

PARALLEL XPATH QUERY EVALUATION ON MULTI-CORE PROCESSORS

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAI'I IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

INFORMATION AND COMPUTER SCIENCES

MAY 2012

By
Ben H. Karsin

Thesis Committee:

Lipyeow Lim, Chairperson
Henri Casanova, Philip Johnson

We certify that we have read this thesis and that, in our opinion, it is satisfactory in scope and quality as a thesis for the degree of Master of Science in Information and Computer Sciences.

THESIS COMMITTEE

Chairperson

©Copyright 2012

by

Ben H. Karsin

Acknowledgments

I want to thank my advisors, Dr. Henri Casanova and Dr. Lipyeow Lim, for their encouragement and support. Their editing and instruction has been invaluable, without which I would never have completed this work. I also want to thank my committee members, whose passion and expertise in their respective fields is an inspiration.

I want to thank Starbucks in Manoa for providing me with a place free of distractions and with plenty of coffee.

I also want to thank my friends and girlfriend for keeping me sane. Their presence in my life inspires me to strive for excellence.

Finally, I'd like to thank my family for their love and support. Their encouragement enables me to conquer anything.

Abstract

XML and the XPath querying language are global standards that are used in industrial settings. The high latency of queries over large XML databases remains a problem for many applications. While this latency could be reduced by parallel execution, issues such as work partitioning, memory contention, and load imbalance may diminish these benefits. To our knowledge, no previous attempt has been made to analyze the performance of multiple parallelization techniques on XPath queries. In this paper, we model the behavior of XPath queries and analyze the benefits of several parallel algorithms. Using synthetic XML databases, we model the query execution time using three quantifiable machine-dependent parameters. Our performance model allows us to estimate a lower-bound of parallel execution time for arbitrary queries and databases on a given hardware environment. We propose five distinct XPath "query engines", including four parallel engines, and compare their performance on synthetic and real-world XML databases. These engines attempt to solve the issue of load-balancing while minimizing sequential execution time. We find that load-balancing is easily achieved for most synthetic data sets, though results on real-world data sets are inconsistent. Our two dynamic query engines (Dynamic Work Queue and Producer-Consumer) achieve the best results on all tests, with Producer-Consumer achieving near-ideal speedup on some queries on real-world data sets. Across two distinct multi-core hardware environments, we find that our models accurately estimate both sequential and parallel execution time of synthetic data sets, though our parameter estimation is not sufficiently precise to prove accurate estimates for real-world data sets.

Table of Contents

Acknowledgments	iv
Abstract	v
List of Tables	viii
List of Figures	x
1 Introduction	2
2 Related Work	6
3 XPath Queries and Performance Models	9
3.1 XML and XPath	9
3.1.1 XPath and XML Document Trees	9
3.1.2 XPath Queries	11
3.2 Sequential Performance Model	12
3.2.1 XPath Query Process	12
3.2.2 Sequential Model Definition	12
3.3 Parallel Performance Model	14
3.3.1 Benefits of Parallelization	15
3.3.2 Parallel Model Parameters	15
3.3.3 Parallel Model Definition	15
4 Evaluation of Performance Models	19
4.1 Test Environments	19
4.2 Synthetic Workloads	19
4.3 XPath Evaluation Engines	21
4.3.1 Sequential Query Engine Details	22
4.4 Instantiation of the Model	23
4.4.1 Measuring β	23
4.4.2 Measuring τ	24
4.4.3 Measuring α	26
4.4.4 Determining $f(L)$	27
4.5 Evaluation of Sequential Model	29
4.5.1 Depth (D)	30
4.5.2 Branch Factor (B)	31
4.5.3 Selectivity (S)	31
4.5.4 Tag Length (L)	32
4.6 Evaluation of Parallel Model	33
4.6.1 Fixed Work Allocation	34
4.6.2 Work Queue Implementation	34
4.6.3 Context Depth (C)	36
4.7 Conclusions	37
5 Unbalanced Synthetic Trees	38
5.1 Randomized Selectivity	38
5.2 Match Skew	39
5.3 Branch Factor Skew	43
5.4 Conclusions	46

6	Evaluation with Real Workloads	47
6.1	Data Sets	47
6.2	Initial Results	48
6.2.1	DBLP	48
6.2.2	XMark	49
6.2.3	xOO7	51
6.3	Weaknesses of the Performance Model	53
6.4	Improving the Performance Model	55
6.5	Conclusions	56
7	Parallel Performance Improvements	58
7.1	Dynamic Work Queue Engine	58
7.1.1	Implementation Details	58
7.1.2	Strengths and Weaknesses	59
7.2	Producer-Consumer Engine	61
7.2.1	Implementation Details	61
7.3	Parameter Calibration	62
7.3.1	Dynamic Work Queue Parameters	62
7.3.2	Producer-Consumer parameters	67
7.4	Empirical Results	69
7.4.1	Branch Factor Skew Revisited	69
7.4.2	DBLP	70
7.4.3	XMark	72
7.4.4	xOO7	73
7.5	Conclusions	74
8	Conclusion	76
	Bibliography	78

List of Tables

<u>Table</u>	<u>Page</u>
3.1 Important parameters used throughout this thesis. For a more complete definition of a parameter, see the section where it is defined.	17
4.1 Description of the hardware test environments used for experimentation.	20
4.2 Fixed Work engine vs. Xalan query engine. Runtimes on 20D Binary tree with 4 threads on the Greenwolf environment	21
4.3 Matching vs. Non-Matching leaf-nodes on dual-core Navet server.	24
4.4 Linear fits of τ and α on our two hardware environments. While α is much larger than τ , τ dominates execution time with large tag lengths.	26
4.5 Results of tests to determine the impact of tag mismatches on comparison time. Run on Greenwolf with a D10 B5 tree.	28
6.1 Query execution runtimes and performance model estimates using the DBLP data set on the Greenwolf environment. We see poor parallel speedup for most queries and very inaccurate performance model estimates. Details about each query are listed in Table 6.5	49
6.2 Measured execution times and performance model estimates for two queries run on 3 different xOO7 data sets. We see good parallel speedup on all queries but completely incorrect performance model estimates. Speedup shown is the average speedup for both parallel query engines. The "small" data set is 4.4MB, "medium" is 44MB, and "large" is 128MB.	52
6.3 Table of depth-specific average parameters for two queries on the large xOO7 data set. The large variance between individual S and $B(q)$ values and the average is the cause of many errors in performance model estimates Note that the query-specific branch factor, $B(q)$ is listed for each of the two queries. Details about each query can be found in Table 6.5.	53
6.4 Estimates generated by the new performance model. Note that this performance model uses level-specific parameters to count the number of nodes evaluated by a query. Estimates are much more accurate than with the previous performance model.	55
6.5 Details about each query for real-world data sets. Note that for XMark, each query is run on several different data sets, each returning a different number of query matches. Likewise, x007 queries are duplicated on a large (128MB, medium (44MB), and small (4.4MB) data set.	57
7.1 List of parameters used by the Dynamic Work Queue and Producer-Consumer query engines. For a more complete definition of a parameter, see the section it is defined in. The impact of each parameter on query execution time is measured in Section 7.3.	61
7.2 Query execution times of our give XPath query engines using the DBLP data set on the Greenwolf environment. We see poor parallel speedup for most queries, though the Producer-Consumer engine performs well on Q_{dblp2} . Details about each query are listed in Table 6.5	71

7.3	List of depth-specific selectivity values and total query matches for each query run on the DBLP data set. We postulate that the relatively higher selectivity of Q_{dblp2} is the reason we achieve much better parallel performance with it.	71
7.4	Results of running two additional queries on the DBLP data set. We identified these two queries as ones that we expect our parallel XPath query engines to perform well on. DWQ is Dynamic Work Queue and P-C is Producer-Consumer.	72
7.5	Results of running 2 queries (details in Table 6.5) on 3 different data sets on the Greenwolf environment. Results listed for our 5 XPath query engines. We see that our two new XPath query engines achieve good parallel speedup using larger data sets. However, the Dynamic Work Queue engine shows slower execution times on very small data sets due to overhead. Note that results with the Work Queue engine are similar to Fixed Work but were omitted for space. The "small" data set is 4.4MB, "medium" is 44MB, and "large" is 128MB.	74

List of Figures

<u>Figure</u>	<u>Page</u>
3.1 Example tree used to illustrate XML trees and XPath queries.	10
4.1 The asymptotic nature of τ as the tag length increases. This is due to the diminishing relative contribution of other parameters (such as α). With a large enough tag length, we assume the runtime is completely dominated by τ and calculate it directly from query execution time.	25
4.2 The τ parameter is linearly dependent on query depth D . Using linear regression, we compute a linear fit of τ to the experimental data, which is plotted on the graph.	25
4.3 The α parameter is dependent on D . Using linear regression, we compute a linear fit of α to the experimental data, which is plotted on the graph.	27
4.4 A simple example of a test to measure the impact of the parameter $f(L)$. We determine $f(L)$ by the difference in execution time of the Q_{match} and $Q_{mismatch}$ queries.	28
4.5 The effect of D on query execution time on Navet. Comparison of test results to performance model estimates for three different tree shapes.	30
4.6 Query execution time is linearly dependent on B . Performance model estimate is linear and test results show a linear dependency as well. Test results are averaged over 100 runs and error bars show min and max values. Most error rates are very small, except the case of $B=170$ on this graph. We assume this is an anomalous occurrence.	31
4.7 Average query execution time while increasing S . Model estimate and runtime both exhibit exponential behavior. The rate of exponential increase is dependent on D	32
4.8 The effect of L on execution time for two different tests. Both model estimate and test results display a linear pattern. Slope and intercept are based on τ and XML document tree and query parameters.	33
4.9 Comparison of model estimate and test execution time while varying work unit count. The 15-30% overhead from the use of the work queue is visible.	35
4.10 Comparison of performance model estimate with empirical results using the Fixed Work engine, while varying context depth. The Fixed Work query engine performs well on these balances synthetic tests.	36
5.1 The impact of adding a random range (R) to selectivity for each node of the tree. Average over 100 runs (re-creating the XML file for each run) on Navet with a D6 B20 T1 synthetic tree. We see that the mean execution time remains stable, but the error rate increases as we increase R	39
5.2 Example of what a match skewed, left-weighted tree may look like with a query of <code>"/A/B/C/D/..."</code> . All query sub-matches occur on only the S left-most children of each node while all other nodes (denoted "X") are mismatches. Parameters for this example are B4 S2 T1 and $I=0$	41
5.3 The impact of match skew and heavy branch placement on Greenwolf. Test is performed while increasing I so that each position from left-weighted to right-weighted is tested. We see that match skew has no significant impact on sequential or parallel execution time. Note that the x-axis is the heavy branch index, I	42

5.4	An example of how the Fixed Work engine distributes work for match skewed trees, using query <code>"/A/B/C/D/..."</code> . Heavy children are sub-match nodes while light children are mismatches (denoted as "X"). Although the example tree is completely left-weighted ($I = 0$), we see an even work distribution.	42
5.5	Using query <code>"/A/B/C/D/..."</code> , an example of what a branch factor skewed, left-weighted tree ($I = 0$) may look like and the Fixed Work engine would distribute work. Notice that light children are now query sub-matches, though they have no children. This means they are still counted for parallel work distribution. Parameters for this tree are B4 S4 T1, I=0 and H=2.	43
5.6	Results of the impact of branch factor skew on query execution time on Greenwolf. While there is no significant impact on sequential execution time, the Fixed Work engine shows dips and spikes in execution time at certain values of I . Note that the x-axis is the heavy branch index, I	44
5.7	The results of testing the impact of branch factor skew on the dual-core environment, Navet. We see only one dip in execution time with the fixed work allocation query engine. This helps us explain the reason for the dips and spikes we see on Greenwolf in Figure 5.6. Note that the x-axis is the heavy branch index, I	45
6.1	Average execution time of Q_{xmark1} on Greenwolf with each of our three XPath query engines. Execution time increases linearly as we increase the XMark scaling factor. We see good parallel speedup on Greenwolf for all queries of on XMark data sets. Details about Q_{xmark1} and other XMark queries available are in Table 6.5.	50
6.2	Average query execution time of Q_{xmark1} on Navet while increasing the number of threads for both the Fixed Work and Work Queue query engines. We see that, although Navet has 2 cores, using 2 threads does not yield good parallel speedup. We determine the ideal number of threads to be 4 for Fixed Work and 8 for Work Queue. Details about the query Q_{xmark1} are available in Table 6.5.	51
6.3	Empirical execution time and performance model estimates for Q_{xmark1} executed sequentially on both Navet and Greenwolf. Performance model estimates increase exponentially at a very fast rate. The scale between performance model estimates and execution times is several orders of magnitude.	52
7.1	The combined impact of the parameters N_{read} and F_{write} on query execution time using the Dynamic Work Queue engine. We see that both parameters have an impact on query execution time, and that the best configuration for our test environments is $N_{read} = 1$ and $F_{write} = B$. This result is corroborated by our other tests.	65
7.2	The impact of the N_{write} parameter on query execution time using the Dynamic Work Queue engine. We see that N_{write} has no significant impact on query execution time. Results shown for several values of N_{read} and F_{write}	66
7.3	The impact of the work depth parameter (W_{depth}) on query execution time using several synthetic tests. We see that, at low values, execution time is erratic, followed by a gradual increase as W_{depth} increases. The fastest runtime is obtained when $W_{depth} \sim \frac{D}{2} - 1$	66
7.4	The impact of the parameters N_{read} and F_{write} on query execution time for the Producer-Consumer engine. We see that the ideal parameter configuration for our test environments is $N_{read} = 1$ and $F_{write} = 1$	67
7.5	The impact of the parameter N_{keep} on query execution time with the Producer-Consumer engine. Results shown for several synthetic trees. All results are obtained on complete trees with all matching nodes, thus we see that a larger N_{keep} value prevents consumer threads from getting work. We revisit this parameter in Section 7.4	68

7.6	The results of repeating queries on branch factor skewed trees with our two new XPath query engines. Results also shown for query execution time using our three previous XPath query engines. We see that our two new query engines perform significantly better than previous engines. Both achieve good parallel speedup and do not suffer from the load imbalance our previous parallel engines experienced.	70
7.7	Demonstration of how low-selectivity queries on the DBLP data set can cause load imbalance. We see that the producer thread must perform more work than each consumer thread. This is an unavoidable consequence of the shape of the DBLP data set and the corresponding query selectivity.	72
7.8	Results of running $Q_{xmark}1$ on a range of XMark data sets on the Greenwolf environment. We see that our two new XPath query engines achieve significantly better parallel speedup than our previous engines.	73

Chapter 1

Introduction

XML is a standardized method of encoding data in a hierarchical structure. Once parsed, an XML document can be represented in memory as a rooted tree. These tree structures can be queried for specific elements using XPath, the XML Path language. XPath queries use path notation to navigate through tree representations of XML documents and can be embedded in host languages such as XQuery, XSL, and SQL. With the abundance of web services being developed, XML and XPath are becoming widely used industry standards.

With the increased proliferation of XML for large-scale applications, it is becoming important to maximize XPath query performance. There has been extensive work toward optimizing performance of XML file parsing [14] [16]. However, there have been relatively few studies focused on optimizing XPath query evaluation by making use of the underlying processor architecture. Several studies have attempted to leverage parallelization for XPath query evaluation [11] [7], though results are inconsistent on large real-world data sets and XML benchmarks [15]. This issue has become increasingly important due to the recent trend toward multi-core processors and multi-processor architectures. Currently, most commercially available desktops and laptops are multi-core. Desktop machines support up to two 6-core processors, i.e., 12 cores. Most state-of-the-art XPath processing libraries, such as Apache Xalan, can support concurrent XPath queries (i.e., multiple threads issuing XPath queries simultaneously against the same Xalan instance) However, each XPath query is still executed sequentially. While this concurrent evaluation can increase overall throughput, it provides no improvement in query latency, which can be a limiting factor when querying large XML documents. Concurrent evaluation of XPath queries can even *increase* individual query latency due to sharing of limited resources (i.e., memory).

The problem of high query latency can be alleviated through parallel query evaluation. If a query can efficiently be split into several computationally independent tasks, all processor cores can work concurrently on a single query. This can potentially decrease the single query latency by a factor equal to the number of processor cores. In addition, the efficient hardware allocation of parallel query evaluation may provide throughput gains similar to that of concurrent query evaluation.

While the theoretical benefits of parallel evaluation are compelling, practical factors may diminish the performance. Issues such as memory contention, load imbalance, and limited cache can be detrimental to the performance of any parallel application. To determine the effects of poor paralleliza-

tion, it is important to model the theoretical behavior and compare it to empirical results. Performance models have been developed for many applications, though, to our knowledge, no studies have attempted to model XPath query execution time.

In this study, we first develop theoretical performance models of the sequential and parallel execution times of XPath queries. We define these models by a set of parameters that are dependent on:

- the XML Document - Parameters such as tree depth and average branching factor,
- the XPath Query - Parameters such as query length and average number of matches, and
- the Compute Platform - Parameters such as average time to compare two strings.

Using these parameters, we first create a model to estimate the sequential execution time of XPath queries on an XML document on a given hardware environment.

We extend our sequential performance model to estimate *parallel* execution time by introducing several new parameters (e.g., number of processor cores). Parallel execution of XPath queries is performed with an initial *sequential* stage followed by a *parallel* stage. During the sequential stage, the first levels of the XML document tree are queried to obtain computationally independent work units. These work units are then processed in parallel by all processor cores. The parallel performance model incorporates the performance implications of each stage of execution but ignores the effects of poor parallelization. The parallel model thus estimates the ideal performance of a given query on a specific hardware environment.

To determine execution efficiency and evaluate model accuracy, we develop a custom query processing engine that performs a conditional tree-traversal over an XML document. This query engine uses an XPath query to evaluate which tree paths to traverse and which to ignore. Our query engine is designed to be simple, correct, and parallelizable.

We initially propose two parallel query engines using the custom sequential implementation we developed as a base: (1) Fixed Work and (2) Work Queue. These two query engines differ in their methods of distributing work among processor cores. The Fixed Work engine evenly distributes all computationally independent work units among available processor cores statically, prior to parallel computation. The Work Queue engine allows processor cores to dynamically request work units as needed. Since the Work Queue engine dynamically distributes work, it attempts to alleviate load imbalance. However, the cost of maintaining and locking the shared work queue results in computational overhead.

We evaluate our sequential and two parallel query engines using various synthetic tests and real-world data sets. These synthetic tests consist of custom-made XML documents and corresponding XPath queries designed to measure the contribution of each performance model parameter individually. To test the performance of our query engines on real-world data sets, we use a set of XPath queries over one large real-world XML document (DBLP) and two XML benchmarks (XMark and xOO7). Experimental results demonstrate that our performance model is able to accurately estimate the execution time of XPath queries on our hardware environments for synthetic data sets. Although our parallel

performance model ignores the effects of poor parallelization and work queue overhead, we obtain accurate estimates of parallel performance and speedup for most synthetic tests. However, our data and query-dependent parameters (e.g., selectivity and branch factor) lack sufficient granularity to accurately estimate query execution time for real-world data sets. Our results indicate that our parallel implementations achieve significant speedup, with near-linear performance increase as we increase hardware resources (processor cores) for most synthetic and real-world data sets. Several weaknesses of our parallel query engines, however, become apparent. We see that certain aspects of skew that is typical of some real-world data sets cause load imbalance and degrade parallel performance of our two parallel query engines, especially the Fixed Work engine.

In an attempt to improve on the shortcomings of our two parallel query engines, we propose two additional parallel query engines: Dynamic Work Queue and Producer-Consumer. The Dynamic Work Queue engine extends the Work Queue engine by allowing all threads to dynamically read from and write to the shared work queue. This further balances the workload among processor cores by allowing threads to work on portions of work units and write unfinished work back to the queue. However, we note a computational overhead associated with frequent writes to the shared work queue. The Producer-Consumer engine attempts to reduce this overhead by designating one Producer thread that performs small sub-queries and writes results work units to the shared work queue. All other threads are Consumers and simply read from the shared work queue and evaluate work units. Both of these new parallel XPath query engines improve over our previous query engines by completely eliminating the need for sequential context processing prior to parallel evaluation. Our results indicate that these new query engines significantly improve performance over our previous engines, especially on XML benchmarks (XMark and xOO7).

Our study makes the following contributions:

- To the best of our knowledge, there have been no previous attempts to model the execution behavior of XPath queries either running sequentially or in parallel. We develop a performance model that can be used to estimate the performance of XPath queries based on the hardware environment.
- Through our analysis of XPath query execution, we identify parameters that encapsulate the query process. We develop a small software package that measures these parameters for a given hardware environment. Using these values and our performance model, we are able to accurately estimate XPath query execution time.
- While there are several studies dedicated to parallelization of XPath queries, few propose query engines that perform inter-query parallelization. In response, we propose four distinct parallel XPath query engines: (1) Fixed Work , (2) Work Queue, (3) Dynamic Work Queue, and (4) Producer-Consumer.

- We verify the performance of our XPath query engines on a range of synthetic and real-world data sets, including one large real-world data set (DBLP) and two well-known XML benchmarks (XMark and xOO7).
- Our experiments determine that XPath queries can be parallelized efficiently. We identify key parameters specific to parallel execution and evaluate their impact on query execution time. The parameters that contribute the most to overall query performance are incorporated into our parallel performance model.

The remainder of this thesis is organized as follows: Chapter 2 reviews the related work in XML performance and parallelization. In Chapter 3 we introduce XML documents and XPath queries and also define our sequential and parallel performance models. Chapter 4 contains our evaluation of our performance models with respect to empirical results obtained using our sequential and two parallel implementations when querying synthetic data sets. In Chapter 5 we briefly discuss the effect of heavily skewed synthetic datasets on query execution time and performance model accuracy. In Chapter 6 we present the performance results of queries of real-world datasets and benchmarks using our sequential and two parallel querying methods. In Chapter 7 we propose two additional XPath query engines and present results of experiments with them. Finally, in Chapter 8 we present conclusions and discuss future work.

Chapter 2

Related Work

With the proliferation of more advanced computer hardware, including multicore and multi-processor architectures, the field of High-Performance Computing (HPC) has grown greatly over the past 10 years. There has been increasingly more research done in this field to try and effectively exploit multicore architectures to improve application performance. The research in this thesis is at the intersection of the fields of Parallel processing (HPC), XML databases, and performance modeling. Our goal is to apply methods from HPC and performance modeling to better understand and improve the performance of query evaluation on XML databases.

Although much of the current research done in the field of HPC is beyond the scope of this thesis, we employ several well-known HPC techniques and concepts, detailed in [17], to improve performance of XPath query evaluation. In [17], the authors provide a broad overview of the field of HPC and also provide details to many concepts used in this thesis (e.g., parallel speedup, load balance, and locking).

Orthogonal to our work, though employing many of the HPC concepts we use, is parallel evaluation of relational database queries. Modern relational databases are increasingly complex and there is a demand for high-throughput and low-latency query evaluation methods. The work in [8], published over 20 years ago, evaluates the efficacy of parallel evaluation of relational database queries. They concluded that query evaluation can attain significant performance benefits from parallelization, even with the hardware available in 1990. Since then, there has been many advances in computer hardware and parallel computing, though many of the techniques developed in the 1990s are still used. Graefe et al. [9], published in 1993, contains a detailed survey of large-scale database systems and techniques to improve performance. In [9], the authors describe several parallel query evaluation methods that are still used to improve query evaluation performance. Some of these methods of parallel processing (e.g., the bracket model), though able to achieve significant performance improvements over sequential evaluation, exhibit high overhead from inter-process communication. The recent work of [19] proposes an improved model for parallelization of relational database queries that reduces the amount of inter-process communication and is extensible to a range of hardware architectures. However, [19] contains no empirical data to validate their methods. Additionally, [10] provides a recent analysis of frequently used relational query processing algorithms and proposes an analytical model to estimate parallel performance. They find that

estimates from their analytical model match parallel query performance, and that they achieved significant parallel speedup with most algorithms. Although many of the concepts employed for parallelization of relational database queries are related, the focus of this thesis is hierarchical database (i.e., XML) queries.

XML (eXtensible Markup Language) is a standardized format that is used in many domains, including web applications. Many real-world applications utilize XML databases to maintain large amounts of data, such as the DBLP database [12]. This open-source XML database contains data relating millions of academic research articles, organized in a hierarchical format. In addition to real-world data sets, the work of [15] and [18] provide details of several XML database benchmarks. The first paper gives an overview of XML database benchmarks that are currently available. The second paper outlines the details of the XMark benchmark. We use the real-world DBLP data set and two XML benchmarks for evaluation of our XPath query engines in Chapter 6.

These large XML databases necessitate the use of efficient XML parsing and loading algorithms. Current XML parsing algorithms are memory and CPU-intensive, so there is extensive research to improve performance. The work in [14] introduces a parallelization method for parsing large XML files. Their method uses a sequential preparsing phase to determine XML file structure, followed by a full parallel parse. They achieve significant parallel speedup using this technique on multicore environments. Similarly, in [16], the authors employ a preparsing phase and claim that a static load-balancing algorithm provides the best performance. Their method achieves good parallel speedup on test environments using up to 8 cores. Although XML document parsing is related to the work of this thesis, we focus our efforts on improving query evaluation performance.

For the work of this thesis, we use XPath, a language used to query XML documents that have been parsed and loaded into memory. Queries executed on large XML documents, such as DBLP [12], can be computationally intensive, especially when queried sequentially. The work in [20] attempts to improve query throughput by exploiting parallelization on multicore hardware environments. They incorporate large numbers of queries into a single NFA (Non-determinist Finite Automata), which they fragment to enable parallelization. The results in [20] indicate that they achieve significant throughput improvement over previous methods. Although this method increases overall query throughput, it does not reduce single-query latency which is the focus of our efforts.

The work in [7] aims to increase query throughput as well as reduce query latency. They decompose XPath queries into several smaller subqueries for parallel execution and merge results. [7] propose parallelization methods for XPath queries on both distributed cluster systems and multicore environments. They obtain significant speedup using all methods, though there are concerns about redundant processing of query nodes. Although query decomposition results in several smaller subqueries, portions of each subquery may require duplicate processing. Additionally, the process of merging results of subqueries can be computationally costly, especially if there are many query matches. The XPath query engines proposed in this thesis avoid these problems by implementing inter-query parallelism through distribution of XML document tree nodes to processor cores.

Another method aimed at reducing query latency is proposed in [11]. They propose an XPath querying method using a series of data structures that index nodes of the XML document tree. These indexes aim to improve sequential performance and identify parallelizable components of the XPath query. The results obtained in [11] are impressive for sequential execution, though parallel performance is inconsistent for their tests. One concern is the computational overhead from the locking of data structures to ensure mutual exclusion. The tests performed in [11] utilize a 24-core machine with 96GB of memory, though the memory requirements of maintaining the large data structures may be problematic for lesser hardware environments. The XPath query engines proposed in this thesis use limited additional memory, aside from a fixed-size shared buffer used by the query engines introduced in Chapter 7. We would like to compare the performance of the methods used by [11] and [7] with that of our XPath query engines by duplicating their tests, though we do not have identical hardware at this time.

A portion of the work done in this thesis (inter-query parallelization of XPath queries) builds on the work of [4] and [5]. The first paper performs an overview of the problem of parallelization of XPath queries and performs preliminary experiments with the Xalan query engine. The second paper compares results using data, query, and hybrid partitioning techniques, with limited success. We extend this work by introducing several query engines that utilize data-level partitioning to achieve parallelization.

In this thesis, we also analyze the XPath query evaluation process and propose two performance models: sequential and parallel. While the area of performance modeling of parallel applications is extensively studied, to our knowledge no studies have attempted to model parallel execution of XPath queries. Performance of parallel applications tend to be complex, with many influencing factors (e.g., resource contention). This is especially the case for distributed (cluster) systems, as [6] and [2] demonstrate. [6] proposes an analytical model that uses distributed memory management and page faults statistics to estimate the performance of applications across cluster systems. They obtain good results, with accurate model estimates for a range of tests. The work in [2] is concerned with modeling the performance of heterogeneous cluster systems to predict the performance of parallel applications on a range of variant hardware platforms. This study achieves good results, with their performance model being able to accurately predict the performance of several parallel applications. These papers focus their efforts on performance of distributed systems, where performance is dependent on memory allocation and management. Our modeling and analysis is performed for shared-memory, multicore systems, where CPU utilization is the primary concern.

Chapter 3

XPath Queries and Performance Models

In this chapter, we formally define XML Document trees and create an analytical model of the sequential and parallel execution times of XPath queries. We begin by providing some background information on the technology used, such as XML and XPath. We explain how XPath queries are executed and define several parameters that describe the process. We use these parameters to create a performance model of total query execution time in Section 3.2. In Section 3.3 we use additional parameters to extend our performance model to estimate optimal parallel execution time.

3.1 XML and XPath

XML (eXtensible Markup Language) is a flexible, self-describing method of encoding data in a hierarchical structure. An XML file is made up of *tags* and *data*. Tags are defined in opening/closing pairs. We refer to an open/close tag pair simply as a tag. Tags can be *nested*, allowing each tag to contain other tags and/or data. This nesting of XML tags produces an hierarchical structure that allows each tag to define its own hierarchical subset of the entire XML file.

3.1.1 XPath and XML Document Trees

One method of processing XML files is XPath, the XML Path Language. XPath is a standardized language that is used to execute queries on XML Document *trees*. These trees are generated by modules that parse XML files, load them into memory, and generate XML tree structures. For this work, we use the Xerces XML parsing module to perform this task.

Xerces is a C++ library that includes a set of XML file parsing and loading methods that can be used to create document trees in memory. It forms a tree by assigning XML tags to *nodes* of the tree, with tag names as identifiers. Each node contains whatever data is contained in its tag, and has children nodes corresponding to tags nested within it. As a simple example, consider the following XML file:

```
<A>
  <B>
    <C></C>
    <D></D>
```

```

</B>
<E>
  <F></F>
  <F></F>
</E>
</A>

```

We have 8 tag pairs, with the ”/” tags being the closing tags. Using Xerces, we create a tree in memory with 8 nodes that can be visualized as shown in Figure 3.1. Note that duplicate tag names are allowed in XML files.

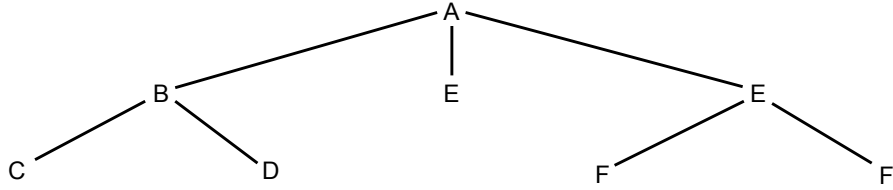


Figure 3.1. Example tree used to illustrate XML trees and XPath queries.

Formally, we define a *tree* T as a directed acyclic graph $T = (N, E)$, where N is a set of *nodes* and E a set of *edges*, where an edge connects two nodes. Given two nodes n and m in N , we use the notation $n \rightarrow m$ to indicate that there is an edge connecting n to m . n is called the *parent* of m , and m is called a *child* of n . We denote by n_{root} the root of the tree, i.e., the only node in the tree without a parent. For a node n , we define the set of children nodes of n as $children(n) = \{m \in N | n \rightarrow m\}$. Similarly, we define the parent of m as $parent(m) = \{n \in N | n \rightarrow m\}$. We say that a node n and a node m are on a *path* if $n \rightarrow^+ m$, where \rightarrow^+ denotes the transitive closure of the \rightarrow operator. We define the set of *ancestors* of m as $ancestors(m) = \{n \in N | n \rightarrow^+ m\}$. Likewise, we define the set of *descendants* of n as $descendants(n) = \{m \in N | n \rightarrow^+ m\}$. Note that every node is a *descendant* of n_{root} and n_{root} is an *ancestor* of every node of the tree, i.e., $n_{root} \rightarrow^+ m$, for all $m \in N - n_{root}$, where $-$ is the set difference operator.

We define the depth of a node n , $d(n)$, recursively: $d(n_{root}) = 0$ and $d(n) = d(parent(n)) + 1$. A node n such that $children(n) = \emptyset$ is called a *leaf*. We use N_{leaf} to denote the set of leaves in the tree. Conversely, we define $N_{non-leaf}$ to be the set of all non-leaf nodes in the tree, i.e., $N_{non-leaf} = N - N_{leaf}$. We define the lineage of a node n with depth $d(n)$ as the sorted set $lineage(n) = \{n_1, \dots, n_{d(n)+1}\}$, such that $n_1 = n_{root}$, and $n_i \rightarrow n_{i+1}$ for $i = 1, \dots, d(n) + 1$. The lineage(n) is the set of all nodes on the *path* from n_{root} to n . Note that $lineage(n)$ is the set of *ancestors*(n), ordered by depth. Finally, a *tag* (i.e., a character string) is associated with each node. We denote the tag for node n by $tag(n)$, and use $|tag(n)|$ to denote the tag length.

We now introduce three parameters that describe the overall structure of a tree:

- Branch Factor (B) - The average number of children of non-leaf nodes:

$$\frac{1}{|N_{non-leaf}|} \sum_{n \in N_{non-leaf}} |children(n)|.$$

- Depth (D) - The length of the longest path in the tree:

$$\max_{n \in N} d(n)$$

- Average tag length (L) - The average number of characters per tag.

$$\frac{1}{|N|} \sum_{n \in N} |tag(n)|$$

3.1.2 XPath Queries

An XPath query consists of a sequence of tag names that corresponds to zero or more paths down the tree. Formally, for a given tree $T = (N, E)$, we define a query q as the ordered set $\{q_1, \dots, q_k\}$, where each q_i is a tag string. The query tag at each depth i is denoted q_i . The query result is then the set:

$$N_{match}^q = \{n \in N | lineage(n) = \{n_1, \dots, n_k\} \text{ and foreach } i = 1, \dots, k \text{ tag}(n_i) = q_i\}$$

We define three additional subsets of nodes, based on the query q :

$$\begin{aligned} N_{sub-match}^q &= \{n \in lineage(m) | m \in N_{match}^q \text{ and } n \notin N_{match}^q\} \\ N_{non-match}^q &= \{n \in children(m) | m \in N_{sub-match}^q \text{ and } tag(n) \neq q_{d(n)}\} \\ N_{unvisited}^q &= \{n \in children(m) | m \in N_{non-match}^q \cup N_{unvisited}^q\} \end{aligned}$$

Note that the sets N_{match}^q , $N_{sub-match}^q$, $N_{non-match}^q$, and $N_{unvisited}^q$ are mutually exclusive and their union is the set of all nodes of the tree.

Using this method of node categorization, we define the *selectivity* of a query, S , as:

$$S = \frac{1}{|N_{sub-match}^q|} \sum_{n \in N_{sub-match}^q} |children(n) \cap (N_{sub-match}^q \cup N_{match}^q)|$$

The selectivity (S) of a query q is the average spread of query matches: it measures the average number of matching children per non-leaf match (sub-match) node, averaged over the entire tree T .

As an example, we run the following query on the tree shown in Figure 3.1:

”A/E/F”

This query results in 2 match nodes (the two ”F” nodes) and an average selectivity of 4/3 (”A” has 2 matching children, the non-leaf ”E” has 2, and the leaf ”E” has 0). This query has 3 sub-match nodes (”A” and the two ”E” nodes) and one non-match node (”B”). The ”C” node is unvisited because it is never evaluated in the query. Note that large portions of the tree may be unvisited for a given query.

3.2 Sequential Performance Model

As seen in the previous section, the performance of an XPath query over an XML database is dependent on many factors. No single element of an XML document or XPath query lets us estimate query execution time. To the best of our knowledge no previous attempt has been made to model XPath query behavior across multiple hardware environments. In this section, we define several parameters and create an analytical model of sequential XPath query execution time. Using this performance model, we hope to gain insight into performance bottlenecks and methods of improving overall query performance. To achieve an accurate model, we must clearly understand the XPath query process and outline the major factors that contribute to query execution time.

3.2.1 XPath Query Process

The XPath query operation can be seen as a selective tree-traversal using the query string to determine which branches to traverse. The process begins at the root node and compares the root tag value with the first value of the query path. If it is a match, the operation moves to each child of the root. The tag value of each child is compared to the second element of the query path. The children of these matching nodes are then compared to the third tag value, and so on. This operation is performed recursively on every sub-match node (as defined in Section 3.1.2). Each node that has a *lineage* that matches the entire query is a match and is saved to be reported to the user.

We divide the query process into three subtasks: tag comparison, tree traversal, and match reporting. We define three atomic, machine-dependent parameters that correspond to each type of task, respectively:

- τ : The time to compare a single character of the tag; a tag of L characters takes at most $\tau \times L$ seconds.
- α : The time to perform all general processing of a single node, such as pointer chasing.
- β : The additional time to process a query match (saving and reporting).

We estimate query execution time for specific hardware environments using these three parameters. Additional contributing factors that scale in line with one of the parameters will also be captured by it. For example, if additional processing is required for every node of the tree, it will be included within the α parameter. As long as each computationally significant task scales with one of the three parameters, the model will be accurate.

3.2.2 Sequential Model Definition

Using our three parameters (τ , α , and β), we estimate the total time to evaluate a single node. For every visited node, the tag is compared to the query and additional node computation is performed (including pointer chasing). However, matches and mismatches differ in total computation time. When

the tag value matches the query, all characters of the tag are compared. When we have a mismatch, comparison can halt once the first mismatched character is found. This can occur at any point during the tag comparison, thus we introduce the function $f(L)$ to be the average number of characters compared on a tag mismatch. We estimate the total time to evaluate a single (non- N_{match}^q) node as:

$$\begin{aligned} \text{Match node} & : \tau \times L + \alpha \\ \text{Mismatch node} & : \tau \times f(L) + \alpha \end{aligned}$$

Note that α acts as a "catch-all" for any additional computation performed at every node.

We estimate the total execution time for each type of node described in Section 3.1.2, ignoring unvisited nodes (i.e., elements of $N_{unvisited}^q$) since they are never visited and do not impact query execution time. Each node type differs slightly in computation time. Sub-match nodes require comparison of each character of the tag and have the additional computation cost of α . Each query match node requires the added time associated with β . Since non-match nodes have tags that do not match the query, we only compare $f(L)$ characters. We therefore estimate the per-node computation time as:

$$\begin{aligned} T_{match} & = \tau \times L + \alpha + \beta \\ T_{sub-match} & = \tau \times L + \alpha \\ T_{non-match} & = \tau \times f(L) + \alpha \end{aligned}$$

We formulate our model by counting the total number of nodes of each type for a given query and XML Document tree. By applying known results for the node counts of tree structures, we get:

$$\begin{aligned} N_{total} & = N_{sub-match} + N_{match} + N_{non-match} \\ N_{sub-match} & = \frac{S^{D-1} - 1}{S - 1} \\ N_{non-match} & = \frac{S^{D-1} - 1}{S - 1} \times (B - S) \\ N_{match} & = S^{D-1} \end{aligned}$$

This gives us a general estimated execution time for a given query and tree:

$$\begin{aligned} T_{total} & = N_{sub-match} \times T_{sub-match} + N_{non-match} \times T_{non-match} + N_{match} \times T_{match} \\ & = \frac{S^{D-1} - 1}{S - 1} \times (\tau \times L + \alpha) \\ & + \frac{S^{D-1} - 1}{S - 1} \times (B - S) \times (\tau \times f(L) + \alpha) \end{aligned}$$

$$+ (S^{D-1}) \times (\tau \times L + \alpha + \beta)$$

In Section 4.4 we determine that the contribution of β is negligible on our test environments. Furthermore, our tests reveal inconsistencies in the contribution of β . Removing β yields the following simplified expression for T_{total} :

$$T_{total} = \frac{S^D - 1}{S - 1} \times (\tau \times L + \alpha) + \frac{(B - S)(S^{D-1} - 1)}{S - 1} \times (\tau \times f(L) + \alpha)$$

Note that this is the general case that includes the possibility that $f(L) \neq L$. We discover in Section 4.4.4 that the function $f(L)$ is non-linear and, for small values of L , $f(L) \sim L$. Additionally, for our test cases, we find that $f(L)$ varies from L by no more than 15%, regardless of tag length. Thus, when we use the approximation $f(L) = L$, we can further simplify the model:

$$T_{total} = \left[\frac{B(S^{D-1} - 1)}{S - 1} + 1 \right] \times (\tau \times L + \alpha)$$

Using the above performance model, we revisit the example in Section 3.1 and estimate the computational cost. Recall that applying the query "A/E/F" to the tree in Figure 3.1 results in the parameter values $S=4/3$, $B=2$, $D=3$, and $L=1$. Applying these parameters to our simplified model (for small L values) yields:

$$\begin{aligned} T_{total} &= [2((4/3)^{3-1} - 1)/((4/3) - 1) + 1] \times (\tau \times 1 + \alpha) \\ &= 5.666 \times (\tau + \alpha) \end{aligned}$$

Looking at the example, we see the query visits exactly 6 nodes. At each node, we compare 1 character (regardless of match or mismatch) and perform additional operations (α) at each node. For this example, the estimated required work is off by 5.5%. In the general case, it is difficult to determine model inaccuracy analytically, so we measure it empirically in Chapter 4. For reference, all parameters defined in this section, as well as many important parameters used in subsequent sections of this thesis are detailed in Table 3.1.

3.3 Parallel Performance Model

In the previous section, we defined several parameters and used them to create an analytical model of sequential execution time of XPath queries. In an effort to improve overall performance and

lower single-query execution time, we explore the effect of parallelization of XPath queries. In this section, we extend our sequential performance model to estimate ideal (ignoring poor parallelization effects) parallel execution time. In Chapter 4, we compare this model to empirical results and attempt to identify parallel performance bottlenecks and measure parallel speedup.

3.3.1 Benefits of Parallelization

For many years, we have seen constant and rapid increase of processor speeds for single-core microprocessors. Recently, physical limitations in heat dissipation and power consumption have prevented this trend from continuing. Instead, we are seeing many architectures with multiple processors and processors with multiple cores. The high latencies associated with queries of large XML databases can be alleviated by leveraging these multi-core architectures. When running a single query in parallel, we distribute the workload among several processor cores. While this approach can substantially reduce query latency, parallelization of XPath queries requires an altered querying method, resulting in a more complex performance model.

3.3.2 Parallel Model Parameters

Parallel execution of XPath queries requires a two-phase process. The first phase involves sequential evaluation of the first few tree levels to generate computationally independent work units. The second phase distributes these work units among our processor cores and evaluates them in parallel. We model the first phase using the sequential performance model defined in Section 3.2, though it runs a smaller sub-query. The depth of this sub-query defines a new parameter: context depth (C). C is defined as the depth executed sequentially before parallel execution commences.

Once the sequential phase reaches the context depth (C), the computationally independent work units are distributed to our P processor cores and the remainder of the query is performed in parallel. We define one additional parameter for the parallel performance model: processor core count P . The parameter P is dependent on the hardware environment and dictates the maximum speedup achievable through parallel execution. We explore the performance implications of the parameters C and P in Chapter 4.

3.3.3 Parallel Model Definition

Using these two new parameters and the general sequential performance model defined in the previous section, we define a model of parallel query execution time:

$$\begin{aligned}
T_{total} &= T_{sequential} + T_{parallel} \\
&= \frac{S^C - 1}{S - 1} \times (\tau \times L + \alpha) + \frac{(B - S)(S^{C-1} - 1)}{S - 1} \times (\tau \times f(L) + \alpha) \\
&+ \left(\frac{1}{P}\right) \times \frac{(S^D - 1) - (S^C - 1)}{S - 1} \times (\tau \times L + \alpha)
\end{aligned}$$

$$+ \left(\frac{1}{P}\right) \times \frac{(B - S)((S^{D-1} - 1) - (S^{C-1} - 1))}{S - 1} \times (\tau \times f(L) + \alpha)$$

As with the sequential model, at small values of L we assume $f(L) = L$, giving us the further simplified model:

$$\begin{aligned} T_{total} &= T_{sequential} + T_{parallel} \\ &= \left[\frac{B(S^{C-1} - 1)}{S - 1} + 1\right] \times (\tau \times L + \alpha) \\ &+ \left(\frac{1}{P}\right) \times \left[\frac{B(S^{D-1} - 1) - B(S^{C-1} - 1)}{S - 1} + 1\right] \times (\tau \times L + \alpha) \end{aligned}$$

Note that the sequential phase of execution is modeled by the sequential performance model from Section 3.2 but only to depth C . We estimate the execution time of the parallel phase by applying the sequential model and dividing by the total number of processor cores (P). We then subtract the portion of work that has already been done sequentially by the first phase. This model assumes perfect parallel speedup for the parallel phase and ignores the effects of poor parallelization (e.g. load imbalance, synchronization overhead) that may degrade performance. The parallel model estimate is therefore the ideal parallel performance of an XPath query using the selective tree-traversal algorithm. In Chapter 4 we quantify the discrepancy caused by poor parallelization and attempt to determine its causes.

Parameter	Name	Defined in	Informal Explanation
D	Depth	Section 3.1.1.1	Maximum depth of an XPath query on an XML document.
B	Branch Factor	Section 3.1.1.1	Average number of children per sub-match node.
S	Selectivity	Section 3.1.1.2	Average number of query-matching children per sub-match node.
L	Tag Length	Section 3.1.1.1	Average number of characters per tag string.
τ	String Comparison Time	Section 3.2.1	Time, in seconds, for a hardware environment to compare two characters.
α	Node Processing Time	Section 3.2.1	Time to perform all general processing on a node (besides character comparison).
β	Match Reporting Time	Section 3.2.1	Time for a hardware environment to save and report a query match.
f(L)	Mismatch Function	Section 3.2.2	Average number of characters compared on a query mismatch.
P	Processor Core Count	Section 3.3.2	Number of processor cores on the hardware environment.
C	Context Depth	Section 3.3.2	Query depth executed sequentially before parallel processing begins.
W	Work Unit Count	Section 4.6.2	Number of work- units used by the Work Queue XPath query engine.
R	Selectivity Range	Section 5.1	The range from the mean value that selectivity is randomly selected from.
I	Heavy Branch Index	Section 5.2	Index of heavy children when experimenting with skewed trees.
H	Heavy Child Count	Section 5.3	Number of heavy children per sub-match node.

Table 3.1. Important parameters used throughout this thesis. For a more complete definition of a parameter, see the section where it is defined.

Chapter 4

Evaluation of Performance Models

In the previous chapter we defined sequential and parallel performance models for estimating the execution time of XPath queries on XML databases. These models depend on a series of query-specific parameters (e.g., S and D) and machine-dependent parameters (α , β , and τ). In this chapter, we instantiate these models on two distinct hardware test environments and compare performance model estimates to empirical results. We generate synthetic datasets and queries that evaluate fixed numbers of nodes and have low variability in execution time. Using synthetic data allows us to isolate and measure specific parameters to determine their effects on overall execution time. Our goal is to determine the accuracy and resilience of our performance models and identify the impact of each parameter on total XPath query execution time.

We begin this chapter by describing our testing methodology and hardware environments. In Section 4.3 we define our sequential XPath query engine. We instantiate our performance models by measuring each hardware-dependent parameter in Section 4.4. In Section 4.5 we then evaluate our sequential performance model and the impact of each parameter (D , B , L , S) on execution time. Finally, we propose two parallel XPath query engines and measure the impact of our parallel execution parameters (C and P) on parallel speedup and query execution time in Section 4.6.

4.1 Test Environments

We use two dedicated test environments for experimentation. Table 4.1 describes the characteristics of each environment. Note that Navet is only dual core, so the maximum achievable speedup is 2.0. The Greenwolf environment, however, has 4 cores so ideal speedup is 4.0. This difference has a significant impact on parallel execution time, as seen in Section 4.6.

4.2 Synthetic Workloads

To test the accuracy of our model and determine the effects of each parameter on query execution time, we create a series of synthetic XML documents and corresponding XPath queries. These synthetic tests allow us to precisely control each data- and query-specific parameter. We use these tests

Name	Processor	Clock Speed	Cores	Memory	Cache Size
Greenwolf	Intel Core i7	2.67 GHz	4	12GB	8MB
Navet	Intel Xeon	3.20 GHz	2	4GB	2MB

Table 4.1. Description of the hardware test environments used for experimentation.

to measure each machine-dependent parameter (α , β , and τ) on each test environment and quantify the individual contribution that each parameter has on total query execution time.

In Section 3.1 we defined several parameters that specify XML trees and XPath queries. We now use them to define a notation to describe our synthetic tests. These tree/query tests are described by their average Depth (D), Branch Factor (B), Selectivity (S), and Tag Length (T). For example, D20 B2 S2 T16 defines a 20-level binary tree with tags of 16 characters and a query with all nodes as query matches ($S = B$).

To generate these synthetic data sets, we create a perl script that generates XML files based on the given test parameters (D, B, and T). The script runs recursively to create nested XML tags with tag names composed of consecutively increasing ASCII characters at each nesting level (if T=2, the root would have tag "AA", its children would have "BB", etc.). Algorithm 1 describes the synthetic XML file generation process:

```

function GENERATE(depth) begin
  for TagLength times do
    | tagName = tagName + (asciiValue("A") + depth);
  end
  write "<" + tagName + ">";
  if depth < MaxDepth then
    | for BranchFactor times do
    | | GENERATE(depth+1);
    | end
  end
  write "</" + tagName + ">";
end

```

Algorithm 1: Pseudo-code outlining the process we use to create synthetic XML data sets.

We also create variants of this script to control selectivity. To vary selectivity, we generate $B - S$ tags that do not match the query tag nested within each matching tag. This results in S matching children and $B - S$ mismatching children of each sub-match node of the tree. The matching and mismatching children of each node are randomly shuffled to prevent cache and memory effects from affecting query execution time.

Our synthetic tests allow us to control each parameter while measuring execution time. We then compare the measured execution time with our model estimates based on the parameters used when

Context — Tag	Fixed Work (sec.)			Xalan (sec.)		
	1	2 ⁴	2 ⁸	1	2 ⁴	2 ⁸
3	1.46	3.23	10.86	47.09	49.1	50.28
4	1.34	3.28	10.87	15.4	17.26	16.88
5	1.33	3.44	10.66	3.84	3.59	3.7
6	1.29	3.5	10.75	1.03	1.12	1.11
8	1.28	3.34	10.94	0.21	0.25	0.32
10	1.35	3.29	10.87	0.11	0.12	0.23
12	1.35	3.34	10.84	0.13	0.14	0.36
15	1.4	3.6	11.22	0.17	0.27	1.27
18	1.41	3.81	9.35	0.44	0.57	3.89
19	1.43	3.37	7.25	0.75	0.81	3.91

Table 4.2. Fixed Work engine vs. Xalan query engine. Runtimes on 20D Binary tree with 4 threads on the Greenwolf environment

generating the synthetic tests. We perform and analyze these tests with sequential execution in Section 4.5 and parallel execution in Section 4.6.

4.3 XPath Evaluation Engines

For initial experimentation, we use the Xalan XPath query package. Xalan is an XML processor that we use along with Xerces to transform XML files into XML document trees in memory. Xalan also includes an XPath querying engine that can be used to query these XML document trees. We perform a series of synthetic tests (described in Section 4.2) using this Xalan querier. We find that we are unable to get consistent query execution times using the Xalan query engine.

In an attempt to determine the cause of the experimental inconsistencies we experience, we develop a custom querying engine. The custom XPath query engine (henceforth known as the Sequential query engine) performs a selective tree-traversal over the XML document in an attempt to obtain consistent and reliable results. The details of the Sequential query engine are described in Section 4.3.1.

We duplicate all tests previously run with the Xalan query engine using our Sequential query engine. Additionally, we run a series of tests using the Fixed Work parallel XPath query engine, described in Section 4.6.1 to determine parallel performance when using the Xalan query engine. These tests reveal that the Xalan query engine is the cause of our inconsistent results. By comparison, our custom querier generates consistent results that we would expect. The results of some of our parallel tests using both our Fixed Work engine and Xalan are outlined in Table 4.2.

We see that query execution times using the Xalan engine vary by a maximum factor of 428 (42800%) when varying only context depth. Under the same conditions, the Fixed Work query engine has a maximum variance of 33.2%. In addition, tag length does not affect Xalan queries in a consistent way. In bold-face are results that demonstrate faster execution with significantly longer tag lengths, illustrating the inconsistent behavior of the Xalan query engine.

We speculate that the inconsistent execution times of Xalan queries, particularly during parallel execution, are caused by techniques used to increase sequential query performance. Xalan uses complex data structures, such as lookup tables and indexing, to decrease the runtime of commonly executed queries. This, however, causes parallelization difficulties such as memory contention. The issue of memory contention is dependent on the hardware environment and is difficult to model or estimate. Due to this complex behavior and the inconsistent results we obtain using Xalan, we opt to use our Sequential and other custom XPath query engines for the remainder of experimentation in this thesis.

4.3.1 Sequential Query Engine Details

Due to the inconsistent results using the Xalan query engine, described in the previous section, we implement a custom XPath query engine, the Sequential query engine. Our goal is to create a query engine that executes an XPath query on an XML document tree. We want the Sequential query engine to (1) return the correct result, (2) exhibit consistent execution times that we would expect from an XPath query, and (3) be extensible to a parallel implementation.

The Sequential query engine uses the Xerces library, described in Section 3.1, to parse an XML file. We then use the Xalan library to create a tree of XalanNode elements using the tags and data from the parsed XML file. The query engine selectively traverses this XML document tree using a given XPath query to find matching nodes.

This tree traversal compares the query string with tag name of each XalanNode and follows only matching branches down the tree. It begins at the root node and compares it to the first query tag. If it is a match, the method proceeds to compare the second query tag with each child of the root. The operation is performed recursively until all matching paths are exhausted. Any nodes that are reached that match the last tag of the query are saved as the set of query matches. Algorithm 2 outlines the process used by the Sequential query engine.

```

function TRAVERSE(node, depth) begin
    foreach child of node do
        if child.tag == query[depth] then
            if depth+1 == MaxDepth then
                | MatchResults.add(child);
            end
            else
                | TRAVERSE(child, depth+1);
            end
        end
    end
end

```

Algorithm 2: Pseudo-code outlining the execution process of the Sequential query engine

XalanNode elements are made up of data components, pointers to their children, and tag names saved as XalanDOMString objects. XalanDOMString objects contain data structures and comparison operations that are used by the Xalan querying method to improve sequential execution of commonly used queries. The complexity of these objects prompts us to create a simpler comparison method for the Sequential query engine.

The Sequential query engine converts the XalanDOMString objects into standard, C-style strings to improve performance. C-style strings are character arrays manipulated by low-level functions. The query engine uses the strcmp() function to compare each query tag to the corresponding node tag. We find that this tag comparison method yields up to a factor 10 performance increase over the XalanDOMString compare. Since, for many queries, tag comparison constitutes the majority of execution time, this has a large impact on the overall query performance.

We use the Sequential query engine as a consistent method for testing our performance model. Additionally, we are able to use this query engine as a baseline for sequential query execution time and calculate the speedup obtained with our parallel query engines. The Sequential query engine also gives us a foundation for further testing and enables us to accurately measure machine-specific parameters that the performance model is dependent on. In the next section, we instantiate our model by measuring the machine-specific parameters on our test environments. We then evaluate the accuracy of our performance model estimates using experiments with synthetic and real-world data sets.

4.4 Instantiation of the Model

Recall that in Section 3.2 we defined a sequential performance model and the parameters that it uses to estimate query execution time. Among them are 3 machine-dependent parameters, α , β , and τ . While all other parameters are obtained by knowledge of the database and query, these three parameters must be obtained for each hardware environment. Without accurately measuring these parameters, we cannot estimate XPath query execution time or perform a detailed analysis of sequential or parallel performance.

We measure these parameters with synthetic queries and data sets that isolate certain computational characteristics. The aggregative nature of these parameters in our model makes precise measurement imperative. A small error in the measurement of any of the three parameters can completely change our model estimates for all XPath queries.

4.4.1 Measuring β

The parameter β is described in Section 3.2 as the time, in seconds, to save and report a query match. Since the contribution of β is only reliant on one type of node (query matches), we can easily isolate it. To measure β , we run two queries on the exact same tree: one that matches all nodes and one that matches all non-leaf nodes only. Both queries have to process the same number of nodes, so the α and τ components do not change. The number of query matches, however, is non-zero (and in fact very

Tag Length	With β (sec.)	Without β (sec.)	Effect of β
1	0.450	0.414	8.02%
2	0.466	0.422	9.42%
4	0.490	0.467	4.58%
8	0.544	0.536	1.61%
16	0.677	0.640	5.45%
32	0.884	0.846	4.34%
64	1.258	1.185	5.85%
128	1.921	1.824	5.05%
256	3.227	3.103	3.84%
512	5.791	5.685	1.82%
1024	11.114	11.010	0.93%

Table 4.3. Matching vs. Non-Matching leaf-nodes on dual-core Navet server.

large) for one query, and 0 for the other (no leaves match). To further reduce the contribution of τ , these tests have minimal tag lengths (1 character). We run this test on a variety of tree shapes, the results of which are displayed in Table 4.3, averaged over 100 runs.

The contribution of β is the difference between the "Matching" and "Non-Matching" columns. While the largest contribution of β is 9.61%, the average over all tests is 2.02%. Furthermore, there is no clear trend with respect to the tag length. We opt to remove β from our model, which simplifies the calculation, as seen in Section 3.2. We see in Section 4.5 that removing β from our model has no significant impact on accuracy and that the simplified non- β model is adequately precise for all of our synthetic tests.

4.4.2 Measuring τ

The parameter τ is the time, in seconds, required for a given hardware system to perform the comparison of two single-character tags. We define the comparison of two tags as the process of loading the tag strings into memory, performing the comparison, and returning either true (match) or false (mismatch). As described in Section 4.3.1, we use a low-level comparison method to improve performance. We initially assume that τ remains constant as tag length increases (e.g., comparing two 3-character tags takes 3 times longer than comparing two 1-character tags). To test the behavior of this parameter, we repeatedly run the same synthetic test, using the Sequential query engine, while increasing the tag length from the minimum (1 character) to the maximum (limited to 2^{14} characters by XPath query string length). We measure the relative value of τ by dividing the runtime by the total number of characters compared (total nodes \times tag length). As tag length becomes very large, the runtime contribution of this parameter dominates all others (α) and we estimate τ to be the asymptotic limit as T becomes large, as seen in Figure 4.1.

We repeat this experiment for a variety of tree shapes to determine the value of τ as other parameters vary. While we expect the cost of character comparison (τ) to remain constant, it seems to

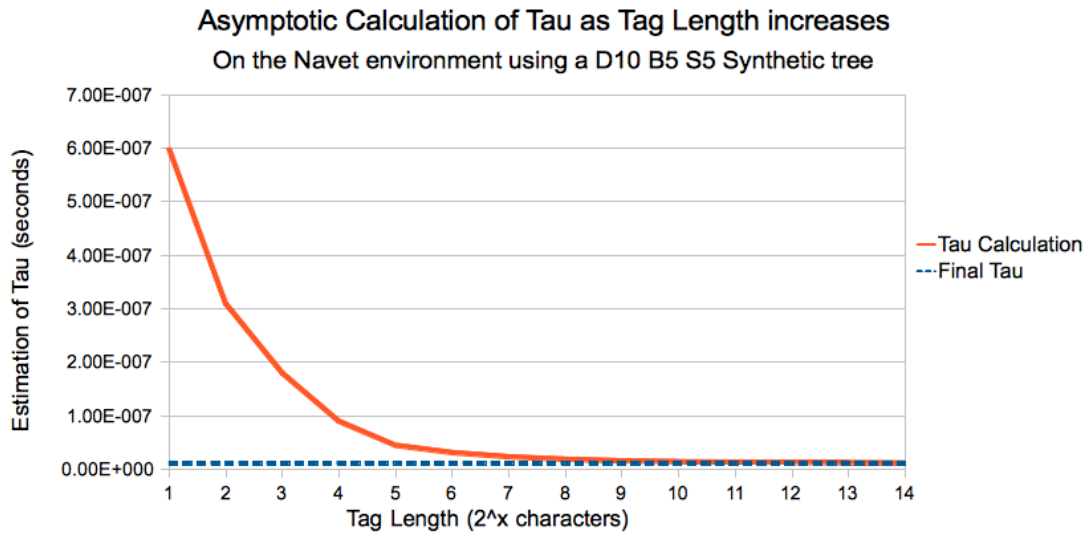


Figure 4.1. The asymptotic nature of τ as the tag length increases. This is due to the diminishing relative contribution of other parameters (such as α). With a large enough tag length, we assume the runtime is completely dominated by τ and calculate it directly from query execution time.

be dependent on maximum tree depth. To measure the value of τ with respect to tree depth, we create a series of trees with long tag lengths (2048 characters) and varying tree depth. Figure 4.2 demonstrates the results of these tests.

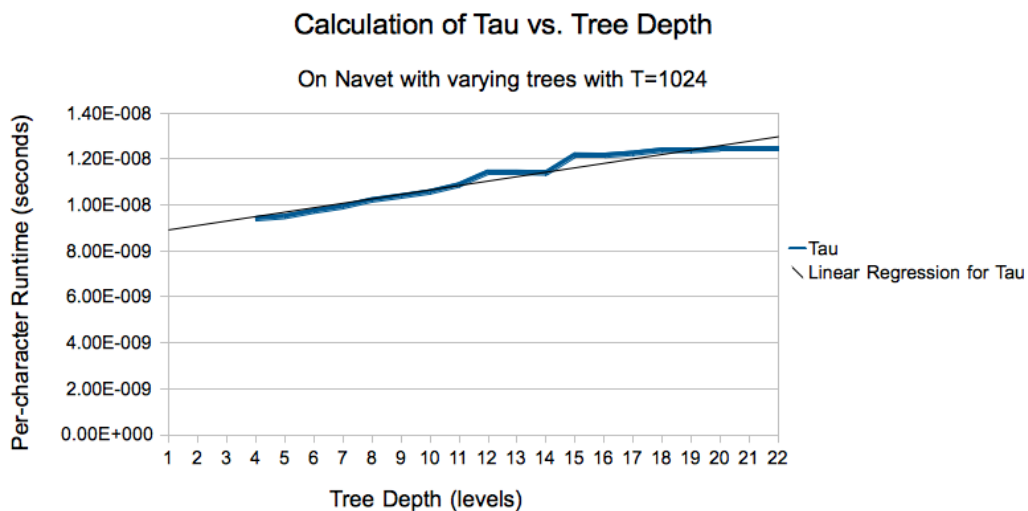


Figure 4.2. The τ parameter is linearly dependent on query depth D . Using linear regression, we compute a linear fit of τ to the experimental data, which is plotted on the graph.

The results in Figure 4.2 are obtained by creating trees of varying depth from 4 to 22 and adjusting branching factor to keep the average number of nodes between 50,000 to 5,000,000 so that average runtime is reasonable for experimentation. All results are repeated 100 times on each of our test environments to prevent error by measurement fluctuation. This experiment illustrates that τ is linearly dependent on tree depth. By plotting a regression line and calculating its formula, we get a tree-dependent

equation for τ for the specific hardware environment. This experiment (or a simplified version) must be repeated on any environment on which the query execution time is to be modeled.

The dependency of τ on maximum tree depth is unexpected. In our original performance model, defined in Section 3.2, we assumed that the parameters τ and α would be dependent only on the hardware environment. However, hardware factors such as memory management and bus speed are difficult to incorporate into generalized performance models and can produce unexpected results.

4.4.3 Measuring α

The parameter α is defined in Section 3.2 as the time, in seconds, to perform all operations other than tag comparison on a single node. Unlike β , this is the processing time that is required by all nodes, regardless of type. The contribution of α is independent of tag length and node type. Although we presume α consists of the cost of pointer-chasing and function recursing, it is a "catch-all" that absorbs the contributions of all operations that act equally on all nodes. Since we measure α empirically, these additional operations are accurately applied to the model as part of α .

Using the τ estimate we obtained from the previous section, we calculate α as the remaining per-node execution time. Assuming we have an accurate value of τ that is stable for many different tag lengths and tree shapes, we calculate α as:

$$\alpha = \frac{Runtime_{L=1}}{N_{nodes}} - \tau \quad (4.4.1)$$

Using a query with the minimum tag length (1 character), we estimate α as the per-node runtime minus τ . Since the tag length is only a single character, the contribution of τ is minimal and the overall query is therefore dominated by α . The additional subtraction of this small contribution by τ (for 1-character tags) gives us a more accurate estimate of α .

Similar to how we calculated τ in the previous subsection, we measure α on all trees from depth 4 to depth 22 while varying branch factor to maintain reasonable node count (50,000 to 5,000,000). All tests are run for 100 iterations to ensure consistency. Like τ , we expect α to be dependent only on the hardware environment. However, Figure 4.3 shows that, like τ , α is linearly dependent on maximum tree depth.

We see a remarkably similar dependency on tree depth (D) with α as we did with τ . We fit a linear regression line and calculate a formula for α for the given hardware environment. The formulae for α and τ for our two hardware environments are shown in Table 4.4.

Environment Name	α ($\times 10^{-9}$ seconds)	τ ($\times 10^{-9}$ seconds)
Greenwolf	$6.79 \times D + 269$	$.184 \times D + 8.44$
Navet	$9.29 \times D + 350$	$0.193 \times D + 8.72$

Table 4.4. Linear fits of τ and α on our two hardware environments. While α is much larger than τ , τ dominates execution time with large tag lengths.

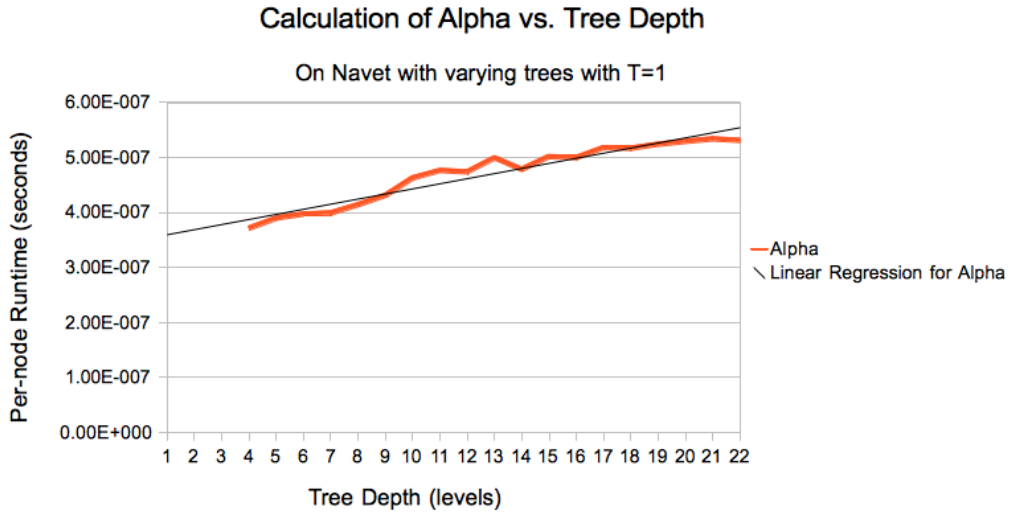


Figure 4.3. The α parameter is dependent on D . Using linear regression, we compute a linear fit of α to the experimental data, which is plotted on the graph.

Using these parameters, we are able to estimate the query-specific execution time on our test environments. However, we must measure one more parameter for our performance model to accurately mimic the behavior of XPath queries. In the next section we measure $f(L)$ using synthetic tests to further refine the accuracy of our model.

4.4.4 Determining $f(L)$

In Section 3.1 we defined the parameter $f(L)$ based on the assumption that a tag mismatch does not require that all characters of the tag be compared. We incorporated this function in the model, but could not measure its effect without first knowing τ and α . With empirical estimates of these two parameters, we develop tests to measure the effect of mismatches and tag length (L) on node processing time.

The function $f(L)$ is defined as the average number of characters compared when evaluating a mismatch between two tags of length L . We assume this value is smaller than L because mismatches can usually be determined without comparing the entire tag. To test the impact of $f(L)$ on string comparison time, we create a series of experiments that isolate this parameter. We create an XML tree with minimal tag lengths (1 character) for non-leaf nodes and a query that begins by matching all non-leaf nodes. We then create two versions of the query:

- Q_{match} : matches all characters except the last for all tags of leaf nodes.
- $Q_{mismatch}$: mismatches all characters for all tags of leaf nodes.

Figure 4.4 illustrates an example.

Since all non-leaves are matches for both queries, the total number of nodes processed is equal. In addition, neither query matches any leaf nodes, removing any contribution by β . However, while

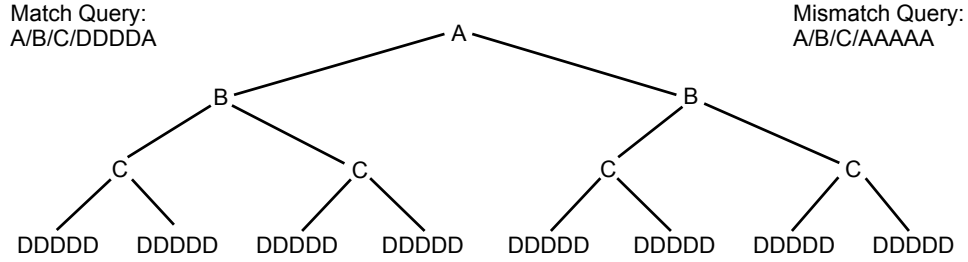


Figure 4.4. A simple example of a test to measure the impact of the parameter $f(L)$. We determine $f(L)$ by the difference in execution time of the Q_{match} and $Q_{mismatch}$ queries.

Leaf Tag Length (chars)	Leaves Match (sec.)	Leaves Mismatch (sec.)	Mismatch Speedup
1	0.7898	0.7744	1.95%
2	0.9060	0.9384	-3.58%
4	1.0889	0.9427	13.42%
8	1.0343	1.1343	-9.67%
16	1.2821	1.2360	4.02%
32	1.7270	1.7042	1.32%
64	2.3435	2.0758	11.42%
128	3.7202	3.4439	7.43%
256	5.8814	5.2932	10.00%
512	10.2664	9.1381	10.99%
1024	19.3491	17.3834	10.16%
2048	37.6182	34.0532	9.48%

Table 4.5. Results of tests to determine the impact of tag mismatches on comparison time. Run on Greenwolf with a D10 B5 tree.

Q_{match} must compare L characters at each leaf before it determines mismatch, $Q_{mismatch}$ compares only $f(L)$ characters, giving us a direct comparison between these two parameters. Note that, for the mismatch test, all leaf node tag strings are completely different from the query string (all characters are different). We repeat this test while varying leaf node tag length and on a variety of tree shapes and hardware environments. Table 4.5 contains the results of one of these tests.

The results in Table 4.5 show us that tag mismatches have a limited effect on query execution time. For all of our tests of the impact of $f(L)$, we experience an average mismatch speedup of 6.91% with a maximum of 14.37%. Furthermore, for smaller tag lengths, the difference in execution time between Q_{match} and $Q_{mismatch}$ is negligible (average difference $< 3\%$ when $L < 100$ characters). This tells us that, on average, $f(L) = L$ is an adequate estimate, especially for real-world applications, where we expect L to be small (< 100 characters).

Though our synthetic testing tells us that the impact of tag mismatches is small, this is not what we expect, given the nature of string comparison. In Section 4.3.1 we describe our Sequential query

engine and its method of string comparison. The `strcmp()` function we use to compare strings functions by comparing each character of the string individually, in order. When a mismatching character is found, the function terminates and returns a non-zero value, indicating mismatch. Based on this method of string comparison, we expect mismatches to be faster than matches for all cases where $L > 1$.

Considering the ability of `strcmp()` to terminate early on character mismatches, we probabilistically determine the average number of characters that must be compared on a mismatch. If we assume an even distribution of all characters and digits (a-z, A-Z, 0-9), the probability of the first character being a match is $\frac{1}{62}$. The likelihood of the first two matching is $\frac{1}{62}^2$, and so on. Since the first character of the string must be compared regardless of mismatches, the average number of characters compared between two random strings is $1 + \frac{1}{62} + \frac{1}{62}^2 + \dots = 1.0163$. This tells us that, on average, a mismatch will be found on the very first character being compared. If we consider the distribution of letters for words in the English language, the average character count becomes 1.0802. The early termination of string comparison reduces the average number of characters needed to be compared between two random strings to approximately 1.

Although, probabilistically, the computational cost of comparing two mismatching strings is very low (nearly the same as comparing two characters), our results show that, for most cases, $f(L) = L$ is an adequate approximation. Furthermore, our results indicate that execution time is not strongly dependent on the number of matching characters in each tag. This tells us that CPU is not the only limiting resource. Our results suggest that memory and cache may be the bottlenecks for this application. If memory, not CPU, is the bottleneck, the early termination of mismatches will have little impact on execution time. This is because blocks of memory are loaded for each of the two strings being compared, regardless of how many characters are used during comparison.

The impact of memory management is difficult to model analytically and varies depending on the hardware and software environment. Therefore, we use the estimate $f(L) = L$ for our performance models, which we evaluate in Sections 4.5 and 4.6.

4.5 Evaluation of Sequential Model

In this section, we use Sequential query engine (Section 4.4) on a range of synthetic tests (Section 4.2) and compare the results to estimates generated by our performance models. Our goal is to determine the accuracy of our sequential performance model and identify the contribution of each parameter to total query execution time. An analysis of XPath query execution time will help us identify methods of improving performance in Chapters 6 and 7. We use a broad set of synthetic tests to get a general error profile of our performance model.

To analyze the performance Sequential query engine, we create a series of synthetic tests designed to independently vary each parameter of our performance model. The combined contribution of some of these parameters (D and B for example) is complex and difficult to test exhaustively. Therefore, we create tests that gradually vary a single parameter and repeat this test for different combinations of

the other parameters. All tests are repeated 100 times to obtain a stable average execution time. Relative error rates over the 100 iterations are shown as error bars on each graph (though they are nearly invisible for many tests). We duplicate tests on our two hardware environments described in Section 4.1. This section is divided into subsections for each parameter that we attempt to individually test and compare with our performance model estimates.

4.5.1 Depth (D)

The Depth parameter (D) is described in Section 3.1 as the maximum number of levels of an XPath query over an XML Document Tree. This parameter, combined with the Branch Factor, directly corresponds to the total number of nodes of the XML document tree. In our performance model, D forms the exponent of the total nodes visited, so we expect that increasing D will have an exponential effect on the total query execution time.

To test the impact of D on query execution time, we run synthetic tests that start with a shallow depth relative to the branching factor (B). We repeat the test while increasing the query depth until query execution becomes too long to test (we stop at 100 seconds per run). This gives us a profile of how the parameter D affects total execution time. We run this test with a variety of different values of B and T . Figure 4.5 shows an example of three of these tests.



Figure 4.5. The effect of D on query execution time on Navet. Comparison of test results to performance model estimates for three different tree shapes.

These tests demonstrate that our performance model accurately estimates the exponential impact of D on query execution time. For these tests, our model estimates execution time with an average relative error of 5.85% and a maximum error of 16.38%. The majority of data points with $> 10\%$ error are seen where total query execution time is very fast (< 0.01 seconds). A high relative error at such short execution times may be caused by measurement inaccuracies.

4.5.2 Branch Factor (B)

The Branch Factor (B) of an XML Document Tree is the total number of children of each non-leaf node (Section 3.1). It is a parameter that is completely dependent on the tree, though it is related to the query-dependent parameter, selectivity (S). The maximum selectivity of a query is the branch factor (i.e., S cannot be $> B$). In our performance model, B is multiplied by the exponential S^{D-1} , so we expect it to have a direct linear impact on query execution time.

Our tests to determine the impact of B are accomplished by running synthetic tests while varying only B . We start with a small query where $S = B$ and increase B while keeping S and all other parameters constant. We continue by repeating the test and increasing B . We duplicate this test with a variety of values for all other parameters. Figure 4.5.2 shows an example of our tests for the parameter B .

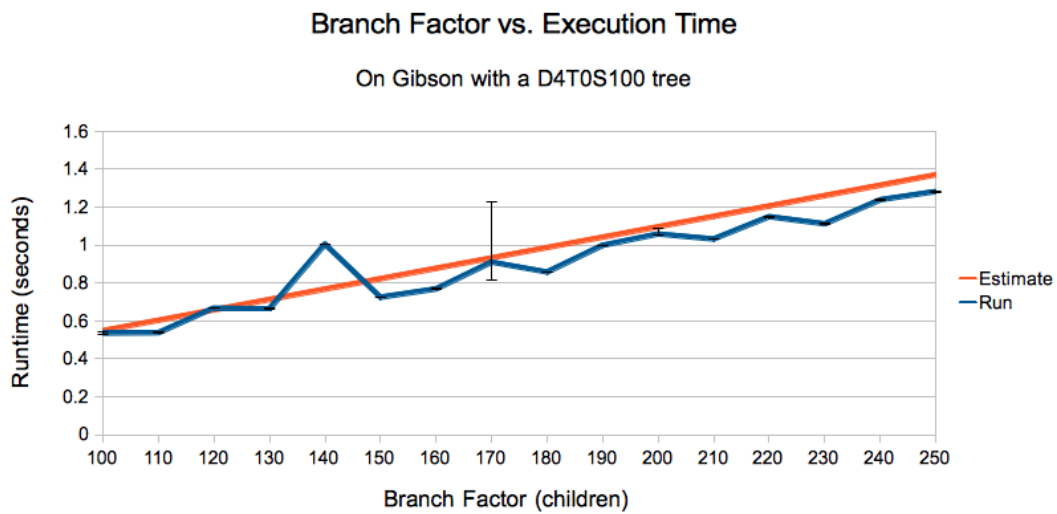


Figure 4.6. Query execution time is linearly dependent on B . Performance model estimate is linear and test results show a linear dependency as well. Test results are averaged over 100 runs and error bars show min and max values. Most error rates are very small, except the case of $B=170$ on this graph. We assume this is an anomalous occurrence.

Our synthetic tests of the parameter B verify our assumption that B has a linear impact on query execution time. Figure 4.5.2 demonstrates that our performance model correctly estimates this behavior. Over all synthetic tests while varying B , our average performance model error rate is 8.6% with a maximum error of 30.89%.

4.5.3 Selectivity (S)

The selectivity parameter (S), like B , measures the spread of the tree. However, S is determined by the average number of *matching* children of each node. While B is dependent on only the tree, S is based on the query and the tree.

The selectivity parameter (S) has a large impact on the total query execution time. In our performance model, S is the base of the exponent, but also divides the entire formula. Because of this, we assume it has an exponential impact on query execution time. However, we expect it to have a somewhat complex behavior, especially at low values and when D is very small.

Like previous experiments, we test the impact of S by keeping all other parameters static and repeating tests while slowly increasing S . Since, for any query, $S \leq B$, we choose trees with somewhat large values of B and increase S until $S = B$. Figure 4.7 shows an example of one of these tests.

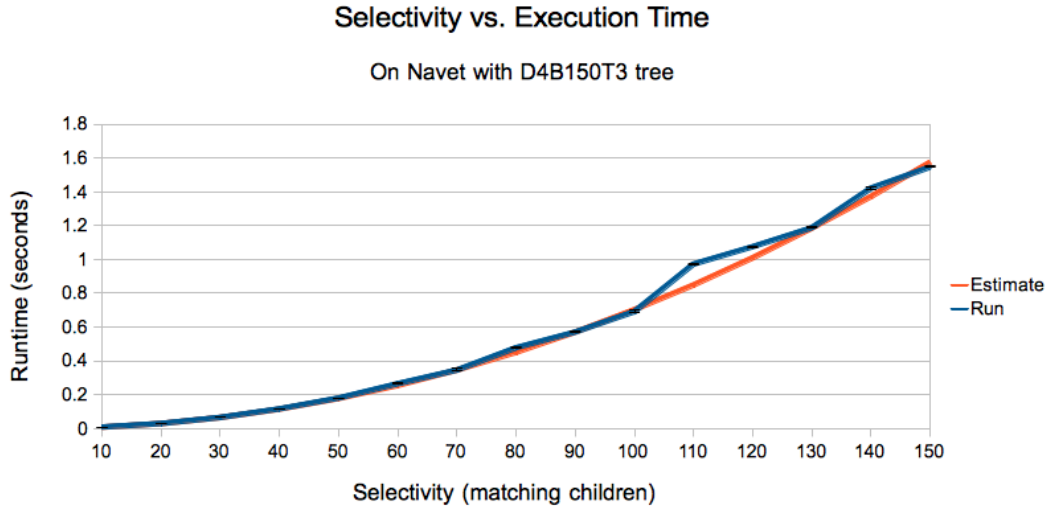


Figure 4.7. Average query execution time while increasing S . Model estimate and runtime both exhibit exponential behavior. The rate of exponential increase is dependent on D .

We see an exponential increase in query execution time as we increase S . Our performance model correctly captures the exponential impact of S on query execution time. The rate of exponential increase is dependent on D . For our tests measuring the impact of S , our performance model is adequately accurate, with an average relative error rate of 2.9% and a maximum of 14.56%.

4.5.4 Tag Length (L)

Tag Length (L) is the average number of characters per node tag. This parameter has a direct impact on execution time by increasing the size of the string that must be compared at each node. The exact impact L has on execution time is dependent on τ , a hardware-dependent parameter (Section 4.4.2). We have seen that string comparison is linearly dependent on L , so we expect L to have a linear impact on query execution time.

Our tests to determine the impact of L involve keeping all other parameters static while increasing L by 100 characters at a time. This gives us a range of values of L without having to run thousands of tests. We repeat this process using synthetic tests using several different combinations of all other parameters. Figure 4.8 shows the results of two of these tests.

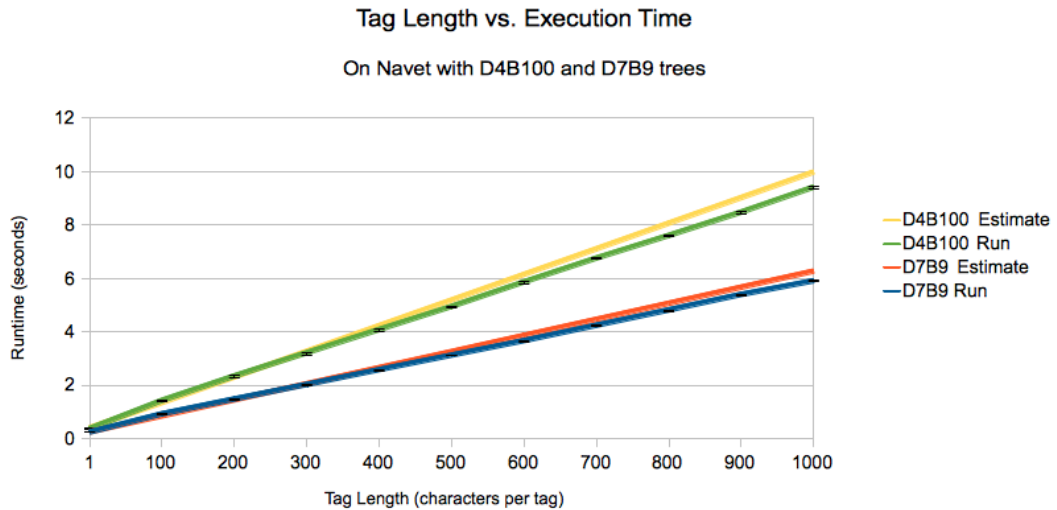


Figure 4.8. The effect of L on execution time for two different tests. Both model estimate and test results display a linear pattern. Slope and intercept are based on τ and XML document tree and query parameters.

Figure 4.8 shows us that execution time is linearly dependent on L . The smooth lines and small error bars demonstrate that the results of these tests have low variance and that our performance model correctly estimates the impact of L . For these tests, our performance model has an average relative error rate of 4.53% and a maximum error of 9.38%. As with our previous tests with the other parameters, our performance model estimates execution time with adequate accuracy.

The results obtained in this section provides us with an understanding of query execution time and how it is affected by each parameter of our performance model. We have a profile of execution time for a broad set of hardware environments, XML trees, and XPath queries.

4.6 Evaluation of Parallel Model

In this section, we continue our evaluation of our XPath query performance models. In Section 4.5 we used synthetic tests to quantify the effects of each parameter (defined in Section 3.1) on our sequential performance model and measured the accuracy of the overall model. In this section, we evaluate the accuracy of our parallel performance model.

As with the evaluation of the sequential model, we create a range of synthetic tests (Section 4.2) to determine the accuracy of the parallel model. We repeat all tests for 100 trials and duplicate all tests on our two test environments (Section 4.1). For parallel execution, the differences between our test environments are more significant, as the number of processor cores differ. We also expect a discrepancy between our parallel performance model and test results due to imperfect parallelization. Our model assumes perfect parallelization of parallel execution, which is rarely achieved in practice.

To estimate the complexities of parallelization, our parallel performance model uses more parameters that we must also independently test. In addition to tree depth (D), branching-factor (B), selectivity (S), and tag length (L), the parallel model incorporates:

- Context Depth (C) - The depth that must be sequentially queried before beginning parallel execution.
- Processor Core Count (P) - The number of processor cores available. While this parameter is dependent on the hardware environment, we can control it by altering the number of threads used by the query engine.

To independently tests these parameters, we must have a stable parallel querying method. In Section 4.3.1 we introduced the Sequential query engine. We extend this query engine to create two parallel engines that differ in how they distribute the workload among processor cores.

4.6.1 Fixed Work Allocation

Our first parallel XPath query engine, the Fixed Work engine, uses fixed work allocation to distribute the workload among threads. With this engine, we first query sequentially to reach the context depth, C , and then evenly distribute the work among each thread. Each thread maintains a list of node matches and, once all are complete, we join the match lists to get a complete query match list. Since work distribution is fixed, we avoid critical sections and have low overhead. However, load imbalance could be problematic using this query engine if trees are very unbalanced (as we test in Chapter 5).

The context depth (C) is a crucial parameter for this query engine. A shallow context depth may lead to load imbalance while a deep context depth requires more sequential execution and diminishes the gains of parallelization. In Section 4.6.3 we determine the impact of this parameter for this query engine and our other parallel query engine, described in Section 4.6.2.

4.6.2 Work Queue Implementation

Our second parallel XPath query engine, the Work Queue engine, uses a work-queue system to balance the workload among available processor cores. As with the Fixed Work engine, we first sequentially query the XML document tree to the context depth (C). The master thread then creates the work-queue by dividing all matching context nodes into W work units. It spawns $N-1$ new threads to perform the work, where N is the number of desired threads. All threads (including the master thread) obtain work from this shared work queue and perform the work, each maintaining its own list of query matches. Since the work queue is shared among all threads, consistency is maintained through the use of a pthread spinlock. Once all work is complete, the master thread assembles all the matches to be given to the user.

For the Work Queue query engine, we have an additional parameter that we use to control the overhead and load imbalance:

- Work Unit Count (W) - The number of units the total work must be split into. If W is less than the number of threads, additional threads will be idle.

This implementation has additional overhead due to the creation and maintenance of the work queue. However, we postulate that the overhead may be outweighed by the potential gains of greater load balance. The gains, however, are difficult to quantify and are dependent on the XML tree and XPath query being run. In Chapter 6 we attempt to verify this postulation with real-world data sets.

We test the impact of work unit count (W) on query execution time using synthetic tests. Each set of tests for W uses a single XML tree and XPath query but varies the work count from 1 to the maximum (based on context depth). Figure 4.9 shows an example of this test on the Greenwolf environment.

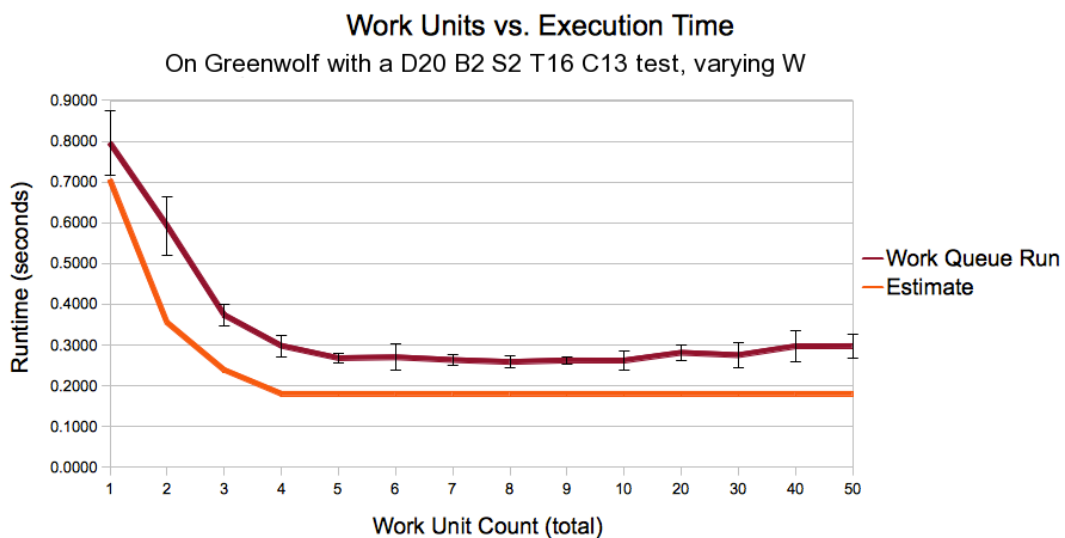


Figure 4.9. Comparison of model estimate and test execution time while varying work unit count. The 15-30% overhead from the use of the work queue is visible.

The model does not incorporate the effects of varying work unit count once there are enough to provide work for each processor core (4 for the Greenwolf environment). The large speedup seen in Figure 4.9 when increasing W from 1 to 4 is the result of providing work to all processor cores. We see that the average execution time using the Work Queue engine is, averaged over all tests, 15.07% slower than our model estimate, with maximum error of 30.83%. This may be due to shared work queue and parallelization overhead. Recall that our model estimate assumes perfect parallelization during parallel execution.

Over all of our tests, the largest average speedup is obtained when $W = 8$. We postulate that this is the ideal number of work units to compromise between load imbalance and work queue overhead. In all experiments performed hereafter, we use $W = 8$, unless specified otherwise.

In Chapter 5 we investigate how the Work Queue engine combats the issue of load imbalance compared to the Fixed Work engine. For balanced synthetic tests, however, we expect to see faster query

execution time with the Fixed Work engine, regardless of the values of other parameters, such as P and C .

4.6.3 Context Depth (C)

The context depth (C) is defined in Section 3.3 as the depth the query is executed sequentially before parallel execution begins. This sequential execution is required to create independent work units to be distributed among processor cores. We test this parameter using synthetic tests and varying C from 1 to the maximum ($D-1$) and repeat it for a range of values of each other parameter.

Figure 4.10 demonstrates the impact of context depth on average query execution time when using the Fixed Work query engine. For completely balanced tests, like this one, we expect this query engine to perform well, since it has less overhead compared to the Work Queue engine.

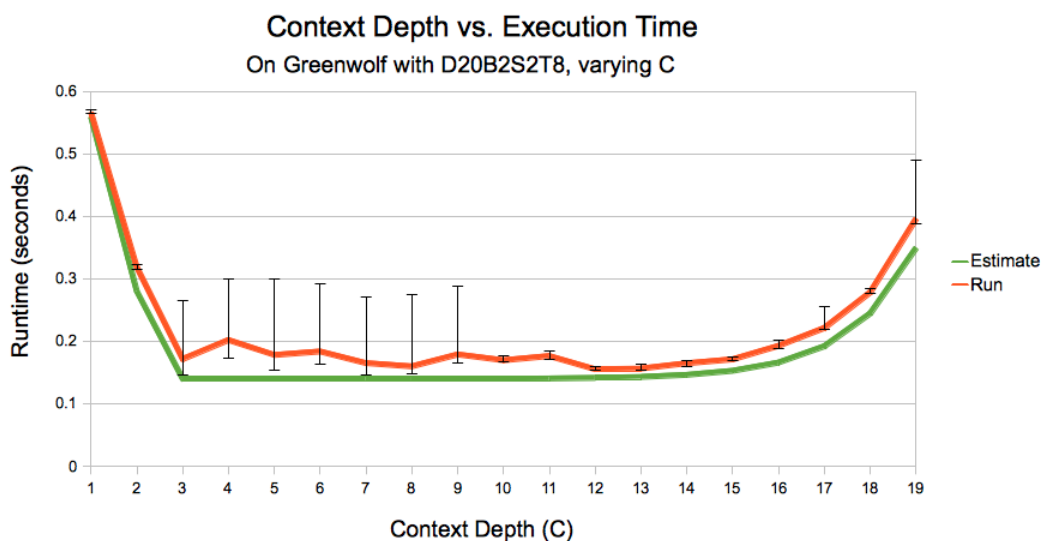


Figure 4.10. Comparison of performance model estimate with empirical results using the Fixed Work engine, while varying context depth. The Fixed Work query engine performs well on these balanced synthetic tests.

As we see in Figure 4.10, context depth has a large impact on execution time. At $C=1$ and $C=2$, there are not enough context nodes to partition among the 4 threads, so cores remain idle through execution. Once $C=3$ is reached, all processor cores are working and we have efficient parallel execution. Execution time remains somewhat even until very deep context depth ($C=15$). At this point, the sequential execution to reach the context depth becomes significant and we see an increase in overall query execution time. For these tests, the trees are perfectly balanced so we do not see poor parallel performance caused by load imbalance.

4.7 Conclusions

In this chapter, we proposed our three XPath query engines: Sequential, Fixed Work, and Work Queue. Using these query engines, we evaluated the impact of each parameter on query execution time. We began by describing our hardware test environments and synthetic testing methods in Sections 4.1 and 4.2, respectively. In Section 4.3 we introduced our Sequential query engine. Using this engine, we measured the machine-dependent parameters (α , β , τ , and $f(L)$) on our two hardware environments (Section 4.4). We verified the accuracy of our performance model while varying each data- and query-specific parameter in Section 4.5. We determined that our performance model is adequately accurate for all synthetic tests on our two hardware environments. In Section 4.6 we introduced our two parallel XPath query engines and measured the impact of each engine-specific parameter (C , P , and W) on query execution time. We conclude that, while the Work Queue engine has significant processing overhead, our parallel query engines perform well and our parallel performance model estimates execution time relatively well.

In the next chapter, we introduce unbalance and skewed synthetic data sets to determine how our XPath query engines perform on non-complete (i.e., all nodes have equal children) XML document trees. In Chapter 6, we further evaluate the performance of our query engines using real-world data sets and XML benchmarks.

Chapter 5

Unbalanced Synthetic Trees

In Chapter 4 we experimented with a range of XPath queries on synthetically generated XML files. We compared query execution times for each of our three XPath query engines with estimates generated by our performance models (defined in Chapter 3) to determine overall performance model accuracy. We determined that, for the synthetic tests used in Chapter 4, our performance models were able to estimate average query execution time with adequate accuracy. However, all synthetic XML trees used for testing thus far have been complete trees (all non-leaf nodes of the tree have equal branch factor) with equal selectivity across all nodes. Data from real-world applications rarely produces these types of trees. Therefore, in this chapter we attempt to measure the impact of the randomness and imbalance that we expect to see with real-world data. To accomplish this, we introduce more complex synthetic data sets using several additional parameters. These new parameters introduce randomness and/or skew by modifying the parameter values (i.e., B or S) of individual nodes. Our goal is to quantify the impact of random variance and parameter skew on query execution time so that we can improve the accuracy of our performance model estimates on real-world data sets and determine weaknesses in our XPath query engines.

5.1 Randomized Selectivity

When querying real-world data sets, we will rarely see uniform selectivity across different tree levels or even amongst siblings. Since our performance model assumes a fixed selectivity across the entire tree, it may be inaccurate when estimating the execution time of queries on trees with a large selectivity variance. In this section, we test the impact of adding a random variance to the selectivity for each node.

For this test we create more complex synthetic trees that have a fixed mean selectivity (S) and a selectivity range (R). The selectivity (number of matching children) of each node is individually calculated as a random integer between $S - R$ and $S + R$. Since the selectivity is used to determine the number of branches the query must traverse, it may have a significant impact the overall query execution time. However, when we repeat a query over many randomly generated trees (with the same parameters),

we expect the *average* execution time to remain consistently close to the estimate from our performance model.

We create a series of synthetic trees and queries and fix all parameters except the selectivity range (R). We repeat each query while varying R to measure the impact of selectivity variance between nodes on query execution time. Each query is executed 100 times, with the XML data set being re-created each time to ensure an even distribution of selectivity values for each node. All tests are repeated for several different combinations of S , B , and T and duplicated on our two hardware test environments, described in Section 4.1. Figure 5.1 shows the results of one of these tests on the Navet environment.

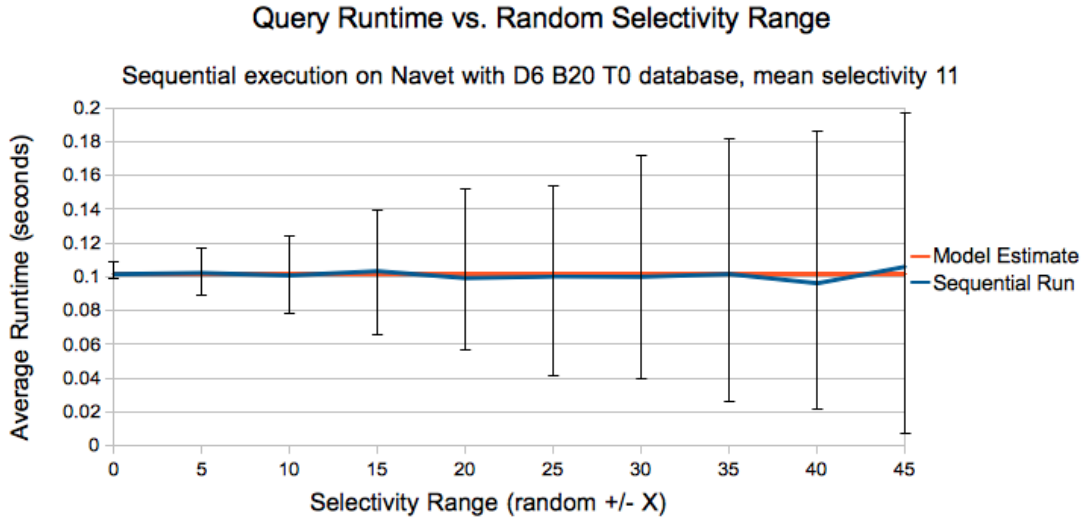


Figure 5.1. The impact of adding a random range (R) to selectivity for each node of the tree. Average over 100 runs (re-creating the XML file for each run) on Navet with a D6 B20 T1 synthetic tree. We see that the mean execution time remains stable, but the error rate increases as we increase R .

As we expect, the results of our tests while varying R show an increase in runtime variance as R increases. In addition, the mean execution time remains close to our performance model estimate. This tells us that while, individually, random selectivity can have a large impact on execution time, it is the average selectivity that we are most concerned with when estimating *average* query execution time. Throughout these tests, the mean selectivity remained the same, giving us a stable average execution time.

5.2 Match Skew

All synthetic tests performed in Chapter 4 utilize complete trees (i.e., trees with nodes that all have an equal number of children). While the tests performed in the previous section introduce a random difference between each node, the *average* values remained the same.

In this section, we create *match skewed* trees by introducing the concept of branch weight. We define a *branch* as a node and all of its descendants. We consider a branch's weight to be the amount of

query sub-matches it has (recall from Section 3.1.1 that sub-match nodes have paths (lineages) that are a proper prefix of the query). Therefore, a *heavy* branch would contain many query sub-match nodes, while a *light* branch would have mismatches. A tree with an uneven distribution of heavy and light branches is *match skewed*. Since weight is based on query sub-matches versus mismatches, it is query specific. Therefore, we create synthetic tree/query combinations to test the impact of match skew on query execution time.

We create our match skew tests by altering our synthetic test generation script, described in Section 4.2. We introduce a *heavy branch index* parameter, I , that defines the section of the tree that contains *heavy* branches. Recall that synthetic trees are created recursively, with each node having B children, S of which are query sub-matches or matches (depending on depth). With the parameter I , we define the indices of the S query sub-match children. Algorithm 3 describes the process of creating match skew synthetic trees:

```

function GENERATE begin
  write "<" + tagName + ">";
  if depth < MaxDepth then
    for childIndex from 0 to BranchFactor do
      if childIndex > I and childIndex < I + S then
        | GENERATE;
      end
      else
        | GENERATE-MISMATCH;
      end
    end
  end
  write "</" + tagName + ">";
end

```

Algorithm 3: Pseudo-code describing the method we use to create match skewed synthetic XML files. Note that the GENERATE-MISMATCH() function simply generates a node with a mismatching tag and no children.

This new test generation script creates XML trees with each node having S sub-match children, all of which are from indices I to $I + S$. All other children are query mismatches. An example of what a match skewed, left-weighted tree ($I = 0$) may look like is shown in Figure 5.2. Note that all heavy branches contain query sub-matches and light branches are just query mismatches.

By isolating all sub-match nodes to a single section of the XML tree, we can measure the impact of match skew on query execution time. We expect real-world data sets to be match skewed, so these tests may help us improve the accuracy of our performance model. While match skew should have little impact on sequential execution time, we postulate that it may increase load imbalance and, therefore, increase parallel query execution time. However, we expect our two parallel query engines

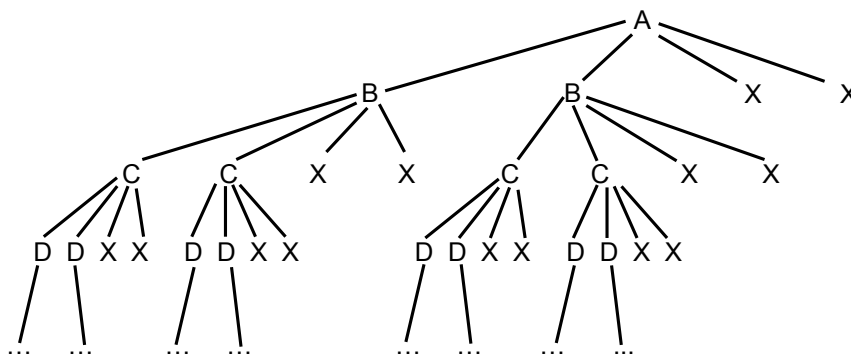


Figure 5.2. Example of what a match skewed, left-weighted tree may look like with a query of `"/A/B/C/D/..."`. All query sub-matches occur on only the S left-most children of each node while all other nodes (denoted "X") are mismatches. Parameters for this example are $B4\ S2\ T1$ and $I=0$

to be effected differently, since they use different load-balancing techniques (see Chapter 4). To get a profile of the impact of match skew, we create a series of tests that vary the size and placement of *heavy* tree branches.

The parameter I allows us to control the placement of matching branches, and we can vary number of such branches with S . Therefore, we begin by creating trees with $I = 0$, which corresponds to a left-weighted tree (e.g., Figure 5.2), and execute the corresponding query using our three XPath query engines. We repeat this test while increasing I until $I + S = B$, which corresponds to a completely right-weighted tree. Query execution time is averaged over 100 runs and we repeat all tests for a range of tree shapes on our two test environments. Our goal is to determine if isolating query sub-matches and matches to certain branches of the tree has an impact on query execution time. Figure 5.3 contains the results of one of these tests on the Greenwolf environment.

As Figure 5.3 demonstrates, isolating sub-match nodes to specific branches has no significant impact on sequential query execution time. Surprisingly, it also has no significant impact on parallel query execution time. We obtain good parallel speedup for these tests using both parallel query engines (Chapter 4), with an average speedup on Greenwolf of 2.92 with the Fixed Work engine and 3.01 using Work Queue. Perfect parallel speedup on Greenwolf (a 4-core machine) is 4.0, so we achieve 73% and 75.25% of perfect speedup, respectively. This is better parallel performance than expected, as we expected match skew to cause load imbalance and degrade parallel performance.

If we look closer at the work distribution for both parallel query engines, we see why parallel performance is not degraded by match skew. As Figure 5.2 demonstrates, all sub-match nodes, while located only on the left-most branches of the tree, have the same branch factor and selectivity. When work is distributed to separate threads for parallel execution, only sub-match nodes are considered as work. This means that all non-matching nodes on the non-left branches of the tree are ignored for parallel execution. Figure 5.4 illustrates how work would be distributed for this example tree.

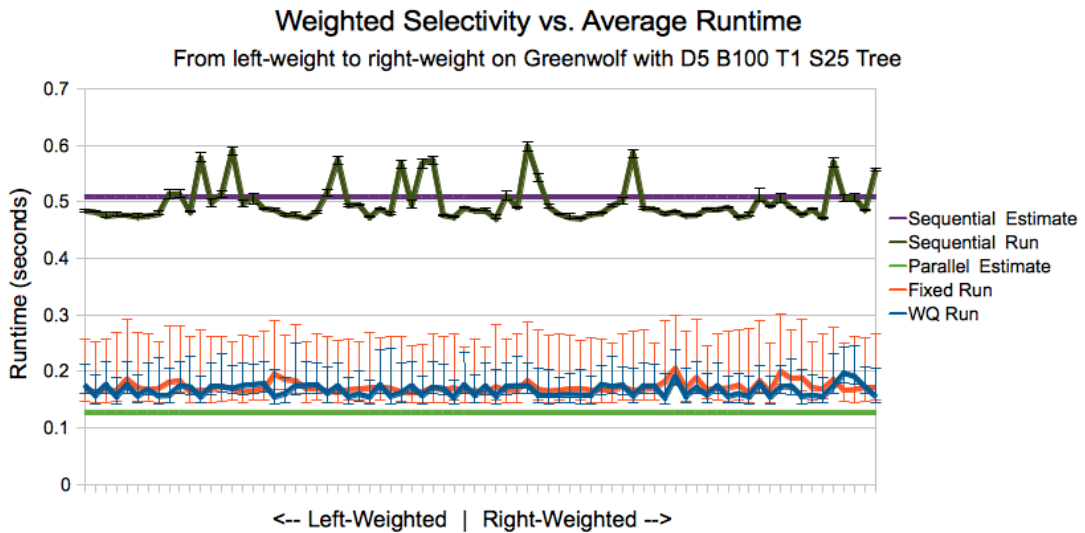


Figure 5.3. The impact of match skew and heavy branch placement on Greenwolf. Test is performed while increasing I so that each position from left-weighted to right-weighted is tested. We see that match skew has no significant impact on sequential or parallel execution time. Note that the x-axis is the heavy branch index, I .

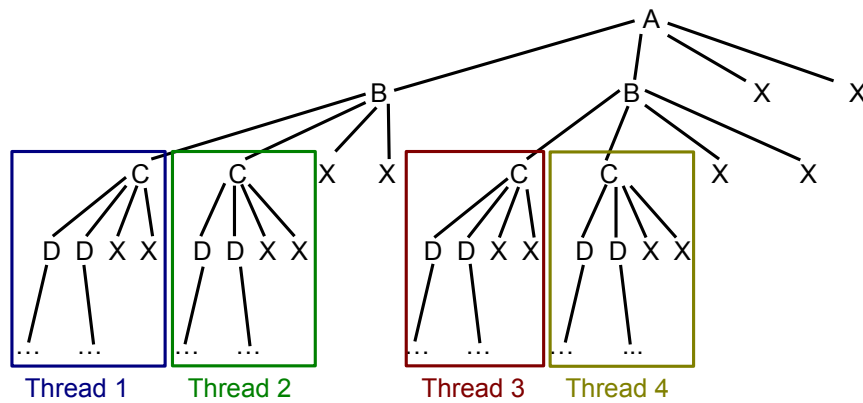


Figure 5.4. An example of how the Fixed Work engine distributes work for match skewed trees, using query `"/A/B/C/D/..."`. Heavy children are sub-match nodes while light children are mismatches (denoted as "X"). Although the example tree is completely left-weighted ($I = 0$), we see an even work distribution.

We conclude that match skew has no significant impact on parallel performance. For these types of trees, all sub-match nodes have the same number of children, so all threads are given the same amount of work. Our parallel query engines are not affected by uneven distribution of matches and it has no significant impact on either sequential or parallel query execution time. Although match skew, which skews selectivity, has no measurable impact on query execution time, we continue experimentation by looking at skew of another parameter.

5.3 Branch Factor Skew

In the previous section we determined that isolating sub-match nodes (i.e., nodes with a path from the root that is a proper prefix of the query string) to certain branches has no significant impact on load balance or parallel execution time. In this section, we look at the impact of trees skewed by branch factor (B) rather than matches.

Recall that branch factor (B) is the *total* number of children of a node. This means that a tree that is branch factor skewed has nodes with many children on one branch, and fewer on the others. Therefore *heavy* nodes are those with many children and *light* nodes are childless. We use this different classification of weight to test the impact of branch factor skew on query execution time.

Unlike the tests in Section 5.2, we create trees where every node is a query sub-match ($S = B$), and introduce a new parameter, heavy child count (H). The parameter H is the number of *heavy* children (those with many children) each node has. We use the parameters I and H to control the section of the tree that contains child-heavy nodes, like we did with I and S in Section 5.2. Synthetic trees for these tests have all heavy nodes in the indices between I and $I + H$. Note that all *light* children (those not within the heavy branch indices) are childless. Figure 5.5 illustrates what a branch factor skewed, left-weighted ($I = 0$) tree may look like. Note the difference between this and the example of the match skewed tree in Figure 5.2.

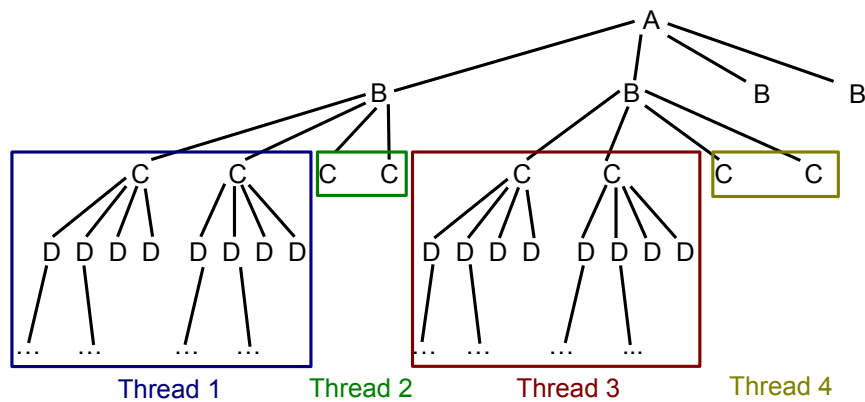


Figure 5.5. Using query `"/A/B/C/D/..."`, an example of what a branch factor skewed, left-weighted tree ($I = 0$) may look like and the Fixed Work engine would distribute work. Notice that light children are now query sub-matches, though they have no children. This means they are still counted for parallel work distribution. Parameters for this tree are $B4\ S4\ T1, I=0$ and $H=2$.

As Figure 5.2 illustrates, we expect branch factor skew to cause load imbalance and degrade parallel performance. In this example, using the Fixed Work query engine, threads 1 and 3 get branches with heavy workloads while threads 2 and 4 have very little work and remain idle for the majority of

parallel execution. We expect the Work Queue engine, however, to perform better than Fixed Work due to its load-balancing benefits, described in Section 4.6.2.

Using tests similar to those in Section 5.2, we measure the impact of branch factor skew on sequential and parallel query execution time. The test results provide us with an average query execution time while varying I (from $I = 0$ to $I + H = B$). The result of one of these tests on the Greenwolf environment is shown in Figure 5.6.

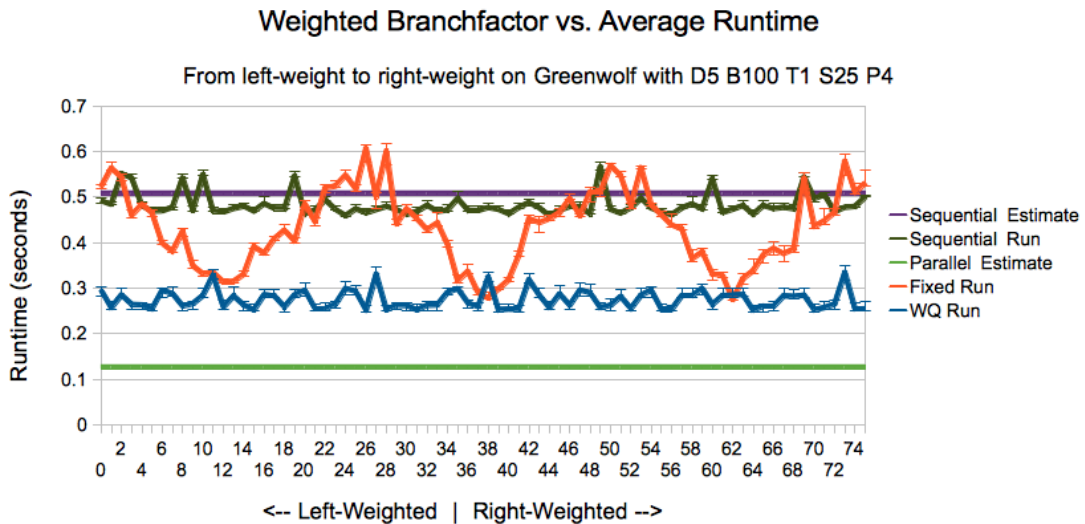


Figure 5.6. Results of the impact of branch factor skew on query execution time on Greenwolf. While there is no significant impact on sequential execution time, the Fixed Work engine shows dips and spikes in execution time at certain values of I . Note that the x-axis is the heavy branch index, I .

The results of our tests to determine the impact of branch factor skew indicate that, while it has no noticeable impact on sequential query execution time, it has a significant impact on parallel query execution time. The load-imbalance introduced by branch factor skew degrades overall parallel performance. In addition, the results indicate that the *position* of heavy branches has an impact on parallel execution time. As seen in Figure 5.6, there are pronounced spikes in query execution time at certain values of I when using the Fixed Work engine. Execution time at the spikes is slower than sequential query execution, and we see an average parallel speedup of 1.78 during the three dips. We see no spikes in query execution time with the work queue query engine and achieve an average parallel speedup of 1.79. This speedup is 68% less than what we see with match skew, though it is still a consistent speedup over sequential execution.

To help explain the cause of the dips and spikes we see in query execution time using the Fixed Work engine, we look at the same test on our dual-core test environment, Navet. Figure 5.7 shows the results of the same test executed on Navet.

In Figure 5.7, we see two distinct spikes in query execution time using fixed work allocation. Furthermore, these spikes occur when we have completely left-weighted and right-weighted trees. We

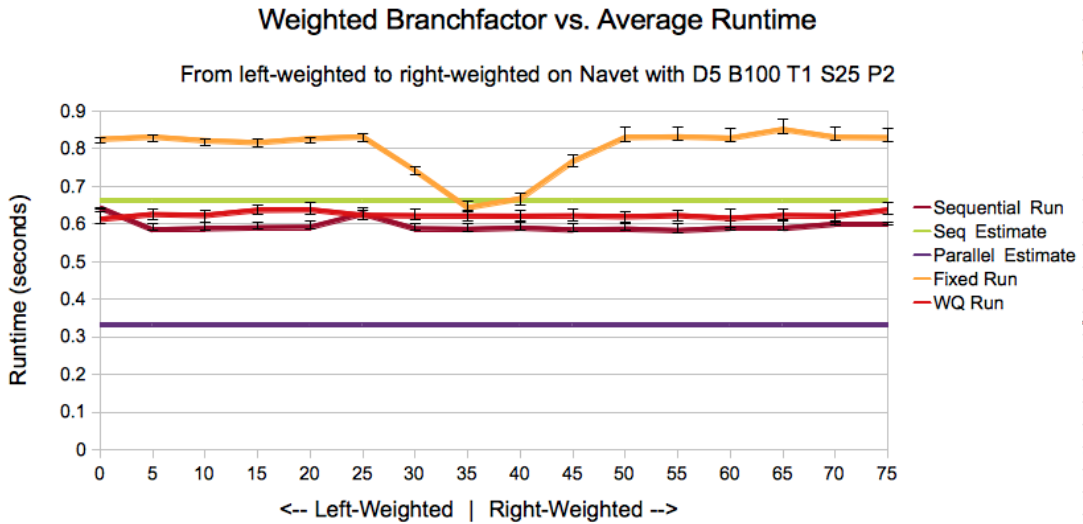


Figure 5.7. The results of testing the impact of branch factor skew on the dual-core environment, Navet. We see only one dip in execution time with the fixed work allocation query engine. This helps us explain the reason for the dips and spikes we see on Greenwolf in Figure 5.6. Note that the x-axis is the heavy branch index, I .

see the fastest execution time when our tree is completely center-weighted (all heavy nodes isolated to the center-most indices).

The occurrence of two spikes in query execution time on Navet and four on Greenwolf indicates that these are dependent on the number of processor cores on the hardware environment. This phenomena does not occur when using the Work Queue engine, so we focus on the Fixed Work engine to attempt to explain it.

Recall that, once context depth is reached, the Fixed Work engine distributes work evenly among processor cores. It does this by assigning the first $\frac{B}{P}$ nodes to core 1, the next $\frac{B}{P}$ nodes to core 2, and so on. With just 2 cores on Navet, the left $\frac{B}{2}$ nodes are given to core 1 and the right $\frac{B}{2}$ nodes to core 2. Therefore, on Navet, we only achieve a balanced workload when the tree is completely center-weighted.

This insight helps explain the 4 spikes in query execution time when running this test on the 4-core environment, Greenwolf, as we see in Figure 5.6. These spikes are caused by an increase in load imbalance, with the top of the spikes occurring when 1 core is assigned all heavy branches. The dips in execution time occur when the *window* of H heavy children falls between the work allocation blocks of two processor cores. Thus, the first spike occurs when we have a completely left-weighted tree; all H heavy nodes are assigned Core 1, while all other cores have only light children. We see the first dip when $I = \frac{H}{2}$. At this point, the H heavy children fall directly between the work allocation blocks for Core 1 and Core 2. We see a speedup of 1.79, which is a reasonable speedup when using only 2 cores. This pattern continues, with each spike occurring when all heavy nodes are assigned to one core, and all dips occurring when the H heavy nodes are evenly distributed among two neighboring cores.

This test shows us a weakness of the Fixed Work query engine. While it uses a simpler work distribution method, if child-heavy branches are all concentrated at one section of the tree, one processor

may have a much larger work load than the others. This results in load imbalance and a degradation of parallel performance. The Work Queue engine, with a more complex method of load balancing, does not suffer from this weakness, provided enough work units are used. However, the average speedup achieved using the work queue method is less than ideal, indicating that it suffers from load imbalance as well.

5.4 Conclusions

In this chapter, we introduced several tests using unbalanced synthetic XML document trees. These tests are designed to introduce some of the randomness and skew we expect to see with real-world data sets. We determine that, while our Sequential query engine was unaffected by these factors, our parallel XPath query engines suffer from load imbalance when dealing with certain types of skew (i.e., branch factor skew). We conclude that, while our parallel query engines perform well on well-balanced trees, they may suffer from load imbalance when querying real-world data sets. In the next chapter, we test our XPath query engines and performance models on real-world data sets and XML benchmarks.

Chapter 6

Evaluation with Real Workloads

The XML format is a widely accepted industry standard. It is heavily used in the real world by applications ranging from web development to content management to database development and management. There are many benchmarks and data sets available with which we can test the speed and efficiency of our XPath query engines, as well as the accuracy of our performance model.

In the previous chapters, we measured the execution time of our three XPath query engines (Sequential, Fixed Work, and Work Queue) on a range of synthetic tests and compared the results to estimates generated by our performance model. In Chapter 5 we used synthetic tests with randomness and skew in an attempt to emulate the inconsistencies we expect to see in many real-world applications. Thus far, our performance model has been adequately accurate on all on synthetic tests run on our two hardware test environments.

In this chapter, we use several real-world data sets and XML benchmarks to test the performance of our XPath query engines. In addition, we use our performance model, defined in Chapter 3, in an attempt to estimate average query execution time.

6.1 Data Sets

The proliferation of XML has resulted in a wide range of easily obtainable real-world data sets and benchmarks. We choose three such data sets to test our performance model and XPath query engines. The three data sets we opt to use are:

- DBLP - "Digital Bibliography & Library Project" is a large XML database containing information about over 2.7 million academic publications,
- XMark - an XML Benchmark generator program that uses the works of Shakespeare to generate XML files that mimic the structure of real-world data sets, and
- xOO7 - an XML generator that is customizable with several parameters and includes a set of corresponding queries.

Our goal is to use these data sets to determine the accuracy of our performance model and evaluate the performance of our XPath query engines. For each data set, we identify and run a series of queries that traverse different portions of the XML document tree and exhibit different characteristics (i.e., different values of D , S , T). All results are averaged over 100 runs for consistency and duplicated on our two test environments, described in Section 4.1.

6.2 Initial Results

Our initial testing consists of measuring the execution time of a series of queries on each data set. We use our Sequential and two parallel XPath query engines and compare the results to determine parallel speedup. For these queries, we manually calculate our performance model parameters (D , B , S , and T). This is accomplished by averaging the individual parameters at each level of the tree (the details are further explained in Section 6.3). Applying these parameters to our performance model allows us to estimate the query execution time. The accuracy of our performance model is measured as the relative error between the performance model estimate and the empirical query execution time. It is calculated as $\frac{Estimate-Run}{Run}$. We apply this process to our three real-world data sets.

6.2.1 DBLP

The DBLP data set is a large (900MB) XML file containing nearly 25 million lines [12]. The structure of the XML file, once parsed and loaded into memory, results in a very shallow ($D = 2$) tree with widely varying branch factor. The root node ($D = 0$) has nearly approximately 2.7 million children, yet at the subsequent level, we see a much smaller branch factor (ranging from $B = 5$ to $B = 100$). The resulting tree has a simple structure, though it is characteristic of some large XML repositories used by real-world applications. It is an XML file structure that is indicative of applications that archive large numbers of entries, each of which contains only a small amount of information. These types of applications are relevant to this study, as they frequently require fast and efficient querying to access information on specific entries.

To determine the performance of our XPath query engines on this real-world data set, we identify a series of queries that traverse different portions of the data set. The simple structure of the DBLP data set limits us to shallow queries that exhibit a high branch factor and selectivity. We choose four queries that execute in a reasonable time (1-100 seconds), return diverse results, and have varying parameters (B , S , T). The details of each of these queries are shown in Table 6.5. Table 6.1 outlines the average execution time of these queries, run on Greenwolf by our three XPath query engines, along with the estimates from our performance model.

One difficulty we encounter with this data set is the shallow depth and large initial branch factor. As we can see in Table 6.1, this results in poor parallel performance for most queries, with an overall average speedup on Greenwolf of 1.24 (DBLP queries could not be executed on Navet, as it lacked sufficient memory). The minimum context depth that provides enough independent work units

Query ID	Seq. Run	Fixed Work Run	Work Queue Run	Seq. Estimate
Q_{dblp1}	1.37 sec.	1.29 sec.	1.23 sec.	1194.88 sec.
Q_{dblp2}	5.17 sec.	2.67 sec.	2.61 sec.	46677.99 sec.
Q_{dblp3}	1.31 sec.	1.37 sec.	1.42 sec.	2.06 sec.
Q_{dblp4}	1.25 sec.	1.23 sec.	1.35 sec.	33112.57 sec.

Table 6.1. Query execution runtimes and performance model estimates using the DBLP data set on the Greenwolf environment. We see poor parallel speedup for most queries and very inaccurate performance model estimates. Details about each query are listed in Table 6.5

to begin parallel execution ($C = 1$) results in a large number of nodes. Not only does this require a long sequential execution, a large number of nodes must be distributed among the processor cores. In addition, the Work Queue engine must maintain a large work queue, resulting in even more overhead.

While we expected some difficulties when applying our performance model to real-world data, we did not expect the vastly incorrect estimates that we see in Table 6.1. Our performance model estimates are completely inaccurate for most queries run on the DBLP data set, with estimates several orders of magnitude away from empirical execution time, and an average relative error of 495,275.91%. In Section 6.3, we attempt to identify the cause of these inaccuracies by analyzing the methods we use to determine average parameter values, such as B and S .

6.2.2 XMark

The XMark XML benchmark software consists of an XML file generator that creates sets of tags filled with large amounts of data [18] [15]. The data that fills each tag is created by randomly pulling text from the works of Shakespeare. The size of the data set created is controlled by a single parameter, the scaling factor.

To test the performance of our XPath query engines on the XMark benchmark, we create a set of XML files ranging from 10MB to 1GB using the XMark data set generator. We analyze these files to identify several applicable queries that traverse different sections of the XML document tree. To get a broad profile of the performance of our XPath query engines on XMark data sets, we select queries with varying parameters, including depth. The details of these queries are outlined in Table 6.5. We run each of these queries on every XMark data set we create, using our three XPath query engines. All tests are duplicated on our two hardware environments and, as with previous experiments, we average execution time over 100 runs. The results of one of these queries on the Greenwolf environment are shown in Figure 6.1.

The results of our XMark tests indicate we achieve good parallel speedup of queries on XMark data sets using the Greenwolf environment. We achieve an average parallel speedup of 3.27 over 4 queries on varying XMark data sets. Additionally, speedup remains consistent over all queries and all data sets, with a minimum speedup of 2.59 and a maximum of 3.88. Note that perfect speedup on Greenwolf is 4.0 (4 cores). On the dual-core Navet environment, however, we see no parallel speedup when using 2

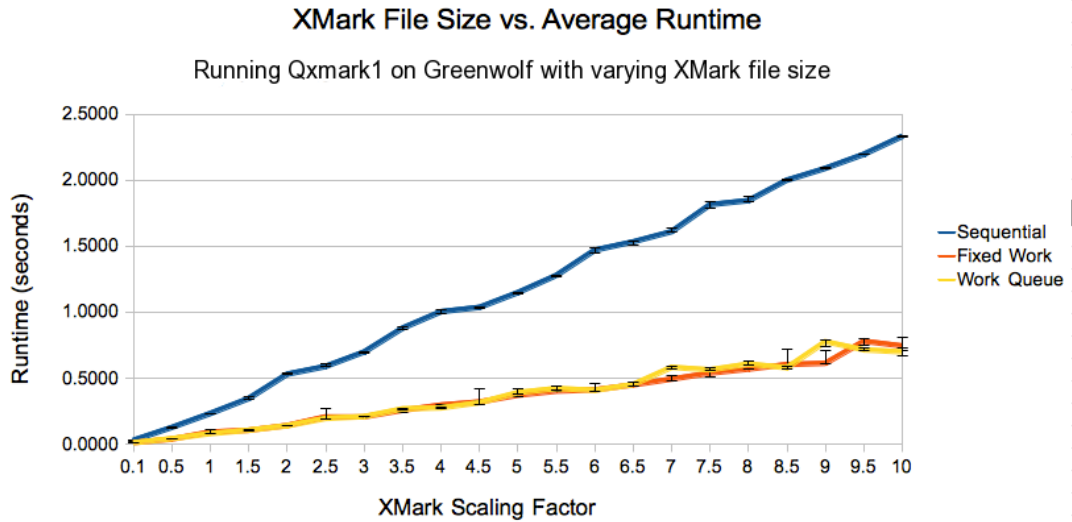


Figure 6.1. Average execution time of Q_{xmark1} on Greenwolf with each of our three XPath query engines. Execution time increases linearly as we increase the XMark scaling factor. We see good parallel speedup on Greenwolf for all queries of on XMark data sets. Details about Q_{xmark1} and other XMark queries available are in Table 6.5.

threads, with an average speedup of 0.92. When we use more than 2 threads, we begin to see parallel speedup. Figure 6.2 shows the average execution time of Q_{xmark1} on Navet while increasing the number of threads.

Despite having only 2 processor cores, we see in Figure 6.2 that the average query execution time on Navet decreases as we add additional threads. This parallel speedup continues until 8 threads using Fixed Work and 4 threads using the Work Queue query engine. At this ideal number of threads, we see an average parallel speedup of 1.39 using Fixed Work and 1.32 with Work Queue. We postulate that assigning additional threads increases parallel performance by alleviating load imbalance, enabling further use of hyperthreading, and reducing memory contention.

Using our performance model, we develop query execution time estimates for each query on each XMark data set. As we did with DBLP, we manually obtain query parameter values (B, S, T) and calculate averages. This process is explained in more detail in Section 6.3. Our performance model uses these parameters to estimate average query execution time. We determine performance model accuracy by comparing these estimates to empirical query runtimes. Figure 6.3 shows the average query execution time and performance model estimate of Q_{xmark1} on both Greenwolf and Navet.

As we see in Figure 6.3, the performance model estimates vary greatly from our empirical results, with an exponentially increasing error rate between them. Our performance model estimates are even more inaccurate with the XMark data set than DBLP, with an average relative error on the order of $10^{11}\%$, and a maximum of $7.48 \times 10^{11}\%$. Note that relative error is calculated as $\frac{Estimate - Result}{Result}$. Like DBLP, XMark data sets have a wide variance of parameter values from level to level. We postulate that this causes the averaging process to be inaccurate, giving us completely inaccurate performance model estimates. We investigate this further in Section 6.3.

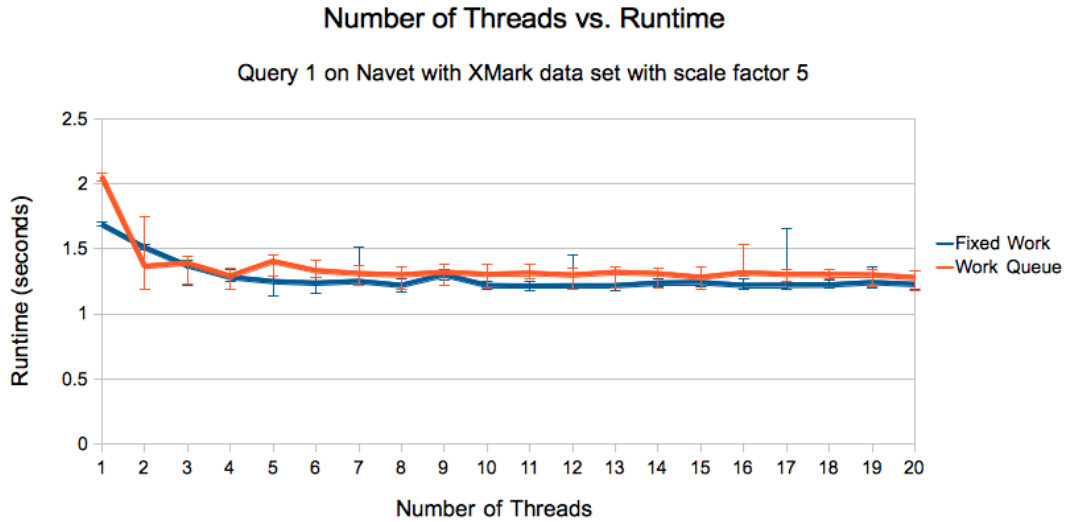


Figure 6.2. Average query execution time of $Q_{xmark}1$ on Navet while increasing the number of threads for both the Fixed Work and Work Queue query engines. We see that, although Navet has 2 cores, using 2 threads does not yield good parallel speedup. We determine the ideal number of threads to be 4 for Fixed Work and 8 for Work Queue. Details about the query $Q_{xmark}1$ are available in Table 6.5.

6.2.3 xOO7

The xOO7 benchmark is an XML benchmark that is freely available online [15]. It includes an XML file generator and a set of queries, and has been used to test several XML management systems. The file generator includes preset configurations to create 54 different XML files. However, the files generated by the xOO7 benchmark are limited to smaller data sets, with the largest being 128MB. The trees created by the xOO7 XML files have a maximum depth of 9 with a large variance in branch factor between levels. At shallow depth ($D = 1$ to $D = 7$), the tree has a consistently low branch factor ($B = 1$ to $B = 3$). At the deepest levels the branch factor drastically increases, depending on the configuration used (up to $B = 3000$).

Due to the relatively larger maximum query depth and the low initial branch factors, we expect our parallel query engines to perform well on data sets generated by xOO7. The small initial S and B values provide our parallel execution engines with independent work units at a low sequential execution cost. Additionally, the large number of nodes at deeper tree levels provides ample work to be done in parallel. We expect to see significant parallel speedup using both parallel query engines (described in Chapter 4).

We identify two queries to run on data sets generated by xOO7. The *teardrop* shape of trees from xOO7-generated data sets (small branch factor until deep query depth) limits our available query options, so we choose two queries that exhibit diverse parameter values. The details of each of these queries are listed in Table 6.5. We perform each query on three data sets: small (4.4MB), medium (44MB), and large (128MB). Note that, $Q_{xOO7}1$ returns the same number of query matches regardless of the size of the data set used. The results of these tests are outlined in Table 6.2.

Average Runtime vs. XMark Scaling Factor

Comparing sequential runtime of Q1 and performance model estimate on Greenwolf and Navet

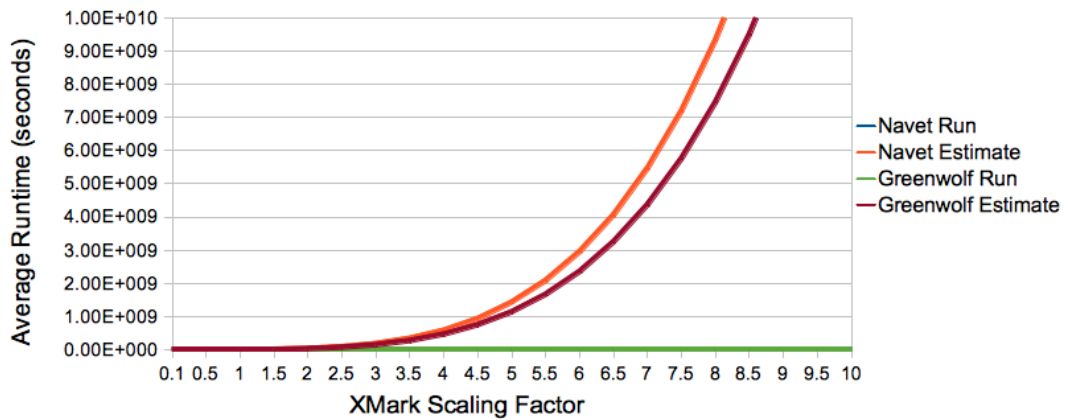


Figure 6.3. Empirical execution time and performance model estimates for $Q_{xmark}1$ executed sequentially on both Navet and Greenwolf. Performance model estimates increase exponentially at a very fast rate. The scale between performance model estimates and execution times is several orders of magnitude.

Query	Seq. Run	Fixed Work Run	Work Queue Run	Speedup	Seq. Estimate
$Q_{xOO7}1$ -Large	0.2488 sec.	0.0968 sec.	0.0892 sec.	2.68	0.0562 sec.
$Q_{xOO7}2$ -Large	0.9188 sec.	0.3426 sec.	0.3650 sec.	2.60	1.29E+012 sec.
$Q_{xOO7}1$ -Med.	0.0813 sec.	0.0334 sec.	0.0282 sec.	2.65	0.0072 sec.
$Q_{xOO7}2$ -Med.	0.3147 sec.	0.1231 sec.	0.1137 sec.	2.66	4.61E+04 sec.
$Q_{xOO7}1$ -Small	0.0076 sec.	0.0036 sec.	0.0035 sec.	2.13	0.0011 sec.
$Q_{xOO7}2$ -Small	0.0309 sec.	0.0127 sec.	0.0355 sec.	2.28	8.213 sec.

Table 6.2. Measured execution times and performance model estimates for two queries run on 3 different xOO7 data sets. We see good parallel speedup on all queries but completely incorrect performance model estimates. Speedup shown is the average speedup for both parallel query engines. The "small" data set is 4.4MB, "medium" is 44MB, and "large" is 128MB.

As expected, we achieve significant parallel speedup with both the fixed work and work queue query engines on the Greenwolf environment. However, as we see in Table 6.2, our performance model estimates are very inaccurate. For $Q_{xOO7}1$ on the Large data set, our performance model underestimates the query execution time by a factor of 4. For $Q_{xOO7}2$ on the Large data set, our model overestimates execution time by a factor of 10^{11} . This overestimation is similar to what we encountered with both the DBLP and XMark real-world data sets. We postulate that the overestimation of query execution time on real-world data sets is caused by an inaccurate estimation of parameter values (B and S). In the next section we investigate this assumption.

Depth	$Q_{xOO71} S$	$Q_{xOO71} B(q)$	$Q_{xOO72} S$	$Q_{xOO72} B(q)$
1	1	1	1	1
2	1	3	1	3
3	3	1.33	3	1.33
4	3	3	3	3
5	3	3	3	3
6	3	3	3	3
7	3	3	3	3
8	1	1351	1800	1351.5
9	2	0.0017	2	3
Average	2.22	152.04	202.11	152.43
Variance	0.94	202151.49	359051.36	202188.38

Table 6.3. Table of depth-specific average parameters for two queries on the large xOO7 data set. The large variance between individual S and $B(q)$ values and the average is the cause of many errors in performance model estimates. Note that the query-specific branch factor, $B(q)$ is listed for each of the two queries. Details about each query can be found in Table 6.5.

6.3 Weaknesses of the Performance Model

In the previous section, we tested our XPath query engines on three real-world data sets. These tests revealed inaccuracies in most performance model estimates, with relative error rates up to a factor of 10^{11} . In this section, we further examine these inaccuracies and attempt to identify their cause.

One aspect of real-world data sets that we encountered in all of our tests is the large level-by-level variance in parameter values, especially B and S . To help us perform query-specific analysis of performance model inaccuracies, we introduce a slightly altered branch factor parameter $B(q)$. This parameter is the query-specific branch factor. It is determined by averaging the branch factor (number of children) of each node that is *visited by the query*. Unlike B , the parameter $B(q)$ can differ greatly between two different queries executed on the same data set. We look at $B(q)$ for individual queries/data sets in an attempt to determine the cause of our performance model inaccuracies.

An example of a data set with widely varying parameters between queries is DBLP. The DBLP data set at $D = 1$ has $B(q) = 1.7$ million, yet at $D = 2$, $B(q)$ ranges from 5 to 100, depending on the query. This wide variance of $B(q)$ between both levels and queries plays a large role in query execution time, yet this granular information is lost when the global average B is used.

For all tests involving real-world data sets, we manually calculate the global averages S , B , and T that are used by our performance model. We calculate these averages by measuring the individual values at each tree level and taking an average. This process of taking a simple average can result in inaccurate parameters for the performance model, which can lead to inaccurate performance model estimates. To demonstrate how this averaging process can cause inaccuracies, we look at two queries for which our performance model greatly underestimates and overestimates execution time. Table 6.3 shows the depth-specific parameter values at each depth of the query, along with the calculated global average and variance between level-specific values.

For Q_{xOO71} , our performance model underestimates query execution time by a factor of 4. For Q_{xOO72} , we have an overestimate by a factor of 10^{11} . The global average values of $B(q)$ and S (shown in the *Average* column in Table 6.3) are used by our performance model to estimate query execution time. We see that, for Q_{xOO72} , our average S is larger than $B(q)$, which is not possible for any query (there cannot be more *matching* children than total children). Additionally, the number of query matches estimated by our performance model for Q_{xOO72} is $202.11^{9-1} = 2.79 \times 10^{18}$, while the number of matching nodes actually returned by the query is 874,800. This overestimate in total node count is clearly a factor that is contributing to the inaccuracy of our query execution time estimate. To identify the cause of this problem, we formally define the process of measuring the parameters.

To define our calculation of depth-specific values of $B(q)$, S , and T , we revisit some of the definitions from Section 3.1. We define the individual $B(q)$, S , and T components at each depth i as:

$$\begin{aligned}
S_i &= \frac{|N_{match}^{q_i}|}{S_{i-1}} |q_i \text{ is the sub-query to depth } i \\
B(q)_i &= \frac{|children(S_{i-1})|}{S_{i-1}} \\
S_1 &= B(q)_1 = 1 \\
T_i &= \frac{\sum tag(n)}{|children(S_{i-1})|} |n \in children(S_{i-1}) \\
T_1 &= tag(n_{root})
\end{aligned}$$

The individual selectivity at depth i , S_i , is determined by the number of query sub-matches at depth i , for each query sub-match at depth $i-1$. Similarly, we calculate $B(q)_i$ by the total number of children of each sub-match from depth $i-1$. The depth-specific tag length, T_i , is just the average tag length of all nodes considered by the query at depth i . We calculate our global averages S , $B(q)$, and T by averaging all of the i components over the entire query depth:

$$\begin{aligned}
S &= \frac{\sum_{i=1}^D S_i}{D} \\
B(q) &= \frac{\sum_{i=1}^D B(q)_i}{D} \\
T &= \frac{\sum_{i=1}^D T_i}{D}
\end{aligned}$$

As demonstrated in Table 6.3, this simple method of averaging is one of the causes of the inaccurate performance model estimates we are seeing for real-world data sets. The use of global parameters degrades the overall accuracy of our model for non-uniform trees, especially when we have inter-level parameter variances are large (such as those of real-world data sets).

Query	Nodes	Characters	Navet Estimate	Navet Run	GW Estimate	GW Run
Q_{xOO71} -Large	439103	5704659	0.250 sec.	0.249	0.203 sec.	0.212 sec.
Q_{xOO72} -Large	1750574	13126731	0.897 sec.	1.191	0.711 sec.	0.920 sec.

Table 6.4. Estimates generated by the new performance model. Note that this performance model uses level-specific parameters to count the number of nodes evaluated by a query. Estimates are much more accurate than with the previous performance model.

6.4 Improving the Performance Model

Any solution to the issues of performance model inaccuracies encountered in the previous chapter requires more granular information about data and query-specific parameters. For example, if our performance model was able to gain the total number of nodes visited by a query, it would be trivial to obtain an accurate estimate. However, as we see in Section 6.3, broad parameter estimates provide us with very inaccurate performance estimates for real-world data sets. Therefore, we must determine how to obtain an accurate performance model without requiring unreasonably accurate and granular parameter values.

Using the depth-specific parameter values defined in the previous section, we develop a method of estimating the total number of nodes evaluated, and therefore the average query execution time. Recall from the previous section that we level-specific values to determine the average number of nodes and characters evaluated by query q at a specific depth. We can use this information to determine the total amount of work done:

$$\begin{aligned}
 W_{total} &= \alpha \times N_{total} + \tau \times T_{total} \\
 N_{total} &= \sum_{i=0 \dots D-1} |children(S_i)| \\
 T_{total} &= \sum_{i=0 \dots D-1} |children(S_i)| \times T_{i+1}
 \end{aligned}$$

These simple formulae allow us to count the total number of nodes visited by query q (N_{total}) and the total number of characters compared (T_{total}). Using this model, we estimate the query execution time of queries on the xOO7 data set, on which our previous performance model was most inaccurate. Table 6.4 contains the estimates using this new performance model for queries on the xOO7 data set.

The results of our new performance model indicate that it is much more accurate, with a maximum relative error of 32.7%. However, we note that this performance model uses level-specific values for each parameters. In most real-world applications, this level of information is not obtainable without explicitly analyzing the query and data set. The benefits of having an accurate performance model will not merit the cost of such detailed analysis. We conclude, therefore, that this new performance model is not practical for real-world applications.

While the performance model described in this section required too detailed of parameters to be practical, our previous performance model (defined in Chapter 3) is not detailed enough. An interesting area of future work may be to determine the least amount of data and query analysis required for accurate performance estimates of queries on real-world data. It is, however, beyond the scope of this thesis.

6.5 Conclusions

In this chapter, we used a series of real-world data sets and XML benchmarks to determine the performance our XPath query engines and the accuracy of our performance models. We determined that, while our parallel query engines achieved significant speedup on several benchmarks, results were inconsistent, particularly on the DBLP data set. In the next Chapter, we propose two new XPath query engines that attempt to achieve better parallel performance.

We found that our performance model was vastly inaccurate when attempting to estimate query execution time on real-world data sets. In Section 6.3 we determined that our performance model inaccuracies were caused by the use of global averages for query and data-specific parameters. We proposed a new performance model that produced much more accurate estimates, though we concluded that the detailed analysis required for its use was impractical for most applications.

Chapter 7

Parallel Performance Improvements

In the previous chapter we tested our previously defined XPath query engines using real-world data sets. For several of these data sets (XMark and xOO7), our parallel implementations achieved good parallel speedup. However, we discovered several weaknesses when querying some unbalanced synthetic data sets (Section 5.3), as well as some real-world data sets (Section 6.2.1). The major weaknesses we encountered with our parallel XPath query engines are load imbalance and sequential overhead. While our Work Queue engine improves load balance for some cases (Section 5.3), both parallel query engines suffer from performance degradation due to sequential overhead. Both the Fixed Work and Work Queue engines require an initial context depth be queried sequentially before parallel execution can commence. For certain types of data sets (Section 6.2.1) this can significantly increase query execution time.

In this Chapter, we propose two additional parallel XPath query engines that attempt to solve the issues we encountered with our fixed work and work queue engines. Our goal is to determine the viability and flexibility of these two engines on a range of data sets, both synthetic and real-world.

7.1 Dynamic Work Queue Engine

Our first new parallel XPath query engine is the Dynamic Work Queue engine. Similar to the work queue engine described in Chapter 4, we use a queue of work units shared by all process threads. With the Dynamic Work Queue engine, however, all threads actively read from and *write to* the queue. The ability of threads to dynamically write to the work queue completely eliminates the need for a sequential phase, as all threads can begin working as soon as there is a small amount of work on the queue. Additionally, since all threads can continually write surplus work to the queue, load imbalance is minimized. Although this new query engine removes the need for a sequential context depth, several additional parameters are introduced.

7.1.1 Implementation Details

Since, with the Dynamic Work Queue engine, all nodes continually both read from and write to the work queue, the queue must be carefully monitored to prevent overflow, lost work, or work duplication. We accomplish this by using a large circular buffer. All operations are performed atomically

through the use of semaphores to ensure mutual exclusion. We also carefully monitor the size of the queue at each write operation, only writing as much as will fit into the work queue. The size of the work queue is controlled through a *buffer size* parameter, N_{buff} . This parameter controls the size of the circular buffer used. The overhead of allocating and maintaining a very large circular buffer is offset by its load balancing benefits. The ideal size of our circular buffer is dependent on several other new parameters.

Each operation performed on the shared work queue has an overhead associated with both the operation itself (read or write) and the locking required to prevent race conditions. We introduce two new parameters, *read size* N_{read} , and *write size* N_{write} that control the number of nodes read from and written to the work queue at each operation, respectively. While large values of N_{read} and N_{write} reduce the amount of locking required, small values promote load balance. These values must be correctly tuned to prevent the work queue from becoming full (threads waiting to write) or empty (threads waiting to read).

To further control how threads interact with the work queue, we introduce the *write frequency* parameter, F_{write} , that specifies how often threads write to the work queue. Each time a thread processes F_{write} nodes, it writes back N_{write} nodes to the work queue. This parameter, along with N_{write} , controls the balance between work queue overflow and thread starvation.

The final parameter introduced for the Dynamic Work Queue engine is the *work depth*, W_{depth} . This parameter controls the number of levels recursed by a thread before writing back to the work queue. Table 7.1 contains the parameters introduced for the Dynamic Work Queue engine, along with those for our other new XPath query engine, the Producer-Consumer engine (described in Section 7.2). We quantify the impact of each of these parameters on query execution time and determine their best values for our test environments in Section 7.3.1.

Incorporating all of these parameters, Algorithm 4 outlines the execution process of each thread using the Dynamic Work Queue engine.

As the pseudocode in Algorithm 4 demonstrates, reading from and writing to the work queue is controlled by the parameters described above. They allow us to specify the frequency and size of reads and writes. Note that each read and write requires locking and unlocking the shared work queue to prevent race conditions, so they are potentially costly operations.

7.1.2 Strengths and Weaknesses

We expect the Dynamic Work Queue engine to provide performance benefits over our previous parallel query engines, described in Chapter 4, with improved load balance and the elimination of the required sequential context. However, the Dynamic Work Queue engine has the additional overhead of allocating and maintaining a large circular buffer, as well as the cost of frequent locking and unlocking when working with the queue.

The operations of reading from and writing to the shared work queue may outweigh any performance gains provided by the Dynamic Work Queue engine. Although the correct parameter choices

```

function THREADWORK() begin
  while other threads still working do
    while localWork.Empty() ≠ True do
      workNode = localWork.frontElement;
      EVALNODE(workNode);
      SharedWorkQueue.write(toWrite, Nwrite);
    end
    SharedWorkQueue.get(localWork, Nread)
  end
end

function EVALNODE(workNode) begin
  foreach child of workNode do
    if child.tag == query.tag then
      if child.depth == query.maxDepth then
        | MatchResults.add(child);
      end
      if recursionCount == Wdepth then
        | toWrite.add(child);
      end
      else
        | recursionCount++;
        | EVALNODE(child);
      end
    end
    if evaluatedSiblings == Fwrite then
      | SharedWorkQueue.write(toWrite, Nwrite);
      | evaluatedSiblings = 0;
    end
    evaluatedSiblings++;
  end
end

```

Algorithm 4: Pseudocode for the Dynamic Work Queue engine. This outlines the functions that are called by each thread. Note that each thread is simultaneously running the THREADWORK() function.

Parameter	Name	Defined in	Informal Explanation
N_{buff}	Buffer Size	Section 7.1.1	Circular Buffer Size
N_{read}	Read Size	Section 7.1.1	Number of nodes taken per read
N_{write}	Write Size	Section 7.1.1	Number of nodes written per write
F_{write}	Write Frequency	Section 7.1.1	Maximum evaluations between writes
W_{depth}	Work Depth	Section 7.1.1	Maximum recursions per work unit
N_{keep}	Keep Nodes	Section 7.2.1	Number of nodes kept by Producer during write

Table 7.1. List of parameters used by the Dynamic Work Queue and Producer-Consumer query engines. For a more complete definition of a parameter, see the section it is defined in. The impact of each parameter on query execution time is measured in Section 7.3.

may help us minimize overhead, we expect it to degrade performance, especially when increasing the number of processor cores.

7.2 Producer-Consumer Engine

In the previous section, we introduced the Dynamic Work Queue parallel XPath query engine, in which all threads dynamically read from and write to a shared circular buffer. While it provides benefits in load balance and parallelization, we expect a significant performance overhead due to the constant interaction with the work queue. In this section, we propose a variation of the Dynamic Work Queue engine called the Producer-Consumer engine.

7.2.1 Implementation Details

The Producer-Consumer XPath query engine uses the concept of the producer-consumer relationship for work distribution and query processing. For this implementation, a single producer thread is assigned while all other threads are designated as consumers. The producer performs small sub-queries and writes results to the shared work queue while consumers only read from the queue. Since writing to the work queue is a costly operation (requires locking), this approach reduces much of the overhead introduced with the Dynamic Work Queue engine while retaining many of the benefits. Like the other query engines proposed in this thesis, the Producer-Consumer engine has its own set of parameters that determine the details of query execution.

The Producer-Consumer engine, like the Dynamic Work Queue engine, employs a large shared circular buffer to distribute work units among processor cores, so it also uses the parameters N_{buff} , N_{read} , and F_{write} (defined in Section 7.1). The remaining parameters used by the Dynamic Work Queue engine are not used by Producer-Consumer, and we introduce one final parameter, the *keep node count*, N_{keep} . N_{keep} defines the minimum number of nodes retained by the producer thread at all times.

Since consumer threads never write back to the shared work queue, the producer thread must retain some amount of work to be able to continue working. If the producer runs out of work and the consumer threads remove all available work units from the queue, the producer thread will terminate

while consumer threads may still have a substantial amount of work to complete. If this occurs early in execution, the consumer threads may be load-imbalanced, leading to poor parallel performance. Therefore, we use the N_{keep} parameter to control the number of nodes written to the shared work queue and to ensure the producer has enough nodes to continue working. The details of execution for the producer thread are shown in Algorithm 5.

The producer thread evaluates nodes one level at a time (no recursion is performed). Note that this thread behaves like each thread in the Dynamic Work Queue engine (Section 7.1), with the W_{depth} parameter set to 0. The consumer threads, however, behave more like the threads of the Work Queue engine (described in Chapter 4). They simply read from the shared work queue and evaluate all descendents of each node. Algorithm 6 outlines the process of each consumer thread.

The Producer-Consumer engine attempts to gain the benefits of the Dynamic Work Queue engine with less overhead. The simplicity of the consumer threads reduces the overhead while the producer thread is designed to improve load balance and eliminate sequential context.

7.3 Parameter Calibration

Our two new XPath query engines use a series of new parameters to control query execution behavior (listed in Table 7.1). These parameters may play a large role on performance, so in this section we measure the impact of each parameter on execution time and attempt to determine the best parameter configuration. All results in this section are averaged over 100 runs for consistency and duplicated on each of our test environments. We repeat all tests on a range of synthetic data sets to determine if the XML tree shape (B, D) or query details (S, T) have an effect on the impact of each parameter on query execution time.

7.3.1 Dynamic Work Queue Parameters

The parameters used by the Dynamic Work Queue engine are: N_{read} , N_{write} , F_{write} , and W_{depth} . In this section we evaluate the impact of each of these parameters on execution time and attempt to find the best configuration.

The parameters N_{read} and F_{write} , together, control the balance between an empty queue (thread starvation) and a full queue (threads waiting to write). Since we expect the interaction between these two parameters to be complex, we measure their combined impact on query execution time by running tests while varying both. The matrix of query execution times for one of these tests is displayed in Figure 7.1.

Our results indicate that increasing N_{read} (read size) increases overall query execution time, especially at small values of F_{write} . We see in Figure 7.1 that the best parameter configuration is $N_{read} = 1$ and $F_{write} = B$ (B is the maximum F_{write} can be when $W_{depth} = 0$). We see the fastest query execution time when N_{read} is minimum and F_{write} is maximum. We postulate that a large F_{write} provides minimal

```

function PRODUCER() begin
  while localWork.isEmpty()  $\neq$  True do
    workNode = localWork.frontElement;
    QUERYPRODUCER(workNode);
    needToKeep =  $N_{keep}$  - localWork.size;
    if needToKeep  $\leq$  0 then
      | localWork.write(toWrite, needToKeep);
    end
    SharedWorkQueue.write(toWrite);
  end
  Producer.isFinished() = True;
end

function QUERYPRODUCER(workNode) begin
  foreach child of workNode do
    if child.tag == query.tag then
      | if child.depth == query.maxDepth then
        | | MatchResults.add(child);
      | end
      | else
        | | toWrite.add(child);
      | end
    end
    if evaluatedSiblings ==  $F_{write}$  then
      | SharedWorkQueue.write(toWrite);
      | evaluatedSiblings = 0;
    end
    evaluatedSiblings++;
  end
end

```

Algorithm 5: Pseudocode for the producer thread of the Producer-Consumer engine. Note that this thread never recurses and simply writes sub-matches to the shared work queue for consumers to take.

```

function CONSUMER() begin
  while Producer.isFinished  $\neq$  True do
    while localWork.isEmpty()  $\neq$  True do
      | workNode = localWork.frontElement;
      | QUERYCONSUMER(workNode);
    end
    | SharedWorkQueue.get(localWork, Nread);
  end
end

function QUERYCONSUMER(workNode) begin
  foreach child of workNode do
    if child.tag = query.tag then
      | if child.depth = query.maxDepth then
      | | MatchResults.add(child);
      | end
      | else
      | | QUERYCONSUMER(child);
      | end
    end
  end
end

```

Algorithm 6: Pseudocode for a consumer thread in the Producer-Consumer engine. Consumer threads never write to the shared work queue. They simply read from the work queue and process nodes' descendents all the way to the leaves.

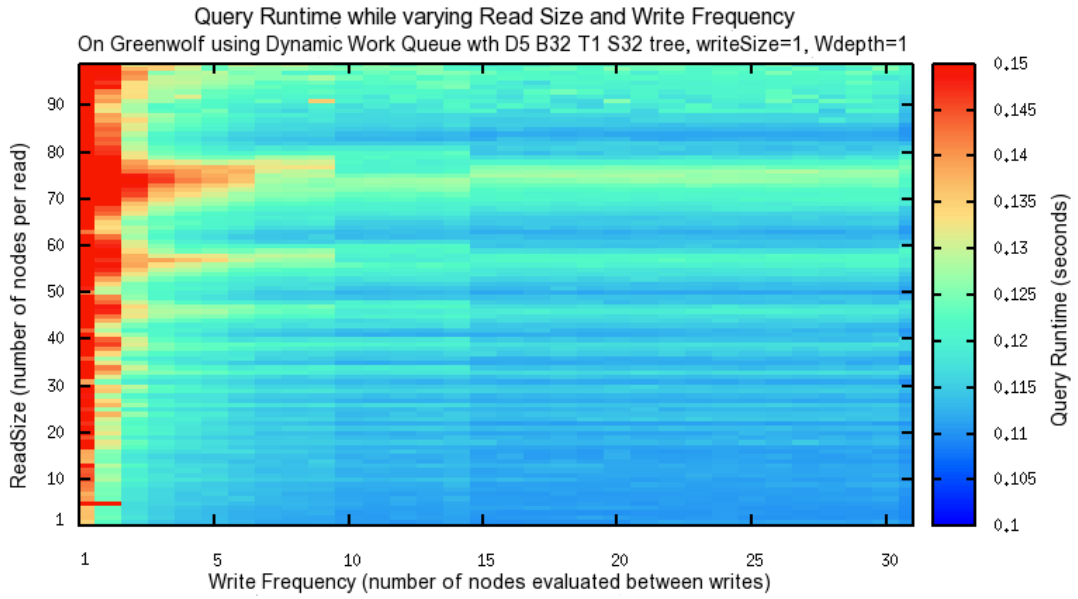


Figure 7.1. The combined impact of the parameters N_{read} and F_{write} on query execution time using the Dynamic Work Queue engine. We see that both parameters have an impact on query execution time, and that the best configuration for our test environments is $N_{read} = 1$ and $F_{write} = B$. This result is corroborated by our other tests.

overhead from writing to the work queue. A minimal N_{read} prevents thread starvation by leaving excess work available on the work queue.

The Dynamic Work Queue engine uses the N_{write} parameter to control the number of nodes written to the work queue per write operation. While small values of this parameter allow threads to retain some amount of work when writing, it may cause load imbalance by depriving other threads from work. To measure the impact of this parameter, we run queries on a range of synthetic tests while varying the N_{write} parameter. We repeat these tests using several values for N_{read} and F_{write} (small values, large values, and best as determined above). Figure 7.2 shows the results of several of these tests on the Greenwolf environment.

The results shown in Figure 7.2 indicate that the parameter N_{write} has no significant impact on query execution time, regardless of the values of the other parameters. Thus, we use the value of $N_{write} = 1$ for further testing, unless otherwise indicated. Additionally, Figure 7.2 verifies the results obtained above for the best configuration of N_{read} and F_{write} . The fastest execution time is measured when $N_{read} = 1$ and $F_{write} = B$.

The final parameter used by the Dynamic Work Queue engine, W_{depth} , determines the number of tree levels recursed by a thread before writing results back to the work queue. Since this parameter helps balance the frequency of work queue operations with load balance, we expect it to have a significant impact on query execution time. If $W_{depth} \geq \max \text{query depth}$, all work is done by one thread and the query is executed sequentially. If W_{depth} is too small, however, we expect overhead from the large amount of reads and writes to work queue. We expect the best value for W_{depth} to be dependent on the

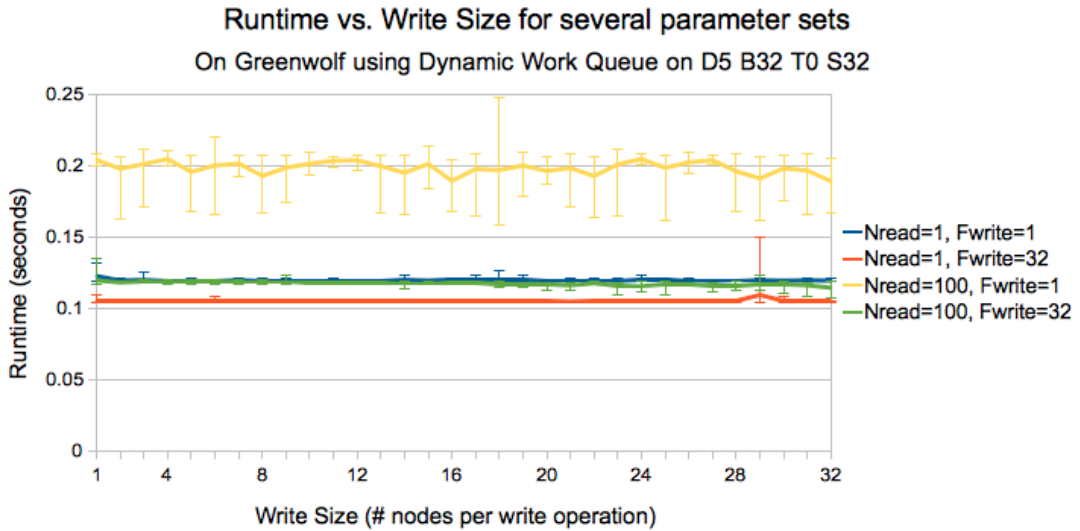


Figure 7.2. The impact of the N_{write} parameter on query execution time using the Dynamic Work Queue engine. We see that N_{write} has no significant impact on query execution time. Results shown for several values of N_{read} and F_{write} .

data set and query, since the maximum query depth is dependent on the XML document. To test the impact of this parameter on execution time, we use the best configuration for our parameters (determined above) and run synthetic tests while varying W_{depth} . Figure 7.3 shows the results of some of these tests on the Greenwolf environment.

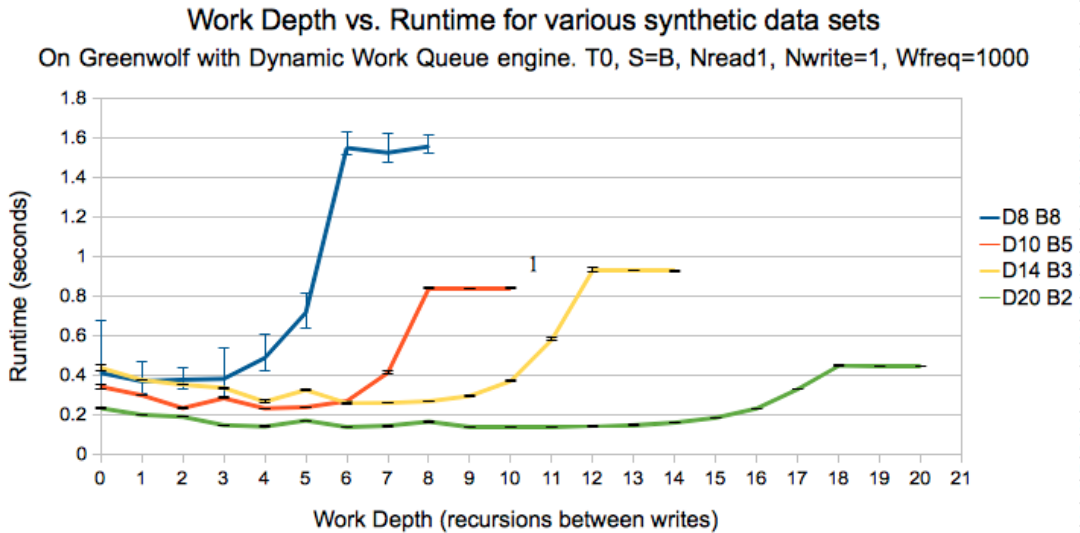


Figure 7.3. The impact of the work depth parameter (W_{depth}) on query execution time using several synthetic tests. We see that, at low values, execution time is erratic, followed by a gradual increase as W_{depth} increases. The fastest runtime is obtained when $W_{depth} \sim \frac{D}{2} - 1$.

Our experiments to measure the impact of W_{depth} on query execution time show us that it is dependent on the total query depth. However, regardless of the data set being used, query execution time follows the same pattern as we increase W_{depth} . We see in Figure 7.3, that the best value for

W_{depth} , regardless of all other parameters, is $\sim \frac{D}{2} - 1$. This formulation is consistently accurate for all of our tests cases on all test environments. We postulate that when $W_{depth} \sim \frac{D}{2} - 1$, work queue interaction is minimized while we achieve adequate load balance. When $W_{depth} = \frac{D}{2} - 1$, nodes are written only at 2 specific depths: $\frac{D}{2} - 1$ and $D - 1$. This means that the first $\frac{D}{2} - 1$ levels are performed sequentially by the first thread. Work is then written to the work queue and distributed among all threads. All threads continue work until Depth $D - 1$ where all work is written to the queue again and work is again evenly distributed. Since the most work is done at the leaf node level, re-balancing the load at this depth is beneficial and provides the fastest query execution time. Therefore, for all further tests we use $W_{depth} = \frac{D}{2} - 1$, unless otherwise mentioned.

7.3.2 Producer-Consumer parameters

As with the Dynamic Work Queue engine, the Producer-Consumer engine uses several new parameters that control the details of query execution. Although it uses some of the same parameters as the Dynamic Work Queue engine, we repeat the tests from Section 7.3.1 using the Producer-Consumer engine to determine if the parameters have a different impact on query execution time.

The parameters N_{read} and F_{write} , as with the Dynamic Work Queue engine, have a combined impact on query execution time by controlling interaction with the shared work queue. Therefore, we measure the combined impact on query execution time, as we did in the previous section. We repeat the test from Section 7.3.1 (results shown in Figure 7.1) using the Producer-Consumer engine to determine the impact of N_{read} and F_{write} on query execution time. Figure 7.4 shows the results of one of these tests on the Greenwolf environment.

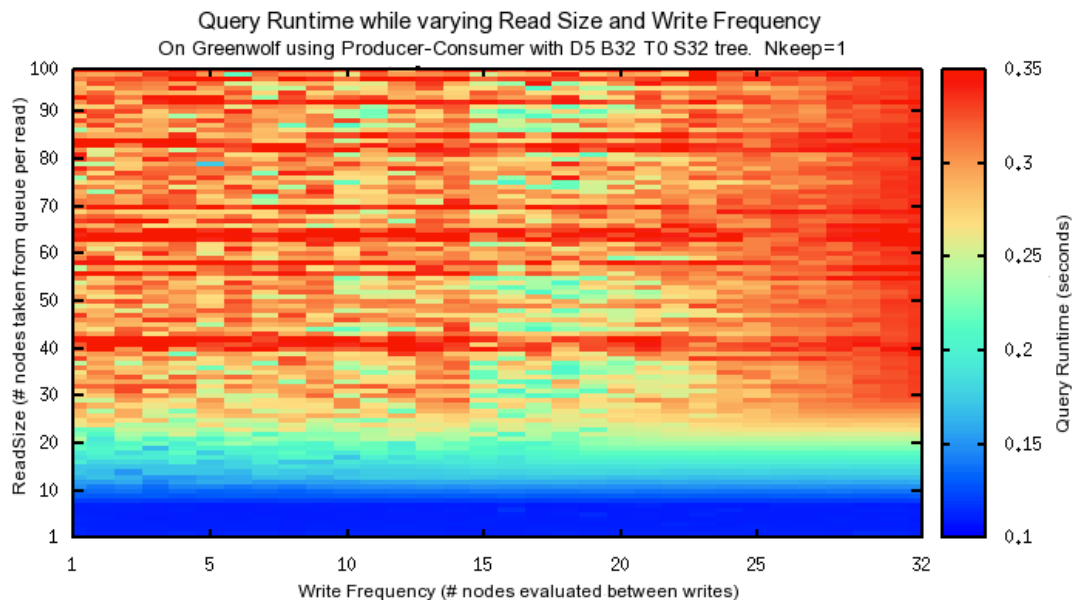


Figure 7.4. The impact of the parameters N_{read} and F_{write} on query execution time for the Producer-Consumer engine. We see that the ideal parameter configuration for our test environments is $N_{read} = 1$ and $F_{write} = 1$.

The results of these tests indicate that, while F_{write} has no significant impact on query runtime, increasing the N_{read} parameter greatly increases query runtime. As seen in Figure 7.4, increasing N_{read} from 1 to 100 greatly increases query execution time, with an increase of 203.27% (averaged over all tests on Greenwolf). While the N_{read} parameter has a similar effect with the Dynamic Work Queue engine (see Figure 7.1), we see a much more drastic increase using Producer-Consumer. We postulate that this is due to thread starvation. Since Producer-Consumer has only a single thread producing work, a high N_{read} value can cause one consumer thread to clear the work queue when reading, causing other threads to wait for work to become available. The Dynamic Work Queue engine is not as affected by this, since all threads continually write to the queue, making more work available. We determine that the best parameter configuration is $N_{read} = 1$ and $F_{write} = 1$ for the Producer-Consumer engine, so we use this configuration for all further testing unless otherwise mentioned.

The Producer-Consumer engine has one additional parameter, N_{keep} , that determines the number of nodes retained by the producer thread. This parameter is used to insure that the producer thread is able to continually produce work for the consumer threads. As noted in Section 7.2.1, this parameter is used to prevent the producer thread from running out of work. If this parameter is too small, the producer runs the risk of terminating prematurely, thus eliminating the load-balancing benefits it provides. If N_{keep} is too large, however, there is a chance the consumer threads could spend time waiting for work to be available. We test the impact of this parameter by using the best values of each other parameter, determined above. We run queries on a series a synthetic tests while varying only the N_{keep} parameter. Figure 7.5 shows the results of several of these tests on the Greenwolf environment.

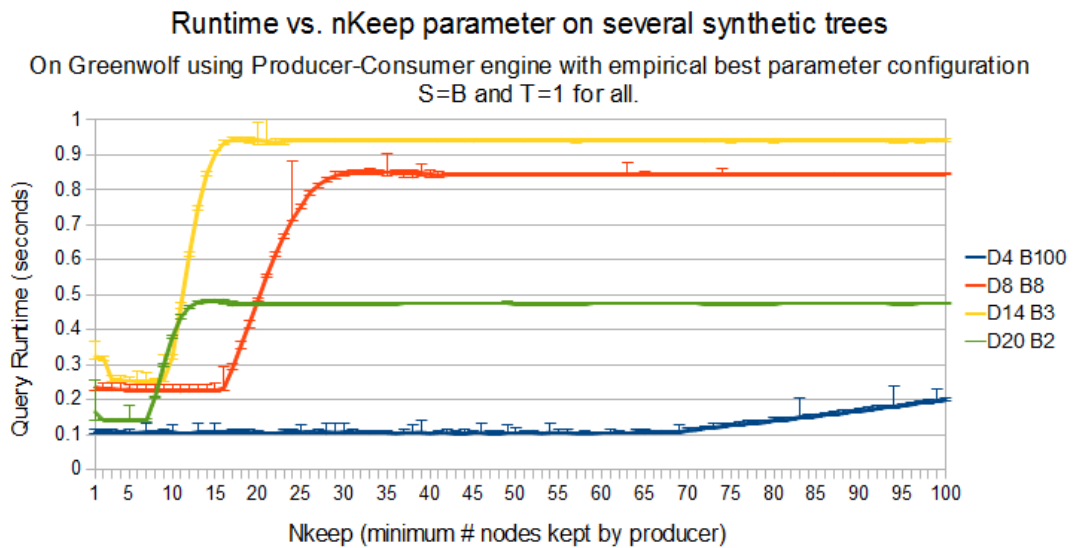


Figure 7.5. The impact of the parameter N_{keep} on query execution time with the Producer-Consumer engine. Results shown for several synthetic trees. All results are obtained on complete trees with all matching nodes, thus we see that a larger N_{keep} value prevents consumer threads from getting work. We revisit this parameter in Section 7.4

As Figure 7.5 demonstrates, increasing N_{keep} increases query execution time by preventing work from being written to the work queue, thus leaving the consumer threads idle. However, the results presented in Figure 7.5 are obtained from queries on *complete* trees (all nodes have equal B). Additionally, the queries on these trees are complete matches ($B = S$). There is no risk of the producer thread terminating prematurely (as explained in Section 7.2.1) in any of these cases. In real-world tests, however, we may have to use a larger value of N_{keep} , especially on queries with low selectivity (S). Therefore, we duplicate the tests performed above with synthetic data sets using the DBLP data set (described in Section 6.2.1). Surprisingly, we find that, similar to the results using synthetic data sets, the ideal parameter value for real-world data sets is $N_{keep} = 5$. Therefore, we use the parameter value of $N_{keep} = 5$ for all future experimentation, unless otherwise noted. In the next section we run queries on synthetic and real-world data sets using the best parameter configurations we determined above.

7.4 Empirical Results

In Sections 7.1 and 7.2, we introduced our two new XPath query engines: Dynamic Work Queue and Producer-Consumer. We postulate that these query engines will provide improved load balance over our previous designs, but may increase overhead. We expect to see reduced overhead with the Producer-Consumer engine, however, compared to Dynamic Work Queue. In Section 7.3 we determined the best parameter configuration for our new XPath query engines. In this section, we use these parameter configurations when executing queries on an array of synthetic and real world tests. Our goal is to validate or invalidate our previous assumptions and compare query performance (execution time) with that of our three other XPath query engines (described in Chapter 4). We focus our experiments on areas where our previous XPath query engines performed poorly to determine if our new engines suffer from the same shortcomings.

7.4.1 Branch Factor Skew Revisited

In Section 5.3 we determined that our two previous parallel XPath query engines (Fixed Work and Work Queue) performed poorly when we introduced *branch factor skew* to synthetic data sets. Recall that a branch factor skewed tree has an uneven distribution of branch factors (number of children). We create branch factor skewed trees by setting an index range that contains nodes with large numbers of children, while other nodes have no children (see Figure 5.5). The results from Section 5.3 indicate that, while neither parallel XPath engine achieved good parallel speedup, the Fixed Work engine produced spikes in execution time that were slower than sequential query execution. We repeat these tests using our two new query engines (Dynamic Work Queue and Producer-Consumer). Figure 7.6 shows the results of these tests, along with the results from previous tests using our three other query engines.

Our results indicate that neither of our new XPath query engines suffer from load imbalance from branch factor skew. As Figure 7.6 demonstrates, both the Dynamic Work Queue and Producer-Consumer engines achieve good parallel speedup for these synthetic tests, with average speedup of 3.37

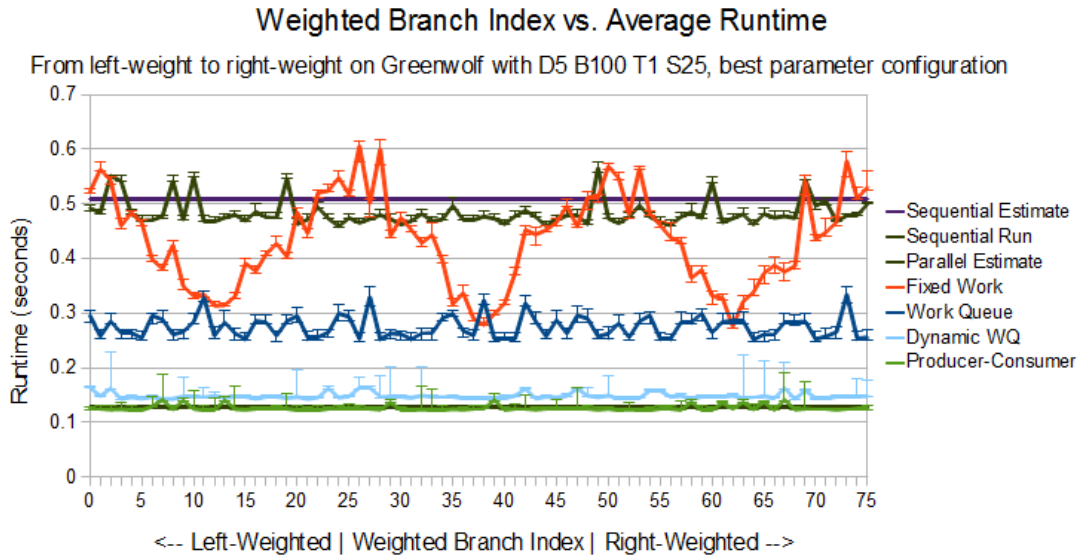


Figure 7.6. The results of repeating queries on branch factor skewed trees with our two new XPath query engines. Results also shown for query execution time using our three previous XPath query engines. We see that our two new query engines perform significantly better than previous engines. Both achieve good parallel speedup and do not suffer from the load imbalance our previous parallel engines experienced.

and 3.96, respectively, on the Greenwolf environment. Note that the Greenwolf environment has 4 processor cores, so perfect speedup is 4. Additionally, our new query engines maintain consistent parallel speedup regardless of the position of heavy nodes. These results indicate that our new XPath query engines perform well, even on tests on which our previous parallel query engines performed poorly. The Producer-Consumer engine performs exceptionally well, achieving nearly identical execution time as estimated by our parallel performance model, seen in Figure 7.6. Since branch factor skew, as well as other tree imbalances, are commonplace with real-world data sets, we expect our new XPath engines to perform well on them as well.

7.4.2 DBLP

Recall from Section 6.2.1 that the DBLP data set is a large (1GB) XML document containing data on a large number of scientific publications. One feature of DBLP is that the data set has a very shallow depth (only 3 levels including the root) with hugely varying Branch Factor. The root node contains 2.7 million children, each of which has between 10 and 100 children. These characteristics make parallelization particularly difficult with our previous query engines (see Chapter 4). Table 6.1 demonstrates the poor parallel performance of our Work Queue and Fixed Work engines.

We test the performance of our two new XPath query engines on the DBLP data set by running each of the 4 queries we selected in Section 6.2.1. Note that the Navet environment is unable to query DBLP due to insufficient memory. Table 7.2 shows the results of these 4 queries using our two new XPath query engines, along with the results using the Sequential, Fixed Work, and Work Queue engines.

Query ID	Sequential	Fixed Work	Work Queue	Dynamic WQ	P-C Run	P-C Speedup
Q_{dbl_p1}	1.368 sec.	1.290 sec.	1.233 sec.	1.302 sec.	1.198 sec.	1.14
Q_{dbl_p2}	4.827 sec.	2.669 sec.	2.607 sec.	2.216 sec.	1.360 sec.	3.55
Q_{dbl_p3}	1.311 sec.	1.371 sec.	1.419 sec.	1.233 sec.	1.220 sec.	1.08
Q_{dbl_p4}	1.254 sec.	1.225 sec.	1.353 sec.	1.254 sec.	1.198 sec.	1.05

Table 7.2. Query execution times of our give XPath query engines using the DBLP data set on the Greenwolf environment. We see poor parallel speedup for most queries, though the Producer-Consumer engine performs well on Q_{dbl_p2} . Details about each query are listed in Table 6.5

The results shown in Table 7.2 indicate that, despite significant parallel speedup on Q_{dbl_p2} , we see only very slight speedup on the other 3 queries. In an attempt to explain this discrepancy in parallel performance despite the queries being run on the same data set, we take a closer look at the execution of each query. Table 7.3 contains details about each query we run on the DBLP data set.

Query	B_1	S_1	S_2	Query Matches
Q_{dbl_p1}	2767177	18077	0.04	738
Q_{dbl_p2}	2767177	712521	2.5	1782468
Q_{dbl_p3}	2767177	9	5.56	50
Q_{dbl_p4}	2767177	8809	0.38	3319

Table 7.3. List of depth-specific selectivity values and total query matches for each query run on the DBLP data set. We postulate that the relatively higher selectivity of Q_{dbl_p2} is the reason we achieve much better parallel performance with it.

As seen in Table 7.3, the query Q_{dbl_p2} has a larger number of matches than the other queries we run on the DBLP data set (by a factor of at least 500). Additionally, the selectivity, S , associated with Q_{dbl_p2} is much larger than the other queries, especially at Depth $D = 1$, seen in boldface. We postulate that the low selectivity of each other query, combined with a large Branch Factor, B , and small query depth, causes load imbalance. Figure 7.7 demonstrates how this type of query can cause an uneven distribution of work among threads.

Since all of the 2.7 million nodes at depth $D = 2$ are children of the root node, they must all be processed by the producer thread. While processing these nodes, the producer writes query sub-matches to the shared work queue. Consumer threads read these nodes from the work queue and process their children. As Figure 7.7 demonstrates, a small selectivity results in fewer nodes written to the work queue, thereby leaving the consumer threads idle. We see in Table 7.2 that all queries of the DBLP data set with low selectivity (Q_{dbl_p1} , Q_{dbl_p3} , and Q_{dbl_p4}) result in poor parallel performance for all of our XPath query engines. However, on Q_{dbl_p2} , which has a high selectivity, we see good parallel performance with our two new query engines, especially the Producer-Consumer engine.

To verify our assumption that selectivity is the cause of our poor parallel speedup, we introduce two new queries for the DBLP data set, Q_{dbl_p5} and Q_{dbl_p6} (The details are available in Table 6.5). Like Q_{dbl_p2} , these two new queries have high S_1 values (selectivity at $D = 1$), so we should see good parallel

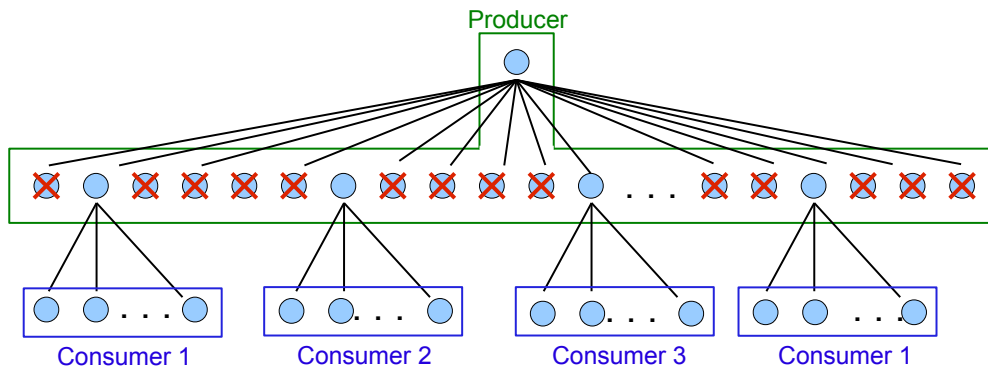


Figure 7.7. Demonstration of how low-selectivity queries on the DBLP data set can cause load imbalance. We see that the producer thread must perform more work than each consumer thread. This is an unavoidable consequence of the shape of the DBLP data set and the corresponding query selectivity.

speedup, especially with the Producer-Consumer query engine. The results of executing these queries are outlined in Table 7.4.

Query	Sequential Run	Fixed Work Run	Work Queue Run	DWQ Run	DWQ Speedup	P-C Run	P-C Speedup
Q_{dblp5}	6.659 sec.	2.639 sec.	2.576 sec.	2.634 sec.	2.53	2.060 sec.	3.23
Q_{dblp6}	2.648 sec.	1.747 sec.	1.761 sec.	1.828 sec.	1.45	1.270 sec.	2.08

Table 7.4. Results of running two additional queries on the DBLP data set. We identified these two queries as ones that we expect our parallel XPath query engines to perform well on. DWQ is Dynamic Work Queue and P-C is Producer-Consumer.

As expected, we see significant parallel speedup with these two queries, especially with the Producer-Consumer engine. We achieve significantly better parallel speedup for Q_{dblp5} than Q_{dblp6} , though with the Producer-Consumer engine, we see significant speedup for both queries.

We conclude that, while this type of data set/query degrades the performance of our new XPath query engines, the type of query and data set that causes it is extreme ($B_1 = 2.7$ million and $D = 3$) and should be a rare occurrence. Thus, we expect these new XPath query engines to perform better on our other real-world data sets.

7.4.3 XMark

The XMark data set generator, introduced in Section 6.2.2, is a set of scripts used to create XML files with high variability designed to mimic the structure of real-world data sets. They use the works of Shakespeare to fill the XML files with data. XMark uses a single *scaling factor* parameter to control the size of the XML file generated. We run a set of 4 queries, described in Table 6.5, on XML files generated using XMark. We repeat the test for a range of XMark scaling factor values. Figure 7.8 shows

the results of one of these queries using the Dynamic Work Queue and Producer-Consumer engines, along with the results using our previous XPath query engines (described in Chapter 4).

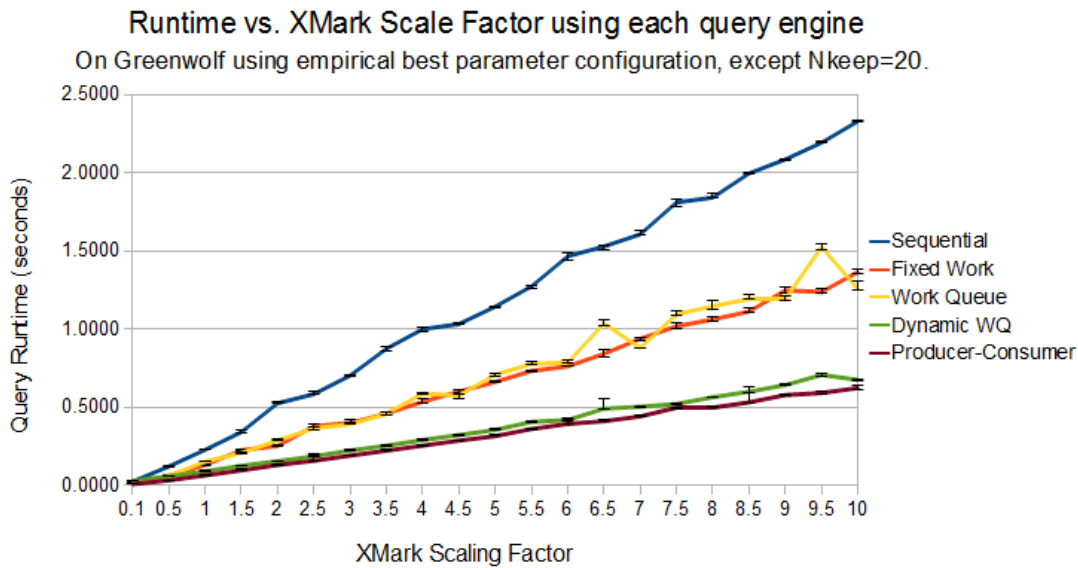


Figure 7.8. Results of running Q_{xmark1} on a range of XMark data sets on the Greenwolf environment. We see that our two new XPath query engines achieve significantly better parallel speedup than our previous engines.

As Figure 7.8 demonstrates, our two new XPath query engines achieve significantly better performance than any of our 3 previous engines (sequential, fixed work, work queue). The Dynamic Work Queue and Producer-Consumer engines both achieve good parallel speedup, averaging 2.11 and 3.42 respectively on Greenwolf. Note that Greenwolf has 4 cores, so perfect speedup is 4, meaning the Producer-Consumer engine achieved an average of 85.5% of ideal speedup. On all XMark tests, the Producer-Consumer engine performs better than the Dynamic Work Queue, with query execution times 39.63% faster, averaged over all XMark tests. We attribute this to the increased overhead required by the Dynamic Work Queue engine. These results indicate that our new XPath query engines, especially the Producer-Consumer engine, achieve significantly better load balance than our previous engines, even on data sets with large parameter skew (B and S vary greatly by level and sibling).

7.4.4 xOO7

Recall from Section 6.2.3 that xOO7 is an XML benchmark data set generator. It can be used to create a range of data sets of different sizes based on a series of configuration files. However, the size of these data sets are limited to at most 128MB. In Section 6.2.3 we identified 2 queries that we executed on 3 different data sets (queries detailed in Table 6.5). We repeat these queries using our two new XPath queries engines and compare execution time with the results obtained using our 3 previous query engines. Table 7.5 shows the results of the experiments on xOO7 data sets.

Our two new XPath query engines achieve consistent parallel speedup on queries over the xOO7 data set. As shown in Table 7.5, both the Dynamic Work Queue and Producer-Consumer engines

Query Name	Sequential Run	Fixed Work Run	Dynamic WQ Run	DWQ Speedup	P-C Run	P-C Speedup
$Q_{xoo7}1$ Small	0.0076 sec.	0.0036 sec.	0.0294 sec.	0.26	0.0060 sec.	1.27
$Q_{xoo7}1$ Med	0.0813 sec.	0.0334 sec.	0.0454 sec.	1.79	0.0268 sec.	3.03
$Q_{xoo7}1$ Large	0.2120 sec.	0.0903 sec.	0.0834 sec.	2.54	0.0743 sec.	2.85
$Q_{xoo7}2$ Small	0.0309 sec.	0.0145 sec.	0.0355 sec.	0.87	0.0137 sec.	2.25
$Q_{xoo7}2$ Medium	0.3147 sec.	0.1231 sec.	0.1070 sec.	2.94	0.1065 sec.	2.96
$Q_{xoo7}2$ Large	0.9487 sec.	0.3318 sec.	0.2673 sec.	3.55	0.3180 sec.	2.98

Table 7.5. Results of running 2 queries (details in Table 6.5) on 3 different data sets on the Greenwolf environment. Results listed for our 5 XPath query engines. We see that our two new XPath query engines achieve good parallel speedup using larger data sets. However, the Dynamic Work Queue engine shows slower execution times on very small data sets due to overhead. Note that results with the Work Queue engine are similar to Fixed Work but were omitted for space. The "small" data set is 4.4MB, "medium" is 44MB, and "large" is 128MB.

achieve good parallel speedup for most queries. However, on very short queries (execution time < 0.1 seconds), the Dynamic Work Queue engine sees significant *slowdown*, compared to sequential execution. We attribute this poor performance to the initial overhead associated with the allocation and management of the large shared work queue. On longer-running queries, both of our new queries engines perform well, with an average speedup of 1.99 with Dynamic Work Queue and 2.56 with Producer-Consumer on the Greenwolf environment (perfect speedup is 4).

The average speedup we see with queries on xOO7 data sets is less than that of results obtained using the XMark benchmark (Section 7.4.3). Since the xOO7 data sets are limited to smaller sizes (maximum of 128 MB), the overhead associated with our new XPath queries is more significant, relative to total query execution time. Despite this, both the Dynamic Work Queue and Producer-Consumers perform better than our previous engines.

7.5 Conclusions

In this chapter, we introduced two new parallel XPath query engines: Dynamic Work Queue and Producer-Consumer. These query engines use shared work queues with dynamic read-write capabilities in an attempt to improve load balance and eliminate the sequential execution component required by our previous parallel query engines (described in Chapter 4). The Dynamic Work Queue and Producer-Consumer engines each have a set of new parameters that control execution behavior. In Section 7.3 we measured the impact of each parameter on query execution time and determine the best configuration for our test environments. Using this empirically best parameter configuration for our two new XPath query engines, we measured the execution time of queries on a series of synthetic and real-world data sets. In Section 7.4 we determined that both the Dynamic Work Queue and Producer-Consumer engines achieve significant faster execution times than our previous parallel query engines. We found that the Producer-Consumer engine performs especially well, with good and consistent parallel speedup

on most real-world queries that we tested (e.g., Section 7.4.3). Overall, the new parallel XPath query engines perform significantly better than previous parallel engines on nearly all synthetic and real-world experiments.

Chapter 8

Conclusion

The proliferation of multi-core and multi-proc (multiple processor) architectures provide us with new opportunities and challenges in many application domains. XPath query evaluation is one such domain that has thus far been limited to sequential execution. While progress has been made to execute multiple queries in parallel [20], [21], attempts to efficiently parallelize single-query execution have been few.

We presented a detailed analysis of XPath query evaluation, introduced a suite of XPath query engines, and tested their performance on a range of synthetic and real-world data sets. In our initial analysis of XPath query execution in Chapter 3, we introduced two theoretical performance models (sequential and parallel). These performance models incorporate a series of query, data, and hardware specific parameters in an attempt to accurately estimate query performance. The goal of these performance models is to identify performance bottlenecks and methods of improving query engine performance (i.e., decrease execution time). We found in Chapter 4 that our performance models are adequately accurate on synthetic data sets, though in Chapter 6 we determined that our performance model parameters lack sufficient granularity to accurately estimate query execution time on real-world data sets. We attempted to improve on our performance model by using more detailed data and query-specific parameters, but determined that the detailed analysis to obtain such information would be impractical for real-world use. We conclude that, while these performance models helped us analyze XPath query evaluation behavior, the parameter granularity required for accurate real-world estimates is beyond what would be available without individual data and query analysis.

In Chapter 4 we presented 3 XPath query engines (Sequential, Fixed Work, and Work Queue) and used them to determine sequential and parallel query execution time on synthetic XML documents. Using our two hardware test environments (a 2-core Intel Xeon and a 4-core Intel Core2 Quad), we found that our parallel query engines performed well on synthetic tests, with good parallel speedup, especially on the the 4-core environment. In Chapter 5, however, we introduced skewed synthetic data sets and found that our parallel query engines (especially the Fixed Work engine) suffer from load imbalance when XML document trees have heavy branch factor skew. We applied our XPath query engines to three real-world XML benchmarks in Chapter 6 and determined that, while we achieved significant parallel speedup on some queries, results were inconsistent and we saw poor parallel performance on some data

sets and queries. Overall, we concluded that our two initial parallel XPath query engines (fixed work and work queue) suffer from load imbalance when querying real-world data sets.

The inconsistent performance of our two initial parallel query engines motivated us to implement two additional XPath query engines in Chapter 7 (the Dynamic Work Queue and Producer-Consumer). These query engines dynamically interact with a shared work queue to improve load balance. We repeated our skewed synthetic and real-world tests using these new query engines and found that, although the Dynamic Work Queue engine suffers from memory allocation overhead, both new query engines are more resilient to skewed XML document trees and achieve better speedup on real-world data sets than our previous parallel query engines.

We conclude that the work presented in this thesis makes the following contributions: (1) A detailed analysis of XPath query evaluation mechanics for both sequential and parallel execution; (2) Two performance models that estimate query, data, and hardware specific execution time; (3) Five distinct XPath query engines, 4 of which achieve inter-query parallelism and are able to increase query throughput and decrease single-query latency on multi-core environments; and (4) Results of tests using our 5 XPath query engines on a suite of synthetic and real-world data sets, including 2 well-known XML benchmarks [15] and one prominent real-world data set [12].

An interesting future direction for this work would be to further evaluate of the performance of our 5 XPath query engines on a wider range of real-world data sets and using more advanced hardware (more cores). The objective would be to determine how the performance of our parallel query engines scales as we increase the number of processor cores. The compelling results obtained with the Producer-Consumer engine, in particular, provide motivation to evaluate its performance further and determine if any weaknesses can be improved. Another compelling area of future work would be the further refinement of a performance model that would seek to obtain accurate performance estimates with limited information about the query and data set. Further work could be done to optimize parallel XPath query evaluation, such as applying indexing techniques to the Producer-Consumer engine proposed in this thesis.

Bibliography

- [1] A. Abounaga, A.R. Alameldeen, and J.F. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. In *International Conference on Very Large Data Bases (VLDB)*, pages 591–600, 2001.
- [2] J. Al-Jaroodi, N. Mohamed, Hong Jiang, and D. Swanson. Modeling parallel applications performance on heterogeneous systems. In *International Parallel and Distributed Processing Symposium*, 2003.
- [3] E-M. Andriescu, A. Azzabi, and G. Gains. Parallel processing of forward xpath queries: an experiment with bsml. Technical Report TR-LACL-2010-11, Universite Paris- Est. Laboratoire dAlgorithmique, 2010.
- [4] R. Bordawekar, L. Lim, A. Kementsietsidis, and B. Kok. Statistics-based parallelization of xpath queries in shared memory systems. In *International Conference on Extending Database Technology (EDBT)*, pages 159–170, 2010.
- [5] R. Bordawekar, L. Lim, and O. Shmueli. Parallelization of path queries using multi-core processors: Challenges and experiences. In *International Conference on Extending Database Technology (EDBT)*, pages 180–191, 2009.
- [6] J. Cai, A.P. Rendell, P.E. Strazdins, and H.J. Wong. Performance models for cluster-enabled openmp implementation. In *Asia-Pacific Computer Systems Architecture Conference (APCSAC)*, pages 1–8, 2008.
- [7] G. Cong, W. Fan, A. Kementsietsidis, J. Li, and X. Liu. Partial evaluation for distributed xpath query processing and beyond. In *Transactions on Database Systems*, 2011.
- [8] D.J. DeWitt and J. Gray. Parallel database systems: The future of database processing or a passing fad? Technical Report 50119, Tandem, 1990.
- [9] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.
- [10] K. Krikellas, M. Cintra, and S.D. Viglas. Multithreaded query execution on multicore processors. Technical report, School of Informatics, University of Edinburgh, 2009.

- [11] M. Krulis and J. Yaghob. Efficient implementation of xpath processor on multi-core cpus. In *Workshop on Databases, Texts, Specifications, and Objects (DATESO)*, pages 60–71, 2010.
- [12] M. Ley. Digital bibliography and library project, 2011.
- [13] L. Lim, M. Wang, and J. S. Vitter. Cxhist : An on-line classification-based histogram for xml string selectivity estimation. In *International Conference on Very Large Data Bases (VLDB)*, pages 1187–1198, 2005.
- [14] W. Lu, K. Chiu, and Y. Pan. A parallel approach to xml parsing. In *Grid 2006: The 7th IEEE/ACM International Conference on Grid Computing*, pages 28–29, 2006.
- [15] I. Mlynkova. Xml benchmarking. In *IADIS International Conference on Applied Computing*, pages 59–66, 2008.
- [16] Y. Pan, W. Lu, Y. Zhang, and K. Chiu. A static load-balancing scheme for parallel xml parsing on multicore cpus. In *International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 351–362, 2007.
- [17] T. Rauber and G. Runger. *Parallel Programming: For Multicore and Cluster Systems*. Springer, 2010.
- [18] A. Schmidt, F. Waas, M Kersten, M.J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *International Conference on Very Large Data Bases (VLDB)*, pages 974–985, 2002.
- [19] P.R. Suri and S. Rani. Mechanisms for parallel query execution. *Journal of Theoretical and Applied Information Technology*, 2008.
- [20] Y. Zhang, Y. Pan, and K. Chiu. A parallel xpath engine based on concurrent nfa execution. In *International Conference on Parallel and Distributed Systems*, pages 314–321, 2010.
- [21] W. Zuo, Y. Chen, F. He, and K. Chen. Load balancing parallelizing xml query processing based on shared cache chip multi-processor (cmp). *Scientific Research and Essays*, 6:3914–3926, 2011.