Methodology for Introducing Concurrency into Sequential Programs

Xinghao Xu

Thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

In partial fulfillment of the requirements

For the degree of

Master of Computer Science

School of Electrical Engineering and Computer Science

Faculty of Engineering

University of Ottawa

Abstract

Efficiency of software application is one of the important metrics that are used to measure the quality of software applications. Nowadays, more and more professionals are focusing on programming technology because suitable programming may make the products more efficient. The emergence of multiprocessor systems and multi-core CPUs makes concurrent programs much more popular than sequential programs. However, a great number of large complex software applications that have already been released and are currently being used by many clients are programmed in sequential fashion. Compared to developing the program from scratch again, code refactoring with the concept of concurrent programming would be a better choice. It saves effort, time, manpower and money.

This thesis studies the problems of introducing concurrency into large and complex software applications and proposes a methodology for transforming sequential programs into concurrent programs. We successfully speeded up a prototype of IBM Security AppScan Source for Analysis by introducing concurrency into the program. The performance of the application was improved, thus demonstrating the usefulness of the proposed methodology.

Acknowledgements

Foremost, I would like to give my sincere thanks to my supervisor, Professor Gregor v. Bochmann and co-supervisor Professor Iosif-Viorel Onut for their encouragement, extraordinary patience and technical support throughout my master degree study. Their guidance and comments have played an important role in my research project. This thesis would not be complete without their knowledge and guidance.

I would also like to thank the other members of the Software Security Research Group at the University of Ottawa, including Professor Guy-Vincent Jourdan and my colleague, Di Zou. Their suggestions are very helpful and facilitated my research work. In addition, I am grateful to IBM Canada for all their help and support. I am also grateful to the AppScan Source development team at the IBM Toronto and Boston Labs.

Lastly and most importantly, I would like to express my deepest gratitude to my parents, Chengda Xu and Dongmei Sun for their unconditional love, continuous encouragement and endless support of my decisions. I am very grateful to my girlfriend, Nanpu Qin, who always stood by me. I am also grateful to my uncle Jiwen Sun and my aunt Xiangjun Zhao for their support during my study for the master degree. I cannot imagine finishing this thesis without the support of my family.

Table of Contents

Table of Figures

List of Tables

1 Introduction

A computer program refers to an organized list of instructions which are used to solve particular problems or accomplish specific tasks. Computer programs form the interface between humans and computers. They help computers to comprehend and execute ideas from humans. Programs can be used to solve huge calculations. They can also be applied to control hardware. Nowadays, more and more professionals are focusing on programming technology because it can make the product more efficient. In a sense, the rational use of programming technology will improve the performance of software applications that are operated in real life.

1.1 Sequential Programs and Concurrent Programs

Before introducing sequential programs and concurrent programs, we first present two basic concepts, process and thread. Processes and threads are the basic units of program execution in the operating system. Both processes and threads are made up of sequences of program instructions. As Abraham introduced in [1], "Although traditionally a process contained only a single thread of control as it runs, most modern operating systems now support processes that have multiple threads." Threads are also called light-weighted processes. In [2], a thread is defined as "the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler." The essential difference between processes and threads is that multiple threads in the same process can share the address space and other memory resources, such as the code

instructions and the contexts, but different processes don’t share these resources.

The sequential program is the most common form among all kinds of programs [6]. The sequential program is a program that is executed by only one thread from beginning to the end. By running a sequential program, the same input data will generate the same output result. It is straightforward for programmers to track down the traces of a sequential program step by step because the execution order is deterministic.

The concurrent program is another form of computer program that has been increasingly researched and used in recent years. The emergence of multiprocessor systems and multi-core chips makes concurrent programs much more popular. Concurrent programs can be implemented at the level of processes and threads. In this thesis, we assume that the concurrent program is a multi-threaded program. All the threads which are in the same process share memory resources belonging to this process. Concurrent programs usually generate better performance than sequential programs since the multiple threads can work concurrently if they use separate cores. However, concurrent programs are much more complicated since they are non-deterministic in nature. The execution orders for multiple threads may be influenced by operating system and hardware, which makes reproducing the same scenario much harder during the debugging and testing process. Programmers cannot determine the execution order of multiple threads before running the concurrent program.

Compared to concurrent programs, sequential programs are much easier to design and implement. However, in most cases, concurrent programs have better performance than

sequential programs. The comparison between sequential programs and concurrent programs are mainly reflected in the following aspects.

1. Processing Speed - With the development of computer hardware technology, more and more computers have more than one CPU, or, more commonly, one CPU with multiple cores. On these kinds of platforms, running concurrent programs can achieve a much faster processing speed than sequential programs for doing the same function.

2. Response Time - Another advantage the concurrent programs have over sequential programs is a shorter response time. If a server application is programmed in sequential form, the request can be received only when the server is listening. However, if this service is dealt with by two threads, then listening and processing can be performed by two threads at the same time. In this way, more clients can be responded to in a short time. The service has become more responsive.

3. Context Switching - The advantages of concurrent programs are not always depending on the operating platform. Even when computers just had one CPU in which there is only one core, concurrent programs were still used. Switching between multiple threads is faster and costs less than switching between processes. Process switching involves all the process information and resources for the new process. It includes memory addresses, mappings, page tables [3] and some other resources. However, threads within the same process share resources and address space, there is no operation involving resource information preservation and address changes when switching to a different thread. Therefore, the operating system overhead is reduced when context

switching happens between multiple threads.

1.2 Research Problem and Goal

Since the concurrent programs hold advantages over the sequential programs with respect to CPU usage, time cost and so on, more software programmers are inclined to achieve their goals by using multiple threads. However, a large number of software applications that have already been released and are currently being used by many clients are programmed in sequential order. Compared to developing the program from scratch again, code refactoring with the concept of concurrent programming would be a better choice. It saves effort, time, manpower and money. “Dealing with concurrency is easier if concurrency is designed into the system from the beginning, rather than being retrofitted later on [4].” Therefore, how to introduce concurrency into sequential programs is a problem that deserves to be considered and explored.

We speeded up a prototype of IBM Security AppScan Source for Analysis [5] (simply called “prototype” in the following sections) by introducing multiple threads into the original program. This application is very large. It implements security analysis for source files in various programming languages such as Java, JavaScript, JSP, C, C++, C# and so on. By transforming a part of the sequential program of the prototype into a concurrent program, we successfully decreased the running time of the application while maintaining correct output results. Much valuable experiences are gained from this project. Based on these experiences and a systematic review of the pertinent literature, in this thesis we try to propose a complete and systematic methodology which can guide

programmers to introduce concurrency into large and complicated sequential programs. The proposed methodology addresses the different activities of the software development process and also integrates some useful techniques for concurrent programming. We expect that the proposed methodology gives guidelines for the big picture involving all steps of the process and will result in a process that minimizes the time required to introduce concurrency into the application.

Three assumptions are made about the target application as follows:

1. The program of the application is large and complex.

2. The software documentation of the application is available to use.

3. The source code of the application is well-organized.

Assumption 1 indicates that our proposed methodology is specific to large and complex applications. Assumptions 2 and 3 make sure that the necessary materials that programmer may need for the program comprehension and decomposition are available. More details are provided in Chapters 3 and 4.

1.3 List of Contributions

The following list describes the contributions of this work:

1. Introduction of an approach for program comprehension - The approach we proposed for program comprehension is systematic. It includes what needs to be prepared before the comprehension process and how to do the program analysis during the

comprehension process.

2. A program decomposition technique for introducing concurrency - The technique we proposed for program decomposition is based on design decisions. It is performed with the aid of software documentation and program comprehension. The technique is very easy to be implemented.

3. An approach to selecting multithreading models for sequential programs - The approach we proposed for selecting a concurrency architecture is based on the decomposition result. We also summarize and present three multithreading models as candidates for selection.

4. A survey of multithreaded programming techniques - We integrated some useful multithread programming techniques from other people's work. We also proposed two techniques based on our project experience. All these techniques can simplify the implementation process.

5. Application of above-mentioned techniques in the IBM prototype - With the aid of approaches and techniques as mentioned above, we successfully introduced concurrency into the IBM prototype and achieved good performance. The speed of the components that we focused on increased over 70%.

In summary, the contribution of this thesis is a complete and systematic methodology for introducing concurrency into sequential programs. The proposed methodology provides solutions for each step of the software development process. By following our methodology, the required time to finish the project will be shortened. The methodology

can be widely used on sequential programs with different levels of complexity, especially suitable for large and complex programs.

1.4 Organization of the Thesis

The thesis is organized as follows:

In Chapter 2, we list challenges that may appear in each software development activity. In Chapter 3, a program comprehension approach is proposed. In Chapter 4, we present a technique for program decomposition. In Chapter 5, we introduce the selection of a concurrency architecture and its implementation. In Chapter 6, we review some useful techniques for multithreaded programming, debugging and testing. In Chapter 7, we describe how the proposed methodology is applied to the IBM prototype and present the performance improvement obtained. Finally, we conclude this thesis and discuss some future work in Chapter 8.

2 Software Development Activities and Challenges

2.1 Software Development Process for Introducing Concurrency into Sequential Programs

Introducing concurrency into large sequential programs is not a simple task, especially for programmers who are not familiar with the original program [4]. This process is made up of more than one software development activity. Bad decisions in one development activity may bring negative effects to the final performance, such as the failure to introduce concurrency, too much time consumed, human and financial resources wasted and so on. We assume that the software development process for introducing concurrency is comprised of the following five software development activities:

1. Program Comprehension
2. Program Decomposition
3. Selection of a Concurrency Architecture
4. Multithreaded Program Implementation
5. Debugging and Testing

Recent research usually provides good ideas to guide programmers in specific steps in this process. However, this is not enough since transforming sequential programs into concurrent programs is a software development process and must be considered as a whole. To the best of our knowledge, among the approaches that focus on introducing

concurrency into sequential programs, no one provides solutions for all software development activities. The goal of our proposed methodology is to offer instructions for each activity to programmers when they transform sequential programs into concurrent programs. We expect programmers can shorten the required time on the project and achieve better performance by performing our proposed methodology. We searched the literature on the first four software development activities. For each of them, we reviewed the existing approaches and proposed new approaches to overcome the challenges. We also integrated other people's work into our methodology for the last two activities. The crux of our proposed methodology is selecting and modifying appropriate loop statements for introducing concurrency, so the approaches presented for different activities are related to the loops, such as program comprehension, program decomposition and selection of concurrency architecture.

2.2 Challenges for Each Activity

2.2.1 Program Comprehension

Understanding the original program is the first and the most important step when trying to transform a sequential program into a concurrent program [23]. All the following steps, such as program decomposition and selection of a concurrent programming model, are based on the programmer's good understanding of the original program. However, this task will be arduous for people who are not involved in the development of the original program because only the author of the source code totally understands the purpose and meaning of the program. Generally, the first step is to ensure familiarity with the

programming language. Naming conventions [95] is another point that should be confirmed. Sometimes, the knowledge of the programming habits of the author would be an asset in understanding the source code, but at the same time the author's programming habits may be difficult to grasp. Even if all the above conditions are satisfied, people may still fail to grasp the essence of the original program because of the complicated structure of the source code. Therefore, reading and understanding the source code is a daunting task that can cost a lot of time and energy. A systematic approach is presented in Chapter 3 to address this problem.

2.2.2 Program Decomposition

After understanding the original program, the programmer should be able to conclude a top-down structure of the source code. Based on this, the next step would be the separation of the program into sequential parts, since modifying the whole program from sequential order into a concurrent program is an unrealistic and impractical idea [96]. Reasonably decomposing the original program into several parts such that each part may be separately transformed into multiple threads poses a big problem. The decomposition results may directly affect model selection and implementation of concurrent programming. If the separation process is too detailed or not thorough enough, it becomes much harder to find an appropriate programming model with concurrency to match the target part. The techniques to solve this problem are discussed in Chapter 4.

2.2.3 Selection of Concurrency Architecture

Achieving success in introducing concurrency into the application does not mean it will

be suitable for users. Software developers and clients care not only about the application's viability, but also its efficiency in solving practical problems. The objective of converting sequential programs into concurrent programs is to improve performance. However, it is possible that the performance of the application as a concurrent program is not good enough and cannot satisfy the clients because of the bad selection of the concurrency architecture [45]. In some cases, synchronization and mutual exclusion problems may increase because of the inappropriate models. These problems would cost so much time for programmers to figure out and fix. All of these reasons could contribute to worsening the application performance. Thus, selecting appropriate multithreading model for the sequential program is a big challenge in concurrent programming. More details about the selection of a concurrency architecture are discussed in Chapter 5.

2.2.4 Multithreaded Program Implementation

In the implementation process, synchronization and communication are big problems that programmers have to face during concurrent programming. In a successful concurrent program, multiple threads must synchronize and communicate very well. As Michel Raynal points out in [6], "synchronization is the set of rules and mechanisms to ensure that two concurrently executing threads do not execute specific portions of a program at the same time." Communication between multiple threads refers to conveying messages which include data and wake-up information. Without synchronization and communication mechanisms, the asynchronism of multiple threads will cause the system to become disordered, especially when critical resources are being used. Other serious problems, such as data race [7], resource starvation [8], and deadlock [9], may come up

when the concurrent programs are running. Thus, synchronization and communication are prerequisites for concurrent program implementation. More detailed theories and practical approaches to accomplish synchronization and communication between multiple threads can be found in [10]. Some useful techniques and approaches are introduced in Section 6.1.

2.2.5 Debugging and Testing

Concurrent programming is much more difficult than sequential programming because synchronization and communication need to be taken into account. Moreover, debugging and testing concurrent programs is even harder due to the non-deterministic nature of concurrent programs [45]. Debugging in concurrent programming is quite a complicated problem compared to sequential programming. In sequential programs, the result of a workflow is predictable if the developer provides the same input data and keeps the system in the same state. Developers can check different parts of the program by setting breakpoints and then locate bugs step by step with the appropriate input data. This approach depends on the predictability in the sequential programs. However, it is very challenging to debug concurrent programs by using breakpoints setting and one step tracking because reproducing the exact context is too difficult. A large number of factors have to be considered during this process, like operating system state [1], processor time slices, priority of multiple threads and so on. To reproduce the exact same environment during debugging requires that each thread is assigned with the same tasks as before. The processor scheduling rules [1] must be known and the execution orders of multiple threads must be reproduced. The status of the memory and context switching must be

reproduced exactly the same at each time. Even the debugging tools may impact the exact environment. This means that “recreating the same sequence of events in order to debug a program is often out of the question” [11]. Alternative debugging approaches for concurrent programming that replace setting breakpoints are introduced in Section 6.2.1.

If the concurrent programs are compiled without errors and are runnable, then the next step is testing. It is possible that some potential problems which lead to wrong results still exist but do not happen during the testing process. 100% successful test cases do not mean that the program is totally correct in concurrent programming [97]. Multiple threads are running concurrently and in some cases they may have dependency on each other. There are many different execution orders for multiple threads when they go through a piece of code. When designing test cases, software testers have to consider the dependency and try their best to cover all different execution orders. Otherwise, potential problems may still arise after the program has been tested hundreds of thousands of times. Therefore, reliable testing strategies and testing tools are required in this field. In this thesis, we discuss testing techniques for concurrent programs in Section 6.2.2.

3 Program Comprehension

3.1 Literature Review

In the software development cycle, software maintenance takes usually much more time than software creation [12]. Among the time for maintenance, most of the time is spent on reading and understanding the original program. Obviously, program comprehension is a very important step in software engineering.

Brooks [13] elaborates a top-down theory for program comprehension. Programmers propose a hypothesis based on the general understanding of the program. Other sub-hypotheses are proposed to refine the first hypothesis. A hierarchy may be established by these hypotheses. Some hypotheses are refined by other hypotheses. The information of the program is connected with the help of hypotheses. Hypotheses are verified or rejected by programmers with the fact and knowledge of the original source code. Programmers repeat this process until they accomplish the goal. Soloway and Ehrlich [14] report their experiments and observations to prove that top-down theory is very useful in program comprehension when programmers are familiar with the source code.

Bottom-up theory is in contrast to top-down theory in program comprehension. Letovsky and Soloway [15] put forward the “chunk” concept which can be considered as a high level abstraction from the source code. Programmers read code statements and divide them into small chunks. Small chunks can be aggregated into big chunks. Programmers

can continue this process until a high level comprehension of the whole program is obtained [16]. Pennington [17] also presents a bottom-up model for programmers to understand the program. Her model uses control flow of the program to extract chunks from different kinds of operations.

Another approach for program comprehension is called hybrid model, which integrates the top-down and bottom-up strategies together. Mayrhauser and her partners presented this approach in [18]. Programmers can use top-down comprehension when they notice that some familiar algorithms are performed. Then they can propose new hypotheses and dig the code to test their assumptions. When an unrecognized section is encountered, programmers can switch to bottom-up comprehension and try to group code statements into chunks for further understanding. Rugaber et al. [19] report their experience on understanding FORTRAN code with the combination of top-down and bottom-up approaches. They use the top-down strategy to search the problems that the program is trying to work out and use the bottom-up strategy to find out the design decisions which are better to know for reverse engineering.

Top-down, bottom-up and hybrid strategies emphasize the comprehension of the entire program. However, for a specific purpose, understanding part of the program is probably enough to achieve the goal. The details about how to perform these three strategies on a relatively small piece of source code needs to be explored.

In addition to top-down and bottom-up theories, the systematic and as-needed strategies are also opposites for program comprehension [20]. In [21], Littman et al. propose the

systematic comprehension that starts at the beginning and the entire program is necessary to be explored. This strategy contains the comprehension of static knowledge [22] and casual knowledge [22]. Lakhotia [23] argues that it costs so much to comprehend the large applications and sometimes it is even impossible. The as-needed comprehension, which emphasizes the understanding a piece of program that is related to the modification, would be a better choice.

Knowledge is the prerequisite for program understanding. In [24], Rugaber elaborates the importance of domain knowledge in program comprehension. Petrenko [25] et al. show how to use ontology fragments [26] to collect knowledge during the understanding process and apply it to concept location [27]. Marcus and his colleges in [28] present concept location techniques, such as String Pattern Matching and Dependency Searching, to speed up the comprehension process. Most previous techniques are presented with a specific programming platform, such as Microsoft Visual Studio, Eclipse and so on. More platform independent approaches that are useful for program comprehension need to be explored.

Some tools are also designed to make the program comprehension process more effective. The tool developed by Murphy [29] and his colleague supports top-down comprehension. Katalin and Sneed [30] designed a comprehension tool based on seven questions which are necessary to be answered during the comprehension process. Recently, visualization tools, such as Codecrawler [31] and sv3D [32] are developed to help programmers be familiar with the entire applications. Most tools can make the comprehension process easier, but usually they cannot provide all information the programmers want to know

[30]. Researchers are still trying to develop new tools which can satisfy the requirements for different purposes.

3.2 Proposed Approach

Compared to the program comprehension approaches mentioned in the literature review section, our approach proposed in the following is used with the specific purpose of introducing concurrency into sequential programs. Locating and analyzing of loop statements is specific to this purpose since an appropriate loop has good structure to be transformed into a concurrent program. The proposed approach follows the idea of Lakhotia's as-needed strategy [22] because "many programs are so large and so complex that they cannot be comprehended in their entirety no matter what forms of representation are used" [36]. What we care about is the part of the program that has opportunities to be refactored with multiple threads. If systematic comprehension [21] is applied, programmers may spend so much time facing extremely complicated parts of the program which is not practical for multithreading. Our approach provides a simple way (looking for loop statements) to shrink the scope of the program that needs to be comprehended and programmers can pay more attention on the valuable content. Instructed by our approach, programmers may refactor the code of the targeted loop one by one without being concerned about the comprehension of the remaining program. To some extent, some existing approaches and our proposed approach are complementary since we can use the ideas mentioned in those existing approaches, such as top-down theory [13] and bottom-up theory [15], to analyze the details of loop operations.

Nowadays, a large number of programmers are working with software applications that are developed by others and up to 50% of the time is spent on the program comprehension phase [33]. Later sections in this chapter present an approach for understanding the program of software applications. This program comprehension approach provides specifics to programmers who are responsible for introducing concurrency into sequential programs. The basic idea of this approach contains two parts, preparation and analysis. Some background knowledge should be prepared first and then the programmers can try to analyze the original program and find opportunities to refactor the source code with multiple threads.

3.2.1 Background Knowledge Preparation

Background knowledge is very useful for new programmers who are responsible for modifying the software application from a sequential program into a concurrent program. Software documentation can help programmers make an overview of the software architecture and learn the functionality of the software application. With the help of a clear view of the architecture and functionality of the software application, program comprehension can become a relatively easy task. Knowledge of the programming language is a prerequisite for transforming sequential programs into concurrent programs.

3.2.2 Naming Conventions

In addition to the software documentation and programming language, naming conventions is another point programmers need to handle. For different developers or different development teams, the difference in naming conventions could be significant.

In most cases, naming conventions include variable names, method names, class names and even some file names. Naming convention seems like a communication jargon constructed by programmers above the programming language. Moreover, programmers can express higher level concepts by complying with naming conventions. At the same time, much richer information about the proprieties and functions of the variables, methods and classes can be conveyed to the programmers from naming conventions. For example, the Hungarian notation [34] is a very famous naming convention in programming. Programmers can easily identify the type of each variable from its name. When programmers are familiar with the naming conventions, they can try to understand the code from the author's perspective, which makes the program much easier to deal with. Therefore, it is necessary to take some time to summarize naming conventions when reading the source code.

3.2.3 Mapping architecture with the source code

Collecting software documentation, learning programming language and exploring naming conventions are served for program understanding and analyzing. These three processes can help programmers to gain a general idea about the software application and some basic knowledge that could be useful during the modification process. After this, the programmer needs to further explore the software architecture. What is described in architecture documents is high-level design and can only present a rough impression since the content is not at the level of the source code. However, all modifications made by the programmer will be present in the source code. The programmer needs to establish a map between the architecture and the source code so that when the programmer wants

to know about the implementation details for some parts of the software application, he or she can locate the precise position in the program.

A good way to achieve this goal is to find the entry point of the software application in the program first and then run the original program in debug mode over and over again. In this way, it is possible to see how the software application works from the beginning to the end. In the meanwhile, programmers may try to clarify the starting point and end point of each software component in the source code to prepare for the following step. It is not necessary to explore details of each line during this step because it would cost a lot of time and usually it is not helpful for programmers to understand the whole picture. At this step, the key point is to master the structure and layer of the program in the code level.

3.2.4 Recording Information for Loop Statements

When running the program in debug mode to know well about the structure of the application, programmers need to finish another very important task, which is looking for loop statements and recording related information, such as the location, data and method inside the loop and so on. Introducing concurrency into loop statements is the key point in our proposed methodology because of their structure and intensive computation feature. The structure of loops usually implies that a certain number of instructions are required to be performed multiple times, which provides the possibility to refactor loops with multiple threads. Oftentimes, most intensive computation is accomplished by loop statements. Decreasing the executing time of loop statements may achieve greater

performance for the entire application.

4 Program Decomposition

4.1 Literature Review

Kim et al. [37] propose a role-based technique for program decomposition. A role hierarchy tree is built during the decomposition procedure. Each component is assigned with a role in the tree. The root node represents the system and the leaf node can be considered as the independent component. This technique takes the role as the main factor to make decomposition decisions. However, the decomposition result is beneficial to improve concurrency in the distributed environment in which there are different independent machines. It is not clear whether Kim's technique is good for multithreaded programs in which multiple threads share resources and execute operations on the same machine.

In [38], Parnas compares two program decomposition approaches. One of them is based on "information hiding" [93]. The module is selected with the reason of "design decisions" which hides itself from others [38]. The second decomposition approach is making a rough flowchart first and then let each major working step in the flowchart be a module. Our proposed approach uses the idea from Parnas's second approach because we can easily get the flowchart from software documentation. Identifying major working steps and making decomposition will be a relatively easy task to finish. On the other hand, Hofmann argues that using information hiding as the criteria for decomposition sometimes is not appropriate because "the importance of so-called hidden assumptions of model abstraction and idealization" [94] is neglected. This issue is particularly knotty

when the program to be decomposed is big and complicated.

The program decomposition techniques mentioned above are from the aspect of the entire application. Programmers can use these techniques to split up the entire program into several parts. There is another way to think about this problem. That is, grouping the small entities into bigger parts, which is usually called clustering technique. In [40], Mancoridis et al. propose a clustering technique based on the dependency graph of small entities. They also design a software tool to analyze the dependency graph and cluster the small entities into modules. Mancoridis and his teammates present "modularization quality" as the measurement criteria for their approach. The tool minimizes the inter-connectivity [40] and maximizes the intra-connectivity [40] in order to provide the best results. A survey by Wiggert [41] concludes the clustering algorithms which are applied to system re-modularization, such as graph theoretical algorithm, optimization algorithm and hierarchical algorithm. In our proposed method, the idea of clustering is used and implemented in a simple way.

Clustering theory is also applied to design software tools [42, 43] for the decomposition of application. However, these tools are not completely automatic. In order to achieve good decomposition results, programmers who are familiar with the structure of the entire application should play an important role in this process. These tools are not perfect for our research since the programmers who are responsible for introducing concurrency may be not familiar with the code at all. Clustering tools can break down the application into several independent parts, but the decomposition result may not be useful for refactoring the source code with multiple threads. Future work needs to be done to make the tools

more powerful for the purpose of transforming sequential programs into concurrent programs.

4.2 Proposed Approach

Multithreading a whole software application that is written as a sequential program is not practical since not all included algorithms can be implemented with multiple threads. Multithreading models are more likely to be suitable for part of the program, not the entire code of the software application. Therefore, it is necessary to break down the program of a large and complex software application and select the parts that are possible to be transformed into concurrent programs. In the proposed methodology, the program decomposition phase has this purpose.

In general, component and module are two different words for the same concept of component-based software. However, in our proposed approach for program comprehension, they have different meanings. Component denotes the functional unit at the architecture level which is used to transmit and process data to accomplish one or more specific sub-functions of the software application. Module denotes the source code cluster that is used to manage part of the source files which are integrated together to perform a function (here used in its non-programming sense) in this thesis. In other words, the application is composed of several components which perform a set of functions (here used in its non-programming sense). Each component can be mapped to one or several modules which integrate a bunch of source files according to the code structure. The reasons why the decomposition hierarchy has two levels are the followings:

1. Decomposing the whole application into components is to identify the part of the entire application which has a chance to be implemented with multiple threads and add those components into a candidate list for refactoring. According to the software documentation and our comprehension of the program, we can know the input of each component and also its function (here used in its non-programming sense). If the input of the component can be decomposed into independent units, then it can be selected as the candidate for further analysis. If the function of the component is to deal with a bunch of stuff, it can also be added into the candidate list. If the component is used to create stuff like a graphical user interface, then it is probably not the part we are looking for. In most cases, the number of component is far less than the number of modules, so the identification of candidates at the component level is simpler than at the module level.

2. Mapping components to modules is to determine the scope of the component at the code level. A clear scope allows the programmers to count the execution time for the component on the candidate list. If the executing time of the candidate component is small compared to the overall time, then the programmers can eliminate the component from the candidate list even if it provides opportunities to be modified with multiple threads. This is because the changes we make should have practical significance. Programmers can also make a priority list for the candidate components according to the executing time. The one with the highest priority will be analyzed and transformed into a concurrent program in the first place.

In summary, the reason for decomposing the entire application into components and then into modules is to shrink the target range and prepare for the Big Loop selection (which is

presented in Section 4.2.3). The key point of our methodology is to find appropriate loops in the original code and refactor them with multiple threads. However, there are probably many loops in the program of a large and complex application. Recording and checking the information of each loop will cost much time. The decomposition process can identify some components which are not necessary to be analyzed. The loops inside those components are eliminated at the same time. We present the details of the two-level hierarchy decomposition next.

4.2.1 Decomposition into Components

A software application is developed because its function (here used in its non-programming sense) can help humans to solve practical problems. Generally, the realization of a function is depending on the realization of several sub-functions. In this perspective, a software application is made up of several components. Therefore, a software application can be broken down into a few components.

The decomposition process can be accomplished by referring to software development documents. Usually, in the design phase, the high-level design of the software application has been determined. The designers break down the software application into several parts according to its function and assign them to different software developers or development teams. Each part in the design phase can be treated as a component. Programmers may try to draw a diagram with input and output information to clarify the relationship between different components. An example of a component diagram is shown in Figure 1.

Decomposing the entire application into components is the first step in our proposed approach for program decomposition. The idea of Parnas's second program decomposition approach [38] is used in our proposed approach. It is based on separating the entire program into several major working steps according to the working flowchart. It provides a quick way to realize decomposition since we can easily obtain the program architecture and working flow information from the software documentation and our comprehension of the program.

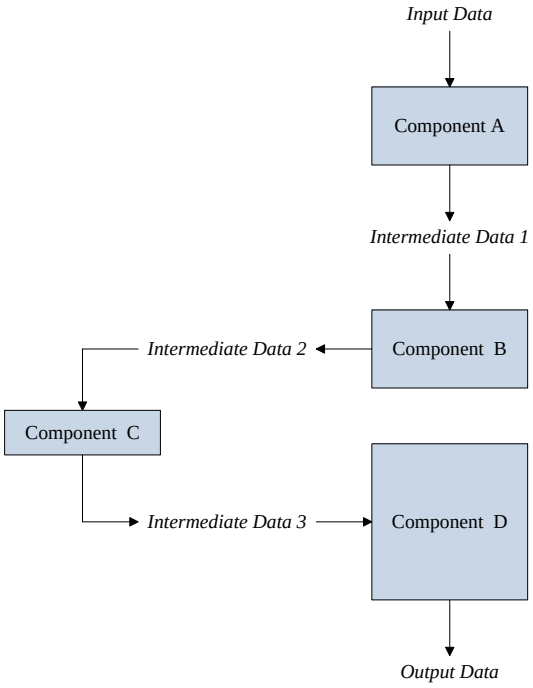

Figure 1: Component diagram example

4.2.2 Mapping Components with Modules

The source code of a software application is often well-organized. For example, in most programming languages, the entire program of a software application is usually made up of many "projects" that contain source files. The "project" can be regarded as a module.

In this perspective, a component which is generated from the previous decomposition process may correspond to one or more modules at the code level. Figure 2 describes the relationship between components and modules in a software application programmed in C++ language.

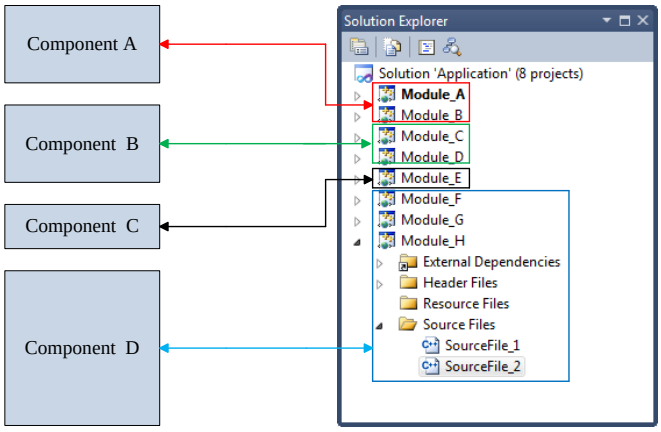

Figure 2: Correspondence relationship between components and modules

At the same time, the entire program of the software application can be broken down into groups of modules, each group corresponding to a given component. Figure 3 provides the program decomposition result after this step.

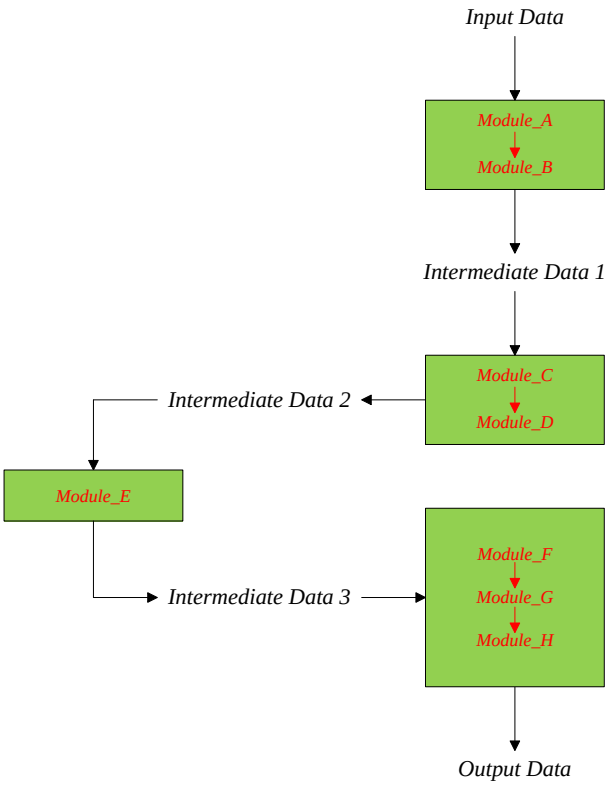

Figure 3: Program decomposition result

Mapping components with modules is the second step in our proposed approach for program decomposition. The basic idea is clustering the source code (or source files) into modules. This clustering process can be easily done by referring to the organization of the source code. Components can be further decomposed into one or more modules according to their functions.

4.2.3 Big Loop Selection

The first two program decomposition steps are quite simple. After that, the next step in the process would be selecting the big loop as the target for introducing concurrency and combining the loop information with the decomposition result to find an appropriate multithreading model. Loops usually imply that a certain number of instructions are

required to be performed multiple times, which provides the possibility to realize data parallelism if each repetition of the loop is independent. Compared to task parallelism, data parallelism would be a good starting point to introduce concurrency into sequential programs when programmers are not that familiar with the source code. Programmers can try to find the Big Loop among the loops which cover the operations of the component in the candidate list. It will save more time and have higher success rate. A Big Loop needs to satisfy the following four rules:

1. It should be a loop statement. Each repetition of the loop should be independent or having chance to be modified to run independently.

2. There should be no operations jumping into or out of the loop.

3. The execution time of the loop should be at least 5% of the total running time of the software application.

4. The program covered by the loop must not be event-based program.

Usually, a loop is the segment that has the most intensive computation [44]. Sometimes a loop processes different data from the same data set. This model is called SIMD (single instruction, multiple data), which is suitable for data parallelism. That is why Rule 1 emphasizes the loop as the key point to solve our problem. Once the loop is found and determined as our target, the next step is to analyze the data and method inside itself to make sure that each repetition of the loop process is really independent or having chance to be modified to run independently. Programmers can achieve this goal with the help of control flow analysis and data flow analysis. Each repetition of the loop can be

considered as the execution of a series of methods, so programmers can follow the call hierarchy of these methods to check if there is dependence between each repetition. Analysis of the data inside the loop is another way to identify the dependence relationship. Independent repetition will not wait for the output data of other repetition to finish its own operations. In other words, the execution order of each repetition should not affect the final result. For those loops whose repetitions need to be modified into an independent form, reentrant programs would be a good candidate. In [80], Jan Wloka and his colleagues stated that "a program is *reentrant* if distinct executions of that program on distinct inputs cannot affect each other." Their point of departure to introduce concurrency is making sequential programs reentrant. The approach they select to accomplish this goal is converting global variables into thread-local variables and creating new threads for execution. Every fresh thread is assigned with a copy of the global variable. When the accesses to the global variables are needed by multiple threads, they can use an index mechanism to find their own copies and execute the operations. They substitute a thread local state for a global state. REENTRANCER is a practical eclipse-based tool [80] developed by the group. In our methodology, the thread local storage which is mentioned in the multi-threaded programming skill section is similar to this idea.

Rule 2 means that in our proposed methodology, we only consider the Big Loop for introducing concurrency. Jumping into or out of the loop may result in difficulty of thread control. In other words, what we want multiple threads to execute concurrently is the contents inside the Big Loop. If there is a thread that jumps out of the loop, it may touch

the sequential program which should be executed by the main thread when all the created threads are killed. Such an unexpected operation may probably lead to wrong results.

Rule 3 disregards the loops with less computation. It makes sure that the modification has practical significance and may bring obvious improvements in efficiency.

Rule 4 implies that our methodology is not applicable to event-based program. An event-based program is usually a sequential program. However, various user actions, such as mouse clicking and key pressing, are frequently used in the application. It is difficult to find an appropriate concurrent programming model to match event-driven mechanism. That is why the sequential program that we focus on does not include event-based program in this thesis. In summary, a loop that meets the Rule 1, 2 3 and 4 can be considered as a Big Loop.

The Big Loop does not necessarily contain all the stuff of the modules (or components). It may only contain part of the operations which are executed by the module (or components) or it may be inside a module (or components). Figure 4 shows an example of big loops combined with the program decomposition result. Every blue dotted ellipse is a separate big loop. In Figure 4, a blue dotted ellipse covering components or modules does not mean the big loop must include the entire components or modules, it indicates that the big loop that contains some operations from those components or modules. This rule also applies to the following figures.

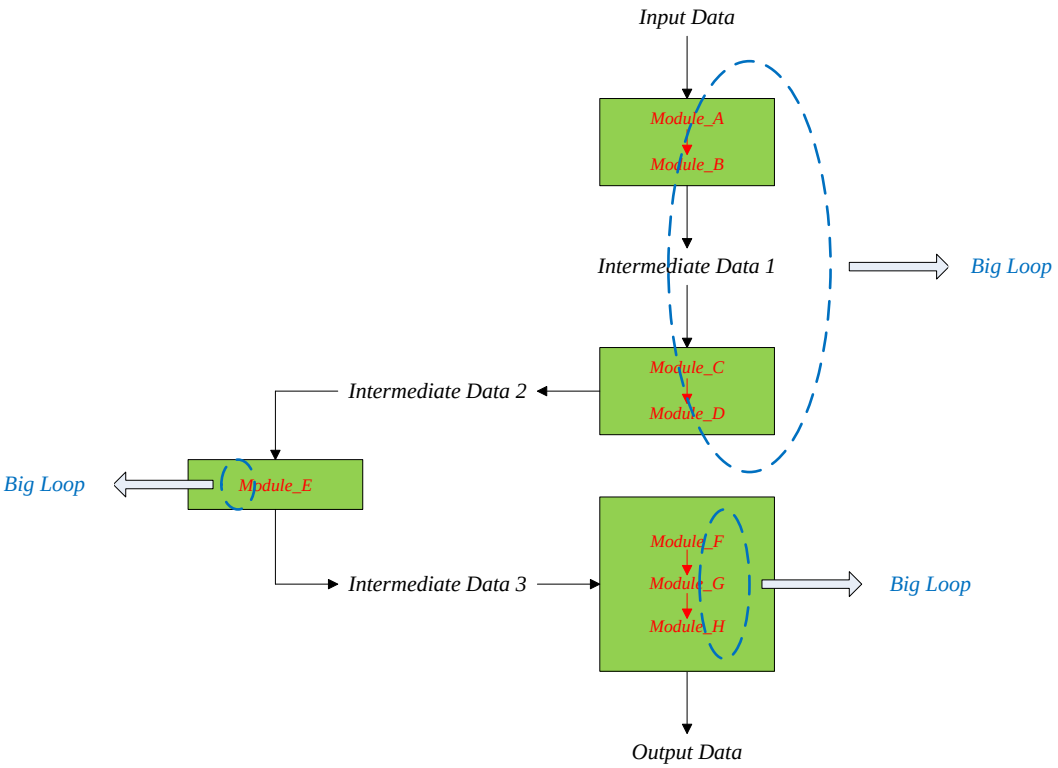

Figure 4: Big Loop with program decomposition result

5 Architecture Selection and Implementation

5.1 Literature Review

Welsh et al. [39] design a framework for concurrent systems based on three components: tasks, queues and thread pools. They also present several patterns which can be used to create concurrent systems with their designed framework. Their framework only serves creating new software applications. Their research does not aim at transforming existing applications into concurrent forms. However, programmers can still bring the design idea of the framework into the transformation process.

Currently, most software products are based on object-oriented programming. In order to bring concurrency concept into object-oriented programming, the researchers in [82] put forward an object-oriented concurrent programming framework based on two important concepts, objects and message passing. The concept of an object is defined as an instance of a class which refers to a module containing attributes and methods. Objects have states and behaviors which are used to model actions in real life. Message passing is identified as a communication mechanism that objects follow to coordinate with each other. Message passing includes all kinds of interactions between objects, such as notifying, demanding data, requesting computation and so on. In this framework, each object uses a buffer to store messages. Each message triggers the execution of a method. Objects are working concurrently and they accomplish synchronization control through sending and receiving messages. This framework provides message passing as a good means to realize synchronization when introducing concurrency into sequential programs.

The issue of concurrent objects communicating with message passing is that it is not efficient enough in the context of inheritance which is a powerful mechanism for code-reuse. It turns out that "using inheritance to reuse synchronization specification (on scheduling of message acceptance) doesn't always work smoothly" [82] in the object-oriented concurrent programming methodology. In other words, "when deriving a subclass through inheritance, the presence of synchronization code often forces method overriding on a scale much larger than when synchronization constructs are absent, to the point where there is no practical benefit to using inheritance at all" [98]. This issue is called inheritance anomaly [83]. In a large and complex object-oriented application like the IBM prototype, there would be a large amount of inheritance relationships. Message passing framework may not be a good choice due to the inheritance anomaly in the original program. In short, the object-oriented concurrent programming framework provides a way to introduce concurrency into sequential problems, but the researchers have neglected a serious problem (inheritance anomaly) caused by this framework. Efficient solutions to overcome the inheritance anomaly in the transformation process could make this framework more popular and useful.

Data parallelism and task parallelism are the most common concepts when we talked about multithreading. Data parallelism refers to the scenarios in which the same operation is performed concurrently on different data in a source collection [84]. From the perspective of multithreading, data parallelism means that each thread executes the same piece of code to process different data which are in the same dataset. Task parallelism refers to one or more independent tasks running concurrently [85]. Task here

means a unit of work multiple threads need to finish. In other words, task parallelism can be applied to the situation that data needs to be processed by different independent tasks and usually one task occurs after the other task has completed in the sequential program.

Since data parallelism and task parallelism have good features for introducing concurrency, some researchers tried to modify the program to make it more suitable for realizing one of these two designs. In [86], Tefft and Lee put forward an approach that uses the idea of data parallelism. A “Mapping Algorithm” is proposed to convert part of the original program into the Single Instruction Multiple Data (SIMD) [46] form which is appropriate for multithreading. A refactoring tool called RELOOPER is designed based on this approach. Their test program has only 10 lines and they leave the performance characteristic discussion as further study. The feasibility of their approach to introduce concurrency into large and complex applications needs more research to confirm. Our approach also involves data parallelism and considers loop as a key point to introduce concurrency. However, we provide experimental performance to show that the proposed approach works fine for large and complex applications, which Tefft’s group does not.

Hughes [45] elaborates the architecture of concurrent programs using the aspect of instructions and data streams. He mainly considers single instruction, multiple data (SIMD) and multiple instructions, single data (MISD) architectures [47]. In multithreading, SIMD means multiple threads perform the same instructions on different data. Usually the data is in the same data set and they need to be processed by the same piece of code. In sequential programs, these operations are often performed with loop statements. The properties of SIMD imply that it can be implemented with the

boss/worker model [48]. MISD means multiple threads perform different instructions on the same data. The implementation of the pipeline model [48] belongs to this architecture, while a purist would say that the data is different since it is transformed at each stage [47].

In our proposed methodology, we select these three concurrent programming models as candidates for matching with the decomposition result. Each model may have more than one implementation form as introduced in Section 5.2. The reason to select these three models instead of others is that programmers can find a good match between the Big Loop and the implementation forms of these three models. Their implementations are more detailed, including the information of the structure and thread management mechanism. The selection process and implementation details of these models are presented in Section 5.3.

5.2 Multithreading Models

Multithreading models provide programmers a common way to design and refactor the program. Rational use of multithreading models may accelerate the modification process. In this section, we summarize three models that are very popular and widely used in concurrent programming. They are the Manager-Worker Model, the Civilian Model and the Pipeline Model.

5.2.1 Manager-Worker Model

The Manager-Worker Model, also called the Master-Slave Model [50] or the

Boss-Worker Model [48], offers good structure for data parallelism. In the Manager-Worker Model, there is a thread which plays a role of a manager. It is responsible for creating worker threads and assigning tasks to each worker thread. The responsibility of a worker thread is to perform specified functions (here used in its programming sense) with the data assigned by the manager. In most cases, the manager thread will wait until all worker threads accomplish their tasks and are then killed. After that, the manager thread performs operations in sequential form. A Manager-Worker Model can be implemented in one of two ways [45].

1. A worker thread pool is created by the manager first, which means that the maximum number of worker threads which can execute concurrently is determined. The manager thread is able to assign tasks to worker threads. If each worker thread has a task queue, the manager thread will push tasks to the queues and the worker thread will pick up tasks from the queue. Figure 5 shows the Manager-Worker Model in which there is a thread pool and each worker thread has a task queue.

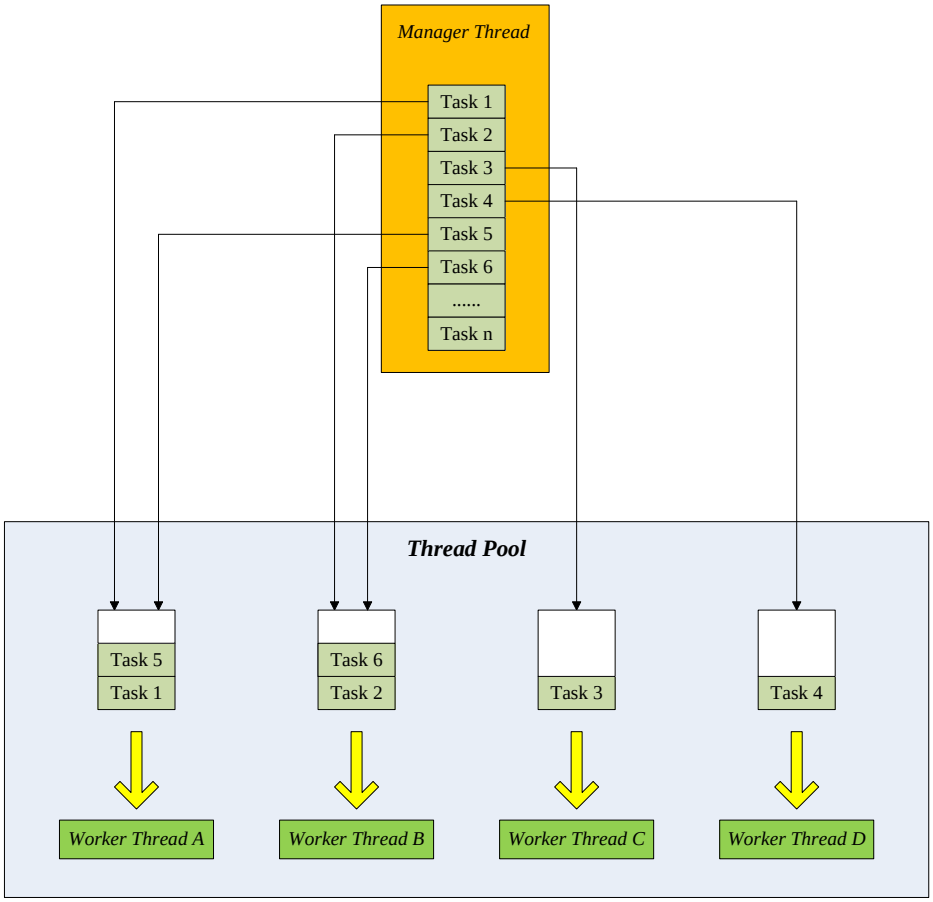

Figure 5: The Manager-Worker Model with pool and queue

If worker threads do not have task queues, then each worker thread can be assigned a new task by the manager thread after it has finished its current task. In this situation, if there are no worker threads in the idle state, the manager thread cannot assign tasks and has to wait. Each worker thread may perform more than one task. The whole process is done when the manager thread has no tasks to distribute and all worker threads are in the idle state. After that, the manager thread will destroy the worker thread pool and get into the next stage. Figure 6 shows the Manager-Worker Model that has a thread pool but without task queues.

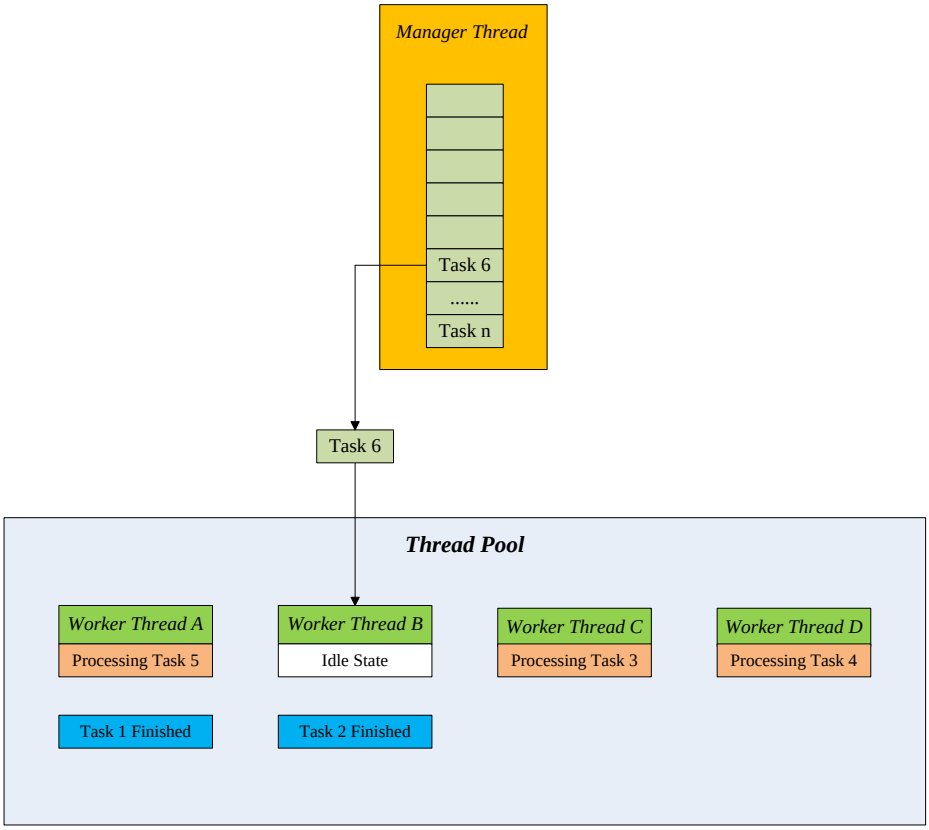

Figure 6: The Manager-Worker Model with pool but without queue

2. The manager thread creates a worker thread for each task. Each worker thread only deals with one task during its lifetime and is destroyed when its current task is finished. The manager thread directly controls the number of worker threads that can be run concurrently. However, hardware requirements and memory resources need to be considered in advance. This is because the number of processor cores is limited. Increasing the number of threads may not achieve better performance if they have to share a core. Having too many threads may also cause an out of memory condition. Figure 7 illustrates the Manager-Worker Model in this case.

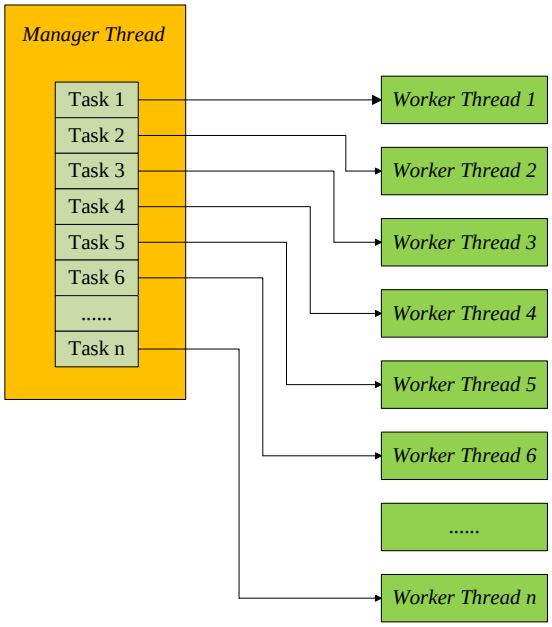

Figure 7: The Manager-Worker Model without pool and queue

5.2.2 Civilian Model

The Civilian Model (here we use “Civilian” as the name of this model to emphasize that there is no central control in this model, all the threads have equal status) bears some similarity to the Manager-Worker Model in that one thread creates all the other threads. The thread that created the others will also perform tasks just like the other civilian threads or wait until others finish. The difference is that all threads have the same working status in the Civilian Model [51]. There is no task assignment. Each civilian thread has to pick up its input data from a shared space or has its own channel for receiving input data [52]. In other words, civilian threads should know the data source in advance. There is no centralized control, either. Threads communicate and coordinate by themselves to realize synchronization. Figure 8 describes the Civilian Model in which civilian threads get input data from a shared space.

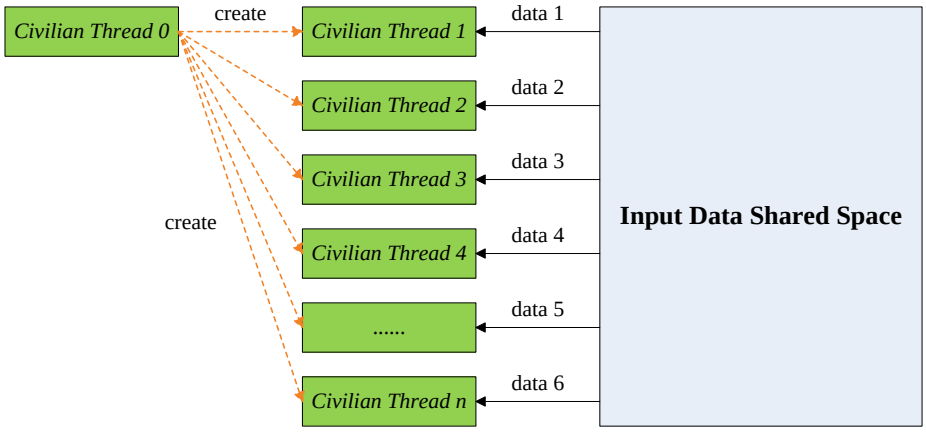

Figure 8: The Civilian Model with shared space for input data

Figure 9 illustrates another case of the Civilian Model in which each civilian thread has its own input data channel.

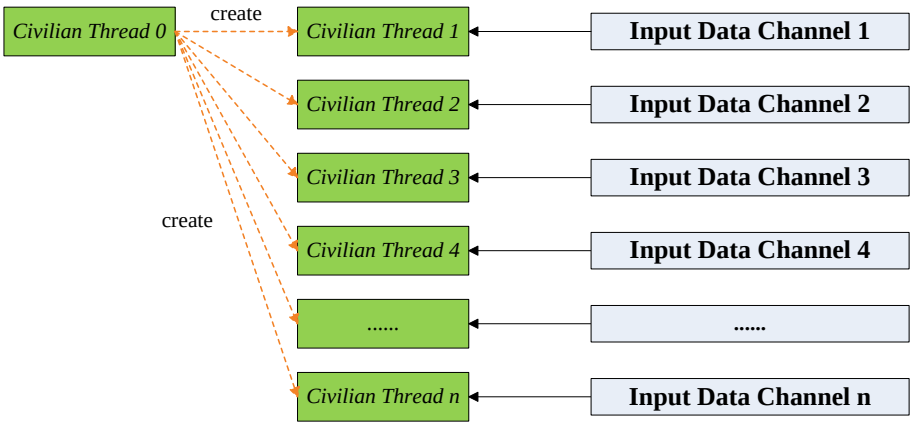

Figure 9: The Civilian Model with input channels

5.2.3 Pipeline Model

The Pipeline Model provides one possible way to achieve task parallelism. In the Pipeline Model, there are several processing stages. For each input data, going through all stages

means that it is completely processed. At each stage, there is usually one thread that processes the input data. Multiple threads in different stages can work concurrently [49]. The processing speed of some stages may be slower than others and the total execution time of the Pipeline Model depends on the slowest stage. In other words, the unbalanced pipeline may result in a waste of resources and low performance. The load balancing [53] issue in the Pipeline Model should be concerned about. One approach to solving this problem is to increase the number of pipeline threads in the stage requiring the most processing time. The other approach to dealing with load balancing problem is bringing work-stealing mechanism [54] into the Pipeline Model. Programmers should try to balance the execution time of each stage in order to achieve the best performance. After all input data are completely processed, their outputs will be integrated together as the output of the pipeline. The Pipeline Model is suitable for programs that can be divided into several stages, such as image processing, text processing and so on. An example of the Pipeline Model is shown in Figure 10.

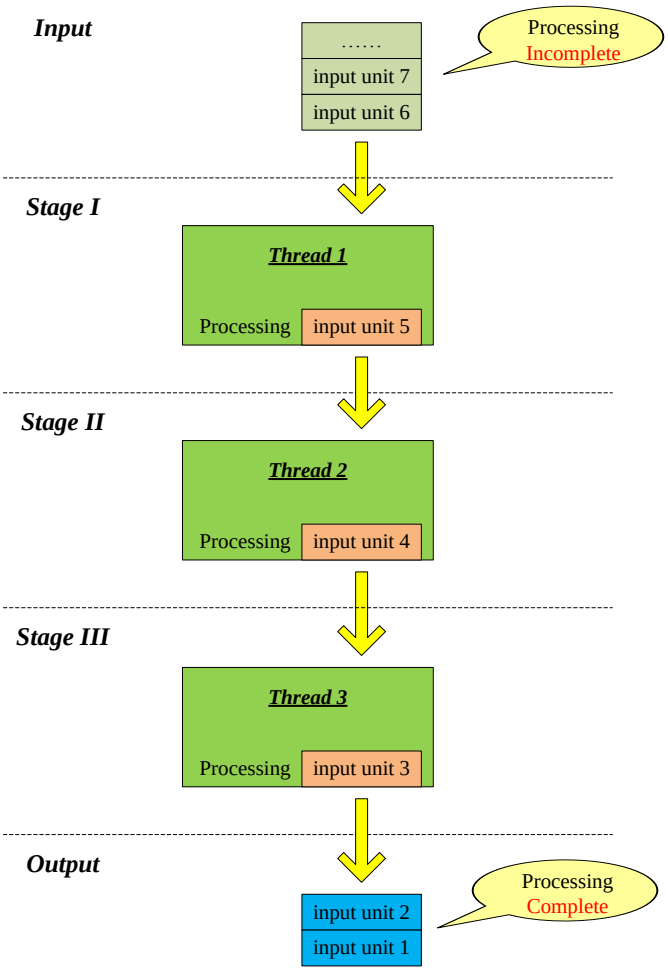

Figure 10: The Pipeline Model

5.3 Architecture Selection and Implementation

5.3.1 Single Big Loop

For a single big loop, the Manager-Worker Model can be selected as candidate. Since each repetition of the big loop is independent, worker threads can perform operations on different data concurrently. The manager thread is responsible for controlling communication among worker threads. The pseudo-code for the Manager-Worker Model is listed as follows:

Manager-Worker Model (with pool and queue)

```
01.  //Manager Thread
02.  pool=createThreadPool(MaxThreadNum);
03.  current_unassigned_task_ID=0;
04.  while(there is still unassigned task existing){
05.     thread_ID = current_unassigned_task_ID % MaxThreadNum;
06.     put current unassigned task into the queue(queue_ID=thread_ID);
07.     current_unassigned_task_ID++;
08.  }
09.  activate all worker threads in the thread pool to work;
10.  wait(until all worker threads finish their tasks);
11.  destroyThreadPool();
12.
13.
14.
15.  //Worker Thread
16.  while(task queue is not empty){
17.     current_task=dequeue();
18.     execute(current_task);
19.  }
20.
```

Manager-Worker Model (with pool but without queue)

```
01.  //Manager Thread
02.  pool=createThreadPool(MaxThreadNum);
03.  while(there is still unassigned task existing){
04.     suspend until there exist worker thread in idle state
05.     find idle-state worker thread and give it current unassigned task;
06.  }
07.  wait(until all worker threads finish their tasks);
08.  destroyThreadPool();
09.
10.
11.
12.  //Worker Thread
13.  while(in idle state){
14.     prepare to accept message from manager thread;
15.     if(the message is "being activated"){
16.        switch to working state;
17.        process task;
18.        switch back to idle state
19.     }
20.     if(the message is "being destroyed"){
21.        prepare to be destroyed;
22.        break;
23.     }
24.  }
25.
```

Manager-Worker Model (without pool and queue)

```
01.  //Manager Thread
02.  taskID=0;
03.  while(taskID<=taskNum){
04.     while(true){
05.        if(RunningThread<MaxThreadNum){
06.           createWorkerThread(…&workerThreadProc)
07.           lock();
08.           RunningThread++;
09.           unlock();
10.           break;
11.        }
12.        else
13.           wait until RunningThread<MaxThreadNum;
14.     }
15.     taskID++;
16.  }
17.  wait until RunningThread=0;
18.
19.
20.
21.  // Worker Threads
22.  workerThreadProc(){
23.     processing assigned task;
24.     lock();
25.     RunningThread--;
26.     unlock();
27.  }
28.
```

In the Manager-Worker Model with pool and queue, the manager thread creates worker threads and assigns tasks to worker threads in advance. In the Manager-Worker Model without pool and queue, the manager thread creates worker threads and assigns tasks as required. In the Manager-Worker Model with pool but without queue, the manager thread creates worker threads in advance and assigns tasks to worker threads as required. The programmers can make the best choice according to the actual situation, such as the number of cores, the properties of the task set, the overhead of thread management and so on.

The Civilian Model can also be selected for a single Big Loop. The difference between the Manager-Worker Model and the Civilian Model is the way to access the data for multiple threads. In the Manager-Worker Model, the worker threads get data from the manager thread. In the Civilian Model, the civilian threads need to obtain their data by themselves. The pseudo-code for the Civilian Model is listed as follows:

Civilian Model

```
01.  //Initial civilian Thread
02.  create civilian Thread(… thread1 … & civilianThreadProc);
03.  create civilian Thread(… thread2 … & civilianThreadProc);
04.  create civilian Thread(… thread3 … & civilianThreadProc);
05.  ……
06.  wait(until all worker threads finish processing);
07.
08.  //Other Civilian Threads
09.  civilianThreadProc(){
10.     while(there is still unprocessed data existing){
10.        pick up data from shared space or data channel;
11.        process data;
12.     }
12.  }
```

In some conditions, the single big loop may contain the operations from more than one component or module. With the number of modules inside the single big loop increasing, the difficulty to introduce multiple threads with the Manager-Worker Model or the Civilian Model may also increase. This is because worker threads in the Manager-Worker Model or civilian threads in the Civilian Model have to do operations for all modules inside the loop. More conflicts may exist and need to be resolved. The Pipeline Model can be considered as a good alternative. The Pipeline Model has several stages, where each stage corresponds to a module through which the data must pass. Consequently, a group of modules inside the big loop may have a chance to be transformed into a Pipeline

Model. The pipeline threads only work on one module so that the conflicts between the threads may decrease in this situation. Figure 11 and Figure 12 provide two cases in which the Pipeline Model may be a better choice for multithreading. In Figure 11, the scope of the single big loop is across two components. The loop includes the operations from four modules.

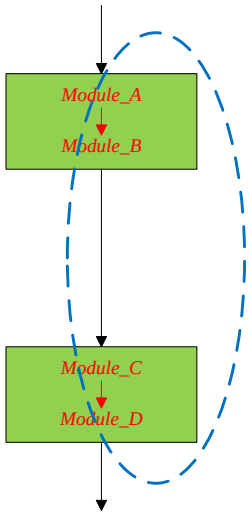

Figure 11: Single Big Loop (a) with the Pipeline Model

In Figure 12, there are two examples of big loops. Each of them contains the operations of more than one module and all these modules belong to the same component. The big loop on the left side includes the operations from all the modules (A, B, C, D and E) in the same component. The big loop on the right side includes the operations from part of modules (G, H and I), but not all modules in the component. The Pipeline Model may be a good candidate for the conditions similar to these two examples. The pipeline can be organized with several stages and each stage contains the operations from one module.

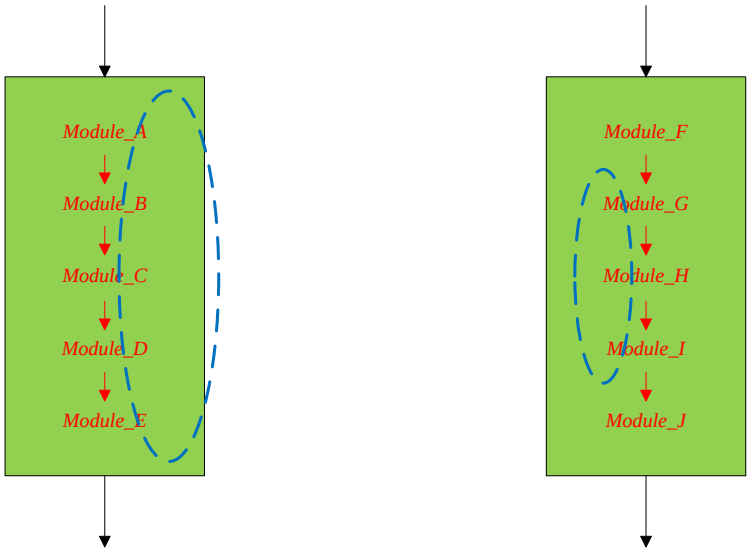

Figure 12: Single Big Loop (b) with the Pipeline Model

If similar situations occur, the Pipeline Model may be more useful than the Manager-Work Model or the Civilian Model. However, the Pipeline Model has to solve the workload balancing problem to achieve optimal performance. How to make a good choice between these three models should depend on the details of the program and requirements of the project.

Below is the pseudo-code for the Pipeline Model with three stages:

Pipeline Model

```
01.  //Main Thread
02.  createPipelineThread(&Stage_1_Proc);
03.  createPipelineThread(&Stage_2_Proc);
04.  createPipelineThread(&Stage_3_Proc);
05.  wait(until all pipeline threads finish their operations);
06.
07.
08.  //Pipeline Stage Threads
09.  Stage_1_Proc(){
10.    do the following process for each program input unit{
11.       get one program input unit from Main Thread as current data;
12.       process current data;
13.       forward result to next stage;
14.    }
15.  }
16.  Stage_2_Proc(){
17.    do the following process for each result of previous stage{
18.       get the result from previous stage as current data;
19.       process current data;
20.       forward result to next stage;
21.    }
22.  }
23.  Stage_3_Proc(){
24.    do the following process for each result of previous stage{
25.       get the result from previous stage as current data;
26.       process current data;
27.       forward result as part of the output of the program;
28.    }
29.  }
30.
```

Sometimes the single big loop only includes the operations from one module, like the two examples shown in Figure 13. The Pipeline Model also has opportunities to be chosen in this case. Since there is no obvious part mapping with the stage in the Pipeline Model, the decomposition process for the module is required.

Example I Example II

Figure 13: Single Big Loop (c) with the Pipeline Model

5.3.2 Multiple Big Loops

Concurrency can be introduced into a single big loop with the multithreading models. In some cases, multiple sequential big loops can also be transformed into the multithreading models. In Figure 14, there are three consecutive big loops.

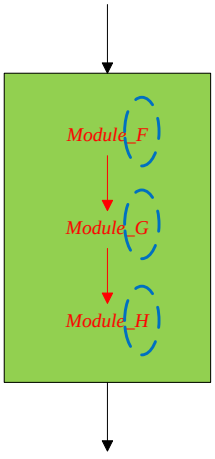

Figure 14: Multiple Big Loops (a)

The most direct way would be selecting the Manager-Worker Model or the Civilian Model for each of them to introduce concurrency. In this case, three big loops are necessary to be multithreaded. However, there are another two ways that programmers can try to realize multithreading.

1. If it is possible that each single big loop can be treated as a stage, then the program of these three big loops can be refactored into the Pipeline Model.

2. If it is possible that three big loops can be merged into a single big loop, then the new big loop can be transformed into a concurrent program with the Manager-Worker Model or the Civilian Model. This situation is shown in Figure 15.

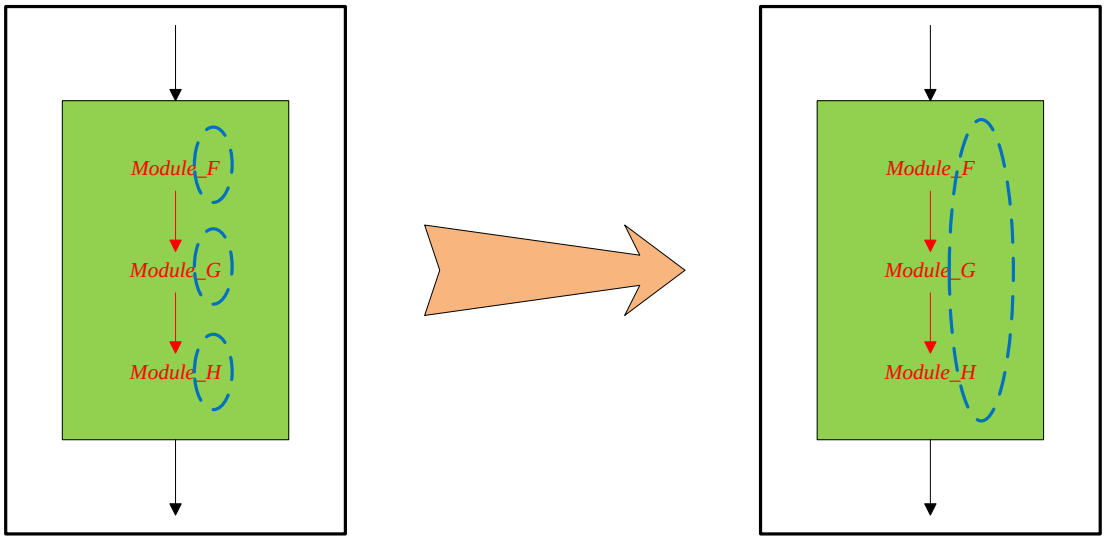

Figure 15: Multiple Big Loops (b)

When facing multiple big loops in sequential order, programmers can try to treat them as a whole (selecting the Pipeline Model) or merge them together (selecting the

Manager-Worker Model or the Civilian Model) to simplify the problem. These two approaches may not work each time, but once they are suitable for some situations, the number of multithreading times will be reduced.

Another special condition of multiple big loops is nested loops. One big loop may contain one or more big loops. An example is illustrated in Figure 16. In this example, Big Loop II is nested in Big Loop I. The best way to solve this problem is multithreading the nested big loop first. There are two reasons for making this choice. The first reason is that the nested big loop is probably taking over the most execution time of the external big loop. Once concurrency is introduced into the nested big loop, the execution time of the external big loop may be also greatly decreased and no further modification on the external big loop is necessary. The second reason is that the scope of the nested loop is smaller. The refactoring process of the nested loop with multiple threads may save more time and effort for the programmers.

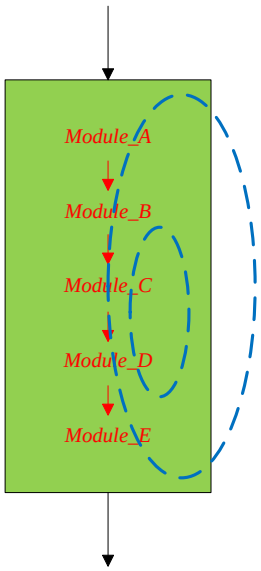

Figure 16: Multiple Big Loops (c)

6 Programming, Debugging and Testing

6.1 Multi-threaded Programming Techniques and Supports

Multithreading with the proposed models focuses on the structural design. The next step would be to modify the sequential program and introduce multiple threads at the code level. Programmers need to deal with code details very carefully. During the modification process, several programming techniques and support may be helpful.

6.1.1 Shared Data Protection Techniques

In concurrent programming, shared data protection is the most important problem that needs to be handled. In some cases, data access is very frequent and shared data is not allowed to be accessed by multiple threads simultaneously. If the requirement is not satisfied, the shared resource may be destroyed and the system may be running in a disorderly manner. The lock is usually used to protect shared data in concurrent program. As is stated in [55], "A lock is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution." Read and write locks are one kind of locks and are commonly used to protect shared data in concurrent programming. With the help of read and write locks, multiple readers can access data at the same time if there is no writer holding the shared data. Only one writer can access the shared data at a given time. Priority mechanisms may be added to solve the writer starvation problem [56].

The lock is very powerful to protect shared data. However, sometimes improper use of

the lock may cause deadlock. The deadlock is a situation where two or more threads are waiting for each other, thus the whole system is blocked. The subsequent operations cannot be performed. The necessary conditions of deadlock include mutual exclusion [57], resource holding, no preemption and the possibility for circular waiting chains [58]. Therefore, the programmer should try to prevent at least one of the conditions mentioned above to avoid deadlocks.

The monitor [59] is a good synchronization construct in concurrent programming. It uses the idea of encapsulating the shared data with their operations and making them into an abstract data type. The monitor is an instance of a class in which the methods should be executed with mutual exclusion. Multiple threads can access the variable and the method in the monitor safely because at any moment, at most one thread can execute a method of the monitor. The monitor also provides the possibility to execute *wait* and *signal* operations for multiple threads. By using the monitor construct, as provided by some languages such as Java and C#, there is no need to synchronize the shared data because the implementation of the monitor can only allow one access to the shared data at one time.

6.1.2 Concurrency Support from Programming Language and Operating System

The Java language is one of the current programming languages that have great support for concurrent programming. The synchronization of multiple threads can be properly coordinated by adding the key word "*synchronized*" before the methods that should be

executed mutual exclusively. The shared data can be accessed only by one thread at any time. In Java, all object instances have the potential to be used as a monitor. This is another way to control synchronization between multiple threads. The package *java.util.concurrent* provides many application programming interfaces for concurrent programming, such as concurrent queue, locks and semaphore. All the methods given by the package *java.util.concurrent* and its sub-packages offer the synchronization primitives. The C# language is similar to Java language and it also has very good support for concurrency.

One popular way to introduce concurrency into sequential programs is to make use of standard libraries. Dig et al. [76] presents the way they use *java.util.concurrent* package to transform sequential programs into concurrent programs. *int* is replaced with *AtomicInteger* to preserve single integer variable. The operations on the preserved variable are atomic so that multiple threads can access and update the variable safely without any lock protection. Similarly, *HashMap* is changed to *ConcurrentHashMap* to support concurrent access. Multiple threads are not required to be blocked when the modification on the map happens.

CONCURRENCER is the refactoring tool designed based on Dig's approach. Lea [77] introduces Java Fork/Join framework which is specific to divide-and-conquer algorithm in sequential programs. A task can be divided into subtasks that would be mapped to several threads for execution. There are other libraries supporting concurrent programming, such as TBB [78] developed by Intel for C++ and TPL [79] developed by Microsoft for C#. These libraries provide good data structure and frameworks for

concurrent programming and offer a possible way to introduce concurrency into sequential programs.

It is a common way for programmers to use concurrent libraries to transform sequential programs into concurrent programs. Researchers in [76, 77] present the way they use *java.util.concurrent* package to introduce concurrency. However, this refactoring approach is tedious, error-prone and omission-prone [92]. This is because in big and complex applications like the IBM prototype, a little change to data structure may cause large pieces of original program to be modified. What needs to change may expand to methods and classes, not just data types. It may be too complicated to be accomplished in the large and complex program. Compared to this approach, our methodology can make fewer changes to the original program. Most changes are concentrated in the target loop and the operations outside the loop will not be affected so much. This is a better way to refactor the code for large and complex applications.

Since concurrent programs are increasingly popular, many new programming languages are designed and created to meet specific requirements of concurrent programming. Ada [60] is a high-level object-oriented programming language. It has very powerful built-in support for concurrent programming, such as object protection, message passing, tasks offering and so on. Erlang [61] is a functional language designed for concurrent programming. The functions in Erlang only return a value and they don’t modify the accessed data, which means that critical section problems can be easily handled since multiple functions can safely access the same data simultaneously. Clojure [62] is another programming language that provides techniques to realize concurrency. Clojure supports

Software Transactional Memory (STM) [63]. STM is similar to database transactions. A transaction occurs when a thread touches data in the shared space. Other threads cannot read or write the data until the transaction is complete. Software developers don't need to use locks to prevent conflicts in Clojure. Many other languages such as Haskell [64], Smalltalk [65] also provide specific mechanisms to support concurrent programming.

In addition to the programming languages, the operating system also provides good supports for concurrent programming. The Windows operating systems support concurrency with the help of synchronization objects and critical section objects. "The mechanism used by the Windows executive to implement synchronization facilities is the family of synchronization objects" [66]. Object types, such as *Event, Mutex, Semaphore* and so on are designed for Windows systems to support concurrency. All the objects have two states, signaled and un-signaled. A thread would be blocked if the object is in the un-signaled state. The blocked thread would be released if the object goes into the signaled state. In synchronization objects, *Wait* functions are used to help a thread suspend its execution.

In Unix-like operating systems, such as GNU/Linux, Mac OS X and so on, POSIX Threads (Pthreads for short) are available for concurrent programming. "Pthreads defines a set of C programming language types, functions and constants, and it is implemented with a pthread.h header and a thread library" [67]. The thread library provides a lot of application programming interfaces for creating and manipulating threads. Thread pool, mutex, read and write locks and more synchronization control mechanisms can be easily implemented with Pthreads.

Usually, the target programming language and operating system can offer support for concurrency to programmers when they transform sequential programs to concurrent programs. It is better for programmers to learn and be familiar with these techniques and support before modifying the source code.

6.1.3 Refactoring Tools

In addition, several tools have been developed to introduce concurrent behaviors into the sequential program. The tools can be separated into two categories, fully automatic tools [87, 88] and interactive tools [76, 80, 89, 90, 91]. The difference between these two categories is that programmers are involved in the process in which interactive tools are used and make some decisions, while in the case of automatic tools no manual work is needed. However, all the tools can help programmers alleviate the burden when introducing concurrency into sequential programs.

Refactoring tools for transforming sequential programs into concurrent programs are usually good at retrofitting small and medium programs, they cannot perfectly introduce concurrency into big and complex applications, like commercial products. Our approach can help programmers overcome this point manually. Dig et al. [4] found that introducing concurrency into a product is not a one-time event. However, most refactoring tools cannot make new changes to the program after first-time modification. Our approach can help programmers revisit concurrency decisions and make further modifications by repeating some steps of the methodology. Most existing refactoring tools [76, 80, 89, 92,] are designed for specific language, like Java. However, our approach is not

language-specific. The idea in our approach can be applied to the software that is developed in any programming language.

6.1.4 Thread Local Storage

Local storage in threads [99] is a very useful concept to make static or global variables local to threads when transforming sequential programs into concurrent programs. Sometimes programmers may copy one static or global variable many times for multiple threads to use and keep them in a container, such as an array, a list, or a queue. Each thread has to use their thread ID to pick up its own variable and then use it in the following operation. In this case, the thread ID has to be set as a formal parameter of the function in which the variable is used.

In a large program, modifying one function signature may lead to a chain of changes on many other functions. This is because the complicated call relationships among different functions. Too many modifications may destroy the reusability of some modules. Therefore, sometimes it is advantageous to let each thread have the static or global variable that is only private to itself. Local thread storage can make a static or global variable declared as a local variable in the thread without affecting any other threads. No copy is needed and no modification is required on the function signature in most cases. The mechanism of local thread storage exists in many programming languages [100], such as C++, C#, Java, Python and so on. It is a powerful approach that can be used when transforming sequential programs into concurrent programs.

6.1.5 Introducing Concurrency Gradually by Using a Moving Lock

In addition to protect shared data, a lock can be used to control thread execution and introduce concurrency gradually when translating a sequential program into a concurrent one. Mostly, when a child thread is created and enters into a module, it will execute a series of methods that belong to the module one by one. A lock can be set at some point between these methods to control the execution of multiple threads. The consequence of adding a lock is that all the methods before the lock can be executed by multiple threads concurrently and all the methods after the lock have to be executed by only one thread at a time. In other words, the program before the lock has been modified into concurrency and the program after the lock is still sequential. By incrementally moving the lock from the first method to the last method of a specified module, the sequential programs can be transformed into concurrent programs step by step. Moving the lock can also be used in the testing process to shrink the scale of concurrency in the program under test.

6.2 Debugging and Testing

6.2.1 Debugging

In sequential programs, bugs can be located by setting different kinds of breakpoints and executing the program line by line. However, this approach doesn’t work smoothly in a concurrent program; the main problems are related to race conditions. Setting breakpoints or single-step tracking may seriously interfere with race conditions between multiple threads. One alternative approach is to write a log file [101]. Log file approaches should not include any complicated operation or any process which costs much time. Log needs

to contain the thread ID to facilitate the extraction of log information from the same thread for further analysis. Adding timestamps into the log is beneficial for analyzing the synchronized relationship between different threads. Debugging with logs can effectively reduce the side effects caused by setting breakpoints and single-step tracking in concurrent programming. The advantage of using logs in the debugging process is not only to pinpoint where the bugs happen, but also to achieve more detailed information about the reason [102].

Another approach for debugging concurrent programs is to use capture and replay tools [103] to record and playback the execution process of the program. In the record phase, the capture systems can dynamically gather useful information, such as the data values and the order of the code execution. In the playback phase, the recorded information is used to repeat the program execution traces and make those scenarios reappear. Capture and replay tools can help the programmer to catch and record a failure run for analysis. It may save much time and energy for the programmer. This is because between two times of the program crashed down, especially caused by the same bug, the programmer may need to run the program hundreds of thousands of times.

6.6.2 Testing

Testing concurrent programs is a notoriously difficult task, because of the program's non-deterministic nature. Unintentional situations, such as deadlocks and races, may not be detected by finite test cases. In order to make sure that a concurrent program has no bugs, all possible execution orders of multiple threads need to be covered. Nowadays,

concurrent programs testing techniques distinguish between design level and implementation level.

- **Design Level**

Model checking [104] is often used to demonstrate the correctness of a concurrent program at the design level. A concurrent program under test is usually abstracted into a model. The model is verified by a model checker in an exhaustive and automatic way. Model checkers usually use state space exploration techniques to traverse the model and check whether it satisfies a given specification [68]. The verification results may include unintentional conditions, such as race conditions, deadlocks, hash conflicts and so on. Programmers can find out unsafe code in the concurrent programs according to the verification results and modify the source code to make the concurrent programs safer and more reliable.

As mentioned in [69], “Spin is a popular open-source software verification tool, used by thousands of people worldwide. The tool can be used for the formal verification of multi-threaded software applications.” The Spin tool analyzes models which are programmed in Promela (Process or Protocol Meta Language) [70]. Promela is a modeling language that supports concurrency. The Spin tool can perform an exhaustive verification of the concurrent program model in order to realize concurrent programs testing. It reports “deadlocks, race conditions, unexecutable code or unspecified receptions” [71]. The Spin tool runs on different operating systems, such as Linux, Windows and Mac OS. In recent years, the Spin tool has been increasingly used to verify

the correctness of concurrent programs.

- **Implementation Level**

Two techniques are mostly used for concurrent programming testing at the implementation level. The first one is to execute conditional sleep statements when a shared resource is accessed or the synchronization condition is triggered. The basic idea of this technique is trying to cover all different execution orders for multiple threads by controlling the running speed of the threads. Whether the conditional sleep statement is executed or not may depend on the selection of "random or coverage-based testing" [72]. If random is selected, then the threads will decide to execute the sleep statement randomly by themselves. If coverage-based is selected, some methods will be marked as interruption available. The thread will execute the sleep statement only when it is in those methods that are marked.

Another technique for testing concurrent programs is called reachability testing. The basic idea of reachability testing is to cover all possible relevant synchronization sequences [73]. With a given input, an execution trace is recorded. A modified trace covers part of the previous trace and stops at some point. Then the modified trace starts to run from that point non-deterministically. The algorithm can be implemented by RichTest, a reachability testing tool. The RichTest has a record mode, which offers the previous trace information to make sure that each trace is executed exactly once.

7 Case Study

The proposed methodology for transforming sequential programs into concurrent programs was applied to a prototype of a commercial product. The objective was speeding up this prototype whose function is detecting and reporting vulnerabilities in applications. We successfully introduced concurrency into a part of this prototype and obtained some good results, thus demonstrating that the proposed methodology is useful.

7.1 The Prototype Program

As is mentioned in [74], “IBM Security AppScan Source for Analysis is a software application for analyzing code and providing specific information about source code vulnerabilities in critical systems.” The program offers a solution to detect potential vulnerabilities in order to protect private and important data, such as customer information. Scanning the source code of an application usually happens early in the development cycle to detect and address the vulnerabilities. The scanned source code could be part of, or the entire program of the application. The program supports applications programmed in different languages, such as Java, .Net, C/C++ and so on. Figure 17 shows an assessment summary report produced by the program. Vulnerability types, vulnerability levels and the locations of vulnerability findings in the source code can be presented by this prototype of the product.

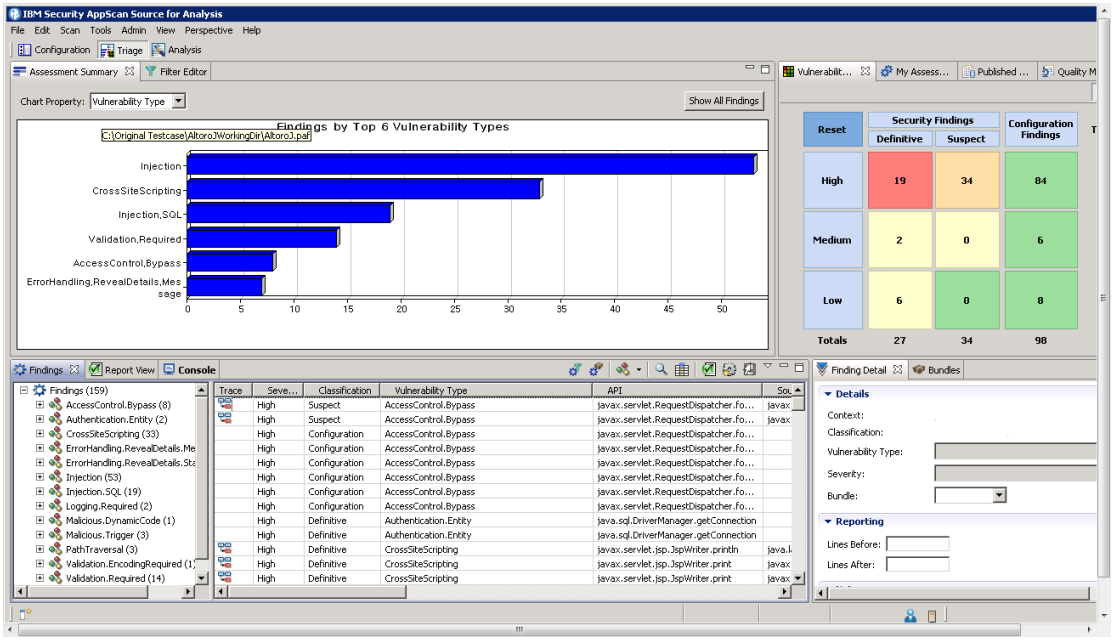

Figure 17: Assessment report produced by the prototype

In this project, we worked on a prototype of this program. We focused on the case that the scanned source code is programmed in Java and JSP. We identified a CPU intensive computation in the prototype and introduced concurrency in order to increase the processing speed.

7.2 Modification Procedure

- **Understanding the Code of the Prototype**

According to the proposed methodology, the first step is to understand the software application. This prototype uses static analysis technology to identify vulnerabilities of other software applications. It can achieve this goal by scanning the source code, performing taint analysis on data flows, reviewing call flows and detecting tracking, managing assessments and creating finding reports.

Figure 18 shows the high level design of the prototype. Accordingly, we identified four main components at the top level. The workflow can be concluded as follows: first, the source code is read by the Java Processor component (Component JP in Figure 18) and is translated into an Intermediate Representation (IR in Figure 18). The Static Single Assignment Transformation component (Component SSAT in Figure 18) produces IR with static single assignment variables [75] (SSA in Figure 18) and sends the result to the Vulnerability Analysis component (Component VA in Figure 18) for further processing. At the last step, the Report Generation component (Component RG in Figure 18) will compare vulnerability candidates that are generated from the analysis processes with the items in the vulnerability database, and then report final vulnerability findings.

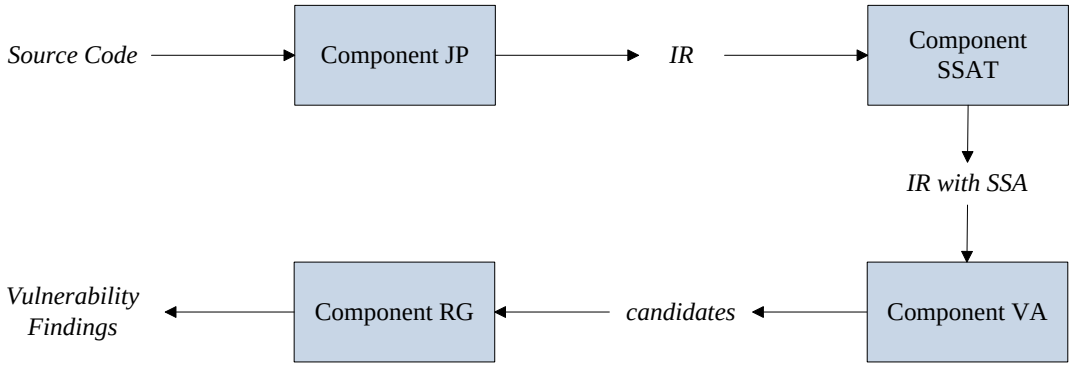

Figure 18: Architecture of the Prototype

Before concentrating on the source code, we first read the software documents that were used to describe the IBM prototype. The high level design and naming conventions of the program were reviewed, as well as the configuration of the application. After running the application in debug mode, we identified the functionality of each component and its

scope at the code level. By using the “as-needed” approach presented in the Program Comprehension Chapter, we skipped the Report Generation component since most operations of this component are dealing with result generation to the graphical user interface. We gave high priorities to other components and spent most time on comprehending the source code of those components. Control flow and data flow analysis were applied when exploring the details of components with higher priorities. UML diagrams were also used to identify the relationship between classes.

- **Program Decomposition**

The prototype that we studied is a complex C++ program of over 8 GB of source code. We first tried to separate the entire application into several components. By referring to the software documentation and based on our comprehension of the program, we found at the design level, it can be decomposed into 4 components as shown in Figure 18. All the source files it contains are separated into approximately 25 projects which are used to perform specific functions (here used in its non-programming sense). Each project can be treated as a module as mentioned in the methodology of Chapter 4. After we mapped components to modules, the whole program was broken down into 4 components which contain one or more modules that are related to the component in question.

Figure 19 shows the component that is related to the Java Processor after the decomposition process. The input of this component is the source code of the software application to be analyzed. This code is usually made up of a large number of source files. The original version of the IBM prototype scans source files one by one to detect

vulnerabilities of the application, which offers us a good sign to introduce concurrency. In this component, there are two modules, Java Loader and IR Conversion.

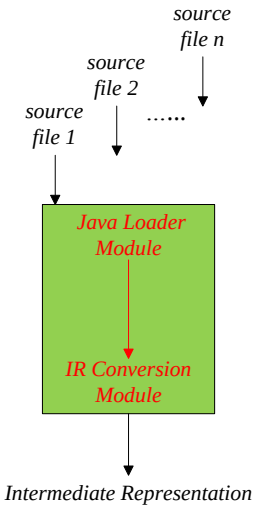

Figure 19: The Java Processor Component

After observation, we found there is a big loop that covers the operations from the Java Loader Module and the IR Conversion Module. It is shown in Figure 20. Each repetition of the big loop is loading and converting one source file, so they are independent of each other. The big loop takes up almost 30% of the total processing time of the prototype, which implies that intensive computation is included in this component. By running the code of the loop in debug mode, we also found that no operations jump into or out of the loop. The data passed to the method does not include events. Therefore, this big loop satisfies the four rules proposed in the Big Loop Selection Section, so it is selected as the candidate for introducing concurrency.

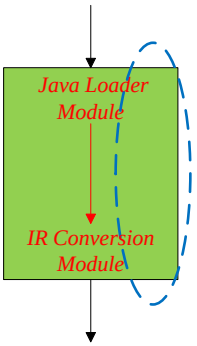

Figure 20: The Java Processor Component with the Big Loop

- **Selection of the Programming Model**

The function of the Java Processor component is translating Java files into an Intermediate Representation. This process can be separated into two phases. In phase I, the Java Loader module loads Java code into memory. In phase II, the IR Conversion module converts the result that was produced by the Java Loader module into the Intermediate Representation.

According to the proposed methodology, we can identify the loop covering the Java Processor component is a single big loop that covers the operations from two modules, so there are two approaches that can be chosen for this big loop. The first approach is using the Pipeline Model. As Table 1 shows, the IR Conversion module spends almost twice as much time as the Java Loader module. The processing time of two modules are not balanced. In order to gain the best performance with the Pipeline Model, we can try to break down the IR Conversion module in a further step or add more threads to the IR Conversion module.

Table 1: Percentage of total time in the Java Processor component

Module	Time Percentage
Java Loader module	35%
IR Conversion module	65%

We also can select the Manager-Worker Model or the Civilian Model as the second solution. We can create multiple threads (worker threads or civilian threads) before they implement the code of the Java Processor component. Each thread takes the source file and goes through the Java Loader module and the IR Conversion module to finish the processing. In this project, we have chosen the Manager-Worker Model. Compared to the Pipeline model, it is not necessary to spend extra time on the load balancing problem when selecting the Manager-Worker Model.

- **Multithreading the Java Processor Component**

The Manager-Worker Model without pool and queue is selected for the Java Processor component. First, the maximum number of worker threads is determined according to the number of CPU cores. This can guarantee that the CPU resource is reasonably utilized. Then the manager thread can create worker threads and assign source files to each worker thread at run time. When the manager thread detects that the number of worker threads is currently smaller than the maximum number and there are still source files that are not assigned, it will create a new worker thread and assign a task to it. When there are no source files left, the manager thread will wait for all worker threads to finish their processing.

For multiple worker threads, the Intermediate Representation is stored in a shared space.

Each thread will translate the java code into the form of the Intermediate Representation and add it into this shared space. Conflicts may happen when multiple threads access data in the IR Conversion module. In most cases, multiple threads will first read the shared list to check if the data has already been processed. If not, multiple threads will perform a series of operations on the data and then write the name of this data into the shared list to indicate the data has been processed. Read and write locks are very suitable to protect shared data and avoid conflicts in this situation. The mechanism of one writer and multiple readers can maximize the parallelism.

We also use the moving lock technique to introduce concurrency into the program step by step. Usually, the worker threads will execute some methods in one source file and then jump into another source file. We use the lock to separate the methods of these source files of the IBM prototype. Concurrency is first introduced into the methods that belong to the source file before the lock, and then we move the lock to separate the methods of the next two source files of the prototype. A big piece of sequential program is transformed into a concurrent program gradually in this way.

The technique of local thread storage is also used to simplify the modification process. Some static and global variables are made local to each thread. Multiple threads can access these variables without using thread ID such that not so much modifications are required on the source code. Local thread storage also decreases the number of read and write locks being used, which reduce the risk of deadlock and resource starvation.

The IBM prototype runs on the Windows platform, so we also use some concurrency

support designed for Windows operating system, such as *Mutex*, *Semaphore* and *Wait* functions. All this support makes the modified code being modified more concise and readable.

- **Multithreading the Static Single Assignment Transformation Component**

After the program of the Java Processor component was successfully modified with multiple threads, another component, Static Single Assignment Transformation (simply called SSAT in the following section), was translated into a concurrent program following the instructions of the proposed methodology. The program decomposition with the big loop selection is illustrated in Figure 21. There are also two modules in the Static Single Assignment component, the DFA module and the TIE module. A big loop contains the operations of these two modules. After calculation we know that the processing time of the DFA module is almost equal to the processing time of the TIE module, so we select both the Manager-Worker module and the Pipeline Module to refactor this big loop and then compare the results (see Section 7.4).

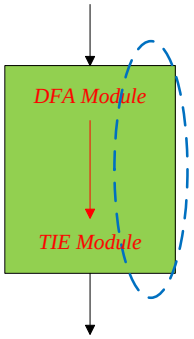

Figure 21: The SSAT Component with the Big Loop

- **Debugging and Testing**

We debugged the modified concurrent program by generating log files instead of setting breakpoints in most cases. The log file, records the thread index, timestamp and brief explanation of the abnormal conditions. We tracked down the error traces with the help of the call stack and the information presented in the log files. We also used capture and replay tools to observe the abnormal scenarios again and again during debugging.

We tested the correctness of the concurrent program by inserting conditional sleep statements into the program to let the multiple threads run in different execution orders. Every time we compared the vulnerability analysis output of the concurrent version with the original IBM prototype to make sure that the transformation from the sequential behavior to the concurrent behavior does not affect the accuracy of the application.

7.3 Experiment Environment

We ran all our experiments on a virtual machine. The configuration information of the virtual machine is shown in Table 2.

Table 2: IBM virtual machine configuration

Component	Configuration
Windows Edition	Windows Server ® Enterprise Service Pack 2
System Type	32-bit Operating System
Processor	Intel (R) Xeon (R) E7-2850 2.00GHz (4 cores)
Memory(RAM)	4.00 GB
Hard Drive	100.00 GB

The virtual machine has 4 CPU cores, which means that usually the optimal result should

be achieved when 4 threads are running concurrently.

7.4 Performance Evaluation

The criterion for test case selection is the number of source files that the software application to be analyzed contains. Two test cases, AltoroJ and WebGoat, were used to verify the performance of the transformed program. The basic information of these test cases is provided in Table 3.

Table 3: Test cases information

	Number of Source Files	Number of Lines	Total Vulnerability Findings
AltoroJ	86	5000	159
WebGoat	402	76314	2044

7.4.1 Processing Time

- **The Big Loop that covers the Java Processor component**

When running the test case AltoroJ, the processing time of the big loop that covers the Java Processor component is provided in Figure 22. Time saving percentage with multiple threads is presented in Table 4. The maximum number of worker threads is 4. “0” means the original sequential program. Compared to the original sequential program, using 4 threads almost saves 68.6% of the total time for the big loop.

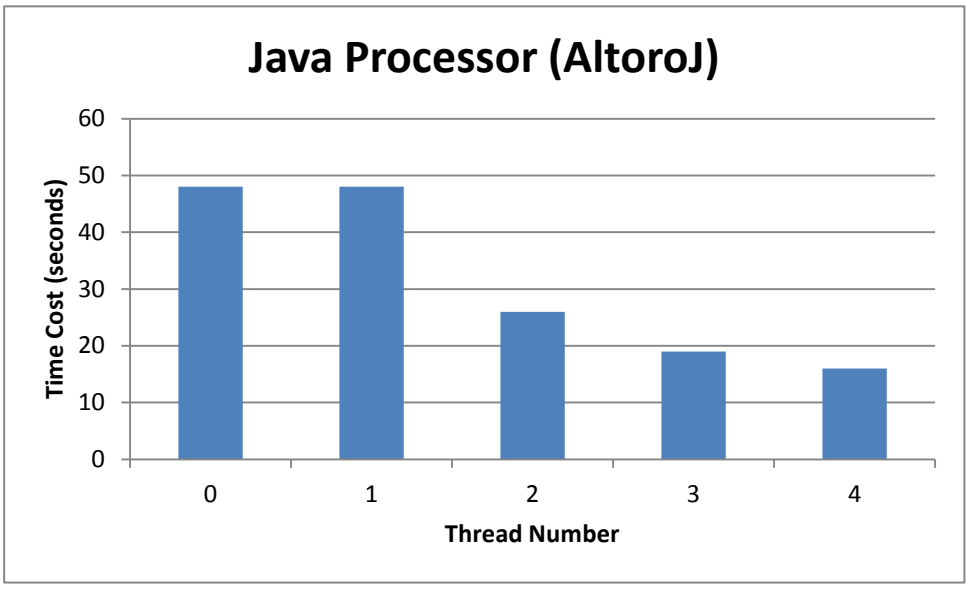

Figure 22: Processing time of the big loop for AltoroJ

Table 4: Time saving percentage for the big loop (AltoroJ)

Working Threads Number	Processing Time (seconds)	Time Saving Percentage	Ideal Time Saving Percentage
0 (Original Program)	48		
1	48	0%	0%
2	26	45.8%	50%
3	19	60.4%	67%
4	15	68.6%	75%

When running the test case WebGoat, the processing time of the big loop that covers the Java Processor component is provided in Figure 23. The time saving percentage is shown in Table 5. Using 4 threads almost saves 71.5% of the total time in comparison with the original sequential program.

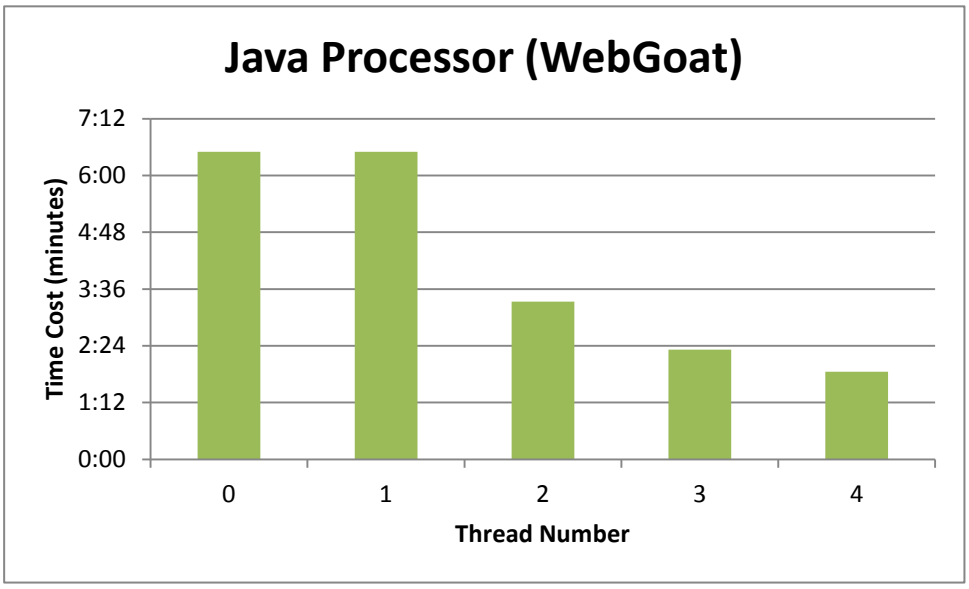

Figure 23: Processing time of the big loop for WebGoat

Table 5: Time saving percentage for the big loop

Working Threads Number	Time Cost (minutes)	Time Saving Percent	Ideal Time Saving Percentage
0 (Original Program)	6:30		
1	6:30	0%	0%
2	3:20	48.7%	50%
3	2:19	64.4%	67%
4	1:51	71.5%	75%

- **The Big Loop that covers the SSAT component**

Table 6 and Table 7 provide the experiment results after introducing concurrency into the big loop that covers the SSAT component for test case AltoroJ and WebGoat, respectively. In order to compare the results, the Pipeline Model and the Manager-Worker Model are both implemented with two working threads instead of four threads. This is because there are only two modules (DFA module and TIE module) inside the Big Loop and it would be

easy to implement the Pipeline Model with two threads, each thread performs the operations of each module. The performance comparison between these two models is for the SSAT component alone, not including the Java Processor component. Since the processing time of the two modules inside the SSAT component are almost equal, there is no problem with load balancing that affects the performance obtained in the Pipeline Model. However, if the executing time of the two stages differs largely, the performance will depend on the optimization of the implementation of the slower stage.

Table 6: Results comparison between two models (AltoroJ)

Working Threads Number	The Pipeline Model (seconds)	The Manager-Worker Model (seconds)
0 (Original Program)	5.94	5.94
2	3.98	3.91

Table 7: Results comparison between two models (WebGoat)

Working Threads Number	The Pipeline Model (seconds)	The Manager-Worker Model (seconds)
0 (Original Program)	68	68
2	45	43

7.4.2 CPU Usage

With increasing thread numbers, the CPU usage is also improved. Table 8 provides the CPU usage for multiple threads. The original sequential program can only use 25% to 35% of the CPU resource. However, after introducing concurrency into the program, the CPU usage can reach almost 100% with 4 threads.

Table 8: CPU Usage for Multiple Threads

Working Threads Number	CPU Usage
0 (Original Program)	25% - 35%
1	25% - 35%
2	50% - 60%
3	75% - 80%
4	97% - 100%

8 Conclusion and Future Work

Concurrent programs are becoming more and more popular with the development of computer hardware. Concurrent programs can achieve higher performance than sequential programs on platforms of multi-processor systems and multi-core processors. For a software application that is developed as a sequential program and is currently being used by clients, introducing concurrency with multiple threads into the source code would be a better choice than designing and developing a concurrent program from the very beginning. Some traditional concurrent programming methodologies and techniques are aimed at creating the software application from scratch. Some are suitable for small programs, but not for large complex software applications. Some have limitations concerning the programming language. Therefore, a general and systematic methodology for transforming sequential programs into concurrent programs for a large complex application is needed.

Dig et al. [4] propose a general pattern for introducing concurrency into existing software applications in three steps: first make the new program runnable, then make it fast, finally make it scalable. This thesis achieves the research goal by proposing a methodology for introducing concurrency into large and complex software applications. It mostly focuses on the first and second steps in Dig's pattern: make the programmer find the opportunities to introduce concurrency in an easy and fast way and then make the program runnable and faster than before. Our methodology is general and can be widely used in different kinds of situations. It includes approaches for program comprehension, program

decomposition, multithreading models, concurrent debugging and testing skills. We have applied this methodology to the IBM prototype and found that the concurrent program obtained by following the proposed methodology performs much better than the original sequential program, thus demonstrating that our methodology is effective and useful.

Our proposed methodology is focusing on introducing concurrency into sequential programs with multiple threads. Compared to the existing approaches, it is more complete and systematic. It does not only include approaches to introducing concurrency, but also contains other necessary refactoring processes, such as program comprehension, program decomposition, debugging and testing concurrent programs. To the best of our knowledge, there is no other methodology that addresses all these software activities on the topic of transforming large and complex sequential programs into equivalent concurrent programs. Most other work only concentrates on one particular step.

8.1 Summary of contributions

The main contributions of this thesis are the following:

1. **Methodology for transforming a sequential program into a concurrent program:** We introduce a general methodology for introducing concurrency into a large sequential program. This new methodology provides the workflow of the transformation process. It does not only include new approaches for program comprehension and decomposition, but also integrates well-known techniques and common sense approaches in concurrent programming. Our proposed methodology is suitable for big and complicated programs. By performing our methodology, the required time that

programmers spend on the modification process may have the chance to be greatly reduced.

2. Case study involving a prototype of an IBM product: Following the proposed methodology, a prototype of IBM Security AppScan Source for Analysis has been partially modified into concurrent code, leading to a better performance compared with the original sequential code.

8.2 Future work

We have presented the proposed methodology with some good experimental results; however, there are still some aspects to be explored in the area of introducing concurrency into sequential programs. We briefly describe few research directions in the following.

The proposed methodology is very general and it can be applied to different kinds of software applications, no matter what programming language the application uses, or what operating system the application is running on. In this perspective, the methodology can be improved by exploring new techniques that can be applied in specific development environments, for example, for applications developed in C within the Linux operating system. Once the sequential program satisfies some specific conditions, the new techniques may speed up the program transformation process. Therefore, exploring new techniques for software applications in different development environments would be a future direction.

We have applied the methodology to only one big and complex application, so another future direction for research would be the evaluation of the proposed methodology with more projects. In order to evaluate its effectiveness, concurrency could be introduced, into different software applications of different types. The applications could be selected based on criteria of programming languages, operating platform and so on. Moreover, further research can focus on efficiency analysis. Most existing approaches are used in refactoring tools, so we can compare the experimental results generated by the tools and ours under the same conditions.

Reference

[1] Silberschatz, Abraham, et al. Operating system concepts. Vol. 4. Reading: Addison-Wesley, 1998.

[2] “Thread”, [online]. Available: http://en.wikipedia.org/wiki/Thread_(computing). [Accessed on 2014 04 15].

[3] Tanenbaum, Andrew S., and Andrew Tannenbaum. Modern operating systems. Vol. 2. Englewood Cliffs: Prentice hall, 1992.

[4] D. Dig, J. Marrero, and M. D. Ernst. How do programs become more concurrent? A story of program transformations. Technical Report MIT-CSAIL-TR-2008-053, MIT, September 2008.

[5] “IBM Security AppScan Source”, [online]. Available: http://www-03.ibm.com/software/products/en/appscan-source/. [Accessed on 2014 04 15].

[6] Raynal, Michel. Concurrent programming: algorithms, principles, and foundations. Springer-Verlag, 2013.

[7] Unger, Stephen H. “Hazards, critical races, and metastability” IEEE Transactions on Computers, 44.6 (1995): 754-768.

[8] “Resource starvation”, [online]. Available: http://msdn.microsoft.com/en-us/library/ee798408(v=cs.20).aspx. [Accessed on 2014 04 15].

[9] Kaveh, Nima, and Wolfgang Emmerich. “Deadlock detection in distribution object systems” ACM SIGSOFT Software Engineering Notes. Vol. 26. No. 5. ACM, 2001.

[10] Andrews, Gregory R., and Fred B. Schneider. “Concepts and notations for concurrent programming” ACM Computing Surveys (CSUR) 15.1 (1983): 3-43.

[11] Hughes, Cameron, and Tracey Hughes. Parallel and distributed programming using C++. Addison-Wesley Professional, 2004.

[12] “EightyTwentyRule”, [online]. Available: http://c2.com/cgi/wiki?EightyTwentyRule. [Accessed on 2014 04 15].

[13] Brooks, R., “Towards a Theory of the Cognitive Processes in Computer Programming”, Int. J. Man- Machine Studies, Vol.9, 1977, 737-751

[14] E. Soloway and K. Ehrlich, “Empirical studies of programming knowledge” IEEE

Transactions on Software Engineering, pp. 595-609, SE-10(5), September 1984.

[15] Letovsky, S. and Soloway, E., "Delocalized Plans and Program Comprehension", IEEE Software, vol. 3, No. 3, May 1986, 41 - 49.

[16] B. Shneiderman and R. Mayer, "Syntactic/semantic interactions in programmer behavior: A model and experimental results", International Journal of Computer and Information Sciences, pp. 219-238, 8(3), 1979.

[17] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs" Cognitive Psychology, pp. 295-341, vol 19, 1987.

[18] von Mayrhauser, A., A. Vans, "From Program Comprehension to Tool Requirements for an Industrial Environment", Proc. 2nd Workshop on Program Comprehension, July, 1993, 78-86

[19] Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc, Jr. "Recognizing Design Decisions in Programs" IEEE Software, 7(1):46-54, January, 1990.

[20] Storey, Margaret-Anne. "Theories, methods and tools in program comprehension: Past, present and future" Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on. IEEE, 2005.

[21] Littman, David C., et al. "Mental models and software maintenance", Journal of Systems and Software 7.4 (1987): 341-355.

[22] Fleming, Scott D., et al. "Refining existing theories of program comprehension during maintenance for concurrent software" Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on. IEEE, 2008.

[23] Lakhotia, Arun. "Understanding someone else's code: analysis of experiences", Journal of Systems and Software 23.3 (1993): 269-275.

[24] Rugaber, Spencer. "The use of domain knowledge in program understanding", Annals of Software Engineering 9.1-2 (2000): 143-192.

[25] Petrenko, Maksym, Václav Rajlich, and Radu Vanciu. "Partial domain comprehension in software evolution and maintenance" Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on. IEEE, 2008.

[26] Wilson, Leon A. "Using ontology fragments in concept location", Software Maintenance (ICSM), 2010 IEEE International Conference on. IEEE, 2010.

[27] Rajlich, Václav. "Intensions are a key to program comprehension", ICPC. 2009.

[28] Marcus, Andrian, et al. “Static techniques for concept location in object-oriented code”, Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on. IEEE, 2005.

[29] Murphy, Gail C., David Notkin, and Kevin Sullivan. “Software reflexion models: Bridging the gap between source and high-level models”, ACM SIGSOFT Software Engineering Notes. Vol. 20. No. 4. ACM, 1995.

[30] Erdos, Katalin, and Harry M. Sneed. “Partial comprehension of complex programs (enough to perform maintenance)” Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on. IEEE, 1998.

[31] Lanza, Michele, and Stéphane Ducasse. “A categorization of classes based on the visualization of their internal structure: the class blueprint”, ACM SIGPLAN Notices36.11 (2001): 300-311.

[32] Marcus, Andrian, Louis Feng, and Jonathan I. Maletic. “Comprehension of software analysis data using 3D visualization” Program Comprehension, 2003. 11th IEEE International Workshop on. IEEE, 2003.

[33] Nelson, Michael L. “A survey of reverse engineering and program comprehension”, arXiv preprint cs/0503068 (2005).

[34] “Hungarian notation”, [online]. Available: http://msdn.microsoft.com/en-us/library/aa260976(VS.60).aspx. [Accessed on 2014 04 15].

[35] “UML”, [online]. Available: http://www.uml.org/. [Accessed on 2014 04 15].

[36] Erdos, Katalin, and Harry M. Sneed. “Partial comprehension of complex programs (enough to perform maintenance)” Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on. IEEE, 1998.

[37] Kim, Tae-Hoon, and Yeong Gil Shin. “Role-based decomposition for improving concurrency in distributed object-oriented software development environments”, Computer Software and Applications Conference, 1999. COMPSAC'99. Proceedings. The Twenty-Third Annual International. IEEE, 1999.

[38] Parnas, David Lorge. “On the criteria to be used in decomposing systems into modules”, Communications of the ACM 15.12 (1972): 1053-1058.

[39] Welsh, Matt, Gribble, Steven D., Eric A. Brewer, and David Culler. A design framework for highly concurrent systems. Berkeley: Computer Science Division, University of California, 2000.

[40] Mancoridis, Spiros, et al. “Using automatic clustering to produce high-level system organizations of source code”, International Conference on Program Comprehension. IEEE Computer Society, 1998.

[41] Wiggerts, Theo A. “Using clustering algorithms in legacy systems remodularization”, Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on. IEEE, 1997.

[42] Müller, Hausi A., Mehmet Ali Orgun, and James Scott Uhl. Discovering and reconstructing subsystem structures through reverse engineering. University of Victoria, Department of Computer Science, 1992.

[43] Schwanke, Robert W. “An intelligent tool for re-engineering software modularity”, Software Engineering, 1991. Proceedings., 13th International Conference on. IEEE, 1991.

[44] “Loop Modifications to Enhance Data-Parallel Performance”, [online]. Available: https://software.intel.com/en-us/articles/loop-modifications-to-enhance-data-parallel-performance. [Accessed on 2014 04 15].

[45] Hughes, C., and T. Hughes. Parallel and Distributed Programming in C++. 2004.

[46] “SIMD”, [online]. Available: http://en.wikipedia.org/wiki/SIMD. [Accessed on 2014 04 15].

[47] “MISD”, [online]. Available: http://en.wikipedia.org/wiki/MIMD. [Accessed on 2014 04 15].

[48] Nichols, Bradford, Dick Buttlar, and Jacqueline Proulx Farrell. Pthreads Programming: A POXIS Standard for Better Multiprocessing. O'reilly, 1996.

[49] Navarro, Angeles, et al. “Analytical modeling of pipeline parallelism”, Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on. IEEE, 2009.

[50] Grama, Ananth, ed. Introduction to parallel computing. Pearson Education, 2003.

[51] Shanthi, Mrs M., and A. Anthony Irudhayaraj. “Multithreading-An Efficient Technique for Enhancing Application Performance”, International Journal of Recent Trends in Engineering 2.4 (2009): 165-167.

[52] Hughes, Cameron. Professional Multicore Programming Design and Implementation for C++ Developers. John Wiley & Sons, 2008.

[53] Kamruzzaman, Md, Steven Swanson, and Dean M. Tullsen. “Load-balanced pipeline parallelism”, Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2013.

[54] Blumofe, Robert D., and Charles E. Leiserson. “Scheduling multithreaded computations by work stealing”, Journal of the ACM (JACM) 46.5 (1999): 720-748.

[55] “Lock”, [online]. Available: http://en.wikipedia.org/wiki/Lock_(computer_science). [Accessed on 2014 04 15].

[56] “Writer starvation”, [online]. Available: http://en.wikipedia.org/wiki/Readers%E2%80%93writer_lock. [Accessed on 2014 04 15].

[57] Raynal, Michel. Algorithms for mutual exclusion. The MIT Press, 1986.

[58] “Deadlock”, [online]. Available: http://en.wikipedia.org/wiki/Deadlock. [Accessed on 2014 04 15].

[59] Hoare, Charles Antony Richard. “Monitors: An operating system structuring concept”, Communications of the ACM 17.10 (1974): 549-557.

[60] Watt, David A., Brian A. Wichmann, and William Findlay. Ada language and methodology. Prentice Hall International (UK) Ltd., 1987.

[61] Cesarini, Francesco, and Simon Thompson. Erlang programming. O'Reilly Media, Inc., 2009.

[62] Emerick, Chas, Brian Carper, and Christophe Grand. Clojure Programming. O'Reilly Media, Inc., 2012.

[63] Shavit, Nir, and Dan Touitou. “Software transactional memory”, Distributed Computing 10.2 (1997): 99-116.

[64] “Haskell”, [online]. Available: http://www.haskell.org/haskellwiki/Introduction. [Accessed on 2014 04 15].

[65] Yonezawa, Akinori. Object-oriented concurrent programming. The MIT Press, 1987.

[66] William Stallings. Operating Systems: Internals and Design Principles, Fifth Edition. Prentice Hall, 2005.

[67] “Pthreads”, [online]. Available: http://en.wikipedia.org/wiki/POSIX_Threads. [Accessed on 2014 04 15].

[68] Holzmann, Gerard J. The SPIN model checker: Primer and reference manual. Vol. 1003. Reading: Addison-Wesley, 2004.

[69] “Spin”, [online]. Available: http://spinroot.com/. [Accessed on 2014 04 15].

[70] Sharma, Asankhaya. “A Refinement Calculus for Promela”, Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on. IEEE, 2013.

[71] Peng, Hong, Sofiene Tahar, and Ferhat Khendek. “Comparison of SPIN and VIS for protocol verification”, International Journal on Software Tools for Technology Transfer 4.2 (2003): 234-245.

[72] Edelstein, Orit, et al. “Multithreaded Java program test generation”, IBM systems journal 41.1 (2002): 111-125.

[73] Hwang, Gwan-Hwan, Kuo-Chung Tai, and Ting-Lu Huang. “Reachability testing: An approach to testing concurrent software”, International Journal of Software Engineering and Knowledge Engineering 5.04 (1995): 493-510.

[74] “IBM Security AppScan Source for Analysis”, [online]. Available: http://pic.dhe.ibm.com/infocenter/appsrc/v8r7m0/index.jsp?topic=%2Fcom.ibm.rational.appscansrc.security.doc%2Ftopics%2Fintro_overview.html. [Accessed on 2014 04 15].

[75] Cytron, Ron, et al. “An efficient method of computing static single assignment form”, Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1989.

[76] D. Dig, J. Marrero, and M.D. Ernst, “Refactoring Sequential Java Code for Concurrency via Concurrent Libraries”, Proc. 31st Int’l Conf. Software Eng. (ICSE), IEEE Press, 2009, pp. 397–407.

[77] Lea, Doug. “A Java fork/join framework” Proceedings of the ACM 2000 conference on Java Grande. ACM, 2000.

[78] “TBB”, [online]. Available: http://threadingbuildingblocks.org/. [Accessed on 2014 04 15].

[79] “TPL”, [online]. Available: http://msdn.microsoft.com/en-us /library/ dd460717 (v=vs.110).aspx. [Accessed on 2014 04 15].

[80] J. Wloka, M. Sridharan, and F. Tip, “Refactoring for Reentrancy”, Proc. 7th Joint Meeting European Soft. Eng Conf. and the Int’l. Symp. Foundations Software Eng. (ESEC/FSE), ACM Press, 2009, pp. 173–182.

[81] Everaars, C. T. H., Farhad Arbab, and Barry Koren. “Using coordination to restructure sequential source code into a concurrent program” Software Maintenance, 2001. Proceedings. IEEE International Conference on. IEEE, 2001.

[82] Briot, Jean-Pierre. “Object-oriented concurrent programming: Introducing a new programming methodology”, Proceedings of the 7th International Meeting of Young Computer Scientists, 1992.

[83] Matsuoka, Satoshi, and Akinori Yonezawa. “Analysis of inheritance anomaly in object-oriented concurrent programming languages”, Research directions in concurrent object-oriented programming 3 (1993): 107-150.

[84] “data parallelism”, [online]. Available: http://msdn.microsoft.com/en-us/library/dd537608(v=vs.110).aspx. [Accessed on 2014 04 15].

[85] “task parallelism”, [online]. Available: http://msdn.microsoft.com/en-ca/library/vstudio/dd537609(v=vs.100).aspx. [Accessed on 2014 04 15].

[86] Tefft, Robert A., and Roger Y. Lee. “Reduction of complexity and automation of parallel execution through loop level parallelism” Quality Software, 2007. QSIC'07. Seventh International Conference on. IEEE, 2007.

[87] R. Allen, D. Callahan, and K. Kennedy, “Automatic Decomposition of Scientific Programs for Parallel Execution”, Proc. 14th ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages, ACM Press, 1987, pp. 63–76.

[88] S.P. Amarasinghe et al., “An Overview of a Compiler for Scalable Parallel Machines” Proc. 6th Int’l Workshop Languages and Compilers for Parallel Computing, Springer, 1993, pp. 253–272.

[89] Kjolstad, Fredrik Berg, et al. “Refactoring for immutability” (2010).

[90] D. Dig et al., ReLooper: Refactoring for Loop Parallelism, tech. report, Dept. Computer Science, Univ. of Illinois at Urbana-Champaign, Sept. 2009.

[91]. R. Fuhrer and V. Saraswat, “Concurrency Refactoring for x10”, Proc. 3rd ACM Workshop Refactoring Tools, ACM Press, 2009.

[92] Dig, Danny, John Marrero, and Michael D. Ernst. “Concurrencer: a tool for retrofitting concurrency into sequential Java applications via concurrent libraries”, Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on. IEEE, 2009.

[93] Parnas, David Lorge. “Information distribution aspects of design methodology”,

Tech. Rept., Depart. Computer Science, Carnegie Mellon U., Pittsburgh, Pa., 1971.

[94] Hofmann, Marko A. “Criteria for decomposing systems into components in modeling and simulation: Lessons learned with military simulations” Simulation80.7-8 (2004): 357-365.

[95] “naming conventions”, [online]. Available: http://en.wikipedia.org/wiki/Naming_ convention_(programming). [Accessed on 2014 04 15].

[96] Meyer, Bertrand. "Concurrency Made Easy." (2011).

[97] Tai, K-C. “Testing of concurrent software.” Computer Software and Applications Conference, 1989. COMPSAC 89., Proceedings of the 13th Annual International. IEEE, 1989.

[98] Wisnesky, Ryan J. “The Inheritance Anomaly Revisited.” (2006).

[99] “thread-local storage, [online]. Available: http://en.wikipedia.org/wiki/Thread-loc al_storage. [Accessed on 2014 04 15].

[100] Carribault, Patrick, Marc Pérache, and Hervé Jourdren. “Thread-local storage extension to support thread-based MPI/openMP applications.” OpenMP in the Petascale Era. Springer Berlin Heidelberg, 2011. 80-93.

[101] Andrews, James H. “Testing using log file analysis: tools, methods, and issues.” Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on. IEEE, 1998.

[102] Andrews, James H., and Yingjun Zhang. “General test result checking with log file analysis.” Software Engineering, IEEE Transactions on 29.7 (2003): 634-648.

[103] Steven, John, et al. jRapture: A capture/replay tool for observation-based testing. Vol. 25. No. 5. ACM, 2000.

[104] Clarke, Edmund M., Orna Grumberg, and Doron Peled. Model checking. MIT press, 1999.