KTH Computer Science
and Communication

Handling XML Data Using Multi-threaded Solutions

Optimization of an XML verification and transformation process

MARCUS BRING

Master's Thesis at NADA
Supervisor: Pawel Herman
Examiner: Anders Lasner
Scania Supervisors:
Jakob Palmheden
Govan Marounsi

TRITA xxx yyyy-nn

Abstract

This project was performed as part of a masters exam in computer Science and Engineering at the Royal institute of technology in Stockholm. Scania AB acted as an employer and has provided both guidance and mentors. This thesis explores the concept of performing computations on XML-data and how multi-threaded programing could improve the performance of such calculations. The thesis begins by discussing the basics of the XML-language and continues with basic concepts for multi-threaded programming. During the project two simulations were developed and benchmarked, one generic simulation to prove the validity of the method and a more realistic simulation that is supposed to indicate if the techniques scale in more real environments. The benchmark result shows good potential and acts as a base for further discussion on the limitations of the system at hand. The results show good potential and multi-threaded solutions could provide big improvements regarding efficiency.

Referat

Hantering av XML-data med hjälp av multitrådade lösningar

Detta projekt utfördes som en del av masterexamen i Datalogi vid Kungliga tekniska högskolan i Stockholm. Scania AB har varit arbetsgivare och har bidragit med såväl vägledning som handledare. I denna uppsats utforskas koncept kring beräkning rörande XML-data och hur multitrådsprogrammering kan hjälpa till att sänka exekveringstider. Rapporten börjar med att förklara grunderna i XML-språket och fortsätter med att presentera grundläggande koncept inom multitrådsprogrammering. Under projektet har två simuleringar utvecklats och prestandatestats, en generell simulering som ämnar att visa att de föreslagna metoderna fungerar och en mer realistisk simulering som ska indikera ifall metoderna även fungerar under mer verklighetstrogna förhållanden. Dessa prestandatester visar på en god potential och ligger till grund för vidare diskussion kring begränsningarna som finns i systemet.

Contents

Chapter 1

Introduction

Scania is a Swedish truck and bus manufacture company started in 1891[1]. In the beginning the company designed and manufactured railway carriages but with the new technology of petrol engines the company changed its course [2]. Since then Scania has become one of the world leading manufacturers of trucks and busses and had eleven factories in five different countries in 2010 [1]. The technology used in these vehicles has gone through tremendous change and today the vehicles are no longer just physical machines, they also consists of a complex network of computers, sensors, actuators and so on. A typical truck can contain a large amount of interconnected embedded computers. Each embedded computer has its own responsibilities and communicates with the rest of the systems in the truck.

As a result of these technological advances in the vehicles there is a need for advanced technological aids that help mechanics to diagnose, interact with and repair these complex systems. Scania Diagnos & Programmer 3 (SDP3) is a software developed and maintained by Scania to give the mechanic a tool for maintaining and updating the computer based system inside these modern trucks. In order to deliver precise and up to date information to the mechanics, Scania has developed and is maintaining a database of guides, maintenance methods and general information. In order to further expand and update this database an internal development platform named SDP3 Production Tool (generally known as PT) has been developed.

1.1 Problem description

In PT there is a process commonly referred to as the build process. This process loads a collection of XML documents and performs certain tasks with the data. The tasks can be divided into three parts: structure verification, consistency checks and transformation. In the end a new library of XML files is generated through the transformation and this library is used as input to SDP3.

Description of the steps:

- **Structure verification:** The structure of each XML document is guided by a schema document that defines rules for what nodes are allowed in the particular XML document and how the structure is supposed to be. In the verification step, the XML documents are verified in regards to the schema to make sure that all documents have the correct structure.

- **Consistency checks:** Data may refer to other data in the input XML documents. In the consistency check these references are verified so that referenced data actually exists.

- **Transformation:** The input XML documents are transformed into other XML output documents. The output documents are a collection of information coming from multiple sources, thus the transformation process can be seen as an information aggregation where information regarding different parts of the truck and its system is aggregated into a more complete representation of the knowledge base.

This process is slow and can take more than five minutes to run on a PC with an Intel Core I5, 8 GB of RAM and an SSD disk. For some users this is a procedure that has to be repeated several times a day. If the process of building in PT could be made to run faster it would decrease the time that the users are unable to perform work and improve the experience of using the software as a whole.

1.2 Aims & scope

In this thesis I have simulated the build process in PT in order to determine if the execution time could be lowered through the use of multi-threading. I will develop two simulations, one generic simulation to test if the method yields good results and one simulation that is more closely modeled upon the build process in PT. The goal for these simulations is to show what performance gain, in terms of lowered execution time, could be achieved by using multi-threading techniques. This thesis will act as motivation and a base for discussing how PT could be further developed in the future. However no development of PT will be performed during this theses work. This is due to the excessive amount of time it would take to obtain a good enough understanding of the PT source code and how the system as a whole is interconnected. All code will be developed in Java due to the fact that PT is developed in Java and thus the simulations will provide more realistic results than if a more optimized language, such as C, was used.

More specific goals:

- Develop a simulation based on a generic XML format without domain specific elements such as references and in order to evaluate the performance difference between single-threaded and multi-threaded execution in terms of execution time .

- Utilize the generic simulation in order to establish how simple optimization techniques, such as hashmaps, affect execution time.

- Develop a simulation modeled on PTs build process to evaluate the performance difference between single-threaded and multi-threaded execution in a more realistic setting then the generic simulation.
- Determine how the the number of threads used in multi-thread executions affects the execution time.

Chapter 2

Background

2.1 Fundamentals of XML

2.1.1 Extensible Markup Language

Extensible Markup Language (XML) is a general way of representing structured textual information. It consists of rules about how information should be represented so that information can be easily shared between different systems and platforms. XML was formally accepted as a World Wide Web Consortium (W3C) recommendation in the year 1998[3][4]. Since then XML has became a widely accepted standard for representing data especially on the Internet. The XML language does not limit the data by the use of certain keywords. XML only consists of rules about how the markup of the data should be structured so that the data is universally readable [3].

Listing 2.1. A simple XML document. The Countries node encloses two country sub-nodes. In this way data is represented as a hierarchical grouping.

```xml
<Countries>
   <Country Name="Sweden">
      <Population>9.5 million</Population>
      <Capital>Stockholm</Capital>
   <Country/>
   <Country Name="Germany">
      <Population>81.8 million</Population>
      <Capital>Berlin</Capital>
   <Country/>
</Countries>
```

In listing 2.1 you can see a simple XML document. Each tag is surrounded by <> and consists of a start tag and an end tag which is denoted by </>. W3C defines (a simplified version) XML documents in extended Backus-Naur form as (for a full definition see [5])

document ::= element
element ::= EmptyElemTag | Stag content Etag
EmptyElemTag ::= '<' Name (Attribute)* '/>'
content ::= CharData? (element CharData?)*
STag ::= '<' Name Attribute* '>'
Attribute ::= Name Eq AttValue
ETag ::= '</' Name'>'

2.1.2 Document Object Model

XML documents are many times stored as Document Object Model (DOM) when they are handled inside software. The Document Object Model, a W3C recommendation, is a programming API for XML documents that defines a set of interfaces for viewing and updating XML documents represented as tree structures [6]. A DOM-tree is a representation of an XML document that stores the tags in the document as nodes and builds the relationships between the nodes as references. The exact structures for building DOM-trees differs between implementations, but in general the nodes form a tree that can be followed from root to leaves. In figure 2.1 we can see representation of the tree build from the XML in listing 2.1. Note that the attribute Name in the country tag is not shown in the tree but is represented as a variable inside each country tag. Further it should be noted that the text leaves are represented as nodes with the actual content stored within.

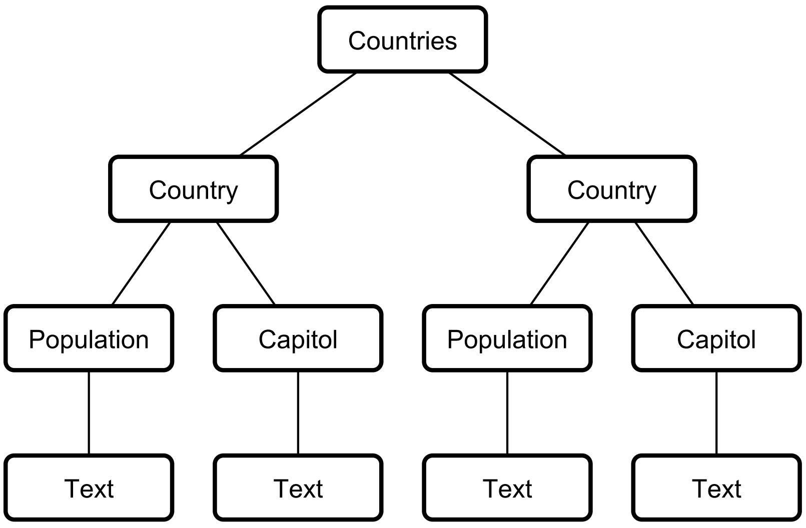

Figure 2.1. One way to represent a simplified view of the internal structure of a DOM tree

2.1.3 XPath and how to search in XML

XML Path language (XPath) is a query language for selecting nodes in XML documents. XPath is not restricted to searching in DOM-trees but as with DOM, Xpath represents the XML document as a tree structure [3]. The structure of XPath looks similar to file locations in Linux. To refer to the whole document or root node '/' is used. To find all subnodes of the root the path '/./' can be used (the dot symbol is used as a wild-card in XPath). Some examples of how to locate different kinds of nodes from the XML document in listing 2.1

XPath: /Country

Yields a list with the two country nodes and their children:

Listing 2.2. The result will be the subset of Countries, in this case two Country nodes.

```xml
<Country Name="Sweden">
   <Population>9.5 million</Population>
   <Capital>Stockholm</Capital>
<Country/>
<Country Name="Germany">
   <Population>81.8 million</Population>
   <Capital>Berlin</Capital>
<Country/>
```

XPath: /Country/Capital

Yields a list with the two Capital nodes and their children:

Listing 2.3. The result will be the capital node for each country node.

```xml
   <Capital>Stockholm</Capital>
   <Capital>Berlin</Capital>
```

XPath: /Country[@Name='Sweden']/Capital

The syntax [@tag=value] is used to locate a tag with a certain attribute that has a certain value. So the above expression yields only the Capital node of the Country with the tag Name that has the value Sweden:

Listing 2.4. The result will be the Capital node of the node where the attribute Name is Sweden.

```xml
   <Capital>Stockholm</Capital>
```

2.2 Multi-threading

2.2.1 What is multi-threading

In order to explain multi-threading we must first define some essential terms.

- **Thread:** Sequential instructions that are executed in the context of a process.
- **Single-threaded:** A process that only executes using one thread of control.
- **Multi-threaded:** A process that executes using two or more threads of control.
- **Parallel execution:** Two or more threads are executed at the same time using different CPUs or CPU cores.
- **Concurrent execution:** Two or more threads are able to execute at the same time but might take turns to execute. The order of execution is in general looked upon as indeterministic.

The alternative to multi-threaded programs are sequential programs. A sequential program consists of a process and a single-thread of control. In a sequential program each instruction is executed one after another in a deterministic way. This means that each time the program is executed, the instructions will be executed in the exact same order[7]. In multi-threaded programs, however, this is not generally the case. A multi-threaded program consists of two or more threads of control. All the threads in the process collaborate and are able to execute in a concurrent or parallel way. A thread and a process have several things in common. It could be said that a thread is a scaled down process that is able to run inside another process [7]. The ability to have more than one thread in one process helps solve the problem of performing calculations while still having a responsive interface. The reason for this is that when a process with only one thread starts a calculation it will be unable to receive input or perform any other tasks until the calculations are complete.

Multiprogramming allows several processes to execute concurrently by allowing access to the CPU and suspending processes, and thus alternating between what process are allowed to execute. In more resent years even consumer computers have multiple CPU cores. This allows threads and processes to execute in parallel and make more use of the CPU time.

All processes have their own address space, global variables, open files etc. and these are all shared between all threads within the process. Each thread however has its own program counter, registers, stack and state. This is why the threads are able to execute in a concurrent fashion. As long as each thread only manipulates its own registers and stack data the treads execute completely independent of each other [7]. But since the address space, and in particular the heap, is shared between threads there is a need to limit the threads from altering shared data from more than one thread at a time.

2.2.2 Why use multi-threading

In the last ten years it has become more and more frequent that normal consumer computers have the capabilities of multicore CPUs. Today almost all new computers have these capabilities. This means that it has become more important to be able to harness these new capabilities and develop programs that are able to perform heavy calculation in parallel using threads.

Even before multi-core CPUs became available threads and multi-thread programming were important concepts, because of context switching (see figure 2.2). In general a program is not able to make full use of the CPU during all of its execution because of the frequent need to wait for other hardware. The most frequent reason for this is that a thread is waiting for some IO response. A thread could be waiting to read data from the hard drive or be in a state where it needs user input in order to progress. A multi-threaded solution can in this case help to make sure as much as possible of the CPUs resources are utilized by letting the program perform calculations in separate threads and the threads that are in a waiting state can yield the CPU time to another thread that might make better use of it. A typical example is software with a graphical interface. The interface is handled by its own thread and thus even if some user input triggers some calculation phase the interface still remains responsive for further user instructions.

2.2.3 Multi-threading in general

Concurrency

The term concurrent execution describes the 0ability to have more than one thread able to execute at one time. Although this does not mean that these threads necessarily execute in parallel. In fig 2.2 we can see an example of how two threads execute concurrently, the execution of the threads are interleaving, they take turns to use the CPU. This is typical behavior of older CPU architectures where the CPU only has one core and thus only one thread can execute at a time. Modern computers often have multiple cores, this allows for true parallel execution as in fig 2.3.

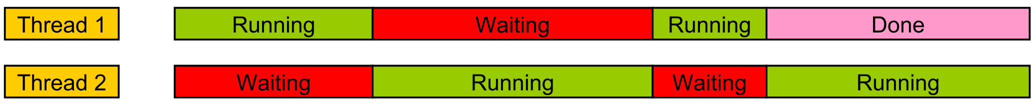

Figure 2.2. Two threads executing concurrently. The threads take turns executing until one of the threads end its execution.

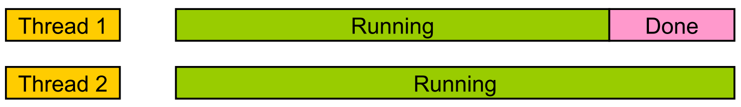

Figure 2.3. Two threads executing parallel. Both treads are able to execute at the same time due to multiple CPUs or multiple cores in a CPU.

Locks

One problem that arises when working with concurrency is race conditions. A race condition is when two or more threads compete over a single resource. The most typical example where this is a problem is when multiple threads try to read and update a shared variable. This could lead to situations where the thread that reads the variable acquires an old value or even a corrupt value. There are also situations where the result of an execution becomes indeterministic because there is no way of knowing in what order the treads execute. In listing 2.5 it could be assumed that the desired behavior is that the variable 'y' is supposed to be either 5 or 8 when the program terminates. However since thread B is allowed to alter the variable 'a' there is a possibility that 'y' might end up having the value 9 at the end of execution.

Listing 2.5. Example of a race condition

```
class W{
        int a = 3;
        class A implements Runnable{
                int y;
                public void run(){
                        if(a == 3){
                                //Thread B might change 'a' here
                                y = a + 5;
                        }
                        else{
                                y = 5;
                        }
                }
        }
        class B implements Runnable{
                public void run(){
                        a++;
                }
        }
}
```

Locks are used to prevent race conditions by limiting the access of certain variables to one thread at a time. There are several different kinds of lock but the two most common is mutual exclusive locks (mutex locks) and condition synchronization.

- **Mutex locks** are used to ensure that a section of code is not executed by more than one thread at a time. Such a section is called a critical section [8]. The mutex lock could be visualized as a locked room with only one key. When one thread wants access through the door it has to ask for the key. If no one else is inside the room the thread receives the key and can access the room. But if another thread is currently in the room the key is not available and thus the thread has to wait until the current holder of the key exits the room and returns the key.

- **Conditional Synchronization** is used to delay a thread until some boolean condition becomes true. The state of the boolean condition is updated by another thread. With the same kind of analogy as for mutex locks this could also be described as a locked room but in this case the door opens with a remote switch. When a thread wants to access the room it checks if the door is unlocked and in other case it waits until another thread opens the door with the switch.

Design patterns

A common design pattern when working with multiple threads is called Producer/-Consumer. The Producer/Consumer pattern is based on separating the responsibilities into tasks of generating subproblems (producers) and solving subproblems (consumers). Depending on the nature of the problem there might be one or more producers and one or more consumers. There are some variations to how the producers and consumers communicate with each other. One solution is to let each producer send data to a certain consumer through a pipe. This one to one relationship between producers and consumers is easy to implement since it is easy to synchronize the data flow. As long as a consumer does not read data from the pipe at the same time as the producer is writing data to the pipe there are no problems with race conditions. But since this ties a certain producer to a certain consumer it may be hard to balance the workload in cases where there are more than one producer/consumer pair and the tasks vary in difficulty. Another approach is to have a common queue that is shared between all producers and all consumers. This balances the workload better, since every consumer can perform a task that is generated from any producer. So even if a certain producer generates tasks that take particularly long time to solve, all consumers will help divide the work of solving these tasks. Although a queue helps distribute the workload it also requires a more sophisticated synchronization policy since there is now several producers that should only be able to write into the queue at once and several consumers who may only take a task from the queue at once.

2.2.4 Multi-threading in Java

Java threads

In Java, any class can be executed as a thread as long as it implements the interface Runnable [1]. A thread may be declared like this:

Listing 2.6. Simple thread that only prints a message and then terminates

```
class Thread_Example implements Runnable{
                @Override
                public void run(){
                        System.out.println("Hello World!");
                }
}
```

When starting a thread, the execution always begins at the method run(). As can be seen from the example the return value of run in void. Void is in fact the only possible return value, since all threads execute asynchronously the code that started the thread will not be able obtain the return value since it will have continued with its own execution. All that is needed to start a thread is a Thread object with the Runnable object that should be executed as constructor parameter and a call to the method start().

Listing 2.7. Create and start a thread

```
Thread_Example runnable = new Thread_Example();
Thread thread = new Thread(runnable);
thread.start();
```

Executorservice

Executorservice is a framework that was introduced in Java version 1.5. This framework makes it easier to create work queues that handle multiple threads and control their execution. When creating an Executorservice you are able to define several parameters regarding how the threads should execute and how many are allowed to execute concurrently.

Listing 2.8. Create an ExecutorService and start three threads

```
ExecutorService executor = Executors.newFixedThreadPool(2);
executor.execute(new Thread_Example());
executor.execute(new Thread_Example());
executor.execute(new Thread_Example());
executor.shutdown();
```

[1] The interface Callable can also be used to create a thread that may return data. Further information about Callable is omitted.

In listing 2.8 we create an ExecutorService with a fixed thread pool of size two. This means that the thread pool will allow two threads to execute concurrently. We then add three tasks to the thread pool, these threads will begin to execute as soon as they are added to the pool. Since only two threads may execute concurrently one of the threads will have to wait until one of the threads have finished their execution. The last step, executor.shutdown(), indicates that no new threads are allowed to enter the thread pool and that the executor should terminate once all the threads in the pool have terminated.

Synchronization

Java offers several solutions to control synchronization. The most common method is Java's intrinsic lock, defined by the keyword synchronized. This lock uses the philosophy that all objects may be used as a lock. One way of using this is to make a method synchronized:

Listing 2.9. A synchronized method

```
public synchronized void metod(){
        //method content
}
```

This will make sure that all calls to this method in an object is made in a serial fashion. This is a very simple construction for controlling synchronization but it also introduces some complexity. Since the synchronized method uses the intrinsic lock on the called object and thus locks he whole object and not just the method, problems with deadlocks might occur if more than one method is declared synchronized.

Listing 2.10. A deadlock example

```
class deadlock{
boolean isReady = false;
public synchronized void setReady(){
        this.isReady = true;
}

public synchronized void doWork(){
        while(!this.isReady){
                //wait until ready
        }
        //do work
}
```

The example in listing 2.11 is a simple error to make. If one thread calls doWork() before setReady() is called a deadlock will occur. This is a simple example but in more complex code this might become a hard to find bug.

Another lock Java provides is the CountDown-latch. This construction makes is easy so set up rendezvous points or synchronization points in the code. This is often needed where several threads perform a similar task and at some point it is important that all threads have reached some point in the code before further execution is allowed. A typical example may be a multi-threaded animation. Each thread calculates part of each frame. Before a frame can be painted on screen every thread needs to be done with the calculation for the frame. In this case a CountDown-latch is useful for signaling when all threads are done with the frame, when this happens, the frame can be printed and the threads may start work on the next frame.

Listing 2.11. An example of the use if the Countdown-latch

```
public void animate(CountDownLatch latch){
       while(continueAnimate){
              //calculate frame
              latch.countDown(); //count down to signal task completion
              latch.await(); //wait until all threads reach this point
       }
}
```

Queues

Queues are useful for communicating tasks between producers and consumers. Queues in computers work the same way as real life queues in the grocery store. An object can be placed last in the queue and the object first in the queue will be the first to exit the queue. A double-ended queue also offers the possibility to place object first in the queue and retrieve the object last in the queue.

Java offers concurrent versions of queues called ConcurrentQueue and ConcurrentDeque. These data structures are internally synchronized, this means that multiple threads can operate towards them concurrently without any need for manual synchronization. While using these structures as pipelines between producers and consumers there is a need to be able to signal when no more data will be added to the pipe. One common way of doing this is to use something called a poison object. A poison object looks similar to other objects that are added to the pipe but instead of containing real data it contains some special data that tells the consumers that all producers are done and that no more tasks will be added to the queue. The reason that this is important is to allow the consumers to complete their execution and terminate when no more jobs are available.

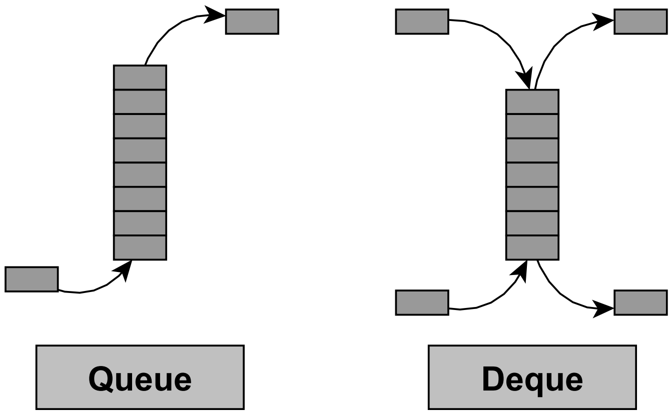

Figure 2.4. A queue and a double-ended queue

2.3 SDP3 and PT

2.3.1 Scania Diagnose & Programmer 3

Modern trucks contain advanced electrical systems and a network of electronic control units (ECU). There are several different ECUs in a truck, each has a different function. There are for example ECUs for controlling brakes, the motor, the climate control and the gearbox. The ECUs are also able to detect when some functions in the truck are compromised in some way, and to keep track of general statistics about the vehicles usage and performance. Scania Diagnose & Programmer 3 (SDP3) is a tool, developed by Scania, for programming the ECUs and communicating with the ECUs in order to diagnose and maintain the trucks. SDP3 is used by mechanics in several countries all over the world. Each time a truck from Scania arrives in a repair shop, the mechanic connects a computer with SDP3 to the truck and can through SDP3 recieve information about how the specific truck is configured, if the ECUs are reporting any fault in the system that need to be repaired or maintained, or if there are new software updated available for the ECUs.

If SDP3 reports that some function in the truck is not working properly, the mechanic can access a troubleshooting guide that helps him/her test the system in order to diagnose where the problem lies and how to repair or replace the faulty hardware.

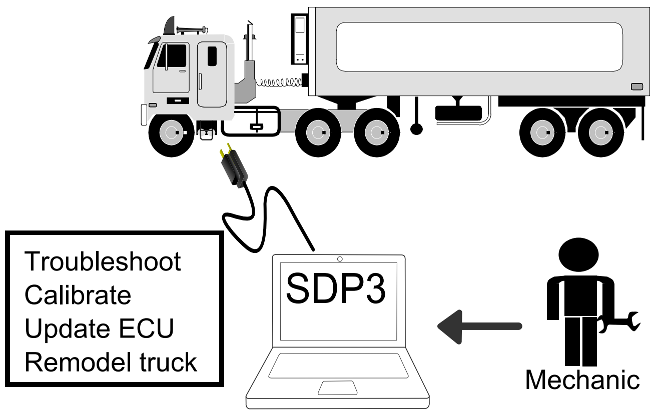

Figure 2.5. SDP3 is a program that serves as an interface to the trucks internal network.

2.3.2 Scania Diagnose & Programmer 3 Production Tool

In order to provide SDP3 with data a tool called Scania Diagnose & Programmer 3 Production Tool (SDP3 PT or PT) is developed. PT has two major functions:

1. Provide method engineers[2] and technical writers a development platform for producing textual information and scripted guide declarations that are to be used in SDP3.
2. Serve as an integration platform for data originating from PT internally and from other fractions of Scania to produce a complete database that serves as input to SDP3.

PT is built in Java as an Eclipse Rich Client Platform (RCP). This means that PT is built using the Eclipse Framework and standard components [9].

[2]Engineers who are responsible for developing and defining methods on how to repair and maintain Scania vehicles

PT as a development platform

PT is a tool that is only used internally at Scania and has two types of users, method engineers and technical writers. Both of these user groups depend on PT in order to produce information that is to be used in SDP3. They rely on PT in their everyday work to continuously integrate their work product into SDP3 and to verify that the new material is correct. In order to perform this integration they are reliant on what is called the PT-build-process.

PT as an integration platform

PT serves as an integration platform in order to compile different kinds of data coming from several different systems and departments inside Scania. This data contains different information about the trucks and how to maintain them. The PT-build-process is used in order to verify that all the data is correct in terms of structure and in terms of reference validity and also to transform the data into a database that is used to present the data to the end user in SDP3.

Chapter 3

Methods

3.1 Proposed approach

3.1.1 Preconditions

In PT there are some preconditions that I have used in order to design a solution that is effective for the specific domain. These preconditions are

1. All input data is known at the beginning of execution.
2. The transformation scheme is know at the beginning of execution.
3. Input data is never modified during execution.
4. Output data is never modified after creation.

3.1.2 Module design

My design is separated into five modules with different responsibilities. Each module is able to run in both sequential and multi-threaded versions. The five modules are

1. **Collection** This module is responsible for providing access to the loaded input data. This module is also responsible for initiating the other modules and serves as the interface for the whole process.
2. **Loader** This module is responsible for loading and parsing the input data and storing it in the collection. The loader threads parse the input XML data and store it as DOM-trees.
3. **Consistency checker** This module is responsible for finding and verifying references in the input data.
4. **Transformer** This module is responsible for transforming input data and producing output data by following the transformation scheme and reading appropriate data from the collection.

5. **Result writer** This module is responsible for writing the output data produced by the transformer to disk.

3.1.3 Communication between the modules

Preliminary approach

The communication or task transfer between modules was initially designed with a one thread per task approach. Each module had a thread-pool and for each task that should be transfered from one thread to another module, the calling thread would create a thread for the specific task and place it in the other modules thread-pool. This approach is easy to implement but it gives rise to a couple of problems. One problem is that it is hard to separate the modules in a clean way, they become more entangled and increasingly hard to develop. The other problem is that there is some performance loss when creating a big quantity of threads.

Final approach

The final approach uses a more strict Producer-Consumer pattern. Each module still has a thread-pool but only a fixed number of threads are created. The actual tasks are created as a data structure and passed to the consumer threads through a synchronized queue. This approach provides a cleaner interface between the modules and the amount of created threads are kept to minimum. This new approach however gave rise to two new challenges

1. How to signal that the producers have finished their execution and that no more tasks will be created.

2. How to determine when all producers have finished their execution.

Problem one was solved by introducing a "poison object" into the queue. When a consumer comes across this object it escapes the consumer loop, however before it terminates the poison object is reinserted into the queue. This method of reinserting the object means that all consumers will be able to terminate regardless of how many producers or consumers were in use.

The second problem is of concern when there are multiple producers. The individual producers have no knowledge of the other producers and thus even if one producer terminates it does not know if the other producers are still executing. This raises the question of who should be inserting the "poison object" into the queue. To solve this I created a small semaphore- like class. When a producer starts it increments the semaphore and before it terminates it decrements the semaphore. If the semaphore is zero after the decrementation has been performed the specific producer is the last one to terminate and thus it takes the responsibility of adding the "poison object".

3.2 Proposed benchmarks

The benchmark is separated into three phases:

1. A profiling tool will be used to identify what components take the most time.
2. Generic format for input data to show what kind of performance gain can be achieved.
3. If the generic version shows good results more realistic data will be tested.

The Netbeans[10] built in profiling tool will be used to evaluate the suggested solution and based on the results some optimizations will be implemented.

To test the difference between multi-threaded and single-threaded implementation a generic format for the input data will be used. The reason for this is that it is hard to write code for a more realistic data format that supports both multi-threaded and single-threaded execution with minimal code difference. This approach provides more accurate for the difference between multi-threaded and single-threaded evaluation. A more realistic format will be evaluated in multi-threading only to show that the method scales for the more realistic data as well.

3.3 Implementation details

3.3.1 Collection

The collection stores all loaded data in a hashmap. The data is represented as a data structure called CollectionData. CollectionData is a class that is designed to store data and give access to the data in a thread safe manner. One reason for providing the CollectionData instead of storing the actual DOM-trees in the collection is to be able to account for the fact that the different modules operate asynchronously. The implication of this asynchronous behavior is that if some module asked for some data from the collection and the data is unavailable the module would be unable to determine if the data is non existent or just not yet loaded. To solve this the collection makes use of the precondition "All input data is known at the beginning of execution". Every file that will be loaded is created as a CollectionData object that holds nothing but the path to the file that represents the specific data object and stores these objects in the collection hashmap. This means that if a module tries to access certain data and the data will be loaded at some point there will exist a CollectionData object. If the actual data is not yet loaded into the CollectionData object the thread asking to access the data will be put into a blocking-wait mode until the data is loaded.

3.3.2 Loader

There are multiple loader threads that receive tasks from a concurrent linkedlist in the form of CollectionData obejct via the collection. A loader picks a task from

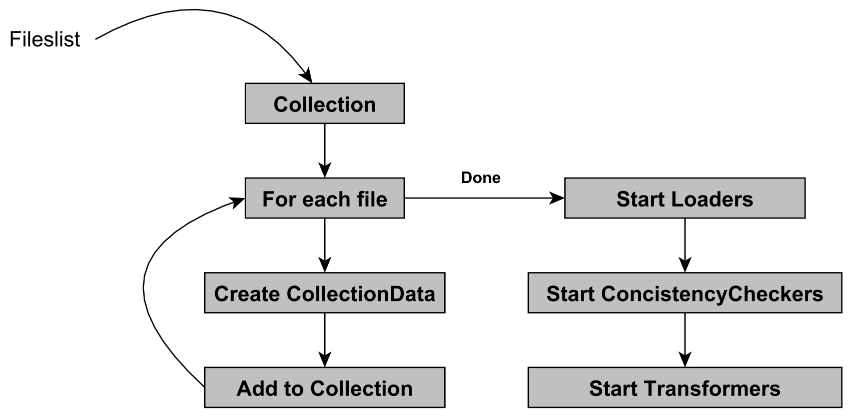

Figure 3.1. Behavior of the collection

the queue and loads the file, specified by the CollectionData object, into a DOM-tree. The resulting DOM-tree is stored in the task object and access to the data is activated. Each time a DOM-tree has been loaded, the loader creates a ConcistencyTask that holds the CollectionData object. The purpose of the ConcistencyTask is to be able to signal that the loaders are done and that no more tasks will be added to the queue. This is done by creating a ConcistencyTask and setting a EndOfJob variable to true.

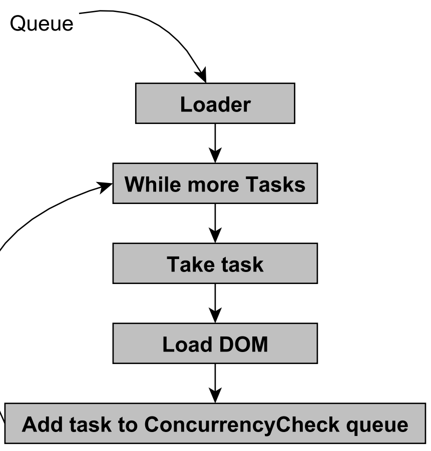

Figure 3.2. Behavior of the loaders

3.3.3 Consistency checker

There are multiple ConcistencyChecker threads that receive tasks from a concurrent linkedlist in the form of ConcistencyTask-objects via the loaders. The concistency-Checker searches the current DOM-tree in search of tags that have the ref-attribute. Each tag that has a ref-attribute is referring to some data. Each such reference is evaluated and verified so that the referenced data actually exists. If the referenced data is not yet loaded, the thread will perform a blocking wait until the needed data has been loaded into the collection.

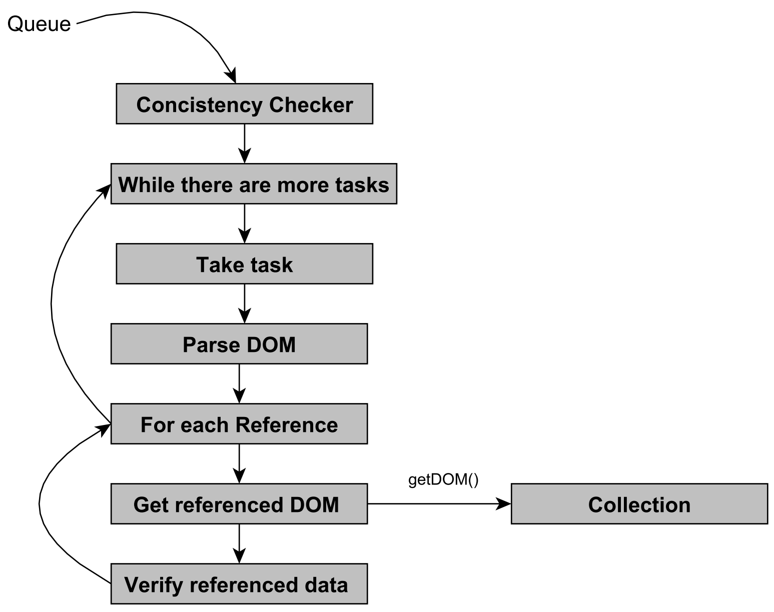

Figure 3.3. Behavior of the concistency checkers

3.3.4 Transformer

There are multiple transformer threads that receive tasks from a concurrent linkedlist in the form of TransformRule-objects via the Collection (the rules are passed to the Collection at the beginning of execution). Each transformer picks a rule from the queue and applies it into a output file. Each rule defines how one output file should be constructed. Each rule refers to data from multiple input files. This means that an output file consists of data from several input files. The data needed to construct the output files are loaded from the collection via the CollectionData objects. If a transformer tries to access data that is not yet loaded into the CollectionData object it performs a blocking wait until the data is available. The transformer builds a DOM-tree according to the specified rule and when the DOM-tree is complete it passes a ResultData object to the result writers through a concurrent linkedlist.

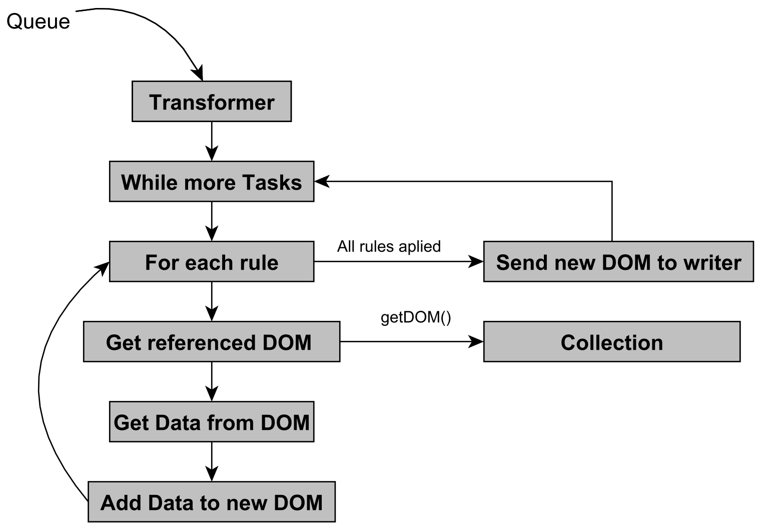

Figure 3.4. Behavior of the concistency checkers

3.3.5 Result writer

There are multiple result writer threads that receive tasks from a concurrent linkedlist in the form of ResultData-objects via the transformers. The result writer retrieves the DOM-tree from the ResultData-object, transforms it into an XML document and writes it to hard drive.

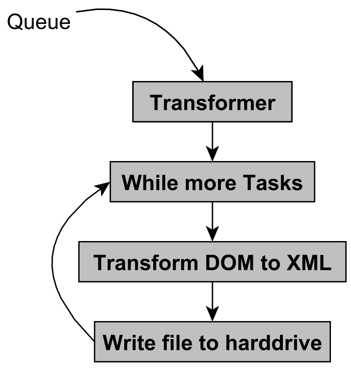

Figure 3.5. Behavior of the result writer

3.3.6 Realistic simulation

There is only one major difference between the generic simulation and the more realistic simulation except from the format of the input data (described in section 3.4).

This difference is due to the fact that the realistic simulation lacks the information of data location. This means that data is only referred to based on identification and there is no information about in what file this data is located. Further this implies that the consistency checker need to be able to evaluate the existence of some data even if it has no information about where this data might be located. To solve this there are two major difference in implementation :

1. The collection holds a hashset that stores what data exists. This hashset only contains the identification of data and not the data in itself. This is possible since the consistency checker only verifies that the data exists, and does not perform any additional operation on the data. The hashset is populated by the loaders as part of the loading process.

2. The consistency checkers have to be aware of the state of the loaders. If a consistency checker fails to find some data in the hashset it will check if the loaders have finished. If the loaders are not yet finished, the reference will have to be verified again after the loaders are done. No data can be declared as unavailable until all input files have been loaded into the collection.

3.4 Benchmark suite

3.4.1 Generic simulation

Input data format

The XML format for the generic simulation is designed with the purpose of being easy to parse and perform operations on. The format can informally be defined as:

Listing 3.1. Example of a simple XML document

```xml
<root>
<name-1 id="1" ref="file-id id">data-1</name-1>
<name-2>data-2</name-2>
<name-3 id="3" ref="file-id id">data-3</name-3>
...
<name-n id="n" ref="file-id id">data-n</name-n>
</root>
```

Description of the XML format details:

- The name and the random-data have the exact same format. They always start with the letter x followed by 32 random numbers or lowercase letters.
- The id attribute is specific for each file and starts at 1 and count up to n.
- The ref attributes are semi-random references to other XML-tags in the same or other documents.

- Half of the data tags contain an id-attribute and a ref-attribute. The rest of the tags are only data tags (see listing 3.1 line 1 and 2).

Methodology

Three benchmark tests where performed. The first one was a benchmark on the generic simulation with comparisons between single- and multi-threaded execution. The test where also designed to show if hashing part of the input while loaded the data can improve the performance of the consistency checker. This means that there will be four test configurations to be considered. Each test will be evaluated 20 times and the result will be evaluated with Anova (see 3.4.1).

ANOVA - Analysis of variance

Anova is used in order to perform a statistical test comparing multiple factors. In this case there are two factors, the effects of optimization and of multi-threading [11]. The result of the Anova test will provide information about the statistical probability that one or both factors are significant. To perform the Anova test a software called ezAnova is used for this project[12].

3.4.2 Realistic simulation

Input data format

The XML-format for the more realistic simulation is more complex, below is an example of how the format is structured.

Listing 3.2. Example of a simple XML document

```xml
<SimulatedGuide>
 <Step>
  <Name>x8f887f50709d49c6a521b617aabab6eb</Name>
  <NextStep>
   <Step ref="Step">x1a38f9aeeb0a4386b6248a5489a3b674</Step>
   <Condition>
    <eq>
     <variable>
      <name>Some Stored Variable</name>
      <storetype>StoredVariable</storetype>
      <datatype>string</datatype>
     </variable>
     <constant>
      <value>Some variable value</value>
      <datatype>string</datatype>
     </constant>
    </eq>
    </Condition>
   </NextStep>
  </Step>
...
<\SimulatedGuide>
```

The main difference between the generic format and this more realistic format in terms of complexity is that the realistic format has more depth in the XML-tree. There is also a significant difference in the distribution between information-data and meta-data. This means that the realistic format will yield a lower computation complexity in relation to file size.

Methodology

The realistic simulation will be developed both as a multi-threaded version and as a single-threaded version and these two version will be benchmarked using the same data input.

3.4.3 Profiling

Software profiling is useful for finding the bottlenecks of an application. A common opinion about execution time distribution is that 90% of the time spent executing is spent on executing 10% of the code [13]. I will use the Netbeans built in profiling tool to evaluate what parts of the code that seem to take significantly longer to execute than other parts and discuss if this is reasonable or if the proposed solution could be improved upon.

Chapter 4

Results and Discussion

4.1 Benchmark tests

4.1.1 Generic simulation benchmark

In order to verify how multi-threading and optimizations affect the execution time I evaluated the generic simulation through a benchmark task. The input consisted of 10000 files. Each had 100 XML-nodes and 50 references. Each input file had an approximate size of 11 kilobytes. The benchmark results for the generic simulation show a significant improvement both in regard to optimizations and multi-threading. In figure 4.1[1] we can clearly see how the execution time improves with the help of multi-threading and optimization. Both multi-threading and optimization offer a significant improvement individually and the best performance is achieved when the two are combined.

A statistical test with Anova shows that the results are statistically significant

[1]Data table can be found in appendix A table A.1

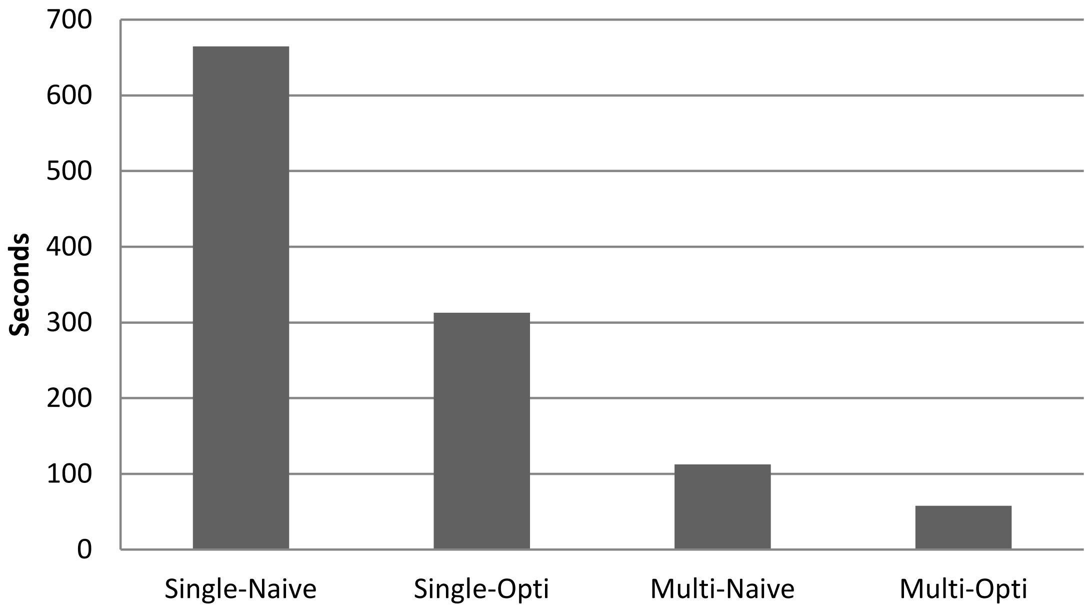

Figure 4.1. Result from benchmark on generic simulation. No error bars are shown since standard deviation is small (lower than 2).

Table 4.1. Anova result for generic simulation shows that each factor is statistically significant both individually and when combined

ANOVA: Design 2 Between Subject Factors
A F(1,76) = 3671196 p<0,000001 SS=3257263,51 MSe=0,89
B F(1,76) = 934216 p<0,000001 SS=828882,08 MSe=0,89
A*B F(1,76) = 497765 p<0,000001 SS=441640,98 MSe=0,89

PAIRWISE COMPARISONS [Q=TukeyHSD: *=p<0.05 **=p<0.01]
-[Multi_Opti]vs[Multi_Naive] t(38)=173,37 p< 0,0001 Q=261**
-[Multi_Opti]vs[Single_Opti] t(38)=819,00 p< 0,0001 Q=1210**
-[Multi_Opti]vs[Single_Naive] t(38)=1843,22 p< 0,0001 Q=2882**
-[Multi_Naive]vs[Single_Opti] t(38)=761,60 p< 0,0001 Q=949**
-[Multi_Naive]vs[Single_Naive] t(38)=1945,70 p< 0,0001 Q=2621**
-[Single_Opti]vs[Single_Naive] t(38)=1270,07 p< 0,0001 Q=1672**

DESCRIPTIVE DETAILS

A	Multi	Multi	Single	Single
B	Opti	Naive	Opti	Naive
	A	B	C	D
Mean	57,75	112,73	312,71	664,89
StDev	1,13	0,85	0,81	0,94
SE	0,25	0,19	0,18	0,21
Var	1,29	0,73	0,65	0,88
CI95%	0,42	0,42	0,42	0,42
N	20	20	20	20
Skew	-1,18	0,49	-0,072	0,61
zSkew	-2,16	0,89	-0,132	1,11

4.1.2 Realistic simulation benchmark

To be able to show that the methods proposed in the generic simulation scaled when implemented in a more domain-realistic setting I benchmarked the realistic simulation with a similar amount of data input. The input consisted of 10000 files each with a size of approximately 11 kilobytes. But the difference in XML format meant that there where fewer top-level nodes and fewer references. To compare the results from the generic simulation and the realistic simulation (figure 4.2[2]) it is interesting to compare the "Single-Opti" bar in figure 4.1 to the "Single" bar in figure 4.2 and corresponding for "Multi-Opti and "Multi".

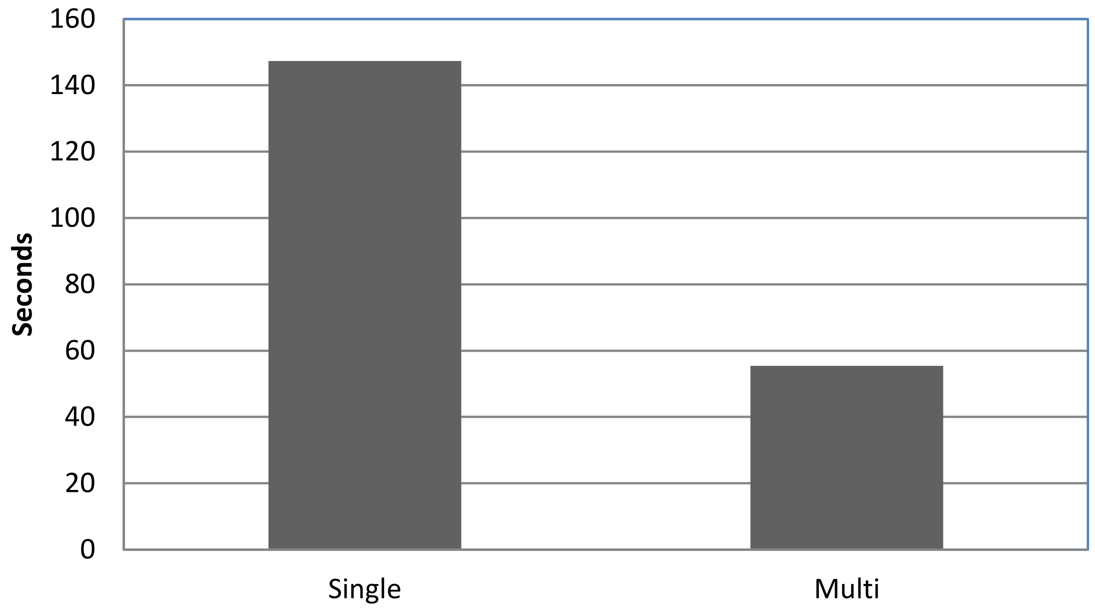

Figure 4.2. Result from benchmark on realistic simulation shows that the execution time for multi-threaded execution is almost a third of the time of that of the single-threaded execution. No error bars are shown since standard deviation is small (lower than 2).

We can see that:

1. "Single" in the realistic simulation is about half the size of "Single-Opti" in the generic simulation.
2. "Multi" in the realistic simulation is about the same as "Multi-Opti" in the generic simulation.

I will discuss more in depth why this is the case in section 4.3

[2]Data table can be found in appendix A table A.2

4.1.3 Amount of threads per thread-pool

In order to find out how many threads were optimal to use for the thread-pools, I performed benchmark tests using different amount of threads. The input data was similar to the one used in the previous benchmarks.

Generic simulation without optimizations

In figure 4.3[3] we can see that the time improves from one down to three threads, but for three up to six threads the difference in performance is negligible.

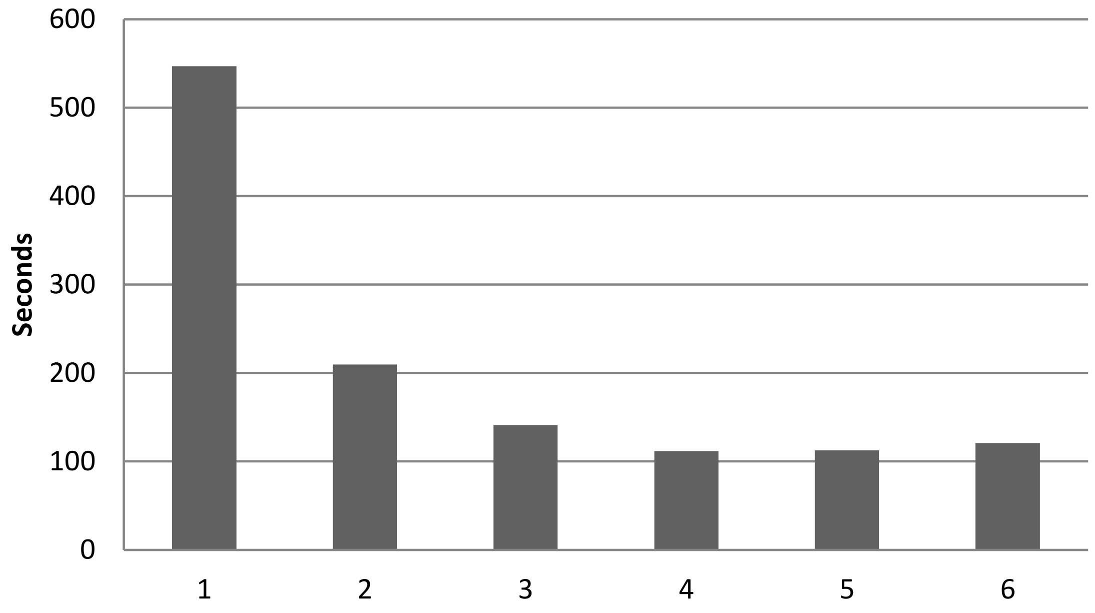

Figure 4.3. Different sizes of thread-pools for the generic simulation without optimizations. No error bars are shown since standard deviation is small (lower than 6). The data shows that execution time is lowest when 4 threads per pool is used.

Generic simulation with optimizations

The results for the generic optimized simulation (in figure 4.4[4]) shows very similar results compared to the naive simulation.

[3]Data table can be found in appendix A table A.3

[4]Data table can be found in appendix A table A.4

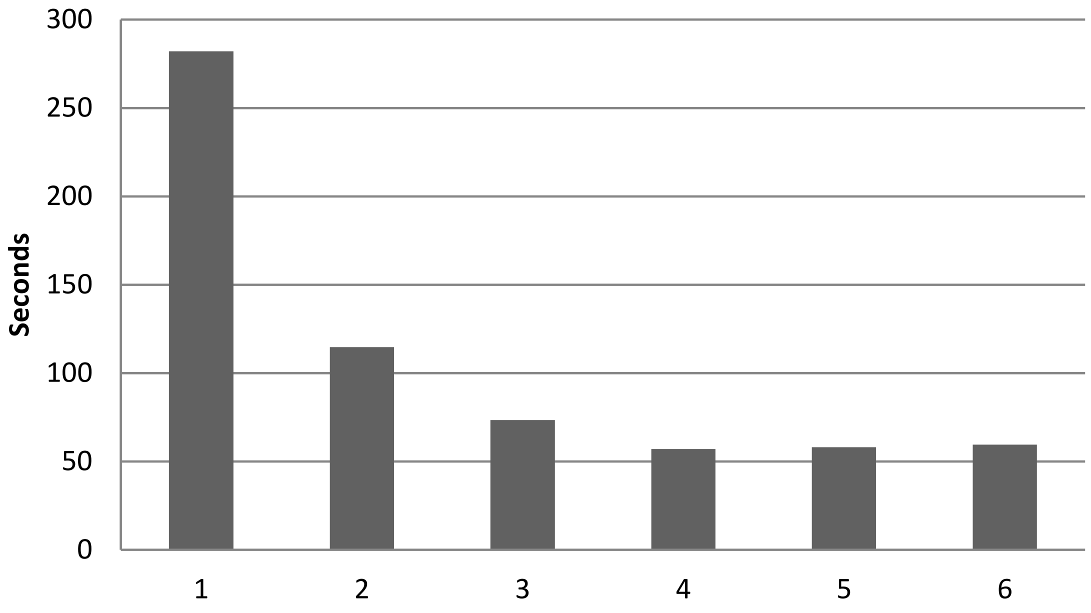

Figure 4.4. Different sizes of thread-pools for the generic optimized simulation. No error bars are shown since standard deviation is small (lower than 4). The data shows that execution time is lowest when 4 threads per pool is used.

Realistic simulation

The realistic simulation (in figure 4.5[5]) shows the same characteristics as the generic simulations when the amount of threads per thread-pool is scaled up.

4.2 Profiler Results

While comparing the results from execution using different amount of threads with a profiling tool the thread-execution-visualization indicates that the amount of time spent on certain task seems to follow a specific pattern. When comparing the time spent in the loader thread while executing with different amount of threads the time spent in each of these loader-threads seemed to be the same when the amount of threads per thread-pool reached four or more. This means that although more threads where performing the same task, there was none or little improvement in time in solving the specific task.

[5]Data table can be found in appendix A table A.5

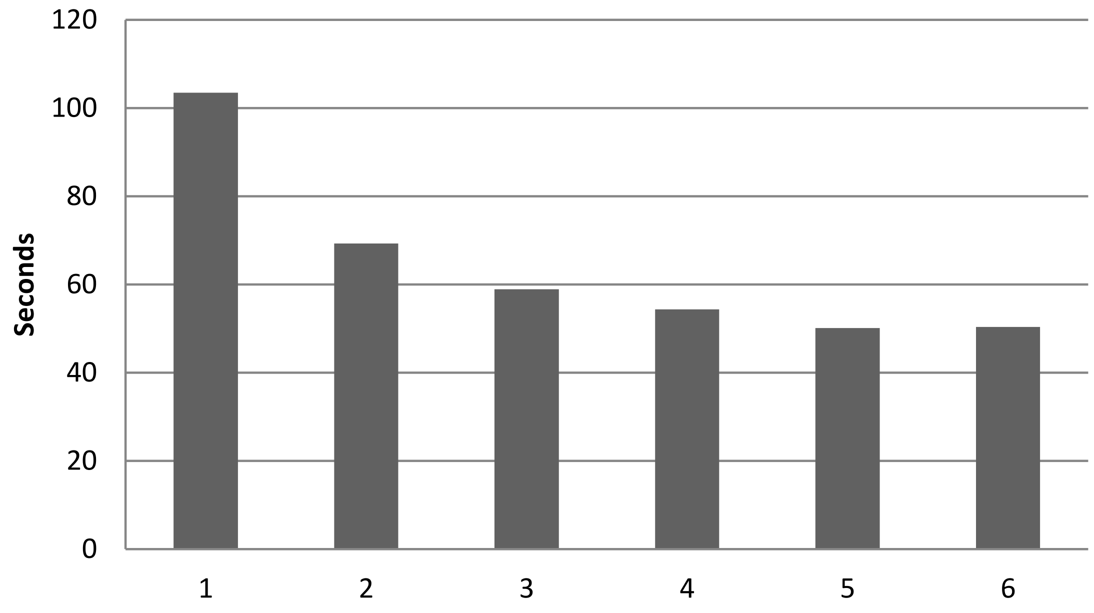

Figure 4.5. Execution time depending on the size of the thread-pools in realistic simulation.

4.3 Discussion

Comparison between Generic and realistic simulation results

When comparing the results from the generic simulation (figure 4.1) and the results from the realistic simulation (figure 4.2) it is interesting to observe that the corresponding execution time for single-threaded execution is about half in the realistic results. This is probably due to the fact that the nature of the realistic XML-format is such that it yields a lower computational complexity than the XML-format in the generic simulation. On the other hand the results for the multi-threaded tests are comparable. My hypothesis of why this happens is that the computational complexity in the simulations becomes negligible in relation to how much time it takes for input and output for the given amount of data. To clarify: it might be the case that all the operations that are dependent on the CPU are able to execute without being restricted by the CPU performance, what is actually limiting the execution is the fact that data needs to be loaded from, and written to, the hard-drive. If this was true it would mean that these results would be close to a lower limit of the execution speed (at least in term of processing speed). To achieve an overall lower processing time the input and output operations would have to be improved.

Amount of threads per thread-pool

The benchmarks for how many threads that should be used (section 4.1.3) show that it does not seem to matter how many are used, as long as there are not too few. The fact that all of the charts show about the same execution time for all simulations with three or more threads per thread-pool could further point towards my theory presented in the previous section. The theory that the execution time is limited by the input and output operations could explain why the execution time does not improve as more threads are added. The speed at which the program is able to read and write to the hard-drive does not increase as more threads are used. If the theory that all CPU dependent computations get free access to the CPU since the processing is dependent on waiting for I/O it would explain why the time does not improve with more threads.

Chapter 5

Conclusion

5.1 Summary

The generic simulation showed that both multi-threading and optimization techniques have a great impact on the performance of the type of XML-processing presented in this thesis. The results from the more realistic simulation shows that the performance gains scale well in a more realistic setting. In the benchmarks performed in order to show how the amount of threads affected the performance, the results points towards a thread count close to the amount of cores in the CPU. The benchmarks performed in this thesis is however not sufficient to indicate if the amount of CPU cores is the main deciding factor when choosing how many threads should be used, there could be other factors such as I/O speed. In order to answer this question more benchmarks on other computer hardware would have to be performed.

5.2 Implications

My work on this matter has given Scania and the team working with PT a good indication that the effort to further develop the buildprocess in PT and implement some of the key ideas I have presented would yield a good result. They have made plans to actually use my ideas as a base for further discussions regarding future development.

5.3 Future work

In the future it would be interesting to look more deeply into how the input and output operations affect the execution time. To do this it would be essential to find an alternative to the DOM-structure when storing the XML documents in memory. The reason for this is that DOM is using the DeferredDocument interface and thus it is impossible to completely isolate the input and output operations from the actual data processing. I have not been able to find a good alternative to DOM but it might be possible to develop a domain specific data structure that serves the purpose better. Although this would be an interesting path to pursue I have no data to support whether or not this would yield a better result.

Appendix A

Single-Naive	Single-Opti	Multi-Naive	Multi-Opti
664,643	313,927	111,594	54,734
664,383	311,721	112,568	55,782
663,464	313,157	113,255	58,224
664,497	313,516	111,39	58,379
665,196	312,823	113,125	58,596
663,962	313,747	114,627	56,182
663,654	312,129	113,526	58,301
665,428	313,69	112,737	58,419
664,74	313,114	112,252	59,144
665,208	311,539	112,833	59,503
664,795	313,69	112,164	57,805
665,864	311,822	112,199	57,211
663,966	312,954	112,466	57,112
666,08	312,379	112,376	58,033
666,173	311,37	114,239	57,722
664,57	312,479	113,206	58,102
665,141	311,949	113,388	58,467
665,117	312,512	111,857	58,158
663,819	312,283	113,125	57,231
667,164	313,492	111,669	57,932

Table A.1. Simulation results from generic simulation benchmark test. Values are in seconds.

multi	single
56,206	146,975
58,342	146,03
55,079	146,861
55,683	150,869
55,517	149,473
55,174	148,502
55,336	148,342
55,648	147,981
55,304	148,161
58,312	146,937
56,84	146,641
54,463	147,808
54,114	146,271
54,698	146,413
55,454	146,875
52,907	146,163
54,56	147,374
55,639	146,718
53,551	146,346
55,433	146,725

Table A.2. Benchmark results for realistic simulation. Values are in seconds.

Threads	1	2	3	4	5	6
	545,408	211,975	141,012	110,009	112,524	123,439
	543,495	208,876	139,777	112,79	111,419	120,269
	557,147	208,409	143,599	111,396	112,938	116,792
	541,428	210,979	143,055	111,321	112,543	120,039
	543,916	211,364	142,298	113,604	113,114	120,302
	544,348	208,204	141,157	112,711	113,664	119,259
	542,428	209,699	142,703	111,324	111,049	120,913
	543,684	211,612	140,795	113,046	114,019	122,334
	557,249	207,692	140,79	113,343	112,796	123,766
	562,025	210,259	142,948	110,802	112,146	125,426
	546,626	209,596	140,908	113,78	112,849	124,943
	543,47	208,653	137,557	110,588	112,232	124,492
	544,765	211,811	142,984	111,821	113,699	122,862
	545,26	209,993	139,728	111,144	112,83	121,759
	546,164	209,473	143,926	110,938	112,114	118,587
	542,891	210,282	141,045	116,034	113,271	117,941
	544,206	208,934	139,857	111,309	111,942	119,87
	545,923	207,965	142,603	110,784	113,618	118,35
	556,036	209,398	138,582	111,993	112,639	119,866
	542,853	210,019	143,085	110,419	112,919	117,801

Table A.3. Different sizes of threads-pools in generic simulation without optimizations

Threads	1	2	3	4	5	6
	288,568	114,478	72,144	58,623	57,333	57,279
	286,698	114,932	74,478	56,409	59,295	58,163
	279,669	116,105	73,289	56,552	56,997	57,391
	286,244	114,763	73,975	56,942	57,75	57,679
	275,769	114,563	74,254	57,273	58,729	58,59
	285	116,061	74,154	57,111	58,856	57,475
	278,746	114,368	73,62	57,143	58,56	58,597
	278,876	114,698	73,512	55,726	58,63	58,813
	281,569	116,481	74,063	57,118	58,375	57,245
	279,124	115,804	72,859	56,414	56,956	58,537
	284,447	114,652	72,461	57,109	57,501	58,426
	282,603	111,927	74,317	57,671	58,127	59,068
	286,201	116,082	73,303	56,772	59,427	58,389
	275,069	115,194	73,607	57,01	59,642	58,196
	280,686	114,676	72,89	56,133	56,322	58,172
	281,155	114,512	73,346	56,708	58,087	64,302
	279,755	112,965	73,432	57,221	57,728	66,718
	284,317	115,871	73,454	57,202	57,403	62,91
	284,15	112,393	72,421	57,248	57,655	63,436
	283,842	113,466	72,851	58,573	57,237	62,154

Table A.4. Different sizes of threads-pools in generic simulation with optimizations

Threads	1	2	3	4	5	6
	107,877	62,616	59,454	54,614	46,063	47,393
	104,31	69,145	59,28	55,899	47,319	53,125
	108,39	65,305	60,007	55,668	45,328	51,897
	108,596	67,425	55,995	49,632	51,294	47,793
	109,234	68,777	58,094	50,531	46,475	52,894
	107,382	69,883	59,551	60,246	47,081	48,179
	101,537	70,414	59,245	53,806	46,065	47,471
	102,273	68,121	57,456	53,493	45,868	53,697
	101,032	69,32	58,784	55,282	52,8	53,496
	105,598	69,369	59,415	55,037	50,218	47,01
	100,83	71,166	59,652	55,461	48,456	57,979
	104,671	71,914	58,853	57,503	52,186	48,081
	99,706	65,683	59,735	54,67	52,03	47,397
	100,822	71,188	61,103	54,061	52,578	52,261
	100,625	72,525	61,112	54,703	52,617	47,743
	103,971	71,028	60,836	56,238	58,539	47,336
	100,541	70,301	59,398	49,723	48,195	52,438
	100,155	71,121	54,927	54,837	54,161	52,944
	97,739	67,826	58,481	53,4	52,196	52,85
	103,577	72,202	57,475	52,039	53,151	45,878

Table A.5. Different sizes of threads-pools in realistic simulation

Bibliography

[1] Scania. The history of Scania. http://www.scania.com/scania-group/history-of-scania/. [Online; accessed 8-07-2013].

[2] Scania. 1900 - Pioneering engine-propelled carriages. http://www.scania.com/scania-group/history-of-scania/1900/. [Online; accessed 8-07-2013].

[3] Deepak Vohra. *Pro XML Development with Java Technology*. Apress, 2006.

[4] W3C. Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/1998/REC-xml-19980210. [Online; accessed 21-10-2013].

[5] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition). http://www.w3.org/TR/2008/REC-xml-20081126/. [Online; accessed 10-07-2013].

[6] Doug Tidwell. *XSLT, 2nd Edition*. O'Reilly Media, 2008.

[7] Andrew S. Tanenbaum. *Modern Operating Systems (3rd Edition)*. Prentice Hall, 2007.

[8] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 1999.

[9] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse Rich Client Platform (2nd Edition)*. Addison-Wesley Professional, 2010.

[10] Netbeans.org. https://netbeans.org/. [Online; accessed 13-04-2014].

[11] Joshua Akey. Lecture 7: Hypothesis Testing and ANOVA. http://www.gs.washington.edu/academics/courses/akey/56008/lecture/lecture7.pdf. [Online; accessed 14-04-2014].

[12] ezANOVA. http://www.cabiatl.com/mricro/ezanova/. [Online; accessed 14-04-2014].

[13] IBM Satish Chandra Gupta. Need for speed – Eliminating performance bottlenecks. http://www.ibm.com/developerworks/rational/library/05/1004_gupta/. [Online; accessed 08-11-2013].