to allow programs to access and modify the content and the structure of XML documents from within applications. Anything found in an

XML document can be accessed, changed, deleted, or added using the DOM, except for the XML internal and external subsets for which DOM interfaces have not yet been provided.

After an XML document has been parsed into a collection of objects conforming to DOM, the object model can be manipulated in any way that makes sense. This mechanism is also known as the "random access" protocol, as any part of the data can be visited at any time. The DOM usually resides in memory (it is the output of an XML parser), but it can also be stored on disk (to save on the time needed to parse the XML repeatedly) as a Persistent DOM (PDOM). When an XML document is large and not likely to change much, as is the case for dictionaries, using its PDOM representation can significantly speed up XQL querying.

2.2.3 XML Path Language (XPath)

The primary purpose of XPath [XPath1999] is to address parts of an XML document by operating on the abstract, logical structure of an XML document, rather than its surface syntax. XPath gets its name from its use of a path notation (as in URLs) for navigating through the hierarchical structure of an XML document. XPath is useful as it provides a common model and syntax to express the patterns required by queries and transformation patterns for XML documents. It is intended primarily as a component that can be used by other specifications and does not define any conformance criteria for independent implementations (of XPath). This is why XQL (compatible with XPath) is used in Kirrkirr instead of a specific XPath implementation.

2.2.4 XQL (Extensible Query Language)

XQL is a set of extensions to the Extensible Style Language (XSL) specification that allows developers using XML to execute powerful, complex queries on XML documents. It has a model that is directly based on the relationships found in XML documents, allowing powerful queries to be simply expressed. Proposed to the W3C by representatives from Microsoft, Texcel and WebMethods in 1998, it competes with the SQL-oriented XML-QL whose specification was submitted to the W3C by AT&T Labs. XML-QL will not be further discussed, as the new data access model of Kirrkirr uses only the XQL API.

Traditionally, structured queries have been used primarily for relational or object-oriented databases, and documents were queried with relatively unstructured full-text queries. Although sophisticated query engines for structured documents have existed for some time, they have not been a mainstream application. XML documents are structured documents – they blur the distinction between data and documents, allowing documents to be treated as data sources, and traditional data sources to be treated as documents. Some XML documents are nothing more than an ASCII representation of data that might traditionally have been stored in a database. Others are documents containing very little structure beyond the use of headers and tables. Kirrkirr is somewhere in between: an e-dictionary that has complex recursive structure, but also much relatively unstructured free text, and clearly needs effective query mechanisms for access.

Database developers have taken for granted the ability to execute queries on data stores for decades. However, XML being a young data technology, querying functionality had been very limited. XQL gives developers the querying functionality they have become used to in the database world, including the following:

- Functionality equivalent to the SQL SELECT statement
- Functionality equivalent to the SQL WHERE statement
- Boolean logic operators (e.g. AND, OR, NOT)
- Comparison operators (e.g. greater than, less than, less than or equal to)
- Wildcard operators (e.g. *)

The major differences between SQL and XQL [Robie1998] are summarised in Table 2. It is clear that XQL is an invaluable tool with huge potential for accessing dictionary information stored in XML format.

SQL	**XQL**
The database is a set of tables.	The database is a set of one or more XML documents.
Queries are done in SQL, a query language that uses tables as a basic model.	Queries are done in XQL, a query language that uses the structure of XML as a basic model.
The FROM clause determines the tables which are examined by the query.	A query is given a set of input nodes from one or more documents, and examines those nodes and their descendants.

The result of a query is a table containing a set of rows.	The result of a query is a set of XML document nodes, which can be wrapped in a root node to create a well-formed XML document.

Table 2: Main differences between SQL and XQL

2.2.4.1 Using XQL in Kirrkirr

The XQL search engine and the PDOM used for the new dictionary representation in Kirrkirr originated from a research project at GMD-IPSI, the Institute for Integrated Publication and Information Systems of the German National Research Centre for Information Technology [Huck1999]. The PDOM implements a persistence version of the Document Object Model mention previously in 2.2.2 using indexed, binary files. The source XML document is parsed once and stored in binary form, accessible to DOM operations without the overhead of parsing them when the information is required. Performance is also boosted by the internal cache architecture. This approach scales very well beyond the limitations of main memory. First, a PDOM is generated from the XML dictionary. Subsequently, Kirrkirr uses XQL to query the PDOM. Parsing of the XML need not be done repeatedly (it is only necessary when the dictionary changes) and access is faster. One advantage of using the XQL engine is that the developer of the current project may be freed of maintenance of the module involving the creation of the PDOM file and the actual query of the DOM to concentrate on improving other aspects of the system.

The following is a simplified XML hierarchy of the dictionary.

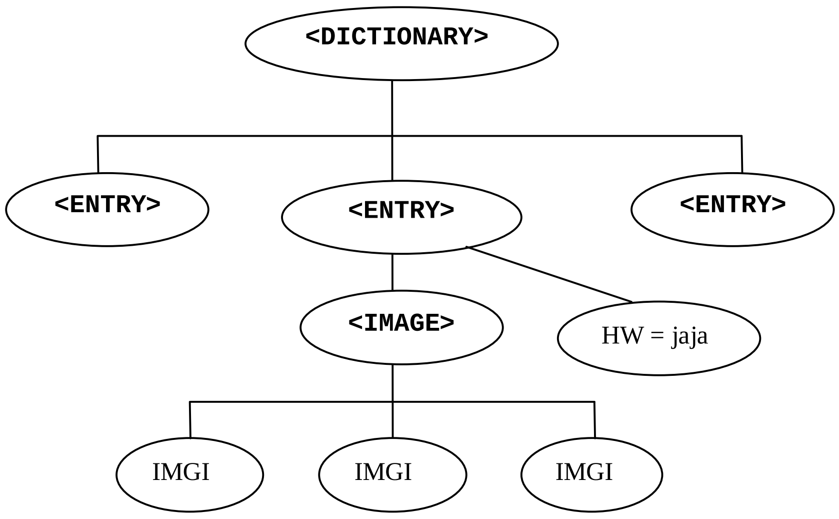


Figure 4: A sample DOM tree

The dictionary is a sequence of many entries, which include some subset of a large number of dictionary components, including a headword (HW element) and perhaps one or more pictures (IMAGE element).

To find an entry whose headword (<HW>) is 'jaja', the following query ([ ] is the filter clause, equivalent to the WHERE clause in SQL) may be used:

```
/DICTIONARY/ENTRY[HW='jaja']
```

Alternatively, if the PDOM index is known, say index = 9 for the word jaja, we can use the query:

```
/DICTIONARY/ENTRY[9]
```

The time taken to execute the above queries is very slow and depends very much on the number of ENTRY nodes in <DICTIONARY>. This is bad news even for the 9300 entries of the current Warlpiri dictionary, and would be totally impractical for something like a large English dictionary, which might have 100,000 or more headwords.

One solution is to split the DOM representation into multiple smaller DOM trees. However, this approach does not scale very well if more words are added to the dictionary in the future and the increase in code complexity is difficult to justify.

Fortunately, implementations of the DOM API provide efficient ways to extract a child node given its index in the parent's node list, and this can be used to execute the above queries more efficiently and extract the required entries. After extracting the required entry from the DOM tree, there may be a need to query the multiple descendant IMGI nodes to extract the filenames of the image files of a dictionary entry (<ENTRY>). This can be very simply achieved by the following query:

```
ENTRY/IMAGE//IMGI
```

The '//' denotes recursive descent and in the above context means finding all the IMGI nodes under the <IMAGE>.

2.2.5 Eliminating Redundancy of Original Data Structure

The new data structure uses a **`DefaultListModel`** implementing the **`TableModel`** interface to store the crucial data needed by program start-up. A sample of the data structure is shown below.

Headword	Polynum	Subword (Boolean)	Frequency	PDOM index
`-ja`	1	False	10	0
`-ja`	2	False	12	1
`jaja`	0	True	165	2
`jaja-na`	0	True	138	3
`jaja-rlangu`	0	True	111	4
`watu`	0	False	77	5

Table 3: A sample example of the new dictionary data structure

This is a more efficient data structure because it

- stores the headwords only once.
- enables sorting to be performed quickly according to frequency, syllabus
- is smaller in size, resulting in a smaller amount of memory required and faster loading time.
- is ready to be used immediately (do not need to extract and add the headwords one by one to the **`DefaultListModel`** used by the **`JList`** as is the case in the original method).

If other information like the definition of the word, the name of the sound file is needed, a look-up to the PDOM will be carried out.

2.2.5.1 Need for a Start-up Index

A timing test is carried out to test the performance of the start-up time. The original program is pitted against two alternatives. The first alternative parses the dictionary information needed into the data structure (described in 2.2.1.1) from the PDOM during start-up, while the second alternative uses a pre-created start-up index for the data structure.

Method	Size of index file	Size of file to be loaded at start-up	Start-up time needed
1. Original Index file method	2.13Mb	2.13Mb	7 min
2.New method (PDOM)	12.5Mb	N.A.	13min 04s
3.New method (PDOM + index)	PDOM - 12.5Mb Index - 520Kb	520Kb	3min 30s

Table 4: Timings justifying the creation of a start-up index

The slowness of method 2 may be attributed to the need to parse the dictionary information from the PDOM at start-up. It is evident that a start-up index created from the PDOM (which effectively halved the start-up time) is required. The next section depicts refinements made to the model to reduce its size, making it more compact and faster to load at start-up.

2.2.5.2 Incremental Load (Refinement 1)

The start-up time can be reduced further by modifying the way the index is created so that part of the dictionary may be loaded and the user interface created and shown to the user while the rest of the dictionary loads in the background.

To achieve this, the dictionary is divided into groups of 500 and the stored in **`ArrayLists`**. At program start-up, a new thread is created to load the **`ArrayLists`** one by one from the persistence storage and presented to the user. Meanwhile, the user interface is being created in the main AWT-EVENT thread and shown to the user when its creation is completed. Concurrency is achieved with the user interface being presented to the user while the rest of the dictionary data loads in the background (see 2.1.6 on Real Speed vs. Perceived Speed).

The size of the index file increases from 520Kb to 544Kb as a result of the overheads involved in the usage of **`ArrayLists`** to group the start-up information. The slight increase in index file size does not cause significant slowing down of start-up.

2.2.5.3 Reduction of Index's size (Refinement 2)

The polynum (polysemy number: number used to label the multiple versions of the same word but with different meaning) is encoded as the number of spaces at the end of a word. This effectively removes the need of a separate column in the model for the polynum. As every dictionary entry is uniquely defined by a word and a polynum, every space-coded word stored in the model is unique. This enables direct looking up of the model (instead of having to match using both the word and its polynum). The usage of the space-coded word can also reduce the number of parameters to be passed between methods, thus reducing the overheads of method calling. The size of the index file is reduced from 544Kb to 454Kb through the removal of the polynum column.

2.3 Speeding up Retrieval of Dictionary Information

The retrieval of the dictionary information (in the form of dictEntry) from the PDOM is slower than getting the data in the form of dictEntry form a **`Hashtable`** (used by the original program). The slight decrease in data retrieval is not obvious when only information for one word is to be retrieved (for example, when a user selects a word from the list), due to the reaction time of the user. Unfortunately, when dictionary information is required for a large number of words (say during a search function), the performance difference becomes significant, due to the accumulation of the extra timing required to retrieve each of the approximately 9300 dictEntries.

Another noteworthy point is that in the original program design, various parts of the program needed and extracted the dictionary information from the **`Hashtable`** from time to time. The original version works fine due to the speed of the **`Hashtable`**. In contrary, this is expensive for the new version, as the PDOM has to be queried repeatedly for the same dictEntry. There is a conflict of interest here: on one hand, we want to avoid loading all the dictEntries (stored in a **`Hashtable`**) on start-up, to reduce the start-up time. On the other hand, we want to achieve quick retrieval of dictEntries and this is not possible with the PDOM implementation.

A solution that realises the speed of retrieval from the **`Hashtable`** and fast start-up time (PDOM implementation) is possible: by adding a "*cache*" (basically a **`HashMap`**, see 3.2.11 for rationale of choice) between the PDOM and the user interface (see Figure 5). All dictEntries retrieved from the PDOM are stored in the cache. Now only the first request for a dictEntry not present in the cache will incur a penalty; subsequent requests for the dictEntry can simply be answered by the cache. This cache, unlike hardware cache does not use any replacement policy (e.g. FIFO, Least Recently Used) to displace dictEntries from it (to maintain the speed of dictEntry extraction). However, if the need arise, the size of the cache can be restricted easily to a suitable number. The memory usage of the program will no doubt increase as more dictEntries are inserted into the cache, but this is in line with the "Lazy Loading principle" practised by commercial programs (for e.g. Netscape Navigator which starts up with bare libraries and proceeds to load in extra plug-in libraries only when necessary).

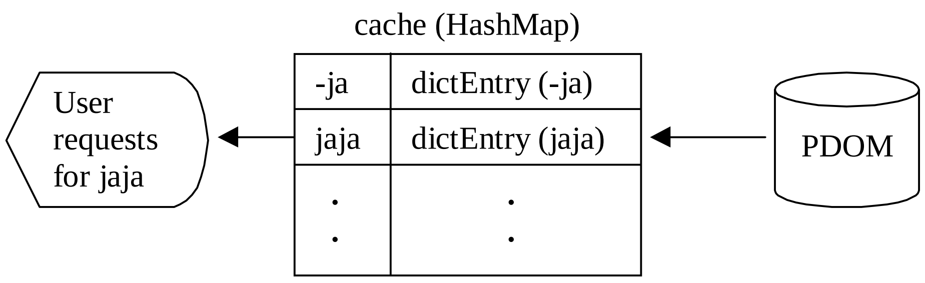


Figure 5: Faster dictionary information retrieving through addition of a cache.

Two schemes for filling the cache was considered. The first one is an aggressive scheme which anticipates the usage of the search function and attempts to load all dictEntries in a background thread to the cache, while the second is a conservative one whereby only dictEntries previously extracted from the PDOM are added to the cache. The first scheme has the advantage of faster first-search performance, but performance degradation (reduced responsive) will result as the loading thread competes with other threads for system resources. Furthermore, the effort will be deemed wasted if the search function is not used. In view of the penalty incurred by the "aggressive cache" and that only the performance of the first-search operation differs between the two schemes, the second scheme is adopted.

2.4 Dictionary Sorting

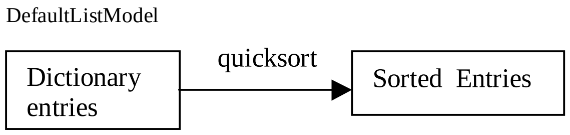


Figure 6: Original method of sorting

The original program makes use of a quicksort algorithm to sort the dictionary entries according to their alphabet, rhythm and frequency (see Figure 6). This method is unstable (entries that are equal may still be sorted in the next round of sorting). This may confuse users when they try to sort the list of dictionary (by the same criteria) repeatedly: they are expecting the list to be already sorted but instead found to their surprise that the list reorders.

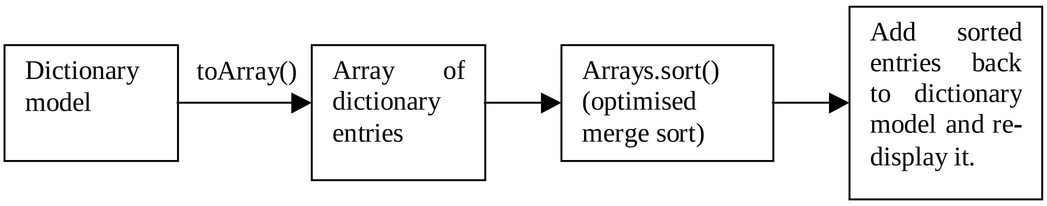

Dictionary model
toArray()
Array of dictionary entries
Arrays.sort() (optimised merge sort)
Add sorted entries back to dictionary model and re-display it.

Figure 7: New method of sorting

The new method of sorting (see Figure 7) uses the sort (Object[] a, Comparator c) from the java.util.Arrays (which is included as part the development kit from JDK 1.2 onwards). The sort operation uses an optimised merge sort algorithm (in which the merge is omitted if the highest element in the low sub-list is less than the lowest element in the high sublist). This method is superior to the original method as it is:

- *Stable*: It does not reorder equal elements. This is desirable if the same list is sorted repeatedly on different attributes. For example, if the user sorts the dictionary by alphabet, and then sorts it by frequency, the user can now expect that the now-contiguous list of words with the same frequency will (still) be sorted by alphabet.

- Fast: The algorithm offers guaranteed O{nlog(n)} performance, and can approach linear performance on nearly sorted lists. Empirical studies showed it to be as fast as a highly optimised quicksort and this is summarised in Table 5 [Wood1993].

Sorting Method	**Worst Case**	**Expected Case**	**Stability**
Merge sort	O{ nlog(n) }	O{ nlog(n) }	yes
Quicksort	O{ n^2 }	O{ nlog(n) }	no

Table 5: Comparison of merge sort against quicksort

Refer to section 5.5 for test result.

Chapter 3: `Other Optimisations

The best way to make the program run and load faster is to code the program in a more efficient manner. This chapter discusses techniques to use and pitfalls to avoid in the quest for a faster and less resource hungry system.

One of the concessions Java programs make to natively compiled programs in other languages is reduced performance. As an interpreted language with a compact bytecode, Java's lack the speed is the most often problem faced by programmers. Unlike most 3GLs (third-generation languages) that provide stable and predictable optimisations that exploit existing software and hardware platforms, most Java compilers do little when it comes to optimising code [Hutcherson2000]. The compiler **`javac`** defaults to doing the following [Bell1997]:

- Constant folding - the compiler resolves any constant expressions such that **`i = (10 *10)`** compiles to **`i = 100`**.
- Branch folding to remove unnecessary **`goto`** bytecodes.
- Limited dead code elimination so that no code is produced for statements like **`if(false) i=1;`**

The level of optimisation **`javac`** (compiler that comes with Sun's JDK) provides should improve as the language matures and compiler vendors begin to compete in earnest on the basis of code generation.

Optimisation in Java can be difficult since the execution environments vary. Code fragments that occur multiple times throughout a program are likely to be size-sensitive, while code with many execution iterations may be speed-sensitive. Using a better algorithm will give a bigger performance increase than any amount of low-level optimisations and is more likely to guarantee improvement under all execution conditions. As such, high-level optimisations should be done before embarking on low-level optimisations.

3.1 General Optimisation Techniques

Some developers have long depended on the compilers' optimisation features to clean up sloppily developed code. Some compilers can hide coding inefficiencies, but none can hide poorly designed code. As such, it is important to exercise good programming practices instead of over-relying on the compiler to do the dirty job.

3.1.1 80/20 Rule

As a rule, 80 percent of a program's execution time is spent executing 20 percent of the code. Optimising the other 80 percent of the program (where 20 percent of the execution time is spent) has no noticeable effect on performance. If the 80 percent of the code can be made to execute twice as fast, the program will only be 5 percent faster [Bell1997]. So the priority task in code optimising is to identify the 20 percent of the program that consumes most of the execution time. Applying this to Kirrkirr, efforts directed at reducing the start-up time (a compulsory process every user has to go through) would result in a definite improvement perceivable by the user.

The following optimisation techniques works independent of the language used, as the techniques basically involve restructuring code and substituting equivalent operations within a method.

3.1.2 Strength reduction

Strength reduction occurs when an operation is replaced by an equivalent operation that executes faster. The most common example of strength reduction is using the shift operator to multiply and divide integers by a power of 2. For example, **`x >> 1`** can be used in place of **`x / 2`**, and **`x << 2`** replaces **`x * 4`**.

Although many modern RISC CPUs have or nearly have eliminated the difference in execution time between multiply and shift. However, divide remains a slow operation on most if not all architectures. So while **`x <<= 1;`** generally has little to no advantage over **`x *= 2`**, **`x >>= 1; on the other hand`** is still an improvement over **`x /= 2`**.

3.1.3 Common sub expression elimination

Common sub expression elimination removes redundant calculations. For example, instead of writing

```
double x = d * (lim / max) * sx;
double y = d * (lim / max) * sy;
```

the common sub expression is calculated once and used for both calculations:

```
double depth = d * (lim / max);
double x = depth * sx;
double y = depth * sy;
```

3.1.4 Code motion

Code motion moves code that performs an operation or calculates an expression whose result does not change (*invariant*). The code is moved so that it only executes when the result changes, rather than executing each time the result is required. This is most common with loops, but it can also involve code repeated on each invocation of a method. The following is an example of invariant code motion in a loop:

```
for (int i = 0; i < x.length; i++)
    x[i] *= Math.PI * Math.cos(y);
```

should be written as

```
double picosy = Math.PI * Math.cos(y);
for (int i = 0; i < x.length; i++)
    x[i] *= picosy;
```

The result of picosy is calculated only once instead of x.length times.

3.1.5 Unrolling loops

Unrolling loops reduces the overhead of loop control code by performing more than one operation each time through the loop, and consequently executing lesser iterations. If we know that the length of **`x[]`** is always a multiple of two, we might rewrite the loop as:

```
double picosy = Math.PI * Math.cos(y);
```

```
for (int i = 0; i < x.length; i += 2) {
    x[i] *= picosy;
    x[i+1] *= picosy;
}
```

In practice, unrolling loops such as this -- in which the value of the loop index is used within the loop and must be separately incremented -- does not yield an appreciable speed increase in interpreted Java because the bytecodes lack instructions to efficiently combine the "+1" into the array index.

3.2 Java Specific Optimisation

3.2.1 Using the compiler

There are just-in-time (JIT) compilers that convert Java bytecodes into native code at run time. Several are already available, and while they can increase the execution speed of the program dramatically, the level of optimisation they are capable of performing is constrained because optimisation occurs at run time. A JIT compiler is more concerned with generating the code quickly than with generating the quickest code. Native code compilers that compile Java directly to native code offers the greatest performance but at the expense of platform independence.

`javac` offers a performance option that can be easily enabled: the invoking of the **`-O`** option will apart from the simple optimisation mentioned previously, inline certain method calls [Bell1997]:

```
javac -O MyClass
```

Inlining a method call inserts the code for the method directly into the code making the method call. This eliminates the overhead of the method call. This overhead can take up quite a significant percentage of its execution time for a small method. Only methods declared as **`private`**, **`static`**, or **`final`** are considered for inlining, because only these methods are statically resolved by the compiler. In addition, **`synchronized`** methods will not be inlined. The compiler will only inline small methods typically consisting of only one or two lines of code.

Unfortunately, the 1.0 versions of the javac compiler have a bug that will generate code that fails the bytecode verifier when the **`-O`** option is used. This has been fixed in JDK 1.1. (The bytecode verifier checks code before it is allowed to run to make sure that it does not violate any Java rules.) It inlines methods that reference class members inaccessible to the calling class. For example, if the following classes are compiled together using the **`-O`** option,

```
class A {
    private static int x = 10;
    public static void getX () { return x; }
}

class B {
    int y = A.getX();
}
```

the call to A.getX() in class B will get inlined in class B as if B had been written as:

```
class B {
    int y = A.x;
}
```

This will cause the generation of bytecodes to access the private A.x variable that will be generated in B's code. This code will execute fine, but since it violates Java's access restrictions, it will be flagged by the verifier with an **`IllegalAccessError`** when the code is executed the first time. The code of applications (as opposed to applets) is not subjected to the bytecode verifier and may therefore ignore the bug altogether. The program is not affected by the bug as it is compiled using javac 1.2.

3.2.2 Looping

Almost all speed problems involve loops, and the loop itself is an easy place to get a quick performance boost. The impact of the loop overhead depends on the number of iterations the loop executes and the amount of work it does during each iteration. Obviously, the more iterations and the less work done per iteration, the greater the benefit with optimising the loop itself. The basic loop that is being discussed is as follows:

```
for (i = 0; i < N; i++) {
    // do something
}
```

The rule is to use a local **`int`** variable for the loop counter. If it is not necessary inside the loop for the loop variable **`i`** to count from **`0`** to **`N-1`**, then the loop can be restructured to use a more efficient. Rather than comparing **`i`** against **`N`** at each iteration, which requires **`N`** to be loaded (assuming **`N`** is not a constant or **`static final`** variable), the loop should be restructured to take advantage of the efficient checks against 0. Comparing against zero is almost always more efficient in any language since the underlying tests are based on < 0,<= 0, == 0, != 0, >= 0 and > 0. To optimise, restructure the loop as:

```
for (i = N; --i >= 0; ) {
    // do something
}
```

By combining the pre-decrement of **`i`** with the comparison against **`0`**, the necessary condition flags for the comparison will already be set, so no explicit comparison is required. Interpreted code does not actually get this free test because the bytecodes used for conditional tests require the condition value to be on the stack. However interpreted code does get a cheaper test because **`i`** is used for the test directly without requiring an explicit comparison. JIT compiler code would almost certainly get the benefit of both the free test and the implicit comparison to zero.

One common use of small loops is to iterate through the elements in an array. If the purpose of the loop simply is to copy elements from one array to another array of the same data type (for example from an array of **`int`** to another array of **`int`** where not data type conversion is required), then use **`System.arraycopy()`**. This is a special-purpose optimised native method for copying array elements. It is by far the fastest way to copy array elements.

If instead of copying, iterating to the end of the array is required, there is an alternative. As all array accesses in Java are bounds checked, the

`ArrayIndexOutOfBoundsException,` which will be thrown when the loop runs past the end of the array can be used to detect the end of an array.

```
try {
    for (i = 0; ; i++)
        array[i] = i;
}
catch (ArrayIndexOutOfBoundsException e) {}
```

The above is not a good style of coding but coding style is frequently compromised by optimisations. For the above code to run faster, the loop must perform enough iteration to cover the cost of throwing the exception. (see the 3.2.8) Because an exception is not cheap to throw, this technique usually would be used only when the loop typically would perform a lot of iterations. In fact, using an exception to terminate a loop should only be used as a last-ditch effort since the results are likely to vary on different platforms.

3.2.3 Using Variable

The performance of a variable is determined by its scope and its type. Local method variables are the fastest to access. There are a couple of reasons for this. One is that there is no reference either to the object instance, in the case of an instance variable, or to the class in the case of a **`static`** variable. There is just an index into the data space. The other reason, for interpreted code at least, is that there are special bytecodes with implicit offsets for accessing the first four local variables and parameters. Variables are assigned to slots in a method according to the following rules:

- If it is an instance method, the first local variable is always the **`this`** reference (**`this`** is implicitly passed as the left-most parameter to an instance method).
- After the **`this`** reference, parameters to the method are assigned to slots, starting with the left-most parameter through the right-most parameter.
- Any remaining slots are filled by local variables in the order that they are declared.
- Each **`double`** or **`long`** occupies two variable slots. The first slot must be within the first four slots to use the special bytecodes for accessing the **`long`** or **`double`** variable.

Under a JIT compiler, some local variables are likely to get a further boost in performance by being kept in a register.

The fastest types of variables are **`int`** and reference variables. For values stored in variables (as opposed to values stored in arrays), **`int`**, **`boolean`**, **`byte`**, **`short`**, and **`char`** variables are all represented as 32-bit values and use the same bytecodes for loading and storing. In arrays, **`boolean`** and **`byte`** values are stored as 8-bit values, while **`short`** and **`char`** values are stored as 16-bit values. **`int`**, **`float`**, and object reference values are always stored as 32-bits each, and **`long`** and **`double`** values are always stored as 64-bit values.

The operations **`byte`**, **`short`**, and **`char`** are performed as **`ints`**, and assigning the results to a variable of the corresponding type requires an explicit cast. In the following, **`(b + c)`** is an **`int`** value and has to be cast to assign to a **`byte`** variable:

```
byte a, b, c;
a = (byte) (b + c);
```

This cast requires an extra bytecode instruction, so there is another penalty for using smaller types.

When using arrays in Java, an array initializer is implemented in the following manner. The array

```
byte[][] myData = {{1, 2}, {3, 4}};
```

is initialised at run time one element at a time and translated literally by the compiler to the following:

```
byte[][] myData = new byte[2][];
myData[0] = new byte[2];
myData[0][0] = 1;
myData[0][1] = 2;
myData[1] = new byte[2];
myData[1][0] = 3;
myData[1][1] = 4;
```

This array initialisation has a couple of ramifications:

- It can bloat the class files to embed data in arrays in the class file. For data of any significant size, it is faster to load the data as a separate element.

- An initialised array as a local variable will execute all this code each time the method is invoked. Therefore the array should be declared as a static or instance member to eliminate the initialisation for each iteration. Even when a fresh copy of the array is needed each time the method is called, it is still faster to store a single initialised copy and make a copy of it for non-trivial arrays.

```
private static int[] data = {1,1,2,3,5,8,13,21,34,55,89};
int[] fibo (int inc) {
    int[] data = (int[])this.data.clone();
    for (int i = data.length; --i >=0; )
        data[i] += inc;
    return data;
}
```

For a multidimensional array, **`System.arraycopy()`** and **`clone()`** both will only do a shallow copy of the outermost dimension. The subarrays will be shared unless they are individually copied or cloned.

3.2.4 Using Method

There are two basic categories of methods: those that can be statically resolved by the compiler and the rest, which must be dynamically resolved at run time. To statically resolve a method, the compiler must be able to determine absolutely which method should be invoked. Methods declared as **`static`**, **`private`**, and **`final`**, as well as all constructors, may be resolved at compile time because the class the method belongs to is known. A statically resolved method executes more quickly than a dynamically resolved method.

The best way to optimise method calls is to eliminate them. The compiler has an option for inlining certain statically resolved methods (as described above in "Putting the compiler to work"). In a critical section of code, calling small, easily inlined methods such as **`Math.min()`**, **`Math.max()`**, and **`Math.abs()`** can speed up those operations dramatically.

The next best option is to convert method and especially interface invocations to **`static`**, **`private`**, or **`final`** calls. If there are methods in the class that do not make use of instance variables or call other instance methods, they can be declared **`static`**. If a method does not need to be overridden, it can be declared **`final`**. Methods that should not be called from outside the class should always be declared **`private`**.

By far the most expensive type of call in Java is to a synchronized method. Not only does the overhead of calling a synchronized method dramatically increase the calling time, especially with a JIT compiler, there is always the potential that the call will be delayed waiting for the monitor for the object. And when the method returns, the monitor must be released, which wastes more time.

There are a few things that can be done to speed up synchronized code. The first line of defence is to eliminate calls to synchronized methods. This is not an area of the program to be overly ambitious in optimising as the problems caused can be difficult to track down. Because of the way a synchronized method is invoked, it is slightly faster to call a synchronized method than to enter a synchronized block. A call to a synchronized method when the monitor is already owned by the thread executes faster than the synchronized method without ownership of the monitor. So for the four combinations:

```
class Test {
    static int x;

    void test () {
        meth1();
        meth2(); // slowest
        synchronized (this) {
            meth1(); // fastest
            meth2();
        }
    }
    synchronized void meth1 () {
  x = 1;
```

```
    }

    void meth2 () {
        synchronized (this) { x = 1; }
    }
}
```

These timings do not include the time taken to enter the synchronized block inside **`test()`**. If there is a heavily synchronized class calling lots of synchronized methods from other synchronized methods, it will be better to have synchronized methods delegate the work to private non-synchronized methods to avoid the overhead of reacquiring the monitor.

```
class X {
    public synchronized void setStuff (int i) {
        doSetStuff();
    }
    private doSetStuff () {
        // setStuff code goes here
    }
    synchronized void doLotsOfStuff () {
        doSetStuff(); // instead of calling setStuff()
        // do more stuff...
    }
}
```

3.2.5 Using Operators

The **`int++`** test has a significantly faster execution than any of the other operators. This is because the only bytecode that directly modifies a variable instead of operating on the stack is the **`iinc`** instruction; this increments a local **`int`** directly with a sign-extended byte (-128 to 127). This bytecode is one big reason why a local **`int`** should always used for a loop counter.

Under an interpreter there is no real difference between writing **`x = x + y;`** vs. **`x += y;`**, because the bytecodes generated are identical when **`x`** is a local variable. However, if **`x`** is changed to an instance variable, array element, or just about any other **`lvalue`**, then **`x += y;`**

becomes the preferred expression. (An **`lvalue`** refers the sub-expression on the left-hand side of an assignment operator.)

3.2.6 Typecasting

Casting object references to refer to different object types in Java (object casts) can get be quite expensive, especially if a lot of vectors or other container classes is used. It turns out that the cost of a cast is high enough that any object cast that cannot be resolved at compile time (an object cast that can be resolved at compile time is an unnecessary cast) takes long enough that it is better to save the cast object in a local variable than to repeat the cast. So instead of writing:

```
boolean equals (Object obj) {
    if (obj instanceof Point)
        return (((Point)obj).x == this.x &&
            ((Point)obj).y == this.y);
    return false;
}
```

write it as:

```
boolean equals (Object obj) {
    if (obj instanceof Point) {
        Point p = (Point)obj;
        return (p.x == this.x && p.y == this.y);
    }
    return false;
}
```

If the cast is to an interface, it is at least twice as bad speedwise as casting to a class. In fact, there is one type of cast that can take much longer to execute. For an object hierarchy like

```
interface X {}
class Super {}
class Sub extends Super implements X {}
```

then the following cast, to an interface implemented by a subclass, takes anywhere from two to three times as long as casting to the subclass.

```
Super cali = new Sub();
X x = (X) cali;
```

The further the separation between the interface and the subclass (that is, the further back in the interface inheritance chain the cast interface is from the implemented interface), the longer the cast takes to resolve.

In addition unnecessary usage of **`instanceof` must also be avoided**. The following cast is resolved by the compiler and produces no runtime code to implement the unnecessary cast.

```
Point q = new Point();
Point p = (Point) q;
```

However,

```
Point p = new Point();
boolean b = (p instanceof Point);
```

can not be resolved by the compiler because **`instanceof`** must return **`false`** if **`p == null`**. Casting data types is simpler and cheaper than casting objects because the type of the data value being cast is known (for example, an **`int`** never is actually a subclass of an **`int`**.) However, since **`int`** is the natural size used by the JVM (**`ints`** are the common data type that all other data types can be directly converted to and from), casting from a **`long`** to a **`short`** requires first casting the **`long`** to an **`int`** and then from the **`int`** to a **`short`**.

3.2.7 Instantiation

Instantiating an object is fairly expensive in terms of CPU cycles. On top of that, discarded objects will need to be garbage collected at some point. The efficiency of garbage collection is difficult to measure. In general, it takes about as long to garbage collect an object as it takes to create one.

Also, the longer the hierarchy chain for the object, the more constructors that must be called. This adds to the instantiation overhead. If there are extra layers in the class hierarchy for increased abstraction, the instantiation time will increase.

The best option is to avoid instantiating objects in tight loops. Where possible, reuse an existing object. The loop

```
for (int i = 0; i < limit; i++) {
    StringBuffer sb = new StringBuffer();
    // do something with sb...
}
```

would be faster if written as

```
StringBuffer sb = new StringBuffer();
for (int i = 0; i < limit; i++) {
    sb.setLength(0);
    // do something with sb...
}
```

One other minor issue with instantiating objects: When an object is instantiated, all of its instance variables automatically are initialised to their default values, which is the value with all bits set to zero for all data types. If the instance variables are explicitly initialised to their default values, this will duplicate the default initialisation, generate additional bytecodes, and make the instantiation take longer. (There are rare cases when the explicit initialisation is needed to reinitialise a variable that was altered from the default value during the super class's constructor.)

3.2.8 Throwing Exception

Throwing exceptions and catching them with try/catch blocks is normally an exceptional circumstance and therefore not typically a performance concern for optimisation. A **`try {}`** statement is "free", i.e. no bytecodes are generated [Bell1997], and unless an exception is thrown, all the **`try {}`** statement adds to the code as far as overhead is the **`goto`** bytecode to skip the **`catch () {}`** statement and one or more entries in the method's exception table. (An exception table keeps track of the beginning and ending bytecode offsets for **`try`** statements and the offsets of the corresponding **`catch`** and **`finally`** statements.) A **`finally`** statement adds a little more overhead as it is implemented as an inline subroutine. When an exception is thrown, a quick check against the exception table for each method in the call chain determines whether or not the exception is handled by the method. Overall, the **`try/catch/finally`** mechanism in the JVM is efficient. When translated to native code, **`try/catch/finally`** can present a challenge to producing optimised code because of issues with register variables, so using these statements may reduce the efficiency of code generated by a JIT compiler.

Instantiating a new exception is not as streamlined as throwing and catching the exception. An exception generates and stores a snapshot of the stack at the time the exception was instantiated. This is where most of the time for instantiating an exception goes. If throwing exceptions regularly is part of the normal operation of the code, a rethink of the design may be necessary. However if there is a need to do this (such as returning a result from several calls deep), and if catching and handling the exceptions is needed, then repeatedly instantiating an exception can be a waste of time. Instead, it is possible to reuse an exception:

```
MyException myexcept = new MyException();
void doit (int val) throws MyException {
    if (val == 0)
        throw myexcept;
}
```

3.2.9 Using threads

There are a number of performance issues related to threads, and they tend to be platform-specific. To start with, each running thread has both a native ("C" stack) and a Java stack, so memory can be an issue. There is an overhead for switching between threads (a context switch), which can vary significantly between platforms. Using threads requires the releasing and acquiring the monitor for an object, so a context switch without the synchronized overhead will be slightly faster. Synchronized access between threads, as discussed in the 3.4, can result in deadlock and delays (due to the overheads for acquiring the monitor).

3.2.10 Using String Buffer

The String concatenation operator + looks innocent but involves a lot of steps: a new StringBuffer is created, the two arguments are added to it with append(), and the final result is converted back with a toString(). This costs both space and time. Hence, when appending more than one String, the StringBuffer should be used instead.

The StringBuffer class still needs to do allocation and recopying, but it maintains a larger array of characters, so copies are less frequent. Each time conversion of a StringBuffer to a String is done a copy is made, so avoid frequent conversions from StringBuffer to String. The only downside to StringBuffer is the fact that it does not use the + operator, and calls to toString() are required every time an actual String object is needed. As discussed, the recommended method of string concatenation:

```
StringBuffer sb;
sb = new StringBuffer("Hello ");
sb.append("World");
```

3.2.11 Using Java Collections

The key difference between the historical collection classes (like Vector and Hashtable) and the new implementations within the Collections Framework is that the new classes are not thread-safe. This was implemented so that the collection will work much faster (without the need for synchronization). For multi-threaded environment, where multiple threads can modify the collection simultaneously, the modifications need to be synchronized. The Collections class provides the ability to wrap existing collections into synchronized ones with another set of six methods:

- Collection synchronizedCollection(Collection collection)
- List synchronizedList(List list)
- Map synchronizedMap(Map map)
- Set synchronizedSet(Set set)
- SortedMap synchronizedSortedMap(SortedMap map)
- SortedSet synchronizedSortedSet(SortedSet set)

Two of the collection classes are worth mentioning. They are the **`ArrayList`** and **`HashMap`**. An **`ArrayList`** is similar to a **`Vector`**: it encapsulates a dynamically reallocated **`Object[]`** array. **`HashMap`** is similar to **`Hashtable`** except that it is unsynchronized and permits nulls values.

The extra overheads of cost incurred by the class **`Vector`** and **`Hashtable`** because of synchronization can be avoided by using the **`ArrayList`** and **`HashMap`** respectively. As the primary focus of the project is to make the application faster, the **`Vector`** and **`Hashtable`** are replaced by **`ArrayList`** and **`HashMap`** respectively in the improved version of the program.

Chapter 4: Other Enhancements

4.1 Using JavaHelp API for the Help System

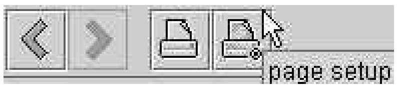
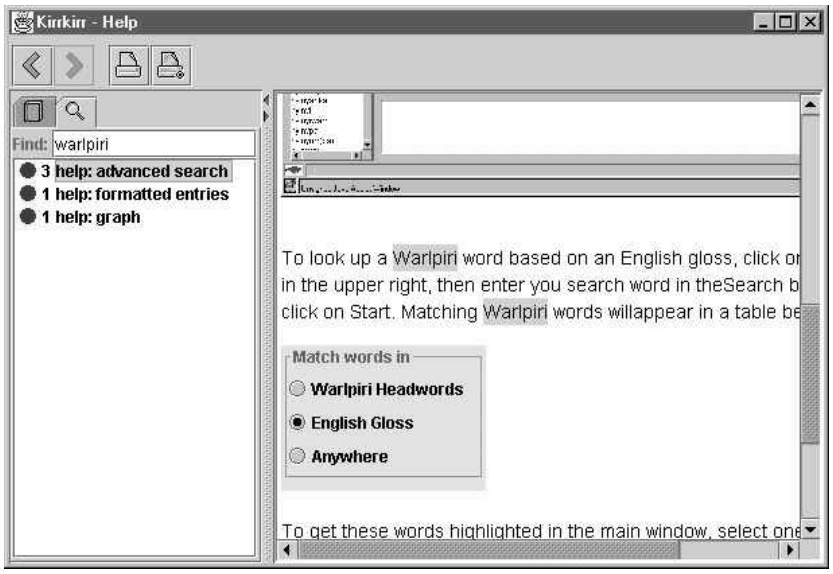


Figure 8: In-built navigation features and printing option of JavaHelp API 1.1



To look up a Warlpiri word based on an English gloss, click o
in the upper right, then enter you search word in theSearch b
click on Start. Matching Warlpiri words willappear in a table be

To get these words highlighted in the main window, select one

Figure 9: Using the search function in JavaHelp API 1.1

The current help system uses a simple **`JEditorPane`** to display the (HTML) help files. Getting to the various help topics is troublesome as the user always has to return to the content page (time wasted in loading the page) in order to do so. The help topics are also limited to those created by the developer (and listed in the content page).

With the new system, the help system is upgraded to use the JavaHelp API to take advantage of its in-built features (see Figure 8). JavaHelp provides a neat way of retrieving the help files according to the chosen help topic and the search ability resulted in an improvement in the usability of the program (see Figure 9). User can now search for help topics using a keyword and is not longer constrained by the limited topics given in the content page. Furthermore, as of JavaHelp version 1.1, the printing out of the help files only requires the simple action of clicking on the print icon in the toolbar of the help window. Maintenance of the files needed for the help system is also enhanced by using easily available authoring tools like HelpBreeze [HelpBreeze2000].

4.2 Installation

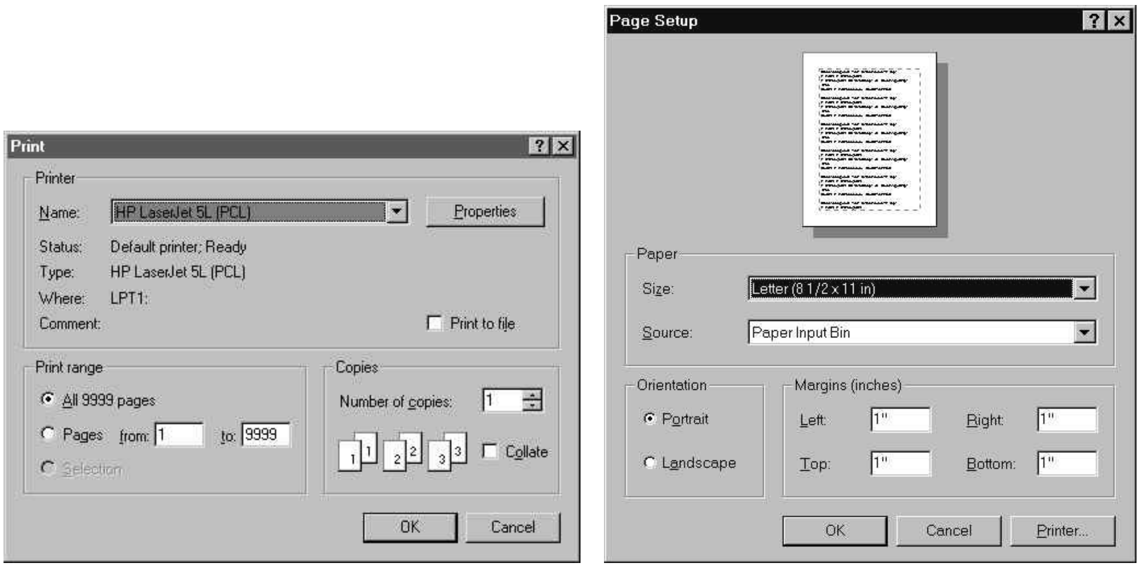
There are around 9304 words in the current implementation of the Warlpiri dictionary thereby dictating the existence of at least the same number of definition files (in the form of HTML files). The presence of the huge amount of HTML files makes the installation process very slow, as the files have to be copied to the target location one by one.

A solution to the above problem is to archive the HTML files using the JAR tool that comes with the JDK). Maintenance of the archive is easy: just use the -u option of the JAR tool to update the archive when necessary. Using the new method, the HTML files can be group together and copied to the destination in one huge chunk, instead of having to open, copy and close the large number of files separately.

There was a problem though: the filenames of the HTML files all began with the '@' character and the jar tool could not detect them. The solution seems clear-cut: remove the starting '@' character. However, there was more to it, the links with the HTML have to be changed too. Due to the large number of files involved, the task was done with a batch program that read in the files one by one, change the filename, parse through the HTML files to update the link. That solved the problem and the HTML files were jar-ed into a single archive.

4.3 Print Function and Print Preview

Presently, to print out the notes taken or the definition of the word, the user needs to highlight the contents and click the ‘copy’ button in the program. Next, he needs to paste the (copied) content into a text editor, before using the user interface of the editor to print. This has a major problem of losing the formatting of the content in the process. Furthermore, there are too many steps involved to achieve printing and the need to copy and paste the content is unintuitive especially to users who are used to printing simply by clicking on a print button.


Print

Printer

Name: HP LaserJet 5L (PCL)

Status: Default printer; Ready

Type: HP LaserJet 5L (PCL)

Where: LPT1:

Comment:

Print to file: ☐

Print range: [✓ All 9999 pages] Pages from: 1 to: 9999 Selection

Copies: Number of copies: 1; Collate: ☐

Page Setup

Paper: Size: Letter (8 1/2 x 11 in); Source: Paper Input Bin

Orientation: [✓ Portrait] Landscape

Margins (inches): Left: 1" Right: 1" Top: 1" Bottom: 1"

Figure 10: Windows 9x Print Dialog & Page Setup

To solve the issue, the print API that is part of the JDK is incorporated into the program. Now, the user can print out his notes and the word definitions by clicking on the print button in the print dialog. In addition, he can also select the paper size using the page setup dialog (see Figure 10).

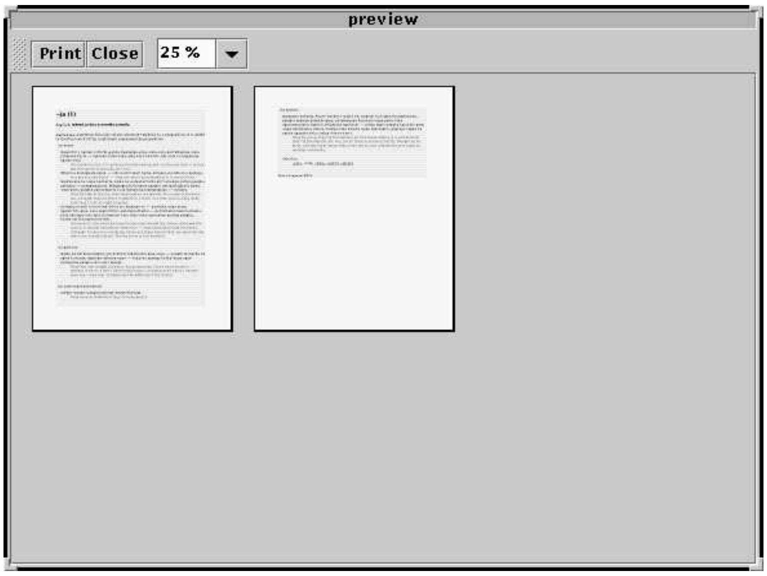


Figure 11: Print preview window in Solaris

Another feature added is the Preview window, which comes with a zoom feature (see Figure 11). This module was modified from a program by Robinson [Robinson2000]. It gives the user an idea of how the actual print out will look like, before commencing the print action. The print preview function is also helpful in debugging print errors: the programmer can detect the errors in the print preview window, rather than having to wait for the contents to be printed out (a more time consuming process).

4.4 Internationization

Internationalization is the process of designing an application so that it can be adapted to various languages and regions without engineering changes to the source code. Sometimes the term internationalization is abbreviated as i18n, because there are 18 letters between the first "i" and the last "n.". In internationalization, textual elements, such as status messages and the GUI component labels, are not hardcoded in the program. Instead they are stored outside the source code and retrieved dynamically. This provides an abstraction of the language definition and allows quick localization of the program. Support for new languages does not require a recompilation of the source code, simply create a new property (text) files containing the language translations and voilà. The new version of Kirrkirr was internationalized and efforts were currently undertaken by linguist to come out with a Warlpiri version of the program. The Warlpiri version will be particularly useful for users who understand only Warlpiri. In addition, it makes the program more customisable as the users is free to choose the type of language he prefers to work in.

Chapter 5: Performance Evaluation

This chapter compares the performance of the new version with original version. The tests in this section were conducted on a Pentium-133 machine with 48 Mbytes of RAM and running JDK 1.2.1. A comparatively slow and old machine was used deliberately because this is more typical of the equipment likely to be available to potential users of Kirrkirr.

5.1 Evaluation of Start-up Time

Method	**Size of index file**	**Size of file to be loaded at start-up**	**Start-up time needed**
XML+ (old) Index	Index file -2.13Mb	2.13Mb	7min
One-PDOM	12.5Mb	N.A.	13min 04s
One PDOM + Index	PDOM - 12.5Mb Index - 520Kb	520Kb	3min 30s
Segmented PDOM + Index	PDOM - 12.5Mb Index file - 454Kb	454Kb	55.48s

Table 6: Comparison of the start-up times for Kirrkirr with different representations.

Table 6 summarises the comparison of the start-up time. XML+(old) index is used in the original Kirrkirr. In one-PDOM, the dictionary information needed is extracted from one monolithic PDOM. The performance of this method is very unsatisfactory. It is evident that a start-up index created from the PDOM is required. This start-up index need not be as complex as the old index – it need only have the bare minimum information (just enough to get the rest by querying the PDOM). The use of such an index reduces the start-up time to half that of the original method. An optimised and enhanced version of this is also considered where dictionary information is broken into blocks. Part of the dictionary can be loaded and the user interface created and presented to the user, while the rest of the dictionary loads in the background. This speeds up the start-up time significantly and allows the user to play with some of the words, while the rest are being loaded in the background (see section 2.2.5.2).

5.2 Evaluation of Creation of Index file

The original method takes around 32min 26s to create the index file. The painstaking process involves parsing the XML tags, retrieving the values and encapsulating the dictionary data in the class dictEntry before storing them in **`CTHashtable`**. Apart from being slow, the index created was far too big and requires too much time to load during start-up (as discussed in section 5.1).

The new dictionary representation requires a smaller index (containing only essential dictionary information) for program start-up. Two steps are involved in creating this new index:

1. Parsing of XML file to get PDOM file.
2. Creation of the index file required at start-up from the PDOM file.

Step	**Time Taken**
XML --> PDOM	3min 2s
PDOM --> Index	10min 52s
Total	13min 54s

Table 7: Time taken for each step of the creation of start-up index (new version).

As shown in Table 7 above there is an improvement of 133% in the time taken to create the index file.

5.3 Dictionary Information Extraction

One of the main justifications for using XQL was speedier access using a standardised mechanism. Here we examine if this is true, and what more needs to be done to fix it.

Method	**Time taken for extract dictionary information for a word**
1. Original method (extracting from **`Hashtable`**)	118 ns
2. Querying the PDOM	3700 ms (first query) 400 ms (subsequent query)

Table 8: Summary of time taken for dictionary information extraction

The result in Table 8 compares the time taken for dictionary information extraction. There are two test results for the second method, this is due to the self-optimizing cache architecture of the XQL engine that ensured faster queries after the first. Extraction of dictionary information by the querying of the PDOM is much slower than getting from the **`Hashtable`**, even with the faster timing (about three orders slower). This issue was alleviated through the addition of a cache (not the cache architecture for the XQL engine) mentioned in section 2.3.

5.4 Evaluation of Memory Requirement

The memory requirement at start-up was obtained using an utility program: TaskInfo2000 [Arsenin2000]. It was found that the new version of Kirrkirr requires 27MB of memory compared to 36MB for the original version: an improvement of 25%.

5.5 Evaluation of Sorting Time

	Time Taken (ms)		**Improvement**
	Original Method (1)	**New method (2)**	**[(1)-(2)]/2 *100%**
Sort by Alphabet	19806.6	4080	385%
Sort by Rhythm	43980	15965	175.48%
Sort by Frequency	32076.6	4065	689.09%

Table 9: Test result for the methods of sorting

The test is performed with the adaptation of the original algorithm in the new version of the dictionary representation. This is to ensure fairness of the test, as the original sorting algorithm in the original program took too long to complete due to the huge amount of memory being used for storing the dictionary information (leaving very little resource for sorting).

The new method uses the sort() method that is part of the Java Core API (see section 2.4) giving rise to faster performance that results from the calling of native methods by the JDK. This is definitely faster than manually sorting the dictionary by methods written in Java. The overheads involved in the new method of sorting may render it to be slower when sorting a small amount of words. However, the main focus here is in making the sorting process faster for a large list (the Warlpiri dictionary consists of ~9300 words). The time taken to sort a smaller list (like a filtered list) is much shorter and therefore does not pose much of a problem.

Chapter 6: Conclusions and Future Work

6.1 Future Work

At present, modifications of the dictionary content have to be done manually to XML document by the administrator. He will also have to rebuild the PDOM and start-up index by running a separate program. Besides being time-consuming, the steps involved are not intuitive and it helps to have a program to guide him along the way. The administrator's program, hereafter will be called CTAdmin. It will be desirable if CTAdmin could provide a user interface to update, insert and delete entries to/from the dictionary, followed by auto-updating the PDOM and index file (without rebuilding). This will greatly simplify the process of dictionary data update, and the process can be turned over to the linguistic department. The CTAdmin provides an abstraction of the steps required to update the dictionary content: the steps involved in the update can be changed without affecting the administrator; simply modify the CTAdmin and the administrator can perform the dictionary maintenance unaffected.

Modern programs like Mcafee VirusScan incorporate an auto-update feature that allows the user to update the data file simply by clicking an Update button. It will be useful to have a similar ‘Update’ button in Kirrkirr to allow the user to download and update the word list (start-up index) and the PDOM from the Internet without having to reinstall the whole program all over again.

To ensure prompt fixing of bugs and continual improvement, it helps to provide a link in the program for users to report bugs and submit suggestions through email to the software developers. This can be achieved by using the JavaMail API, which facilitates development of modules needed for email-ing.

New user unfamiliar with the features of the program might not be able to use the program to its full potential. Therefore, it is recommended that an animated tutorial be created to guide the new user through the commonly performed tasks. It may even be possible to incorporate the tutorial directly into the program using the Java Media Framework API from Sun Microsystems which provides a comprehensive set of capabilities for rich-media handling.

6.2 Conclusions

In this project, the main problems of Kirrkirr specifically slow start-up time and huge memory requirement were addressed. Using XQL to query the information stored in PDOM during runtime, extra information not required at start-up was eliminated from the start-up index, resulting in a smaller index file and a new leaner data structure. To further accelerate the start-up process, the multithreading capability of Java was exploited to load the tab panels and dictionary words (grouped in blocks) in separate threads. These effort results in a very significant reduction of the start-up time from 7 min to 1min. In the process, the memory required at start-up was reduced to 27Mbytes from 36Mbytes and the time taken to create the start-up index file was more than halved. The slow speed of extracting information from the PDOM was also solved by adding a software cache between the PDOM and the query engine.

The speed of sorting the word list was improved by more than 175% after using the faster, stable sorting methods that comes with the Java Collection Framework. The installation process was also enhanced by the archival of the large number of HTML dictionary definition files into a single file. This significantly sped up the process, as only a single copy operation is needed now.

While focusing on performance issues, other aspects of Kirrkirr were not neglected and initiatives had been taken to improve the features and usability of the program. The help system was upgraded to use the JavaHelp API 1.1 to take advantage of the built-in features and ease of use. The usage of JavaHelp also simplified the maintenance of the help system, as cheap and even free visual tools are available to aid the creation of the files needed by the JavaHelp Viewer. Print and print preview modules were also added to reduce the steps needed to obtain a hardcopy of the definition file. Internationalisation of Kirrkirr was also done to enable different languages (for e.g. Warlpiri) to be used for the menus and labels within the program. This is particularly useful in adapting the program for users who are English-illiterate.

In summary, the optimisation efforts undertaken for Kirrkirr have indeed yielded tangible performance improvements and together with the newly added features have resulted in a more enjoyable Kirrkirr experience for its users.

Chapter 7: References

Armstrong, E. 1998, *Collections API Redefines Coding Standards,* in JavaOne Conference 1998.
At: http://www.javaworld.com/javaworld/javaone98/j1-98-collections.html

Arsenin, I. 2000, *TaskInfo2000*, an utility program for resource accounting.
At: http://www.iarsn.com/

Barnea, G. 1999, *Adaptive Java applications -- XML makes it possible*, in JavaWorld (September 1999).
At: http://www.javaworld.com/jw-09-1999/jw-09-xmlmessage_p.html

Bell, D. 1997, *Make Java fast: Optimize! How to get the greatest performance out of your code through low-level optimisations in Java*, in JavaWorld (April 1997).
At: http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html

Bickford, P. 1997, *Interface Design The Art of Developing Easy-to-use Software, Chapter 11 Speed and Feedback*, : Academic Press

Bishop, P. and Warren, N. 1999, *Lazy instantiation - Balancing performance and resource usage*, in JavaWorld (February 1999).
At: http://www.javaworld.com/javaworld/javatips/jw-javatip67.html

Boumphrey, F. 1998, *XML Applications*, Wrox Press Ltd

Chang, D., Harkey, D. 1998, *Client/Server Data Access with Java and XML, Chapter 17: Introduction to XML, Chapter 18: DOM and SAX*, : Wiley Computer Publishing

Christ, R.; Halter, S.; Lynne K.; Meizer, S.; Munroe, S. and Pasch, M. 2000, *SanFrancisco performance: A case study in performance of large-scale Java applications,* in IBM Systems Journal (Vol 39, No 1, 2000 - Java Performance p. 4-20)

Donthy, S. 1999, *Innovative ways to handle events in AWT and JFC*, in JavaWorld (November 1999).
At: http://www.javaworld.com/javaworld/jw-11-1999/jw-11-events_p.html

Eckel, B. 1998, *Thinking in Java, Appendix D: Performance,*.: Prentice Hall PTR.

Hardwick, J. 1998, *Java Optimization*.
At: http://www.cs.cmu.edu/~jch/java/optimization.html

HelpBreeze JavaHelp Edition
At *http://www.solutionsoft.com/*

Huck, G. 1999, *GMD-IPSI XQL Engine Version 1.0.2*, a Java API for XQL and PDOM.
At: http://xml.darmstadt.gmd.de/xql/index.html

Hutcherson, R. 2000, Compiler optimisations, in JavaWorld (March 2000).
At: http://www.javaworld.com/jw-03-2000/jw-03-javaperf_4.html

Jansz, K. 1998, *Intelligent processing, storage and visualisation of dictionary information*, Honours Thesis., Department. of Computer Science, University of Syndey.

Jansz, K.; Manning, C. and Indurkhya, N. 1999, *Kirrkirr: Interactive Visualisation and Multimedia from a Structured Warlpiri Dictionary*, Ausweb 1999

Jansz, K.; Sng, W.J.; Indurkhya, N. and Manning, C. 2000, *Using XSL and XQL for efficient, customised access to dictionary information*, Ausweb 2000

Kessler, P. and Griswold, D. 1997, *High Performance Java: Programming Tips, Techniques and Choices*, in JavaOne Conference 1997.

Kincaid, C.; Phelan, J. and Vianney, D. 1999, *"Heap of trouble" with the wrong heap size*.
At: http://www-4.ibm.com/software/developer/library/tip-heap-size.html

Magee, J. and Kramer, J. 1999, *Concurrency State Models & Java Programs, Chapter 3: Concurrent Execution*,: John Wiley & Sons.

McManis, C. 1996, *Not using garbage collection*, in JavaWorld (September 1996).
At: http://www.javaworld.com/javaworld/jw-09-1996/jw-09-indepth.html

Mitchell, D. and Foote, B. 1999, *Singletons vs. class (un)loading*, in JavaWorld (December 1998).
At: http://www.javaworld.com/javaworld/javatips/jw-javatip52.html

Nguyen, J.; Fraenkel, M.; Redpath, R.; Nguyen, Q. and Singhal, K. 1999, IBM Software Solutions, IBM T.J. Watson Research Center, *Building High-Performance Applications and Servers in Java: An Experiential Study.*
At: http://www.ibm.com/java/education/javahipr.htm*l*

Robie, J. 1998, *The Design of XQL*.
At: http://www.texcel.no/whitepapers/xql-design.html

Robie, J. 1999, *XQL (XML Query Language)*.
At: http://metalab.unc.edu/xql/xql-proposal.html

Robinson, M. and Vorobiev, P. 2000, *Swing, Chapter 22: Printing*, Manning Publications Co.

Sosnoski, D. 1999, *Java performance programming, Part 1: Smart object-management saves the day"*, in JavaWorld (November 1999).
At: http://www.javaworld.com/jw-11-1999/jw-11-performance.html

Sosnoski, D. 1999a, *Java performance programming, Part 2: The cost of Casting*, in JavaWorld (December 1999).
At: http://www.javaworld.com/jw-12-1999/jw-12-performance.html

Venners, B. 1998, *Object finalization and cleanup*, in JavaWorld (June 1998).
At: http://www.javaworld.com/jw-06-1998/jw-06-techniques.html

Venners, B. 1998a, *Design for thread safety*, in JavaWorld (August 1998).
At: http://www.javaworld.com/jw-08-1998/jw-08-techniques_p.html

Wood, D. 1993, *Data Structure, Algorithms and Performance, Chapter 12: Sorting*, Addison-Wesley Publishing Company, Inc.

XPath 1999, *XML Path Language (XPath) Version 1.0, W3C Recommendation 16 November 1999*
At: http://www.w3.org/TR/1999/REC-xpath-19991116.html