ALGORITHMS FOR REDUCED CONTENT DOCUMENT SYNCHRONIZATION

By

AJAY I. S. KANG

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002

I dedicate this work to my parents, Harjeet Singh and Kulwant Kaur Kang. Without their constant encouragement and support over the years, I would not be here documenting this thesis.

ACKNOWLEDGMENTS

I cannot adequately express the gratitude and appreciation I owe to my family, friends and faculty advisors in this space.

My family has always been supportive of all my endeavors and prodded me to push farther and achieve all that I am capable of. However, they have never in that process made me feel that I let them down when I failed to achieve their expectations. I know that I will always have their unconditional love and support as I go through life.

I have many friends I would like to thank, especially the “fourth floor crowd” of the CISE department where I spent most of my time working on my research. Hai Nguyen, Andrew Lomonosov and Pompi Diplan would constanly stop by my office providing welcome interruptions from the monotony of work. I am indebted to Mike Lanham for his encouragement to pursue this area of research. A special mention must also be made for Starbucks Coffee which the aforementioned are addicted to and have consumed many gallons of the overpriced beverage in their time here in (according to them) an effort to bolster the national economy.

I thank my advisors, Dr Abdelsalam “Sumi” Helal and Dr Joachim Hammer, whose patience, encouragement, insight and guidance were invaluable in the sucessful completion of this work.

Finally, I would like to thank the National Science Foundation for funding the UbiData project, which resulted in my receiving a research assistantship this summer.

TABLE OF CONTENTS

page

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

ALGORITHMS FOR REDUCED CONTENT DOCUMENT SYNCHRONIZATION

By

Ajay I. S. Kang

December 2002

Chair: Abdelsalam "Sumi" Helal
Major Department: Computer and Information Science and Engineering

Existing research has focused on alleviating the problems associated with wireless connectivity, viz. lower bandwidth and higher error rates, by the use of proxies which transcode data to be sent to the mobile device depending on the state of the connection between the proxy and the mobile device. However, there is a need for a system to allow automatic reintegration of changes made to such reduced content information into the original content rich information. This also allows users the flexibility of using virtually any device to edit information by converting the information to a lesser content rich format suitable for editing on that device and then automatically reintegrating changes into the original.

Hence, these algorithms allow the development of an application transparent solution to provide ubiquitous data access and enable reducing download time and cache usage on the mobile device for editable documents. They can also be applied to

progressive synchronization of a full fidelity document with the master document on the server.

To develop flexible, generic, synchronization algorithms, we need an application-independent specification of the hierarchical content of data, a generic method of specifying what information is shared between reduced content documents and the originals and a change detection algorithm. This algorithm should produce a script that can be applied to the original to bring it up to date and also allow for version control.

Existing difference detection algorithms do not operate on reduced content documents and the approaches used need to be adapted to this domain. XML has been chosen to represent document content since it allows encoding of any hierarchical information and most applications are now moving towards native XML file formats. The change detection algorithm makes extensive use of the *O(ND)* difference algorithm developed by Eugene W. Myers for matching shared content and approximate substring matching.

Extensive performance analysis has been performed on the system by constructing a test engine to automatically generate modifications of varying degree on source documents and thus produce a set of modified reference documents. These reference documents are converted to text (hence stripping out all formatting content), and the algorithm is run to generate delta scripts that are applied to the original documents to generate the new versions. Comparison of the new versions to the references allows us to quantify error.

CHAPTER 1
INTRODUCTION

With the emergence of smaller and ever more powerful computing and communication devices providing a wide range of connectivity options, computers are playing an ever-increasing role in peoples lives. Portable computers such as laptops, tablet PCs, handheld devices based on the Palm, Pocket PC, embedded Linux platforms and to an extent, smart phones coupled with the proliferation of wireless connectivity options are bringing the vision of ubiquitous computing into reality.

However, the potential provided by this existing hardware has as yet largely gone untapped due to the lack of a software infrastructure to allow users to make effective use of it. Users instead have to deal with the added complexity of manually synchronizing their data across all the devices that they use and handle the issues of file format conversion and integration of their changes made to the same document in different formats. These differences exist due to the constraints on mobile devices of being lightweight, small and having a good battery life, which restricts the memory, processing power and display size available on such devices [1]. Hence, there will always be devices for which applications will not be available since the device's hardware or display capabilities fall short. These devices cannot be used as terminals or thin clients (which would alleviate the problem to an extent) since lower bandwidth, higher error rates and frequent disconnections characterize wireless connectivity.

Most research in the mobile computing area has thus been focused on the minimization of the impact of this form of connectivity on the user experience. The most

common approach used is to use a proxy server to transcode the data [2-5] to be sent to the mobile device based on current connectivity conditions, thus ensuring that latency times in data reception as perceived by the user stay within certain limits at the cost of displaying data with reduced content. If the user were to edit such content, the burden of reintegrating the changes with the original copy on the server would fall on him/her. Hence, there is a definite need for a system to perform this change detection and re-integration automatically without user intervention. Solving this problem would also result in a solution to the first problem outlined, i.e., the unavailability of certain applications on a mobile device by having a "smart proxy" deduce that the device could not support such content and reduce the complexity to allow it to be convertible or directly compatible with applications on the mobile device. Any changes made would subsequently be re-integrated using the change detection system.

In conjunction with existing research on systems that automatically hoard the files a user uses on the various devices he owns for disconnected operation support such as the Coda file system developed at Carnegie Mellon University [6-7], we now have the ability to provide the user access to his data at any time, anywhere and on almost any device.

To allow the development of a change detection system that works on hierarchical data in any format, we need an intermediate data representation form that can be used to represent hierarchical data in any format. The intermediate form is what the change detection system will use as its input and output formats which are by definition easily convertible to the required data representation. The Extensible Markup Language (XML) [8] was chosen as the intermediate data representation since XML is an open standard format designed for this purpose and has a wide variety of parsing tools available for it.

This work is being explored as a component to be used in the ubiquitous data access project [9] under development at the Computer and Information Science and Engineering Department of the University of Florida. The goals of the UbiData system are to provide any time, anywhere access to a user's data irrespective of the device being used by him/her.

To get a better feel of the potential of such a system we explore two use cases in the following paragraphs.

Use Cases

Use case 1: overcoming device/application limitations. Joe is working on a document that he intends to show his advisor at a meeting across campus but he is not quite done yet. He powers up his Palm PDA that connects to the local area network through a wireless interface card. The UbiData system recognizes this as one of Joe's devices and fetches the device's capability list from the UbiData server. The system then proceeds to hoard his data onto the device, including the word document that he was just working on. However, a comparison with the capability list shows that the Palm does not support editing of MS Word documents. UbiData uses a converter plugin to convert the MS Word document to text/Aportisdoc format that is editable on the Palm and then transfers it to the Palm as he walks out of the door. By the time he reaches the campus lawn where wireless connectivity is unavailable, the document is sitting on his Palm. He makes edits to the document as he walks across campus. When he reaches his destination, he requests UbiData to synchronize his updates and the text file is compared to the original document and the changes are imported. By the time he reaches his advisor's door, the document is up to date.

Use case 2: overcoming bandwidth limitations. Jane has to go on a business trip to Paris and has a long flight ahead of her. She wants to use this time to edit some project proposals and cost estimates. She requests UbiData to hoard the files she needs to her laptop. At the airport she realizes that she needs a file that she has not accessed recently and hence has not been hoarded to her laptop. She uses her cell phone to connect to and authenticate with the corporate network. She requests the file download and the smart proxy on her corporate network transcodes the file data by removing the images from the file for fast transfer over the weak connection. She edits the file on the plane and uses the business lounge at her hotel in Paris to upload the changes. The UbiData system recognizes that this is a reduced content document (since it was notified by the proxy that Jane's copy was content reduced) and proceeds to detect and integrate the changes with the original. After her meeting is over, she decides to walk around town and sitting at a café has an idea to incorporate into a proposal. She takes out her PDA and uses her cell phone to connect to and authenticate with her corporate network. Since she is now making an international call, she requests a fast download and the proxy converts the document down to compressed text for minimum transfer size. When she returns to the hotel she reintegrates her changes with the content rich document on the server.

Summary of the Advantages of Change Detection in Reduced Content Information

- A change detection system as outlined above would allow the development of an application transparent system for ubiquitous data access with no need for any support software at the client end for change detection as against other solutions such as the CoFi architecture used in the Puppeteer system developed at Rice University [10-12].

- The Change detection system could be used to hoard reduced content documents on a mobile device instead of full fidelity documents. This would speed up download times and reduce cache space utilization, which could also allow hoarding of a larger working set. Transcoding proxies could also transcode editable content since it would always be possible to reintegrate changes into the original document.

- On the other hand, if the mobile device has a full fidelity version of a document that has been changed, and a high bandwidth connection is unavailable, the document could be "distilled" by removing content such as images. The changes could then be reintegrated with the document on the server allowing other users to see updates earlier. Once a high bandwidth connection is available, the entire document could then be uploaded for incorporation of changes to content that was stripped out. Hence, progressive updates of documents are made possible.

Approach Used to Investigate the Design and Implementation of Reduced Content Change Detection and Reintegration

Since such a system is quite complex, it is easier to focus on a restricted case and apply the approaches developed to the general problem. However, this problem should be carefully chosen since the solution may not be applicable to the overall problem. Hence, the extreme case in document editing was chosen for solution, viz. conversion of a content rich document to text and detecting and reintegrating the changes. To provide flexibility, a diff/patch solution was chosen which would also allow only the differences to be sent to another client requesting the same file in the UbiData network. This would result in significant bandwidth savings. To enable version control, a complete diff format was chosen which has enough information available to revert to a previous version of the file. To ease development, an existing XML diff/patch tool was modified. The tool is part of the XyDiff program suite developed by the VERSO team, from INRIA, Rocquencourt, France, for their Xyleme project [13-14].

The following chapters describe the UbiData system, related work and the algorithms developed to solve the problem of change detection in reduced content XML documents as well as test results obtained.

CHAPTER 2
RELATED RESEARCH

The Ubiquitous Data Access Project at the University of Florida

System Goals

1. Anytime, anywhere access to data, regardless of whether the user is connected, weakly connected via a high latency, low bandwidth network, or completely disconnected;

2. Device-independent access to data, where the user is allowed to use and switch among different portables and PDAs, even while mobile;

3. Support for mobile access to heterogeneous data sources such as files belonging to different file systems and/or resource managers.

System Architecture

The architecture for the UbiData system [15] divides the data it manages into three tiers as follows:

1. The *data sources* which could consist of file systems, database servers, workflow engines, e-mail servers, web servers or other sources.

2. *Data and meta-data collected for each user of the system*. The data is the user's working set consisting of files used by the user. Meta-data collected on a per user basis includes the configuration and capabilities of all the devices used by the user.

3. The *mobile caches* that contain a subset of the user's working set to allow disconnected operation on the various devices owned by the user.

The middle tier (storing the data and meta-data), acts as a mobility aware proxy, filling the client's caches when connected and updating the user's working set information. On reconnection after disconnection, it handles reintegration of updates made to documents on the mobile cache.

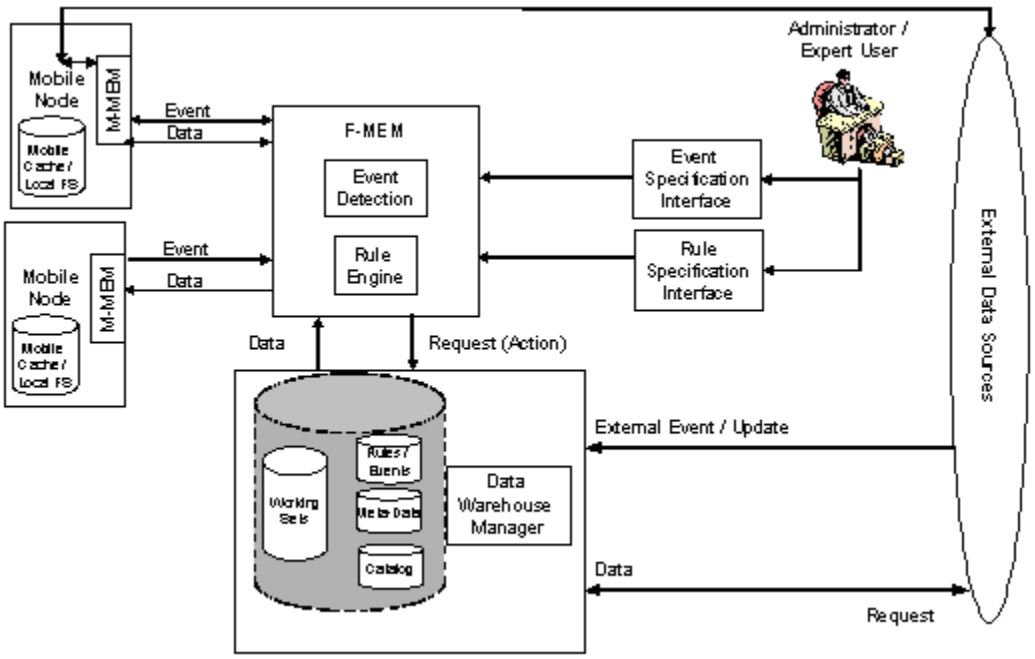

Figure 1. Architecture of the UbiData system

The three tiers are easily identifiable in figure 1; the clients on the left contain the mobile caches on their local file systems. The clients also act as data sources since they can publish data to the system. The other data sources (labeled external data sources) are to the right of the figure. The lower middle part of the diagram depicts the data and metadata collected for each user. The system modules that implement this "three tier data architecture" [9] are detailed below:

Mobile environment managers (M-MEMs)

The functions of the M-MEMs are as follows:

1. To observe the user's file access patterns and discard uninteresting events e.g. operating system file accesses and filter the remaining to determine the user's working set. These are the files that the user needs on the devices he uses to continue working in spite of disconnections.

2. Manage files hoarded to the device and track events such as updates to the file so it can be marked as a candidate for synchronization on reconnection. Also, on reconnection, if the master copy in the middle tier is updated, the M-MEM discards the stale local copy and fetches the updated master copy from the middle tier server (MDSS).

3. Bandwidth conservation by always shipping differences between the old version and the updated version of a file instead of the new file whenever possible. Other activities such as queue optimization of the messages to be sent to the middle tier server also help in this regard.

Mobile data service server (MDSS)

The MDSS consists of two components:

Fixed environment manager (F-MEM)

The F-MEM server side component is a stateless data and meta-data information service to mobile devices. The basic functionality of the F-MEM is to accept the registration of events, detect events, and to trigger the proper event-condition-action (ECA) rules. The main tasks of the F-MEM are to manage the working set and the meta-data in conjunction with the M-MEM component. As a result, the F-MEM needs to perform additional services such as authentication of users and implement a communication protocol between the M-MEM and itself. This protocol should be very resilient to poor connectivity and frequent disconnections, which implies the use of a message queuing system.

Data and meta-data warehouse

The meta-data warehouse is an XML database that stores information related to each user such as the uniform resource identifiers (URIs) for all files in his/her working set and device types and capabilities for all devices used by the user. The data (working set) is stored on the local file system of the MDSS server and accessed through URIs as mentioned before.

The Role of Reduced Content Change Detection in this Architecture

The goal of providing "device-independent access to data" would be unattainable without the presence of such a system. This is due the fact that mobile devices vary widely in terms of hardware capabilities, the operating systems they run and the applications they support. Hoarding files without considering the target environment would result in the user having a number of unusable files since there is no application available on his/her device to handle them. Hence the F-MEM needs to access the device profile and suitably transcode the user's working set before shipping it to the client device. If the transcoded content is editable, changes made to this content must be reintegrated into the original copy on the server. Simply replacing the old file by a newer copy, as done in existing systems of this type e.g. the Coda file system developed at Carnegie Mellon University, would result in lost information loss since transcoding involves loss of information.

Now that it is established that this is a critical component of UbiData, the question that remains is how this system will be integrated into the overall UbiData architecture. Figure 2 shows how the system could be integrated into the UbiData architecture. The sequence of actions for three events has been shown and they are described below:

1. **Mobile host publishing a document**. The actions for this event are shown with solid black arrows. The mobile host transfers the document to F-MEM which on decoding the message recognizes it as a publish request. The F-MEM determines if the document is already in XML format. If not, the appropriate conversion plugin is loaded and the converted document is stored in the repository as part of the user's working set.

2. **Mobile host importing a document**. The actions for this event are shown with solid gray arrows. When an import request is received, the required document format is determined. The connection state is also sampled from the connection monitor. If the connection is weak, an appropriate transcoding scheme is selected. Once the type of conversion is determined, the appropriate plugin is loaded and the conversion performed by loading the latest version of the document from the

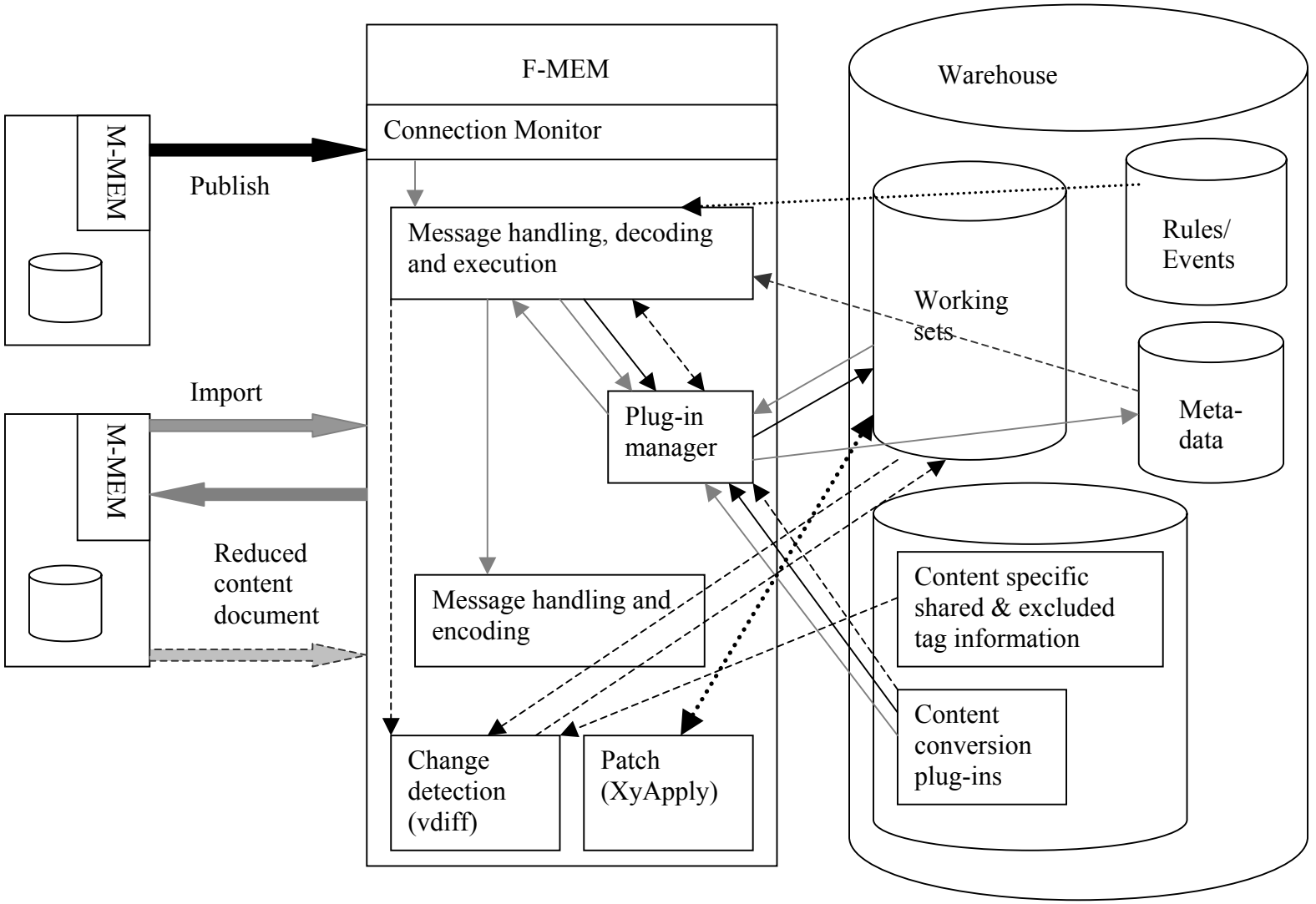

Figure 2. Proposed integration of change detection and synchronization with UbiData

repository and running the plugin to convert it. An entry is made in the meta-data database to mark the client as having a reduced content version of the document. Finally, the document is packaged for transport and shipped to the mobile client.

3. **Synchronization of the possibly reduced content document**. The actions for this event are shown with dashed arrows. Once the message is decoded and recognized as a synchronization request, the meta-data content is checked to determine if this was a reduced content document. If so, the appropriate XML conversion plugin is loaded if necessary, and the document is made available to the vdiff change detection module. The change detection module also uses the application specific shared tag information and the original document from the repository. This information is used to generate a delta script to update the document that is stored in the repository for versioning purposes. The XyApply module is then used to apply the script to the original document in the repository to generate the new version. This new version is stored in place of the original if it was not the first version of the document. Otherwise, the latest version is stored separately. Hence, the repository maintains the original version of the document and the latest version. The stored deltas allow any version in between to be generated from the original document. The latest version is kept for efficiency.

Previous Work on Change Detection for UbiData

Refer to [16] for previous work on change detection and reintegration. This work approaches the same problem in a slightly different way and also extends the previous effort. The previous system was unable to handle deeper and more complex document structures. Updates were often missed and treated as inserts. This has been addressed in this work and the approach to testing the system has been improved making evaluating document correctness more objective.

Research on Disconnected Operation Transparency: the CODA File System

Coda is a location transparent, distributed file system developed at Carnegie Mellon University, which uses server replication and disconnected operation to provide resiliency to server and network failures. Server replication involves storing copies of a file at multiple servers. The other mechanism, disconnected operation, is a mode of execution in which a caching site temporarily assumes the role of a replication site. Coda uses an application transparent approach to disconnected operation. Each client runs a

process called Venus that handles remote file system requests using the local hard disk as a file cache. Once a file or directory is cached, the client receives a callback from the server. The callback is an agreement between the client and the server that the server will notify the client before any other client modifies the cached information. This allows cache consistency to be maintained. Coda uses optimistic replication where write conflicts can occur if different clients modify the same cached file. However, these operations are deemed to be rare and hence acceptable. Cache management by Venus is done according to the state diagram in Figure 3.

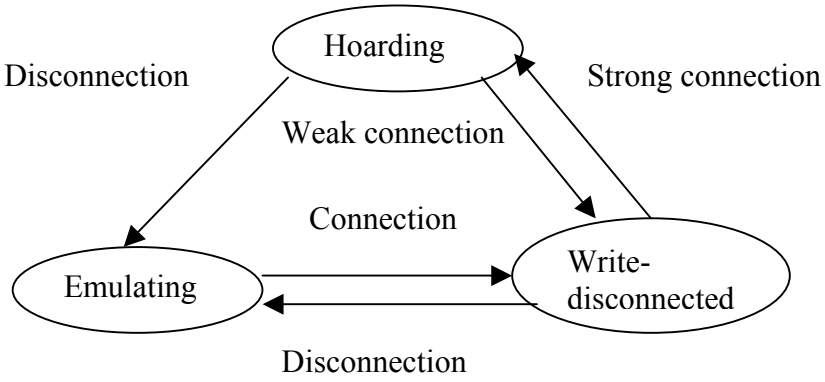

Figure 3. Venus states

Venus is normally in the hoarding state but on the alert for possible disconnection. During the hoarding state, Venus caches all objects required during disconnected operation. While disconnected, Venus services all file system requests from its local cache. All changes are written to a Change-Modify-Log (CML) implemented on top of a transactional facility called recoverable virtual memory (RVM). Upon reintegration, the system ensures detection of conflicting updates and provides mechanisms to help users recover from such situations. Coda resolves conflicts at the file level. However, if an application specific resolver (ASR) is available, it is invoked to rectify the problem. An ASR is a program that has the application specific knowledge to detect genuine

inconsistencies from reconcilable differences. In a weakly connected state, Venus propagates updates to the servers in the background.

The Coda file system has the drawback of requiring its own file system to be present on the clients and servers, which UbiData does not have. UbiData uses the native file system of its host. Hence, bandwidth adaptation and reduced content synchronization is being incorporated into UbiData, due to its greater flexibility in potentially supporting a much wider variety of devices on which such a system can be tested.

Research on Bandwidth Adaptation

Odyssey

Odyssey [17-19] uses the application aware approach to bandwidth adaptation. The Odyssey architecture is a generalization of the Coda architecture. Data is stored in a shared, hierarchical name space made up of typed volumes called Tomes. There are three Odyssey components at each client as follows:

1. The Odyssey API: The Odyssey API supports two interface classes, viz. resource negotiation and change notification class and the dynamic sets class. The former allows an application to negotiate and register a window of tolerance with Odyssey for a particular resource. When the availability of the resource falls out of those bounds, Odyssey notifies the application. The application is then free to adapt itself accordingly and possibly renegotiate a window of tolerance.

2. The Viceroy and the Wardens: These are a generalization of Coda's cache manager, Venus. The Viceroy is responsible for monitoring external events and the levels of generic resources. It notifies clients who have placed requests on these resources. It is also responsible for type independent cache management functions. The Wardens, on the other hand, manage type specific functionality. Each Warden manages one Tome type (called a Codex). Each Warden also implements caching and naming support and manages resources specific to its Codex.

The Odyssey architecture, due to its application aware adaptation design, is not as universally useable as an application transparent approach and hence, unsuitable for ubiquitous data access. It must be noted however, that applications written for such

architectures would have superior performance, as it is easier and faster for an application to adapt to a poor connection than a generic transcoder that would more likely have higher latency and would be far more complex.

Puppeteer

The Puppeteer architecture is based around the following philosophy:

1. Adaptation should be centralized.
2. Applications should not be written to perform adaptation. Instead, they should make visible their Document Object Model (DOM): the hierarchy of documents, containing pages, slides, images and text.

The CoFi architecture [11] standing for consistency and fidelity, implemented in Puppeteer supports document editing and collaborative work on bandwidth-limited devices. This is precisely the goal that the reduced content change detection and synchronization algorithms are being designed for. However, CoFi takes a different approach to solving the problem and this can actually be used to complement the algorithms being developed. CoFi supports editing of partially loaded documents by decomposing documents into their component structures and keeping track of the changes made to each component by the user and those resulting from adaptation. Hence, in cases where the target applications meet the puppeteer requirements of exposing a DOM interface, this approach would be more efficient since only the modified subset of components needs to be dealt with. However, when this is not the case, the hierarchical change detection and synchronization system can still be used. Another requirement for Puppeteer to be used is that a Puppeteer client needs to be present on the target device to perform the function of tracking changes. This may pose a significant overhead for some mobile devices (these devices tend to be resource poor in general), in which case this work can still be used.

Transcoding Proxies: Pythia, MOWSER and Mowgli

Each of these systems performs active transcoding of data on a proxy between the data source and the mobile device to adapt to the varying state of the connection between the mobile device and the proxy. The work done on these systems can be applied to the proposed UbiData integration architecture described earlier.

Congestion Manager

Congestion Manager (CM) is an operating system module developed at MIT [20] that provides integrated flow management and exports a convenient programming interface that allows applications to be notified of, and adapt to changing network conditions. This can be used to implement the connection monitor subsystem in the proposed UbiData integration architecture described earlier.

Intelligent Collaboration Transparency

Intelligent collaboration transparency is an approach developed at Texas A&M University [21] for sharing single user editors by capturing user input events. These events are then reduced to high-level descriptions, semantically translated into corresponding instructions of different editors, and then replayed at different sites. Although the research is not aimed at bandwidth adaptation, it can be adapted for this purpose by capturing all user input events on the mobile device and then converting them to higher level descriptions before optimizing this log for transmission to the server. This was explored as a means of reducing transmission overhead between the mobile device and the server in the UbiData architecture and the work done and results obtained are described in chapter 5.

Research on Difference Algorithms

LaDiff

LaDiff [22] uses an $O(ne + e^2)$ algorithm where n is the number of leaves and e is the weighted edit distance, for finding the minimum cost edit distance between ordered trees. However, there is no support for reduced content change detection. The algorithm functions in two phases, viz. finding a good matching between the nodes of the two trees and finding a minimum cost edit script for the two trees given a computed matching. The matching phase uses a compare function, which given nodes s_1 and s_2 as input, returns a number in the range [0,2] that represents the distance between the two nodes. This is similar to the technique used to generate matches in our reduced content change detection system. The edit script computation operations are similar to the technique used by XyDiff and that is the system that has been followed for edit script construction in the reduced content change detection algorithm.

XyDiff

XyDiff is a difference detection tool for XML documents designed for the Xyleme project at INRIA, Rocquencourt, France. The Xyleme project proposes the building of a worldwide XML warehouse capable of tracking changes made to all XML documents for query and notification. The XyDiff tool had is being used to investigate reduced content change detection since it employs an efficient algorithm to track changes in XML documents and its source is available for modification. The companion tool to XyDiff, XyApply, applies the deltas generated by XyDiff to XML documents. This tool has been used as is for testing the algorithms developed since the XyDiff delta structure has not been modified in any way but the delta construction process has been completely rewritten. Xyleme deltas also have the advantage of being complete, i.e. they have

enough additional information to allow them to be reversed. Hence, they meet the goal of allowing version reconstruction. To assist in versioning, all XML nodes are assigned a persistent identifier, called an XID, which stands for Xyleme ID. XIDs are stored in an XID map that is a storage efficient way of attaching persistent IDs to every node in the tree. The basic algorithm used by XyDiff is as follows:

1. Parse the v0 (original) document and read in the XIDs if available.
2. Parse the v1 (modified) document.
3. Register the source document, i.e. assign IDs to each node in a postorder traversal of the document. Construct a lookup table for ID attributes (unique identifiers assigned to nodes by the creating application for the purpose of easing change detection), if present.
4. Construct a subtree lookup table for finding nodes at a certain level from any parent node for the v0 document.
5. Register the v1 document.
6. Perform bottom up matching.
7. Perform top down matching.
8. Peephole optimization.
9. Mark old tree for deletes.
10. Mark new tree for inserts.
11. Save the v1 document's XID map.
12. Compute weak move operations, i.e. a node having the same parent in both documents but different relative positions.
13. Detect updates.
14. Write the delta to file.
15. All steps before step 8 have been altered for supporting reduced content difference detection. The remaining steps concerned with the construction of the delta have been left relatively untouched.

Xmerge

Xmerge is an OpenOffice.org open source project [23] for document editing on small devices. The goals of the document editing on small devices project are as follows:

1. To allow editing of rich format documents on small devices, using 3rd party applications native to the device.
2. Support the most widely used small devices, namely: the Palm and PocketPC.
3. Provide the ability to merge edits made in the small device's lossy format back into the original richer format document, maintaining its original style.
4. Take advantage of the open and well-defined OpenOffice.org XML document format.
5. Provide a framework with the ability to have plugin-able convert, diff and merge implementations. It should be possible to determine a converter's capabilities at run-time.

The goals of the Xmerge group are virtually identical to what the reduced content difference detection system is trying to achieve. However, the approach used by the Xmerge group is to have format-specific diff and merge implementations, unlike the format independent approach selected for reduced content synchronization. This allows our reduced content synchronization system to be simpler and have lesser space requirements. This would enable it to be installed on a reasonably powerful small device. However, Xmerge's plugin management system can be utilized to implement the plugin manager subsystem for the proposed UbiData integration architecture described above.

An O(ND) Difference Algorithm

The approach used in this difference algorithm [24] is to equate the problems of finding the longest common subsequence of two sequences A and B, and a shortest edit script for transforming A into B, to finding a shortest path in an edit graph. Here, the N and D refer to the sum of the lengths of A and B and the size of the minimum edit script

for A and B respectively. Hence, this algorithm performs well when differences are small, and thus has been chosen as the approximate string-matching algorithm since the application domain for the system is for user edits on mobile devices. These edits are not likely to be very extensive and hence, for this application domain, the algorithm should perform well.

The algorithm uses the concept of an "edit graph", an example of which is shown in figure 4. Let A and B be two sequences of length N and M respectively. The edit graph for A and B has a vertex at each point in the grid (x,y), $x \in [0,N]$ and $y \in [0,M]$. The points at which $A_x=B_y$ are called match points. Corresponding to every match point is a diagonal edge in the graph. The edit graph is a directed acyclic graph.

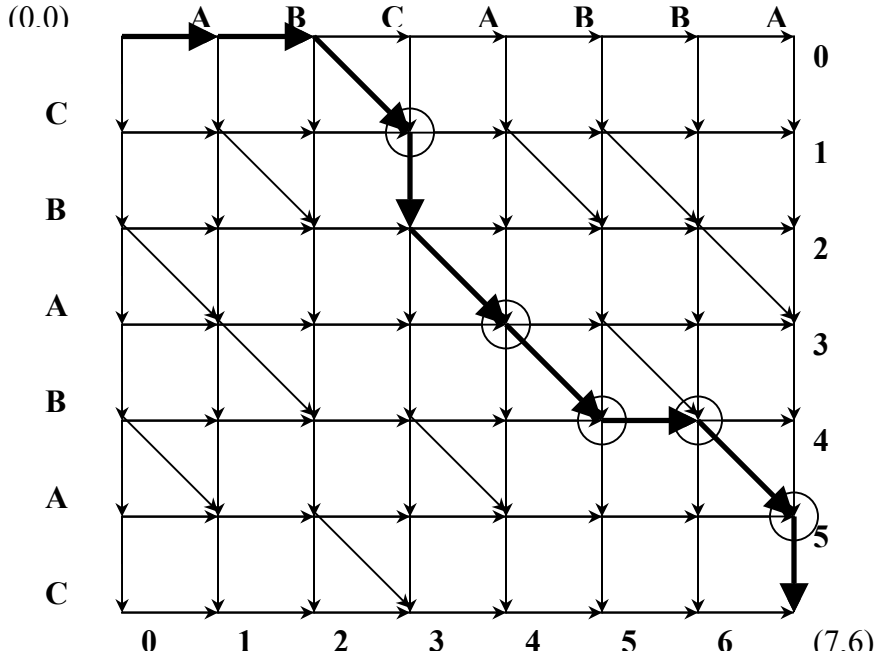

Figure 4. Sample Edit graph; Path trace = (3,1), (4,3), (5,4), (7,5); Common subsequence = CABA; Edit script = 1D, 2D, 2IB, 6D, 7IC

A trace of length L is a sequence of L match points (x_1, y_1), $(x_2, y_2)\ldots(x_L, y_L)$, such that $x_i < x_{i+1}$ and $y_i < y_{i+1}$ for successive points (x_i, y_i). Each trace corresponds to a

common subsequence of the strings A and B. Each trace also corresponds to an edit script. This can be seen from figure 4; if we equate every horizontal move in the trace to a deletion from A and every vertical move to an insertion into B, we have an edit script that can transform A to B. Finding the longest common subsequence and finding the shortest edit script both correspond to finding a path from (0,0) to (N, M) with the minimum number of non-diagonal edges. Giving diagonal edges a weight of 0, and non-diagonal edges a weight of 1, the problem reduces to finding the minimum-cost path in a directed weighted graph which is a special instance of the single source shortest path problem.

Refer to Myers [24] for the algorithms to find the length of such a path and to reconstruct the path's trace. These algorithms have been used for approximate string matching and approximate substring matching respectively as part of the overall change detection algorithm.

CHAPTER 3
PROBLEMS INVOLVED IN RECOVERABLE CONTENT ADAPTATION

Developing a Generic Difference Detection and Synchronization Algorithm for Reduced Content Information

Since document formats vary widely, and there are a vast number of them, the number of possible content reductions is enormous. To reduce the size of the problem and make it tractable, only the extreme case was considered for solution. The system that was developed can now be extended to suit other combinations. This extension is an ongoing task. Document editing has been chosen as a problem domain and to investigate recoverable content adaptation, conversion to text followed by reintegration of changes to the edited text has been chosen as the restricted problem to be solved. This problem is the extreme case of content adaptation in the domain of document editing. Abiword [25], an open source document editor, has been selected as the test editor as it uses XML as its native file format and is available on both Linux and Windows.

Example Document Content Reduction Scenario

Consider the document shown in figure 5 which is reduced to text and edited as shown in table 1. The notation followed in this sequence of changes to the document is to refer to the source document as v_0, the content reduced (converted to text) document as $v_0(-)$, the content reduced document after edits as $v_1(-)$, and the final document with changes integrated as v_1. In the general case, reduction will not be so extreme and the result document will still be in XML format. Hence, we need to impose an XML structure on the $v_1(-)$ document. However, since all formatting content has been lost, it

will only be possible to impose a very basic structure on the document. This basic structure will depend on the particular word processor in use and hence the module handling this will vary for every document format. Therefore, a plugin management system is required to load the correct converter depending on the particular format being handled. In the case of Abiword, imposition of this structure involves extracting text terminated by a new line and inserting it under a <p> tag since in Abiword, paragraphs are terminated by a new line. Other structures that can be safely imposed are an <abiword> tag and a <section> tag since they are present in all Abiword documents. However, this is the maximum extent we can go to, given only the modified text document. The resulting XML DOM tree is shown in figure 6.

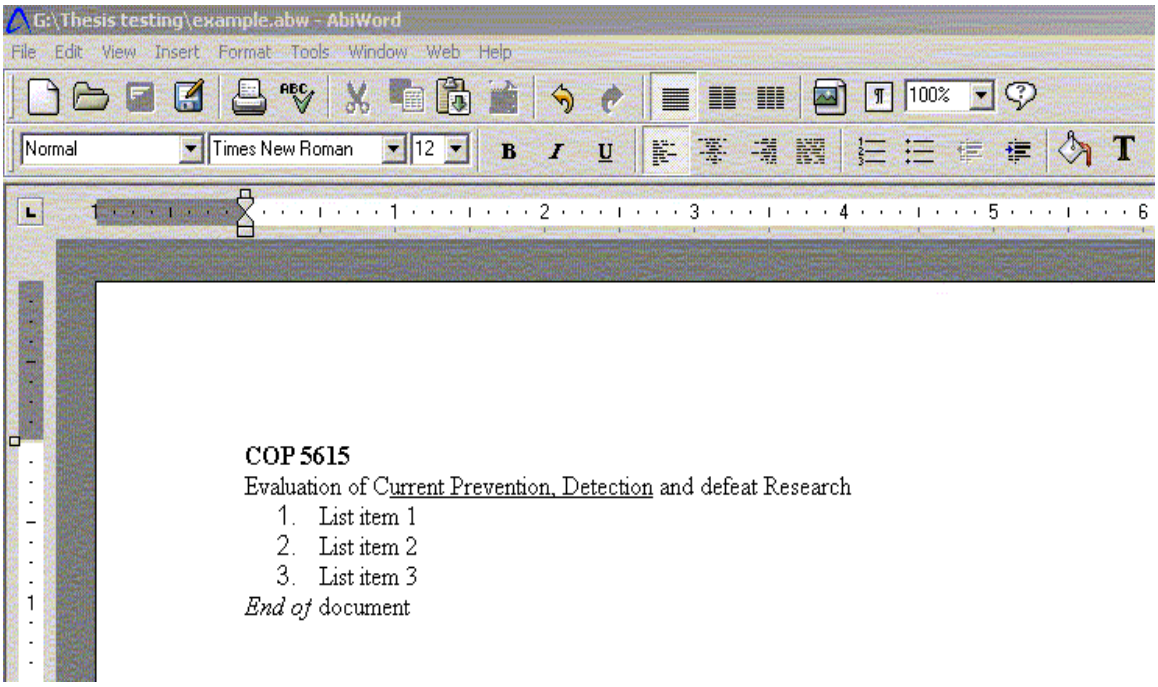

Figure 5. Example Abiword document

Table 1. Document in figure 5 converted to text and edited

Document text content (v_0 (-))	Document edited text content (v_1 (-))
COP 5615 Evaluation of Current Prevention, Detection and defeat Research List item 1 List item 2 List item 3 End of document	COP 5615 Evaluation of Current Prevention, Detection and defeat Research List item 1 List item 2 Updated #1## List item 3 End of document

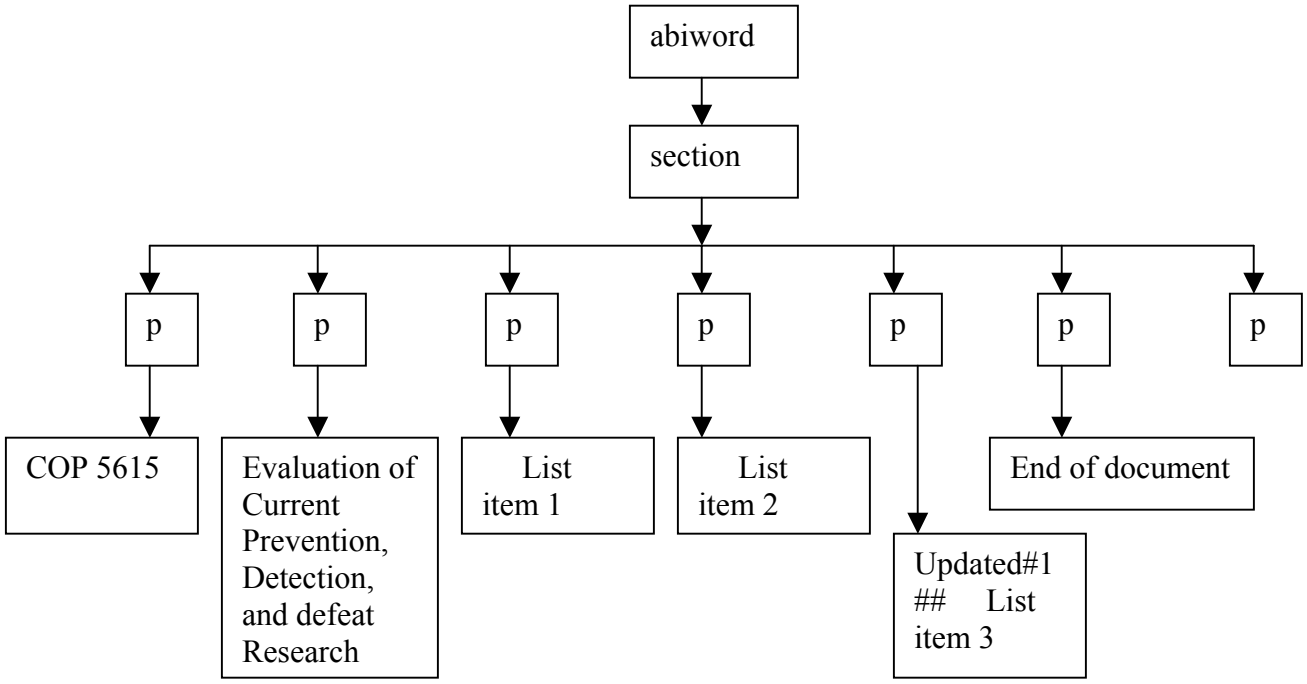

Figure 6. XML structured imposed on modified text document

For comparison, the original document structure is shown in figure 7. Clearly, the modified document has a far simpler structure. To enable integration of changes to the original document, we need to come up with a function which, when given the original document tree and the tree with XML structure imposed, produces a structure with the content in the modified document that is isomorphic to the source document.

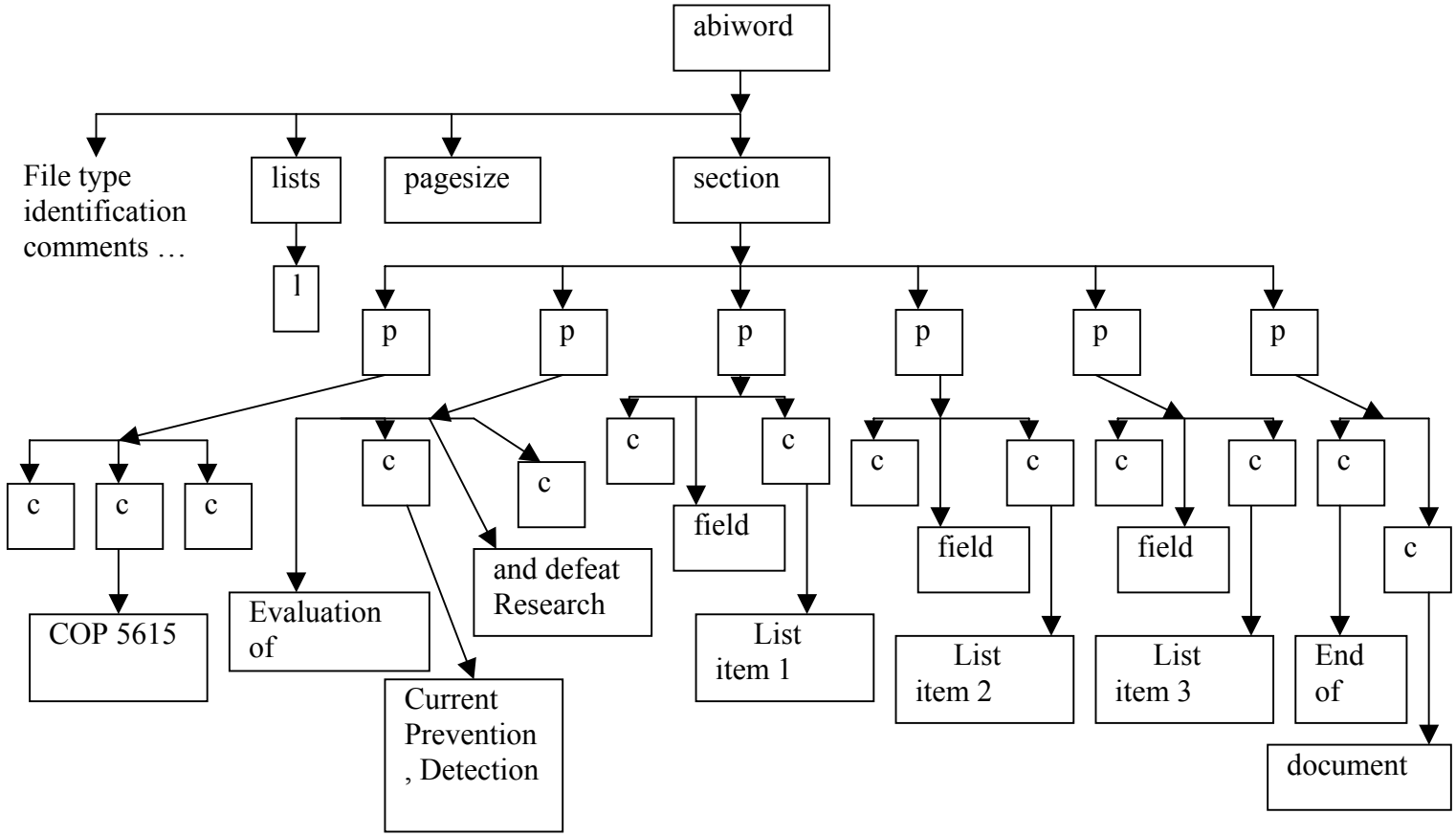

Figure 7. DOM tree for the original document, v_0

Once we have this isomorphic modified tree, the difference script can be constructed using standard techniques employed in other difference algorithms.

Issues in Designing a Function to Construct a Modified Document Structure Isomorphic to the Source Document Structure

1. A format independent method of specifying what content is shared between the modified document and the source is needed to identify what content to consider deleted versus just stripped out. In our example above, the shared content are the abiword, section, p and text nodes as shown in figure 6.

2. A technique for matching content nodes in the modified document to the nodes in the source document is needed. However, this is complicated by the following possibilities:

    a. Node contents can be edited to varying degrees. Hence, an approximate string matching technique is needed and a boundary value for a node to be considered matched to another needs to be found.

    b. An aggregation of nodes in the source document may be present as one node in the result. As an example, consider the string "Evaluation of Current Prevention, Detection and defeat Research" in figure 5. The substring "Current Prevention, Detection", is underlined and hence the DOM subtree for this string in figure 7, has three nodes corresponding to the string. However, on conversion to text, this structure is lost and when a basic XML structure is applied, the three nodes are present as one node in the result tree in figure 6. In addition to possibility a. occurring, the result string may have the subsequences in a different order, which further complicates matching. Since the string may be edited in any way, it is not possible to match each substring in the source to the result split into the same sized substrings. Matching each substring in the source document to every prefix/suffix of the content node in the result tree is also incorrect, since the match may be any substring and not necessarily a prefix or suffix. Hence, structures such as suffix trees are unusable. Matching all possible substrings would be extremely expensive computationally, since all possible substrings would result in $O(2^n)/O(1+2!+3!+\ldots+n!)$ combinations which when rewritten as $O(2^n)/O(n!+(n-1)!+\ldots) = O(2^n)/O((n-1)(n+1)+(n-3)(n-1)+\ldots) = O(2^n)/O(n^3) = O(2^n)$ combinations to be processed. In the extreme case, when no XML structure can be imposed on the document except a document root tag, all content will be present in one node. This extreme case of the aforementioned problem underscores the need to develop an efficient solution to it.

3. Once matches are found, the tree structure from the source needs to be used to construct an isomorphic structure for the result that re-imports all stripped out content.

4. A mechanism is needed to filter out certain content nodes from the match process e.g. Abiword encodes images in base 64 text and thus this could be confused for a content node and impose a heavy, and unnecessary performance burden.

5. Post processing is necessary to ensure that the created isomorphic tree is valid e.g. if the document has all content deleted but the formatting and style descriptions are imported as part of the process above, Abiword crashes on trying to open the document. A format specific plugin could be run at this point, to perform dependency checks such as this. Alternatively, a separate tool could be used to perform this function after applying the delta to the source document.

CHAPTER 4
DESIGN AND IMPLEMENTATION OF RECOVERABLE CONTENT ADAPTATION

Solution Outline for Delta Construction

As mentioned in the previous chapter, the restricted problem being solved is that of converting a document to text, and incorporating the changes made to the textual version back into the original source document. There are three basic steps in the solution approach as follows:

1. If required, adjust the resulting document by imposing a basic XML structure on it. Match the nodes in the source and modified document trees taking into account the fact that an aggregate of source nodes may match a result node.
2. Adjust the structure of the modified document to make it isomorphic to the source document. In this process all content that was stripped out should be re-imported unless it was associated with deleted content.
3. Construct the delta script using the source document structure and the newly created modified document structure.

Before describing the steps in detail, the question of how shared content and nodes to be excluded from the match procedure are to be represented needs to be answered.

Representation of Shared Content

There are two ways shared content between documents could be represented as follows:

1. The intersection of the set of tags in the source and result.
2. The complement of the set above i.e. the set of tags, which are not shared between the two documents.

The former approach has been chosen since in our case, the number of shared tags will be very small (Since all documents are in XML format we treat shared information

as shared tags). This information can also be represented in an XML format. Hence, there will exist one such map for each file format. The corresponding one will be read depending on the format being handled and the tag names will be stored in a hash table (STL hash map) for O(1) time lookup during the matching process. The same idea is used to represent excluded tags (the information which is to be excluded from the matching process).

Matching of Nodes in the Source and Modified Documents

The algorithm has the following basic steps as follows:

1. Recursively construct a list of content nodes (TEXT nodes) from the source (v_0) document excluding any subtrees under excluded tags.
2. Construct a list of content node subtrees from the v_0 document and remove the corresponding nodes from the list constructed in step 1.
3. Recursively construct a list of content nodes (TEXT nodes) from the modified document ($v_1(-)$).
4. Find the least cost edit script matches between the nodes in the v_0 list and the nodes in the $v_1(-)$ list.
5. Find the least cost edit script matches between the v_0 subtree list and the nodes in the $v_1(-)$ list.

Construction of the List of Content Node Subtrees

The algorithm has the following basic steps:

1. For each node in the v_0 list, check if its parent node is a shared node and if the node has a sibling. If so, find out the number of leaf nodes in this subtree by counting each TEXT sibling and recursively counting the TEXT nodes in every non-TEXT node sibling.
2. Otherwise, if the parent is not a shared node, traverse the tree upwards until a shared parent is found. Calculate the number of TEXT nodes in this subtree recursively.
3. If the number of nodes found is greater than one, remove this range of nodes from the v_0 list and insert the nodes into the v_0 subtree list.

Determining the Node Matches between the v_0 and v_1 (-) Trees

The algorithm has the following basic steps:

1. For each node in the v_1 (-) node list, execute step 2.

2. minLES = ∞

    a. For each node in the v_0 node list, perform the following steps:

        i. Compute the least cost edit script length using the algorithm in [24]. If this value is less than minLES, set minLES = the value computed above.

        ii. If the minLES value is below a threshold value, enter the v_1(-) node as a key in a hash table with the value being another hash table containing the (key, value) pair of (v_0 node, the minLES value) inserted into it. Otherwise, if the minLES value exceeds the threshold value, enter the v_1(-) node as a key in a hash table with the value being another hash table containing the (key, value) pair of (v_0 node, ∞) inserted into it.

        iii. If the minLES value is 0, we have found a perfect match for the v_1(-) node and there is no need to consider any other v_0 nodes. Continue execution from step 2.

    b. If minLES < ∞ and if the v_0 node corresponding to the minLES value is already assigned, find the better match by looking up the previous match value in the hash table and comparing it to minLES. If the current one is better, reset the previous match. If not, we need not continue further for this v_1(-) node. Continue execution from step 2.

    c. Mark the v_0 and v_1(-) nodes as matched/assigned to each other.

    d. If minLES is 0, remove the corresponding v_0 node from the v_0 node list since it does not need to be included in any further comparisons.

Determining the Node Matches between the v_0 Subtree List and v_1 (-) Tree List

Before going into the steps involved, we explore the solution approach with an example. Consider the subtree consisting of the nodes AB, C, ABBA and the v_1(-) node CBABAC. For now let us assume that the relative ordering of the substrings in the v_1(-) node is the same as that in v_0, but they may have been edited in any way otherwise. We

need a way to decompose the $v_1(-)$ node content into substrings providing the best approximate matches to the subtree nodes. The approach used utilizes the way the least cost edit script algorithm works. Consider figure 8 which shows the edit graph generated for the concatenation of the substrings (AB, C, ABBA) and the $v_1(-)$ node CBABAC.

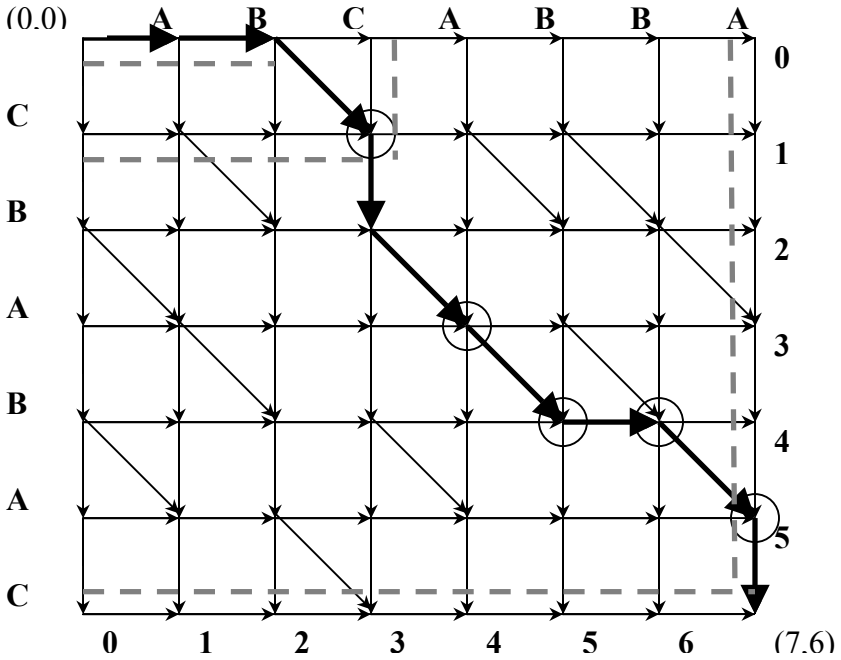

Figure 8. Example of approximate substring matching

We can find the matches for each substring by using the co-ordinates of the path found in the graph. If the end of a substring e.g. AB occurs at point x (in this case 2), the corresponding y value of the point on the path gives the termination of the matching substring (in this case 0 implying that the best match for the substring AB is an empty string). In this example the matches are AB=ϕ, C=C, ABBA = BABAC.

Now returning to the general problem where substring node orderings may not be the same, we can see that the solution is valid provided we find the permutation of the subtree that best matches the $v_1(-)$ node content. However, finding all permutations of a

set of substrings takes O(n!) time which is intractable for large problems. Hence, a condition/heuristic is needed to prune the search. The approach used in the current implementation is to perform the search for n<C where C is a small experimentally determined constant and to assume no reordering in the other cases. Thus, the overall complexity drops to O((M+N)D)+O((M+N))=O((M+N)D) for the strings of lengths M and N with edit distance D, where the second term accounts for the recursive determination of the path through the graph.

The basic steps involved in the algorithm are as follows:

1. For each node in the v_0 substring list, perform the following steps:
    a. minLES = ∞
    b. If the size of the current v_0 subtree is less than a threshold value C, recursively compute the permutations for the nodes in the subtree and store them in a vector. Otherwise, concatenate the nodes in the subtree and store the result in the vector.
    c. For each node in the v_1(-) node list, execute the following steps:
        i. Find the minimum LES value of all the permutations in the vector with the current v_1(-) node subject to the threshold value. If the threshold value is exceeded, the current LES value is ∞.
        ii. If this value is less than minLES, assign it to minLES.
        iii. Enter the v_1(-) node as a key in a hash table with the value being another hash table containing the (key, value) pair, (first node in the v_0 subtree, current LES value) inserted into it.
        iv. If minLES=0, the perfect match has been found and no further comparisons are necessary. Restart execution from step a.
        v. If minLES < ∞, perform the following steps:
            1. If the subtree has already been matched, determine the better match and set the v_1(-) node as the match for the v_0 subtree corresponding to the better match.
            2. Re-compute the best permutation for the matched v_0 subtree and compute its LES match, saving

information needed to reconstruct the trace through the graph.

3. Compute the trace through the edit graph.

4. Use the technique described above, to traverse the trace and split the v_1(-) node into substrings matching the v_0 node. Enter the substrings with their matching counterparts into a hash table.

5. Return the hash table.

2. Otherwise, return an empty hash table.

Adjusting the Structure of the Modified Document to make it Isomorphic to the Source Document

Applying the Substring Matches

The basic steps involved in the algorithm are as follows:

1. For each subtree in the v_0 document, perform the following steps:

   a. If the v_0 subtree has already been matched, execute the following steps:

      i. Set v1Parent = Parent node of the v_1 match node.

      ii. For each node in the v_0 subtree, execute the following steps:

         1. Look up the matching substring the hash table returned from the substring match function.

         2. If the match exists, then execute the following steps:

            a. Create a text node for the match and insert it before the matching v_1 node. If the v_0 node was previously matched, reset the match. Mark the v_0 and v_1 nodes as matched/assigned to each other.

            b. Add the inserted node to the list of v_1 nodes.

      iii. If even one node in the subtree was matched, delete the corresponding v_1 node (it has now been replaced by its components).

2. Return.

Adjusting the Result Document Structure for Unshared Ancestors

The basic steps involved in the algorithm are as follows:

1. For each node in the v_1 node list, perform the following steps:
    a. Lookup the matching v_0 node and its parent.
    b. If the v_0 Parent node is not null and is unshared, execute the following:
        i. Push the parent node into a stack.
        ii. Get the Parent's ancestor, and repeat step b.
    c. If the shared v_0 parent and v_1 node parent match (have the same node names), perform the following steps:
        i. Match the parents if not done already.
        ii. Propagate the matches upward till we reach a matched node or the root. For each match, import attributes of the v_0 node over to the v_1 node.
        iii. Remove the v_1 node (to allow insertion of unshared ancestors).
        iv. Set v1ParentDom = Parent of the v_1 node.
        v. Get the v_1 node's previous sibling and check if its name is the same as the ancestor we are trying to insert. If so, and the v_0 ancestor is assigned, check the remaining nodes in the stack for assignment and a match with the rightmost path down the sibling's tree. Remove each matching node from the stack. Set v1ParentDom = last match from the stack. This takes into account the possibility that we are trying to insert ancestors common to siblings.
        vi. For each node on the stack, execute the following steps:
            1. Append the node as a child to v1ParentDom.
            2. Mark the appended node and the v0 nodes as matched/assigned to each other.
            3. Set v1ParentDom =appended node.
        vii. Append the v_1 node as a child of v1ParentDom.
2. Return.

Adjusting the Result Document Structure for Unshared Nodes

The basic steps involved in the algorithm are as follows:

1. If the document roots are not matched and they have the same tag names, match them. Otherwise, signal an error and halt.
2. Set v0Node = First child of the v_0 Root.
3. Set v1Node = First child of the v_1 Root.
4. Similarly, set v0Parent and v1Parent to the v_0 and v_1 Roots respectively.
5. If v0Node is null, return.
6. If the v0Node is not assigned, and it is unshared with its parent matched to the v1Node parent or has an excluded parent, execute the following steps:
    a. Add the v0Node to the v_1 tree as a previous sibling of the v1Node if it is non-null or the only child of the v1Parent otherwise.
    b. Mark the nodes as matched/assigned to each other.
    c. Recursively process all the children of the v0Node and v1Node.
7. Otherwise, if the nodes are already assigned, recursively process all the children of the v0Node and it's v_1 match.
8. Recursively process all the children of the v0Node's next sibling and the v1Node.

Applying Default Formatting for Inserts

This phase has been added to the overall algorithm to provide the flexibility of assigning any formatting as default for inserted content. This allows inserted content to be easily identifiable. However, to allow any formatting to be applied, we need the flexibility of inserting the new content as a child of a hierarchical structure. This structure is added as a child of a shared ancestor (which would have been the immediate parent of the inserted content). To define the structure, we use an XML document specifying the tree structure where the root of the tree defines the shared tag to which this structure is to

be applied. The insertion point, which is a text string to be replaced by the inserted content, is also defined in this document. The structure is read in along with the other XML configuration files and each time the structure has to be applied, the new content replaces the insertion point. The new content is then set as the new insertion point for the next cycle. When the structure is applied, the replacement is performed as above and the tree is inserted into the position previously occupied by the inserted content.

Post Processing

Before running the XyDiff delta construction phases, we need to set up some basic structures needed by the difference detection algorithm and apply the discovered matches to those structures. Once this is done, a XyDiff optimization phase is run to detect updates that may have been missed by the least cost edit script match phase due to the text differences being too extensive. The optimization function should actually be run before the application of the default formatting for inserts but since it will rarely find missed matches, this is the way it has been kept for now. Another justification for doing so is that the structures used by this function do not exist before the insert formatting is applied. Insert formatting cannot be applied later since the new nodes inserted will require extensive re-organization of the XyDiff structures. Hence, the best approach would be to perform tests on the effectiveness of the phase and if required, re-implement it to function with the structures used by the match phase. A brief description of these phases follows.

Computing signatures and weights for the two documents

The basic steps involved are as follows:

1. Set the first available XID for the source document based on its XID map.

2. Register the source and result document by traversing the documents and assigning identifiers to each node based on a post-order traversal of the tree. Create a list of data nodes storing these IDs and hash values for the subtrees corresponding to the tree nodes in question.

Applying the node matches to the XyDiff structures

Apply all node matches generated as part of the matching phases to the XyDiff structures. If the matched nodes have the same names and values, mark them as NO-OPS. Otherwise, if they have the same parents, they represent an update and hence mark them accordingly. Otherwise, it must be the case that a move has occurred, which will be handled as part of the delta construction phase.

Optimizing node matches

In this phase, we basically look for matched nodes and check if any children need matching. If children are found satisfying the condition that the source and result tag names are the same, and that the result has only one child with that tag name for the current parent, they are matched.

Constructing the Delta Script

The steps involved in delta construction are as follows:

1. Mark the old tree.
2. Mark the new tree.
3. Save the XID map.
4. Compute weak moves.
5. Detect updates.
6. Construct the delete script.
7. Construct the insert script.
8. Save the delta document.
9. Clean up.

Mark the Old Tree

Conduct a post-order traversal of the source document and mark the nodes as deleted and moved. A deleted node can be identified as a node that has not been assigned.

An assigned node that has different parents in the source and result documents is deemed a “strong move”.

Mark the New Tree

Conduct a post-order traversal of the result document and mark the nodes as inserted or moved and adjust the XID map accordingly. Matched nodes are imported into the result document’s XID map with the XID of the node in the source document. A moved node is identified in the same fashion as above and if the node is not assigned, it is deemed inserted and marked as such.

Save the XID Map

Write the result document’s new XID map to file.

Compute Weak Moves

A weak move is one where assigned nodes have the same parents but are in different relative positions among the parent’s children. The function of this phase is to determine the least cost edit script that can transform the old child ordering to the new one. This corresponds to finding the longest common subsequence between the two orderings.

Detect Updates

If a node and its matched counterpart have only one unmatched TEXT node as children, match them and mark the children as updated.

Construct the Delete Script

Construct the delete script by traversing the nodes in post-order with a right to left enumeration of children. Based on the marked values, construct appropriate tags for the delete script, which specifies what operation is to be performed, at what location, and enumerates the affected XIDs. Add the created subtree to the delta script DOM tree. In

this process, situations where a node is deleted and its parent is deleted need to be handled. The rule employed is that if the parent is to be deleted, only the parent's delete operation is written to the delta and all deleted children are specified as part of the parent's delete operation.

Construct the Insert Script

The same procedure as above is also used to construct the operations for inserts. Note that moves and updates are also handled in these two phases.

Save the Delta Document

Write the resulting delta tree to file.

Clean Up

Close the input files and free allocated memory.

CHAPTER 5
RELATED EXTENSIONS TO UBIDATA

Development of a File Filter Driver for Windows CE to Support a UbiData Client

To allow integration of the change detection system with UbiData, and testing on various platforms, it is necessary to first have UbiData support a range of platforms. To allow porting of the UbiData system onto windows CE, a file filter driver is necessary to capture file operations to ascertain when a file on the local file system has been accessed or modified. This filter driver communicates the information to the local UbiData client, the M-MEM that uses the information to build the user's working set or to mark the locally modified file as requiring synchronization with the server. The windows CE .NET 4.0 storage architecture [26] is as shown in figure 9.

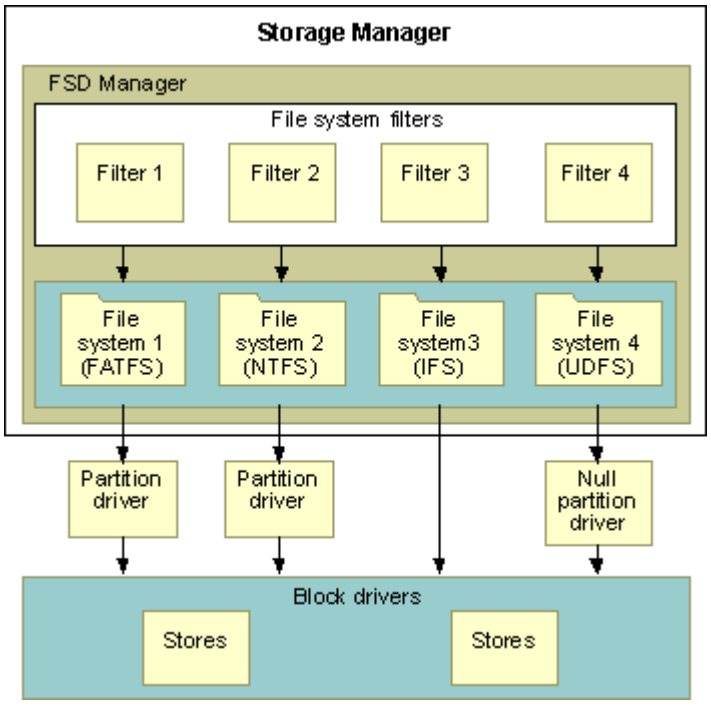

Figure 9. Windows CE .NET 4.0 Storage architecture [26].

As can be seen from figure 9, a file system filter is a dynamic-link library (DLL) that sits on top of the file system and intercepts file system calls. Multiple filters can exist on any loaded file system performing any combination of file manipulations before the file system sees the call. For the purposes of UbiData, we need to trap the file system functions associated with create, open and write. For every open/create call on a file, the file name is stored in a hash table with the key being the file handle value. On a write, the hash table is looked up to get the corresponding file name. These file operations are encoded (using an application specific lookup table) and communicated to the M-MEM application.

An issue with the communication is that the driver and client are separate processes where the client runs with user level privileges. An interprocess communication technique is required that is lightweight and fast. This method should not use an inordinate amount of memory since the mobile device will have limited memory. The solution to this problem was to use memory mapped files in a logical queue (see figure 10) ensuring an upper bound on memory utilization and yet being fast and lightweight compared to pipes and sockets.

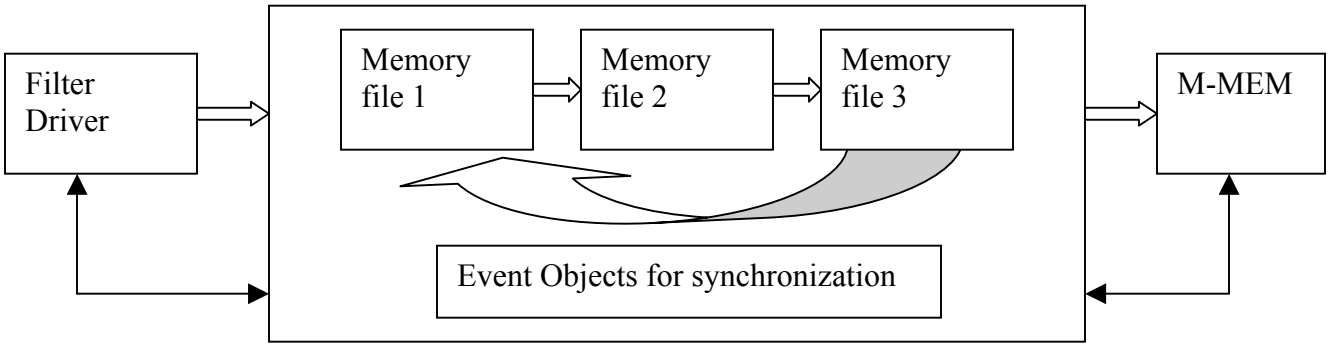

Figure 10. Logical queue of memory mapped files

To complete the port, the UbiData client code and libraries need to be ported to Windows CE .NET 4.0.

Active Event Capture as a Means of UbiData Client-Server Transfer Reduction

The idea behind active event capture is to reduce data transfer between a UbiData client (M-MEM) and the UbiData server (F-MEM) by capturing and logging events generated by an application as it is being used to edit a file. These events are then converted to higher-level descriptions and optimized. The resulting event log would in some cases be smaller than the delta computed by a difference algorithm. The delta can also be computed and compared to the event log to decide which one to ship based on size. Although a promising idea (and one that has been implemented in a desktop environment at Texas A&M University [21]), it was found that the burden of event capture and the resulting context switches (since the application logging events was different from the editor, each time a message was posted to the editor's queue, a context switch to the logging application would be made) was too heavy for a mobile device. However, as mobile devices get progressively more powerful, this may be a promising avenue of research for supporting more resource rich (and perhaps less mobile) devices such as laptops.

Development of an Xmerge Plug-in to Support OpenOffice Writer

As explained in the introduction, a plugin architecture is the cleanest method for handling application/format specific code. Since this is the philosophy guiding the design of Xmerge [22], the Xmerge plugin management code can be used to implement the UbiData integration architecture proposed earlier. As a result, the OpenOffice [27] to text converter and the text to a basic OpenOffice document format converter have been implemented as an Xmerge plugin.

CHAPTER 6
PERFORMANCE EVALUATION APPROACH

Test Platform

Testing was performed on a machine with an AMD Duron 650Mhz processor, 128 Mb RAM and an IBM-DTLA-307030 hard drive running Mandrake Linux 8.2 (kernel 2.96-0.76mdk) with gcc version 2.96.

Test Methodology

The architecture of the test engine is as shown in figure 11. The figure also shows the sequence of steps involved in the testing process.

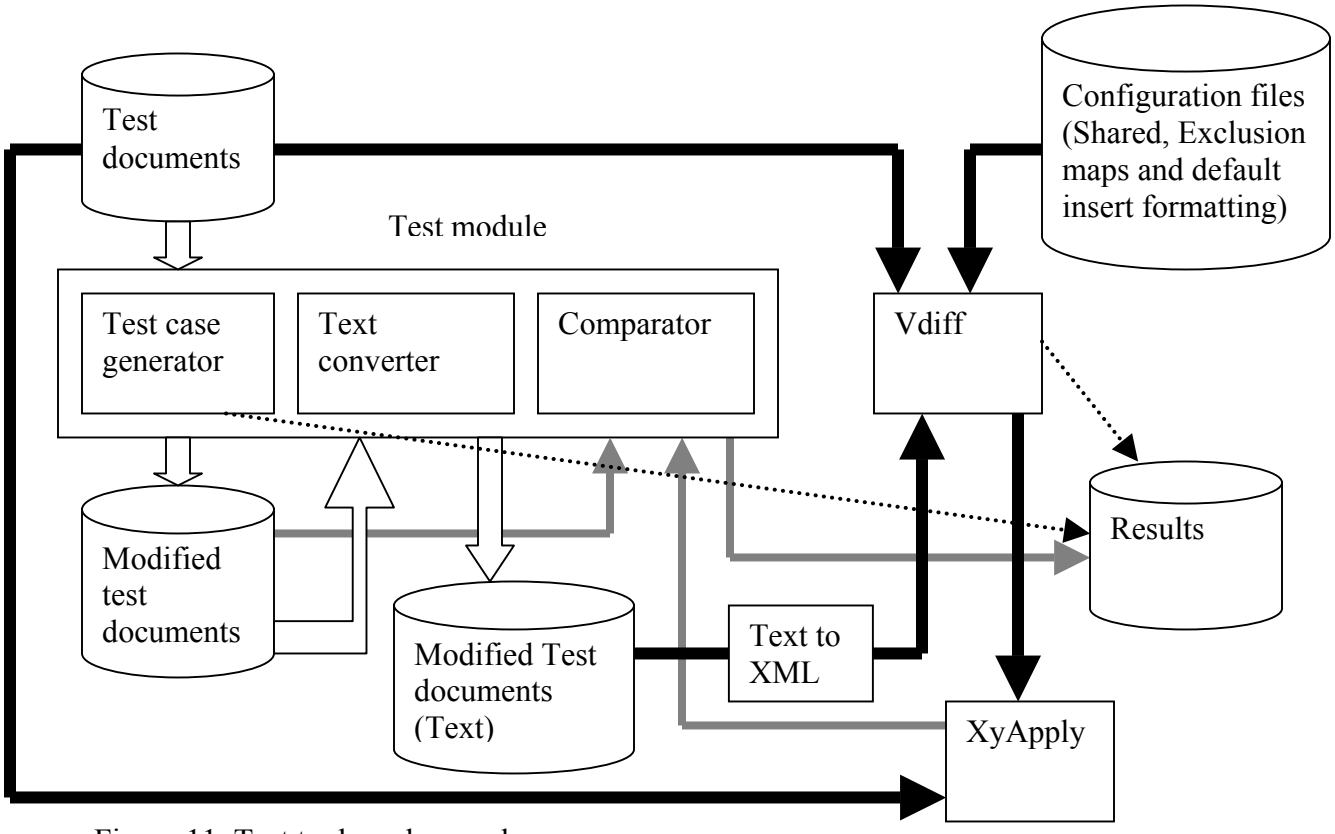

Figure 11. Test tools and procedures

The steps involved in the testing process are as follows:

1. 10 documents were selected as initial test cases. These documents included letters, application forms, term papers, pages from an E-book and a resume.

2. A test case generator module was used to generate 960 modified documents from these source documents by traversing the DOM trees and modifying the content nodes. Each test case had 0-50 percent file level changes and 10-40 percent paragraph level changes for four different edit operations, viz. insert, delete, update and move, applied to it to generate its share of 96 modified documents. Paragraph level operations refer to operations performed within a content node since these nodes represent paragraphs or their components in Abiword. The number of content nodes they are applied to depends on the file change percent. The file level change percent also determines the number of content nodes added, deleted and moved for inserts, deletes and moves respectively. The test case generator also records information on source file statistics, viz. the source document size (in bytes), node count, text node count, number of document subtrees, number of content nodes in all subtrees, average number of children, average height, average content size (in characters) and the average number of attributes which are recorded (shown as a dotted line).

3. The 960 Modified documents are converted to text using a text conversion module which traverses the DOM trees and extracts the content in the TEXT nodes (shown by the block arrows).

4. The modified text documents then have a basic XML structure imposed upon them before being used as inputs to Vdiff, the change detection module described in this thesis, along with the original documents and the configuration files. During execution, Vdiff generates statistical data such as the time taken by each phase of the algorithm, which is recorded (shown by the dotted arrow).

5. The resulting deltas generated by Vdiff are used to patch the original files to generate the modified documents (shown by the black block arrows).

6. These modified documents are compared to the reference modified documents to quantify reconstruction accuracy and the results are recorded (shown by the gray arrows). The comparator generates three data points for each comparison, viz. the number of missing leaves, missing non-leaves and the number of missing attributes. The comparator determines these values by walking the input trees and noting each discrepancy.

7. All data was generated in comma separated variable format, which can be directly read by spreadsheet software and some databases. The software packages used to analyze the data were Microsoft Access and Microsoft Excel. Access was used to run queries to produce data for specific graphs that were then generated in Excel.

CHAPTER 7
TEST RESULTS

Correctness of the Change Detection System

The change detection system, as shown in figures 12 and 13, has good accuracy with a maximum of 14.8 percent total absolute missing nodes or 12 percent missing nodes with a maximum of 6.6 percent missing leaves and 9.2 percent missing non leaves.

The absolute missing leaves parameter refers to the difference between the number of leaves in the expected reference document and the number of leaves in the result document. The other missing node counts (termed "computed" in figures 12 and 13) are determined by the test engine comparator (refer to chapter 6). The computed measures are needed to determine whether the nodes are in their expected locations, which cannot be determined from the absolute measure.

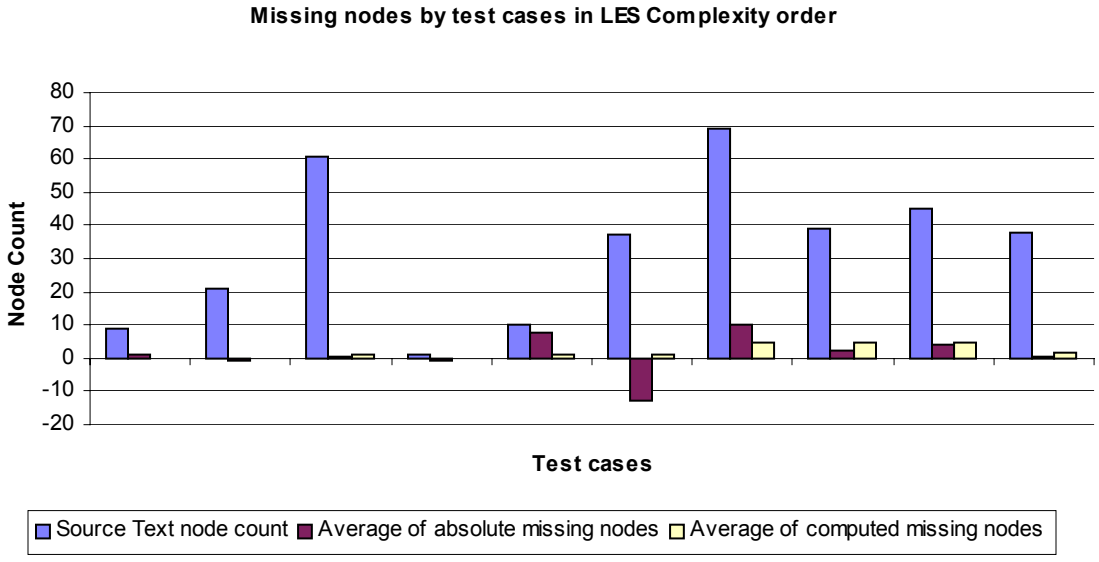

Figure 12. Missing Nodes by test cases in LES complexity order

The error rate seems to depend more on the LES complexity of the document than the substring complexity (refer to the sections on the run time behavior of the phases of the algorithm in the next section on the performance of the algorithm).

The error in the last document in figure 13 may be explained by the fact that its subtree sizes exceed the bound C on subtree re-ordering (refer to chapter 4).

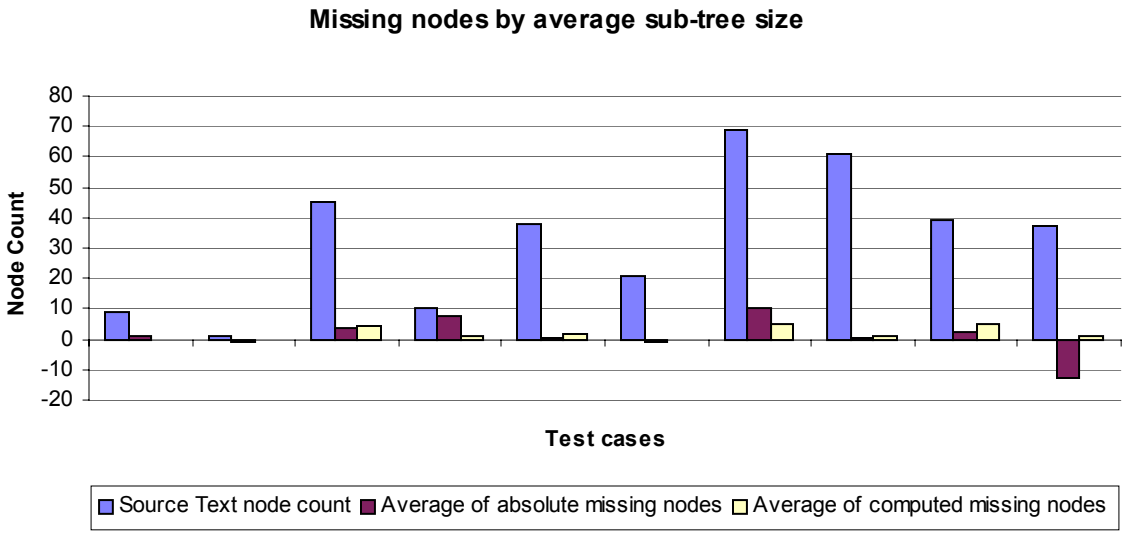

Figure 13. Missing Nodes by test cases in substring complexity order

Performance of the Difference Detection Algorithm

To determine the performance of the difference algorithm and the primary phases on which it depends, we plot the time taken by the various phases of the algorithm for all the test cases in figure 14.

As can be seen from figure 14, the overall time taken (Vdiff2 Compute Delta time in the legend) is dependent almost entirely on only two phases, viz. least cost edit script matching of nodes (Vdiff2 LES Match Nodes) and matching of substrings of the source tree to the modified tree (Vdiff2 Subsequence Match). The substring matching process is clearly the dominant factor in the overall time taken.

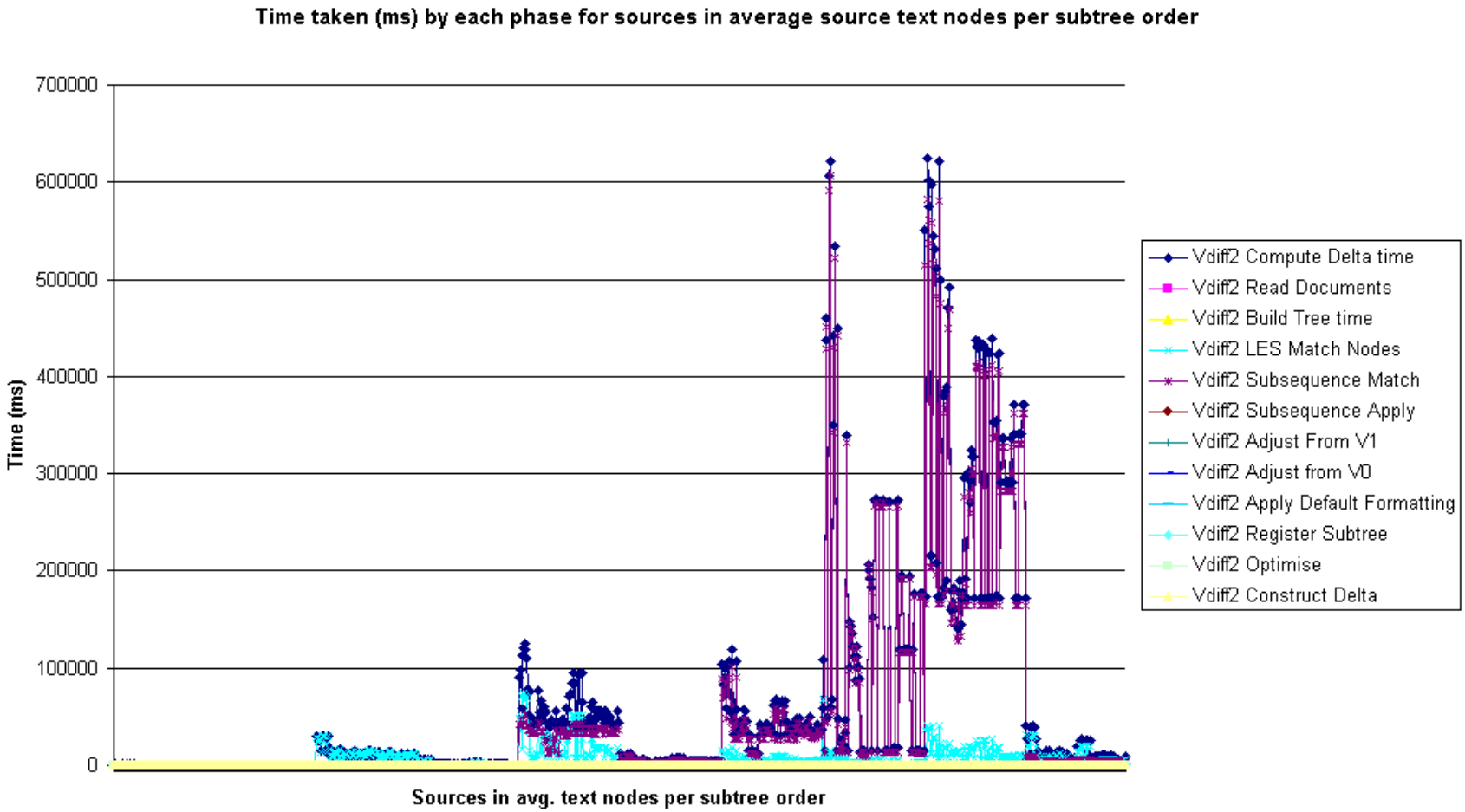

Figure 14. Time taken for the various phases by test case

The test cases have been ordered according to the average number of TEXT nodes per subtree since this is the primary factor determining the substring matching time. The drop in execution time seen for the cases with large subtrees can be attributed to their exceeding the bound C, which determines the limit beyond which no subtree reordering is done (refer to the subtree matching algorithm in chapter 4).

We now analyze the run time for each operation class, viz insert, delete, update and move. We shall consider the node matching and substring matching phases from now on since they are the primary factors determining run time.

Insert Operations

From figures 15 and 16, it is clear that the operations done at file level affect the run time more than the operations related to individual nodes.

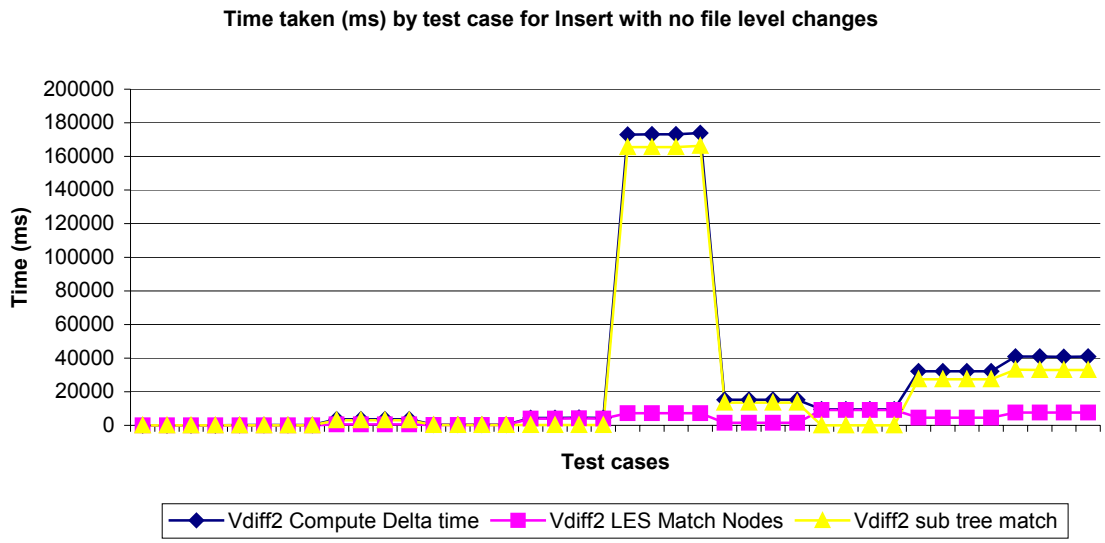

Figure 15. Time taken by test cases for Inserts with no file level changes

This can be attributed to the fact that for each inserted node, it is compared with every node in the source subtree whereas in the case of paragraph level inserts, the number of comparisons remains roughly the same (The only difference being that some comparisons that may have been terminated early due to finding a perfect match now

need to be worked through. However, on average the match would be found halfway through a set of nodes so insertion of a new node would affect run time more).

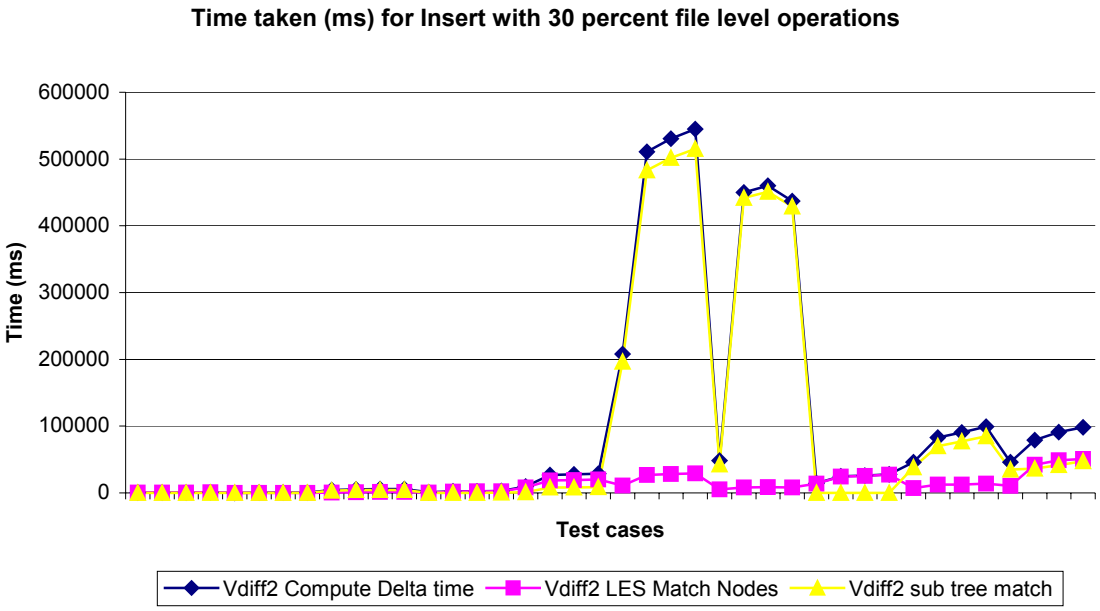

Figure 16. Time taken by test cases for Inserts with 30 percent file level changes

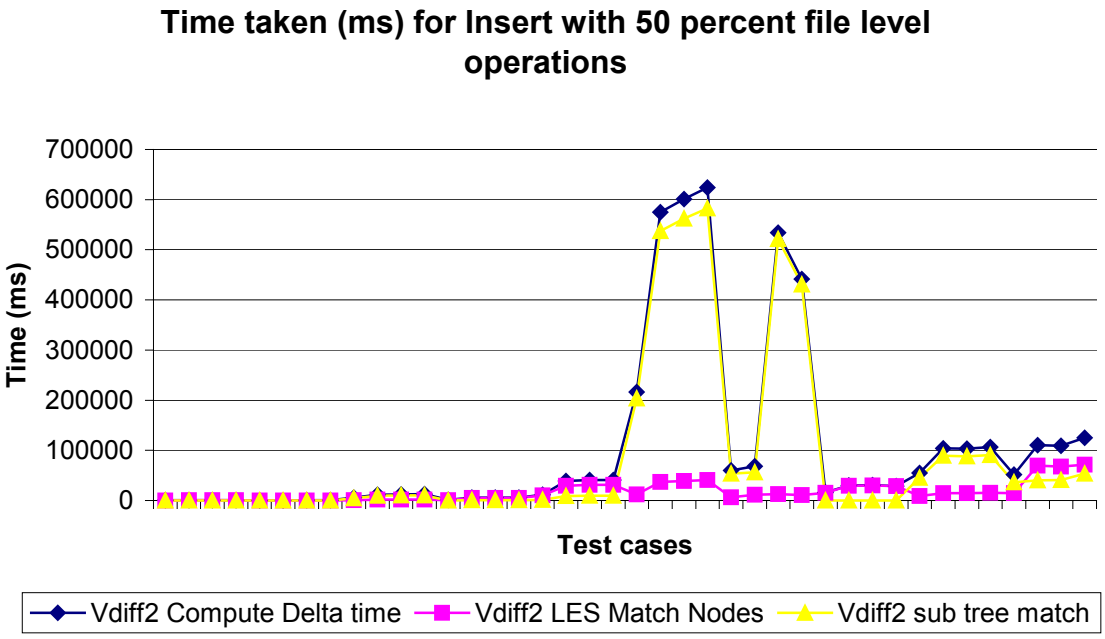

Figure 17. Time taken by test cases for Inserts with 50 percent file level changes

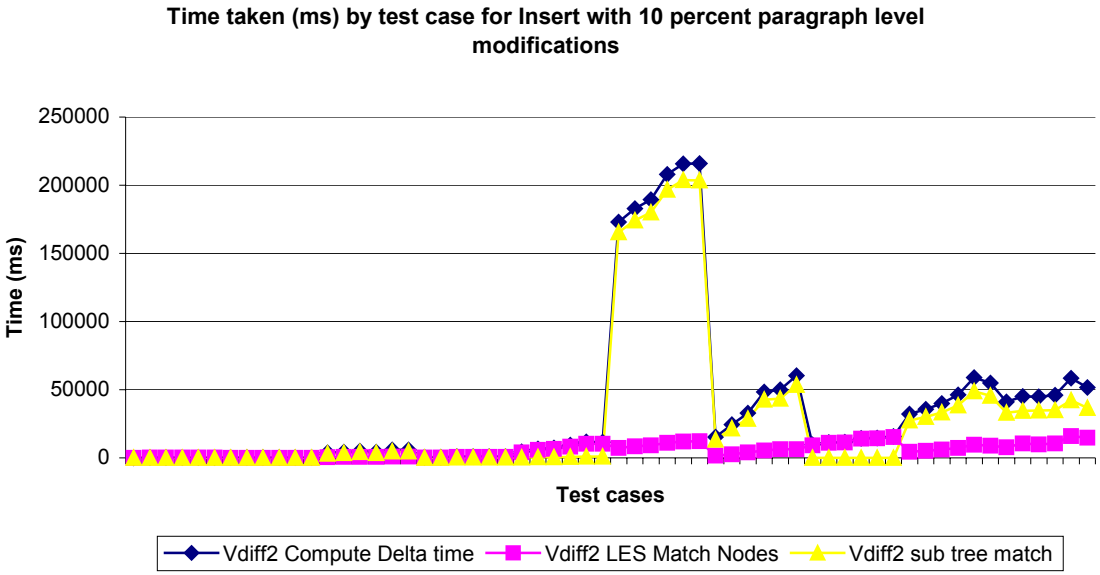

Figure 18. Time taken by test cases for Insert with 10 percent paragraph level changes

The same trend holds for 50 percent inserts at file level as seen in figure 17.

Since we have determined that the operations that affect run time are at file level, the graphs for the remaining operations will be shown in the format of figure 18.

Delete Operations

For deletes, we see that for the highest percentage of file level changes, the delta computation time goes down as expected.

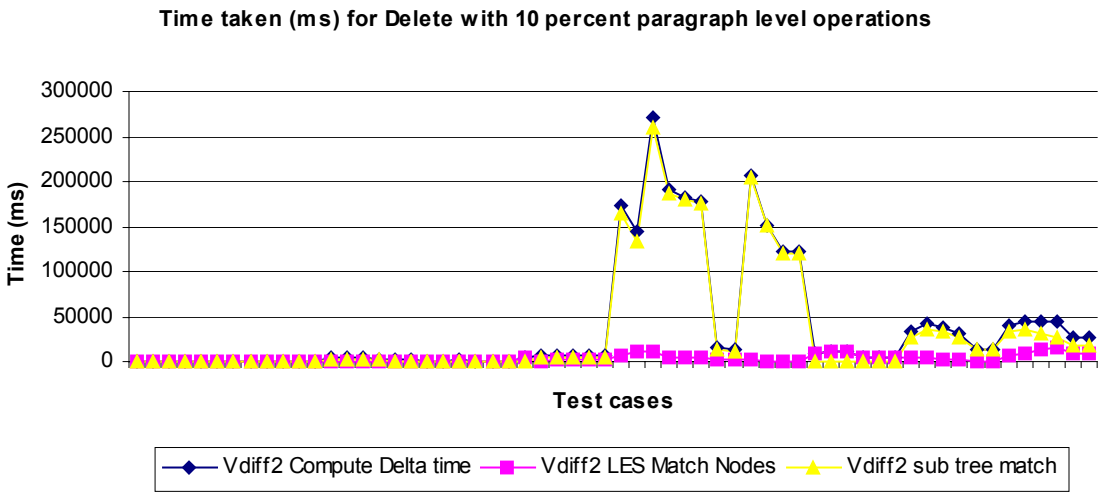

Figure 19. Time taken by test cases for Delete with 10 percent paragraph level changes

However, for the initial deletes, the delta computation time actually rises. This is reflected in both the Node matching phases as well as the substring matching phases.

Update Operations

In the case of updates, the percentage of file operations does not affect the run time since the number of nodes in the files does not change (see figure 20).

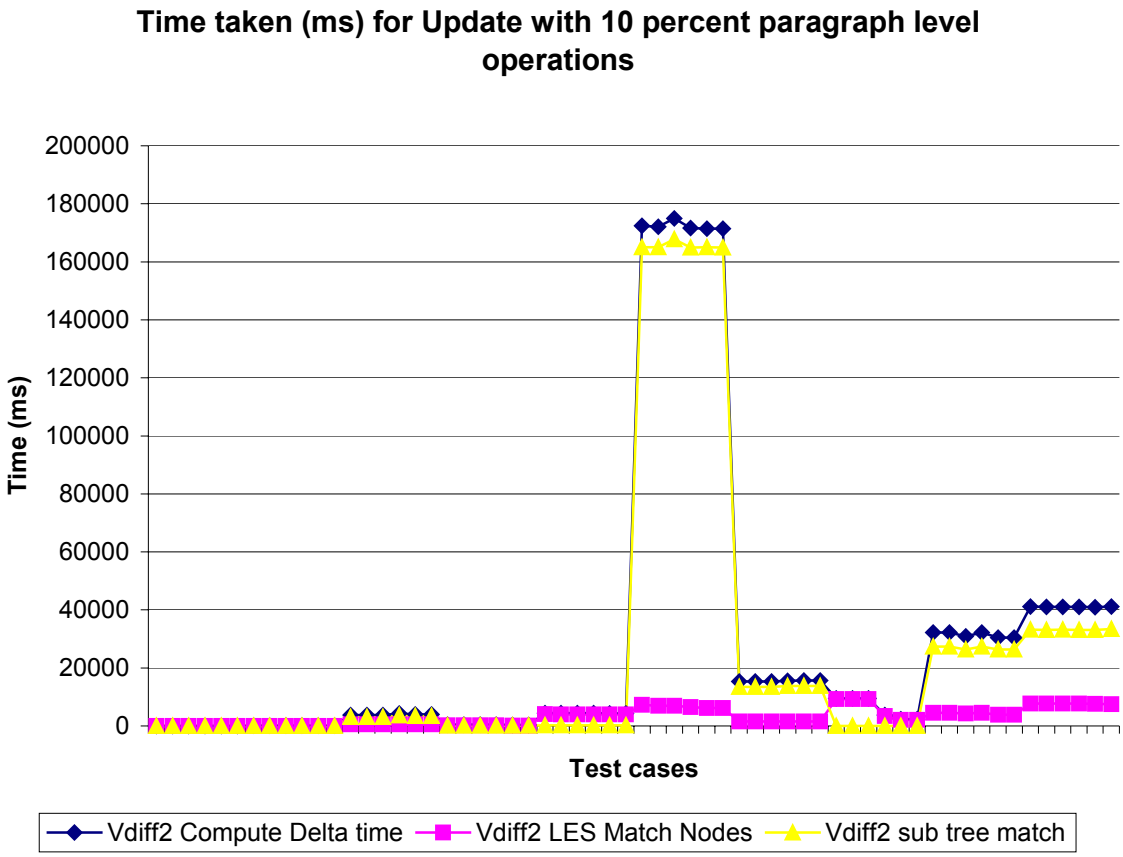

Figure 20. Time taken by test cases for Updates with 10 percent paragraph level changes

However, the same may not hold for paragraph level operations and hence we need to plot paragraph level changes for a fixed number of file level changes as shown in figure 21. As can be seen from figure 21, with paragraph level operations up to 30 percent, there is no significant change in run time.

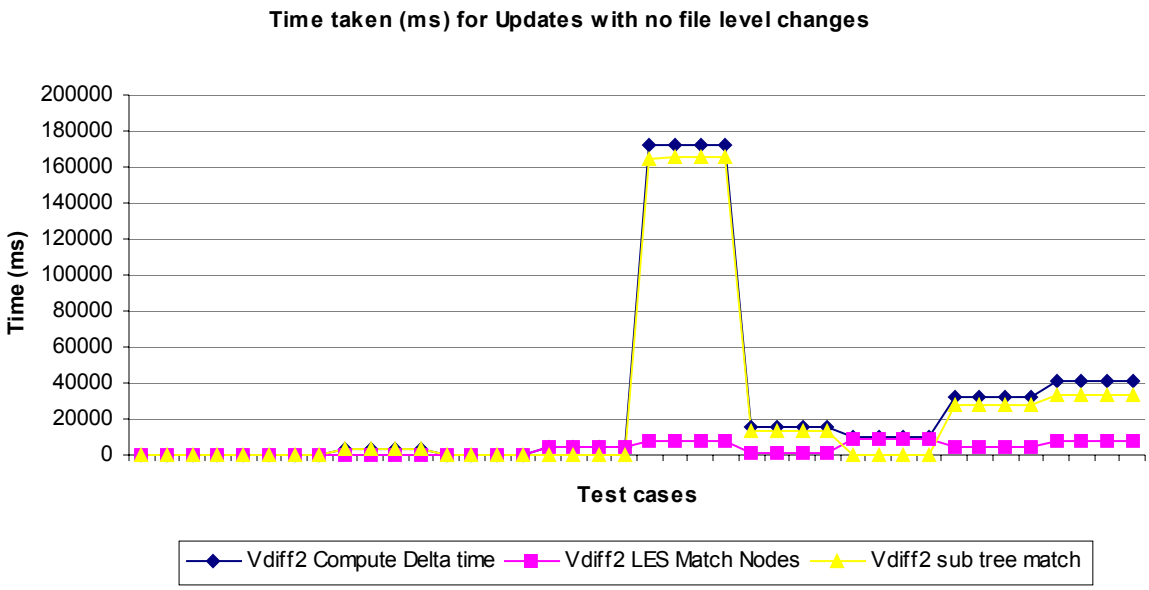

Figure 21. Time taken by test cases for Updates with no file level changes

Move Operations

As can be seen from figure 22, the time taken for the node matching phase remains relatively constant or increases slightly with increasing percentages of moves.

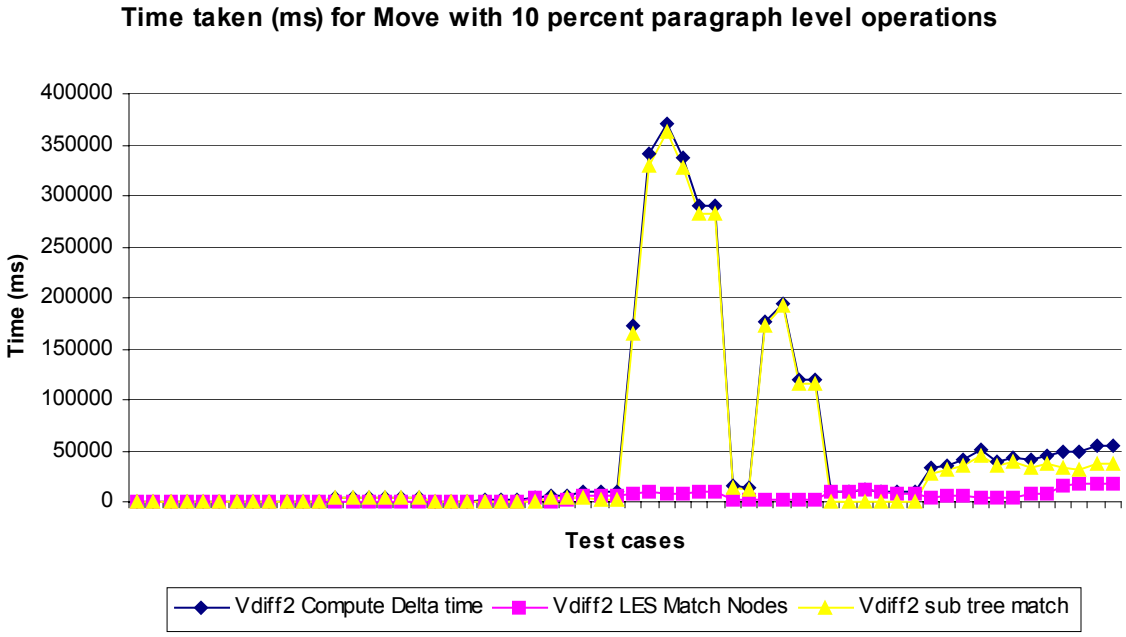

Figure 22. Time taken by test cases for Moves with 10 percent paragraph level changes

This would be expected since the only effect that file level moves have is to affect when a perfect match is found. With increasing move percentages, the nodes are displaced further relative to their previous positions and hence a larger number of nodes need to be compared to find matches. The behavior of substring matching should be independent of the percentage of file level moves and hence no correlation should exist which is borne out by the figure. Hence, we need to investigate the effect of paragraph level changes on run time.

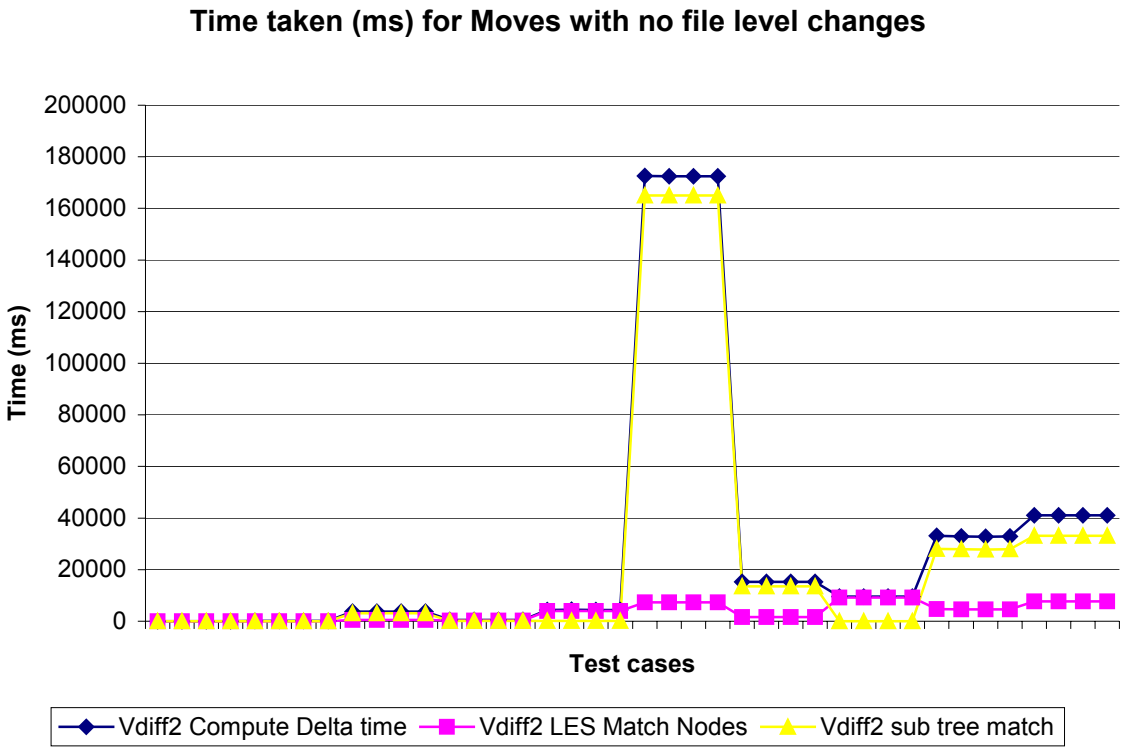

Figure 23. Time taken by test cases for Moves with no file level changes

For increasing paragraph level moves, the run time is not affected. We analyze the run time behavior of the algorithm's phases as a whole in subsequent sections.

Conclusions on Per-Operation Run Time

From the analysis above, we can conclude that the primary factor determining the run time performance of the system is the number of operations at file level. Also, as shown in figure 24, the time complexity is in the order delete, move, update and insert.

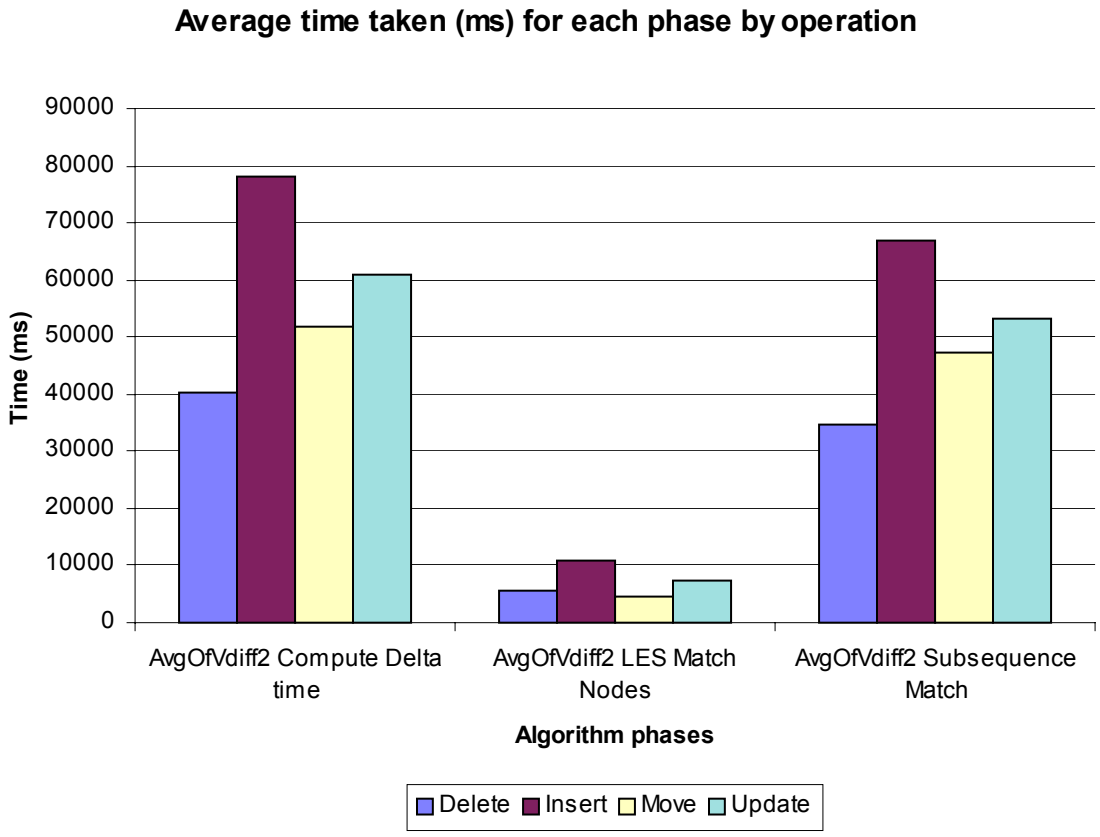

Figure 24. Per-operation relative time complexity

Run Time Behavior of Node Matches

To determine whether the run time behavior of the node matching phase is according to expectations, we need to define a complexity measure to reorder the source documents by and check if the average values for the time taken are monotonically increasing/decreasing. Since we know that the Node matching phase depends on the number of non-subtree TEXT nodes in the source document (since subtree TEXT nodes

are processed seperately in the subtree match phase), and the number of nodes in the result document, we can define our initial complexity measure as follows:

LES Complexity = (non-subtree TEXT nodes)*(non-subtree TEXT nodes + number of subtrees) since the number of nodes in the result document is equal to the number of non-subtree TEXT nodes plus the number of subtrees each of which contribute one node to the result (since the result is a text document, all siblings are clobbered together for lack of a formatting structure). However, we have not considered the time taken by the least cost edit script matching process itself. Since it is *O(ND)* where N is the total size of the nodes being compared and D is the average edit distance for each pair, we can approximate it to 2*Average size of a source text node*D. Since we apply the same percentages of edits to all source document sets, a rough approximation would be to represent D as Average size of a source text node* (Some constant). Hence, an approximate complexity measure works out to the following:

LES Complexity=((Non subtree nodes)^2 + (Non subtree nodes)*(Number of subtrees))* (Average size of a source text node)^2

Ordering the Source file according to this complexity measure as in figure 25, we note that we obtain an increasing trend in terms of average time taken albeit with fluctuations. However, this should be considered in the light of there being relatively few data points, the documents not being large enough to render constant factors small enough to be negligible, and the fact that the measure is itself an approximation.

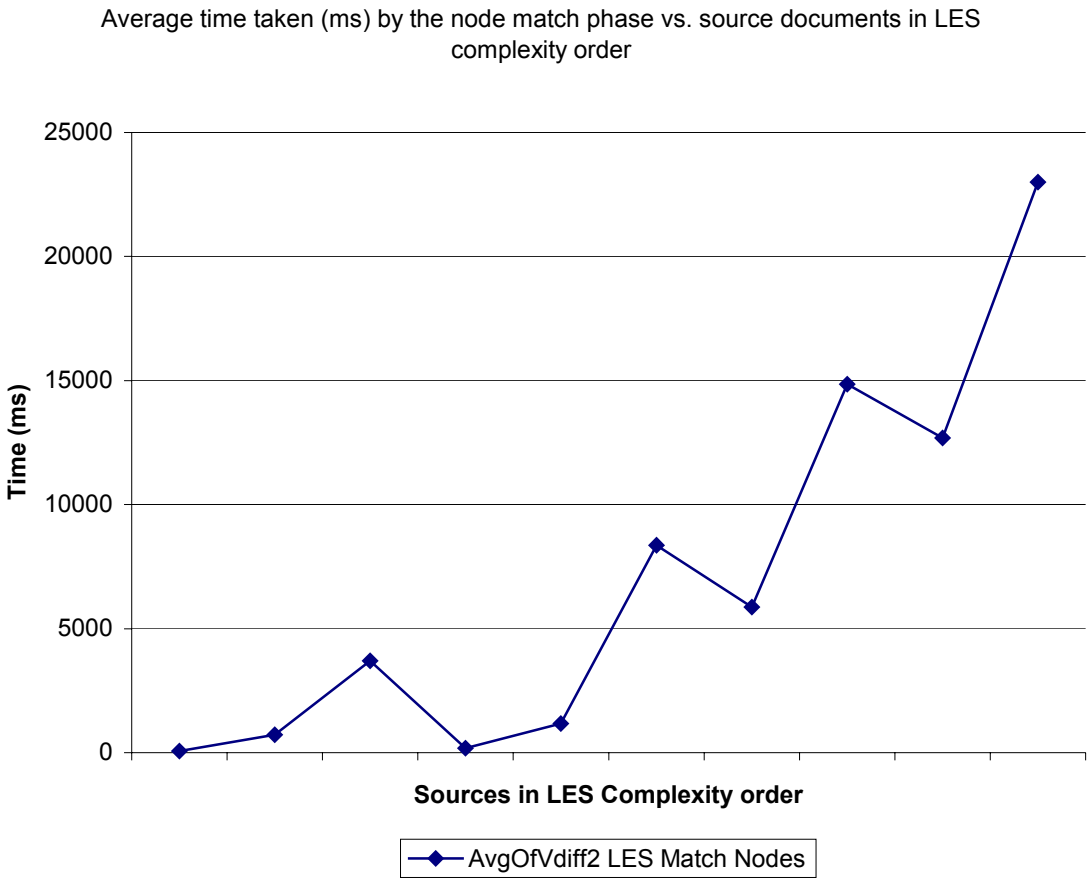

Figure 25. Behavior of the node match phase

Run Time Behavior of Substring Matching

To analyze substring matching, graphs were plotted versus source document parameters and expected complexity measures. However, strong correlation was only shown for the ordering of the sources by the average number of children per TEXT subtree. This is due to the fact that the number of children per subtree determines the number of possible orderings for the nodes in that subtree. The fall in run time for the last source document set in figure 26, is due to the fact that most subtrees in the document have a greater number of children than the re-ordering threshold C. This causes a great drop in the time taken since re-ordering is completely bypassed.

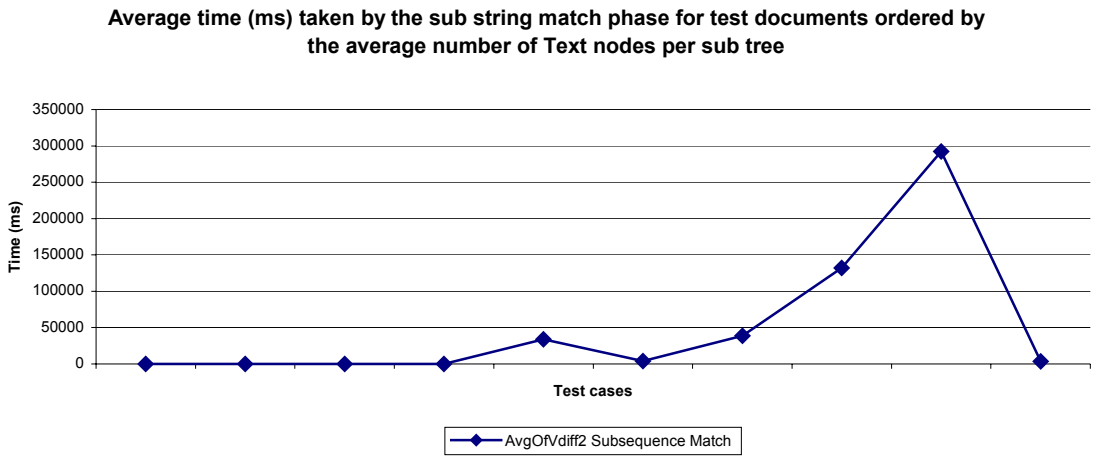

Figure 26. Behavior of substring match phase

Relative Sizes of Files produced

From figure 27, we can see that delta sizes roughly increase with increasing sizes of the source documents.

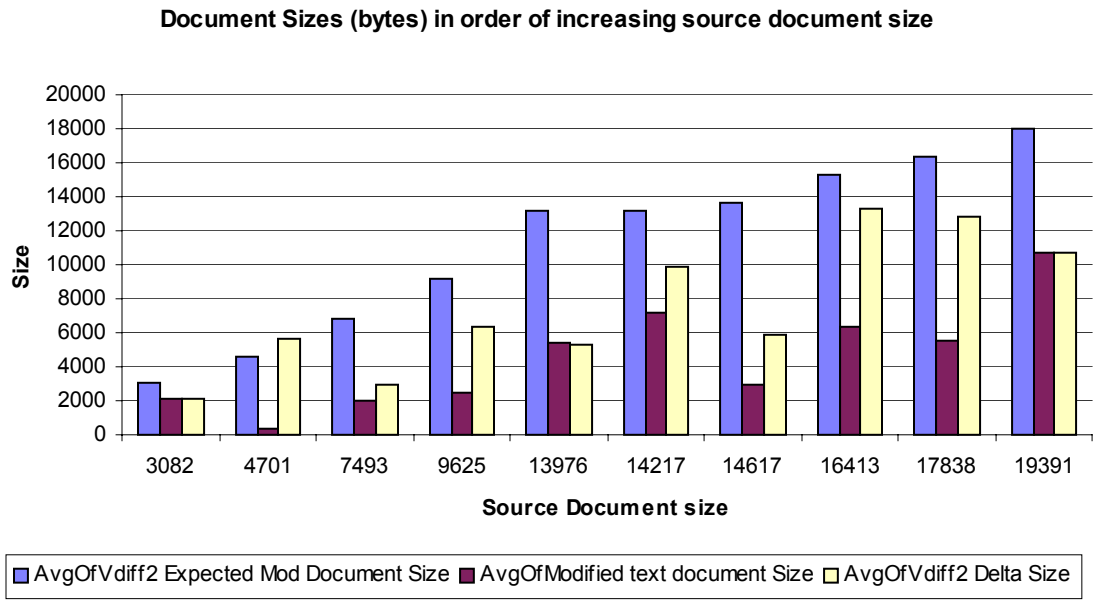

Figure 27. Sizes of the files produced by the algorithm

The deltas are relatively large (still smaller than the sources however) due to their completeness requirement. If version control is not required, these sizes can be reduced further, yielding significant bandwidth savings.

CHAPTER 8
CONCLUSIONS AND POTENTIAL FOR FUTURE WORK

Conclusions

The results obtained in this exploration of a generic algorithm for synchronizing reduced content documents shows that this work can improve upon the state of the art for data access on mobile devices. However, to achieve the goal of ubiquitous data access, the algorithm needs to be tested with other document editing software and extended to handle other classes of applications. This would have to be done in conjunction with the integration of the system into a framework for disconnected operation support e.g. the Coda File System or UbiData.

Potential for Future Work

Determination of Best Performing Values for Algorithm Parameters

The change detection algorithm explored in this thesis depends on the two primary parameters: the threshold value for the least cost edit script matching and the threshold for substring re-ordering before substring matching. Extensive testing is needed to determine the best performing values for the algorithm since a fine balance needs to be struck. For the former, if the match threshold is set too low, heavier modifications will be missed. However if set too high, the possibility of false matches will increase. In the case of the re-ordering threshold, a higher value will result in a significant performance impact but a value that is too low may be completely ineffective.

Exploration of Techniques to Prune the Search for the Best Subtree Permutation

As can be seen from the test results, searching all permutations of subtrees is computationally expensive. Currently a naïve heuristic has been used to prune the search. A better technique may deliver the same level of correctness with lower performance overhead.

Post Processing to Ensure Validity of Reintegrated Document

As stated in chapter 3, post processing is necessary to ensure application consistency requirements are met. This is primarily an issue only for very extensive deletes and rarely occurs in practice. However, it should be handled for a better user experience.

Further Testing and Generalization of the System

The algorithm needs to be tested with a wider range of editors. So far, all tests have been done with Abiword. To test with other editors would require the appropriate converters to be written, appropriate configuration files generated and integration of Xmerge with the test setup (to allow converters to be written as loadable plugins). As a result of such testing, it may be determined that the algorithm does not handle certain conditions arising due to differing document structure. In this case, the code would need to be extended to handle all such conditions.

Implementation of the UbiData Integration Architecture

To validate the usefulness of this work, it is necessary to integrate it with the UbiData system. An overview of this has been described in chapter 2.

Extension to Other Application Domains

The current design and implementation has focused on the document-editing scenario. This caters to the primary usage of most mobile devices (editing documents and

e-mail). However, validating the ideas on a generalized content reduction and re-integration system requires extension of this approach to handle other application classes such as spreadsheets, calendars etc.

LIST OF REFERENCES

[1] Michael J. Franklin, Challenges in Ubiquitous Data Management. Informatics - 10 Years Back. 10 Years Ahead. Lecture Notes in Computer Science, Vol. 2000, Springer-Verlag, Berlin 2001: 24-33.

[2] Richard Han, Pravin Bhagwat, Richard LaMaire, Todd Mummert, Veronique Perret, and Jim Rubas, Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing. IEEE Personal Communications, Vol. 5, Number 6, December 1998: 8-17.

[3] Harini Bharadvaj, Anupam Joshi and Sansanee Auephanwiriyakul, An Active Transcoding Proxy to Support Mobile Web Access. Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems, West Lafayette, IN, IEEE Computer Society Press, October 1998: 118-126.

[4] Mika Liljeberg, Heikki Helin, Markku Kojo, and Kimmo Raatikainen, Mowgli WWW Software: Improved Usability of WWW in Mobile WAN Environments. Proceedings of the IEEE Global Internet 1996 Conference, London, England, IEEE Communications Society Press, November 1996: 33-37.

[5] Armando Fox and Eric A. Brewer, Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation. Computer Networks and ISDN Systems, Vol. 28, Issues 7-11, Elsevier Science, May 1996: 1445-1456.

[6] M. Satyanarayanan, Coda: A Highly Available File System for a Distributed Workstation Environment. IEEE Transactions on Computers, Vol. 39, Number 4, 1990: 447-459.

[7] M. Satyanarayanan, Mobile Information Access. IEEE Personal Communications, Vol. 3, Number 1, February 1996: 26-33.

[8] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, 6 October 2000, available at http://www.w3.org/TR/2000/REC-xml-20001006. August 2002.

[9] J. Zhang, Mobile Data Service: Architecture, Design, and Implementation. Doctoral dissertation, University of Florida, 2002.

[10] Eyal de Lara, Dan S. Wallach and Willy Zwaenepoel, Position Summary: Architectures for Adaptations Systems. Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII). Schloss Elmau, Germany, May 2001, available at http://www.cs.rice.edu/~delara/papers/hotos2001/hotos2001.pdf. August 2002.

[11] Eyal de Lara, Rajnish Kumar, Dan S. Wallach and Willy Zwaenepoel, Collaboration and Document Editing on Bandwidth-Limited Devices. Presented at the Workshop on Application Models and Programming Tools for Ubiquitous Computing (UbiTools'01). Atlanta, Georgia. September, 2001, available at http://www.cs.rice.edu/~delara/papers/ubitools/ubitools.pdf. August 2002.

[12] Eyal de Lara, Dan S. Wallach and Willy Zwaenepoel, Puppeteer: Component-based Adaptation for Mobile Computing. Presented at the Third USENIX Symposium on Internet Technologies and Systems, San Francisco, California, March 2001, available at http://www.cs.rice.edu/~delara/papers/usits2001/usits2001.pdf. August 2002.

[13] G. Cobéna, S. Abitebould, A Marian, Detecting Changes in XML Documents. Presented at the International Conference on Data Engineering, San Jose, California. 26 February-1 March 2002. May 2002, available at http://www-rocq.inria.fr/~cobena/cdrom/www/xydiff/eng.htm. August 2002.

[14] Amélie Marian, Serge Abiteboul, Lurent Mignet, Change-centric Management of Versions in an XML Warehouse. The {VLDB} Journal, 2001: 581-590.

[15] A. Helal, J. Hammer, A. Khushraj, and J. Zhang, A Three-tier Architecture for Ubiquitous Data Access. Presented at the First ACS/IEEE International Conference on Computer Systems and Applications. Beirut, Lebanon. 2001, available at http://www.harris.cise.ufl.edu/publications/3tier.pdf. August 2002.

[16] Michael J. Lanham, Change Detection in XML Documents of Differing Levels of Structural Verbosity in Support of Ubiquitous Data Access. Master's Thesis, University of Florida, 2002.

[17] M. Satyanarayanan, Brian Noble, Puneet Kumar, Morgan Price, Application-Aware Adaptation for Mobile Computing. Proceedings of the 6th ACM SIGOPS European Workshop, ACM Press, New York, NY, September 1994: 1-4.

[18] B. Noble, M Satyanarayanan, D Narayanan, J. Tilton, J. Flinn and K. Walker, Agile Application-Aware Adaptation for Mobility. Operating Systems Review (ACM) Vol. 51, Number 5, December 1997: 276-287.

[19] Brian Noble, System Support for Mobile, Adaptive Applications. IEEE Personal Communications Magazine, Vol. 7, Number 1, February 2000: 44-49.

[20] David Andersen, Deepak Bansal, Dorothy Curtis, Srinivasan Seshan, Hari Balakrishnan, System Support for Bandwidth Management and Content Adaptation in Internet Applications. Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation, USENIX Assoc., Berkeley, CA, October 2000: 213-226.

[21] D. Li, Sharing Single User Editors by Intelligent Collaboration Transparency. Proceedings of the Third Annual Collaborative Editing Workshop, ACM Group. Boulder, Colorado. 2002, available at http://www.research.umbc.edu/~jcampbel/Group01/Li_iwces3.pdf. August 2002.

[22] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, Change Detection in Hierarchically Structured Information. Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM Press, New York, NY, 1996: 493-504.

[23] OpenOffice.org. Xmerge Project. http://xml.openoffice.org/xmerge/. August 2002.

[24] E. Myers, An O(ND) Difference Algorithm and its Variations. Algorithmica, Vol. 1, 1986: 251-266.

[25] SourceGear Corporation. Abiword: Word Processing for Everyone. Software available at http://www.abisource.com/. August 2002.

[26] Microsoft Corporation. Microsoft Windows CE .NET Storage Manager Architecture. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcemain4/htm/cmconstoragemanagerarchitecture.asp. August 2002.

[27] OpenOffice.org. OpenOffice.org Source Project. http://www.openoffice.org/. August 2002.

BIOGRAPHICAL SKETCH

Ajay Kang was born in New Delhi, India. He received his bachelor's degree in computer engineering from Mumbai University, India. He subsequently worked for a year at Tata Infotech, a software and systems integration company based in India, as an associate systems engineer. During this period he worked on a product for financial institutions and programmed in C++ using the Microsoft Foundation Class (MFC) libraries and the Win32 SDK.

During his master's study he was a TA for CIS3020: Introduction to Computer Science and had his first teaching experience in the process.

His interests include mobile computing and algorithms as a result of two excellent classes he took on these subjects at the University of Florida.

On graduation he will join Microsoft Corp. as a Software Design Engineer in Test (SDET).