**Département de génie logiciel et des TI**

**Progress Report**

**LOG792 Projet de fin d'études en génie logiciel**

**Speculative Multithreading System**

**A generic software-only implementation**

**Author**

Rémi Attab  
ATTR05098501

**Supervising professor**

Eric Paquette

**Date**

**22 Juin 2011**

**Changes Tracking**

\***A** – Added **M** – Modified **D** – Deleted

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **VERSION NUMBER** | **DATE**  **yyyy/mm/dd** | **FIGURE, TABLE OR SECTION ID** | **A\* M D** | **SHORT DESCRIPTION OF CHANGES** | **CHANGE REQUEST NUMBER** |
|  |  |  |  |  |  |
| 1.0 | 2011-06-22 |  | A | Initial draft | n/a |
|  |  |  |  |  |  |

**Table of content**

[1. Context and Problem Definition 4](#_Toc296537370)

[1.1 Manual Parallelization 4](#_Toc296537371)

[1.2 Automatic Parallelization 4](#_Toc296537372)

[1.3 Speculative Multithreading 4](#_Toc296537373)

[2. Goals of the Project 5](#_Toc296537374)

[3. Methodology 6](#_Toc296537375)

[4. Progress Report and Recomendations 7](#_Toc296537376)

[4.1 Progress Report 7](#_Toc296537377)

[4.2 Recommendations 8](#_Toc296537378)

[5. Deliverables and Planification 8](#_Toc296537379)

[5.1 Artefacts description 8](#_Toc296537380)

[5.2 Planning 9](#_Toc296537381)

[6. Risks 10](#_Toc296537382)

[7. References 11](#_Toc296537383)

[8. Table of Content for Final Report 12](#_Toc296537384)

[Annexe A: Revised Work Plan 13](#_Toc296537385)

# Context and Problem Definition

With the recent widespread availability of multi-core processors, there has been an increasing demand for software that can make full use of the hardware that it’s running on. To meet the demand, multiple tools have been created to help programmers design and implement parallel code. Unfortunately, there is a large number of existing programs that were not designed to work in a parallel environement. These programs are generally difficult to parallelize due to their size and complexity.

The following sections briefly describe the current state of the techniques and tools used to parallelize a program and the various problems associated with them. Since the project proposition, the section 1.3 was modified to include a discussion of the two major approaches that can be used to implement a speculative multithreading system.

## Manual Parallelization

The first method is to do the parallelisation by hand. This involves the following steps:

1. Finding a section of code that is inherently parallelizable and involves a bottleneck.
2. Split the section of code into separate tasks that can be executed in parallel.
3. Use the primitives provided by the current platform to run the tasks in parallel.
4. Synchronize the communications between the different threads running these tasks.

Each of these steps can be difficult to implement correctly. For example, a bottleneck that is inherently not parallelizable will usually contain a lot of dependencies in its computation. In order to parallelize this bottleneck and keep its computation coherent, a large amount of synchronization would need to be introduced. This would negate most, if not all, of the benefits of the parallelization. Since it's not always obvious if the bottleneck is inherently parallelizable, a programmer could waste a lot of his time. The extra complexity could also introduce defects in the programs which can be difficult to detect and correct. Tools like *OpenMP[[1]](#footnote-1)* have been created to alleviate some of these problems. Nevertheless, they still require a considerable amount of human intervention because they can't solve all the steps involved in parallelizing a program.

## Automatic Parallelization

Because of all the difficulties associated with manual parallelization, another type of tool has been developed with the aim of automatically finding and parallelizing bottlenecks. These are usually part of a compiler and are implemented as optimization passes [[1](#01)]. During the compilation process most of the developer's intent is lost which forces the optimisation process to be conservative in nature to keep the program coherent. By being conservative, the process could miss some good parallelization opportunities.

## Speculative Multithreading

To solve the problems associated with a purely compiler based parallelisation scheme, another type of tool called speculative multithreading has been developed. It uses a runtime component as well as a compiler component to solve the problems described in the previous sections. Unfortunately, this approach is still an academic concept that doesn't have any robust implementation usable in the real world. To date, most of the research papers also concentrate on specialised hardware assisted schemes which are not usable on commodity hardware [[2](#SPMT04)] [[3](#SPMT03)].

There are currently two major approaches that can be used to implement a speculative multithreading system. The first approach is to parallelize the iterations of loops. Loops are natural bottlenecks in programs and can be naturally divided into separate tasks making this approach the easiest to implement. The second approach is called function based speculation using return value prediction [[4](#HuS03)]. The general concept is to delegate function calls to a new thread that is executed in parallel while the main thread continues executing after predicting the return value of the function. This approach is more difficult to implement because of the extra complexity required to instrument the code.

# Goals of the Project

The overall goal of this project is to provide a robust foundation for a speculative multithreading system which doesn't rely on any specialised hardware. The project will require the development a runtime component as well as a compiler component. These components will be used to automatically parallelize a program with minimal user intervention. Since it's not feasible to develop the full system during the 15 weeks allocated, this project instead focuses on building the key components. These key components will act both as a proof of concept and a solid foundation to construct the more advanced features.

The goal of the runtime component will be to run a *for*-loop in parallel while keeping its computation coherent. The coherence of the computation will be achieved by tracking every reads and writes made to the dependencies identified by the compiler component.

The runtime component will also have to achieve a speed up[[2]](#footnote-2) of at least 2 using a machine with 8 physical processors. Since the project proposition, our target has been lowered from a speed up of 4 to a speed up of 2. This reduction is due to the development of the runtime component that took more effort than originally estimated. We decided that it was necessary to cut the time spent on profiling and optimization in order to start working on the compiler component as soon as possible.

The runtime component should be generic and work with any types of programs. This means that it can't assume anything about the layout of the data it is tracking or rely on special hardware support. The runtime component can only take advantage of these special scenarios if instructed by the compiler component.

The compiler component should be able to analyse a *for*-loop, detect any relevant dependencies and instrument them with the required calls to the runtime component. To achieve within the allocated 15 weeks for the project, the component will only work on simplified *for*-loops whose body will also be kept simple.

The compiler component should also be able to instrument a target program with little or no user intervention. The use of user added pragma directives to guide the compiler component has been removed because of the unforeseen delays related to the development of the runtime component.

By the end of the 15th week, the system should be capable of automatically parallelizing a sample program. The project recommendation will indicate whether a speculative multithreading is a viable parallelization scheme in a real world scenario.

In summary, the goals for the runtime components are:

* To run a *for*-loop in parallel while keeping its computation coherent.
* Achieve a speed up of at least 2 using a machine with 8 physical processors.
* Create a generic component that works with any types of programs.

And the goals for the compiler components are:

* The ability to analyse a *for*-loop, detect any relevant dependencies and instrument them with the required calls to the runtime component.
* To process the target program with little or no user intervention.

# Methodology

This section describes the development methodology that will be used during the course of the project. The project is separated into three different processes: the documentation process, the development of the runtime component and the development of the compiler component. The development of the compiler component can only start once all the requirements for the runtime components have been met. The documentation process will be done in parallel with the development activities.

All development and documentation activities will be versioned using *git[[3]](#footnote-3)* as the versioning software and *github[[4]](#footnote-4)* as the remote repository. All the source code and the documents produced during this project will be open sourced under the *FreeBSD*[[5]](#footnote-5) license.

The development of the runtime component will focus on its performance. This focus should be reflected in the conception of the architecture which is documented in the *Architecture* document. Caches and shortcuts should be introduced to keep the load and store operations as small as possible. The synchronisation schemes should also be implemented as lock-free algorithms where possible to avoid concurrency bottlenecks. The entire source code for this component will follow the *C99* [[5](#Ste02)] standard with the exception of the public headers which will follow the *Clean C* [[5](#Ste02)] standard. For every module that is developed, a series of sequential and parallel tests will be created to ensure that the program works as expected and that there are no major synchronisation issues. The testing procedure will be codified in the *Test and performance* document.

To achieve the performance goals, the component should be tested on hardware that matches the requirements. A series of programs will be created to test the performances of the runtime component under different circumstances. These test programs will also be used to calculate the speed up achieved by the parallelisation. A strict performance timing methodology will be established to ensure that the results are correct and consistent. This methodology will be documented in the *Test and performance* document. Before doing any optimization, a bottleneck should first be identified using the profiling tool *gprof*.

The development of the compiler component will require an initial research phase to learn the available tools provided by *LLVM[[6]](#footnote-6)* and *clang[[7]](#footnote-7)*. The architecture which will be documented in the *Architecture* document, should aim to reuse the tools provided by *LLVM* and *clang* as much as possible. The implementation of any additional algorithms will be integrated as much as possible within the *LLVM* and *clang* architecture. The entire source code for this component will follow the conventions defined by the *LLVM* and *clang* project. For every module that is developed, a series of tests will be created to ensure that the program works as expected. The testing procedure will be documented in the *Test and performance* document.

The documentation process will start by the creation of a *Vision* document which will guide future developments once this project is finished. The *Architecture* document will be progressively constructed every time a component is completed. It will describe not only the current implementation but also how future features will be added. Finally the *Algorithms and proofs* documents will be progressively constructed to reflect the development activities of the current iteration.

# Progress Report and Recomendations

## Progress Report

This section will describe the work that has been completed since the project proposal was completed.

Currently the source code for the runtime component is complete and its functionalities have been thoroughly tested. The tests were conducted using sequential unit tests that were created for every compilation unit of the program. The runtime component was also tested using some very simple use cases that were run in parallel several million times to ensure that no major concurrency problems remain. Due to unexpected difficulties, the performance tests have not yet been conducted. Because of these delays, we are not able to determine whether our goals for this component have been met or not. The code and the tests for this component are available on Github[[8]](#footnote-8) and can be built on a Linux based system as long as the *check[[9]](#footnote-9)* library is present.

An initial draft of *Vision* document has been produced which contains the definition of the problem being addressed by our project. Work has also started on defining the architecture of the system in the *Architecture* document. Currently, the document contains a high level view of the components of the system and a more detailed look at the runtime component. Both documents are available as an annexe of this document.

Finally, some preliminary research on data-flow analysis algorithm has been conducted for the compiler. These algorithms will play a key part in analysing *for*-loops and detecting dependencies.

## Recommendations

This section will present what remains to be completed on our project.

Despite the delays, work on the compiler component is almost complete with only the performance tests that were delayed by a week. Since optimization opportunities were identified during development, the process of profiling and testing should not take more than a single iteration to complete. If no major problems are encountered during the performance tests, we recommend that the development of the compiler component should start right away. The delay will not have a large impact because our requirements for the compiler component have already been adjusted.

It is possible that the runtime component is unable to produce a reasonable speed up. In this case we recommend changing our approach for the runtime component. The new approach will use another speculative scheme which parallelizes function calls using return value prediction. It is possible that this scheme could allow the project to achieve its performance goals and a significant amount of our existing code could be reused to implement this scheme. A significant amount of research material on this subject has already been gathered during our initial research phase. Nevertheless, because of the additional effort required to develop this new scheme there would no longer be enough time left to develop the compiler component. It would have to be postponed to another phase of development.

# Deliverables and Planification

## Artefacts description

| **Artefacts** | **Description** |
| --- | --- |
| Vision | Description of the problem and the needs being fulfilled. |
| Architecture | Description of the system modules and their interaction. |
| Test and performance | Description of the testing methodology and the presentation of the performance results. |
| Algorithms and proofs | Description of the testing methodology and the presentation of the performance results. |
| Source Code | Source code for both the runtime kernel and the code injection modules. |

Since the project proposition, the *SRS* document has been removed from the planned artefacts. It added very little value and requires extensive efforts to keep it up to date. There was also content that overlapped with the *Algorithms and proofs* document. Therefore Some extra effort was also allocated to the *Vision* and the *Architecture* documents to fill in any missing gaps.

## Planning

The full breakdown and planning for the project is described in Annexe A.

Since the project proposition, the planning was modified to show the task breakdown per iteration. The development schedule was also adjusted to take into account the extra iteration that was required to finalize the runtime component. This removed an iteration from the compiler component schedule which was mitigated by removing the development of the user added pragma directives. For the documenting process, the removal of the SRS document allowed for more effort to be allocated to the *Vision*, *Architecture* and *Algorithms and proofs* documents.

# Risks

This section describes the various risks associated with the project along with a short analysis for each. The risks are split into general risks and risks specific to the current project.

|  |  |  |  |
| --- | --- | --- | --- |
| **General risks** | **Impact** | **Probability** | **Mitigation / attenuation** |
| Project objectives | Low | Low | Use a vision document to clearly define the objectives. |
| Project size | Medium | Medium | Break the implementation in short 1 week iteration with plenty of testing. |
| Testability | High | Medium | Allocate extra time to isolate and test difficult and critical areas. |
| Performance factors | High | High | Allocate an iteration for performance testing and optimization. |
| Maturity of Technology | Medium | High | Locate and read research papers available on the subject. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Specific risks** | **Impact** | **Probability** | **Mitigation / attenuation** |
| Difficult to adapt LLVM's data-flow analysis algorithms for our needs. | Medium | Medium | Research data-flow analysis and compiler optimizations algorithms so we can implement any missing pieces. |
| Complex and error prone lock-free algorithms. | Medium | Low | Fallback to simpler synchronisation scheme. This could negatively affect performances. |
| Portability issues between different OSs and processor architecture. | Low | Medium | Segregate any potential portability issues and test on multiple platforms. |
| Inexperience with the *LLVM* compiler and the *clang* frontend. | High | High | Allocate time to locate and read available documentation. |
| *For*-loop speculation doesn't provide a sufficient speed up. | High | High | Prepare research material on the *function based speculation using return value prediction* scheme. Consider dropping the compiler component from the project scope. |

Since the project proposition, the definitions of several risks were refined so that they better represent the project. The risk related to hardware constraint was removed because the school is providing this project with hardware that meets our performance test requirements. A new mitigation scenario was also added to address potential problems related to the performance tests. The probability related to the implementation of lock-free algorithms has also been lowered since they have already been implemented and no major problems have been encountered. No other major modifications have been made to this section since the project is still too early in its development to invalidate or confirm any other risks.

# References

x

|  |  |
| --- | --- |
| 1 | Aho, Alfred V., Lam, Modica S., Sethi, Ravi, and Ullman, Jeffrey D. *Compilers: Principles Techniques, & Tools*. Addison Wesley, Boston, 2007. |
| 2 | Porter, Leo, Choi, Bumyong, and Tullsen, Dean M. Mapping Out a Path from Hardware Transactional Memory to Speculative Multithreading. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)* (Washington, DC 2009), IEEE Computer Society, 313-324. |
| 3 | Prabhu, Manohar K. and Olukotun, Kunle. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '03)* (New York, NY, USA 2003), ACM, 1-12. |
| 4 | Hu, Shiwen, Bhargava, Ravi, and John, Lizy Kurian. The Role of Return Value Prediction in Exploiting Speculative Method-Level Parallelism. *Journal of Instruction-Level Parallelism*, 5 (2003), 1-21. |
| 5 | Steele Jr., Guy L. and Harbison III, Samuel P. *C: A Reference Manual*. Prentice Hall, Upper Saddle River, NJ, USA, 2002. |
| 6 | Bhowmik, Anasua and Franklin, Manoj. A General Compiler Framework for Speculative Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems* (2004), 713-724. |
| 7 | Cintra, Marcelo and Llanos, Diego R. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '03)* (New York, NY, USA 2003), ACM , 13-24. |
| 8 | Herlihy, Maurice and Shavit, Nir. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Burlington, MA, USA, 2008. |
| 9 | Steffan, J. Greggory, Colohan, Christopher B., Zhai, Antonia, and Mowry, Todd C. A scalable approach to thread-level speculation. In *Proceedings of the 27th annual international symposium on Computer architecture (ISCA '00)* (New York, NY, USA 2000), ACM, 1-12. |

x

# Table of Content for Final Report

1. Introduction
   1. Scope of the document
   2. Glossary
2. Context and problem definition
   1. Manual parallelisation
   2. Automatic parallelisation
   3. Speculative multithreading
3. Goals of the project
   1. Goals for the runtime component
   2. Goals for the compiler component
4. Implementation methodology
   1. Documentation process
   2. Development of the runtime component
      1. Testing methodology
   3. Development of the compiler component
5. Description of the implementation
   1. Runtime component
      1. Quick introduction to speculative multithreading
         1. Speculative execution
         2. Data dependencies
         3. Example of a speculative execution
      2. Implementation overview
         1. Component overview
         2. Read and write tracking
         3. Scheduling
         4. Performance considerations
   2. Compiler component
      1. Quick introduction to compilers
         1. Compiler architecture
         2. Data-flow analysis
      2. Implementation overview
         1. Component overview
         2. (to be determined)
6. Project results
   1. Runtime component
   2. Compiler component
7. Recommendations

# Annexe A: Revised Work Plan

The **start** and **end** column represents the range of weeks inclusively that each task will be accomplished in.

The **effort** column represents a rough estimate of the number of hours that will be spent on a given task.

This table presents the efforts already spent since the start of the project.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Effort | Component | Task/Milestone | Artefacts |
| 1 | 7 | Iteration 1 |  |  |
| 1.1 | 2 | Documentation | Scoped project planning | Project Proposal |
| 1.2 | 5 | Runtime component | Conception | Algorithms and proofs |
| 2 | 9 | Iteration 2 |  |  |
| 2.1 | 2 | Documentation | Scoped project planning | Project Proposal |
| 2.2 | 7 | Runtime component | yarn\_map implementation and testing | Source code |
| 3 | 11 | Iteration 3 |  |  |
| 3.1 | 1 | Documentation | Scoped project planning | Project Proposal |
| 3.2 | 5 | Documentation | Full system analysis | Vision |
| 3.3 | 3 | Runtime component | yarn\_tpool implementation and testing | Source code |
| 3.4 | 1 | Runtime component | yarn\_pstore implementation and testing | Source code |
| 3.5 | 1 | Runtime component | yarn\_pmem implementation and testing | Source code |
| 4 | 10 | Iteration 4 |  |  |
| 4.1 | 5 | Documentation | Full system analysis | Vision |
| 4.2 | 5 | Runtime component | yarn\_epoch implementation and testing | Source code |
| 5 | 10 | Iteration 5 |  |  |
| 5.1 | 5 | Documentation | Full system analysis | Vision |
| 5.2 | 5 | Runtime component | yarn\_epoch implementation and testing | Source code |
| 6 | 9 | Iteration 6 |  |  |
| 6.1 | 3 | Documentation | Summary report outlines | Progress report |
| 6.2 | 7 | Runtime component | yarn\_dep implementation and testing | Source code |
| 7 | 12 | Iteration 7 |  |  |
| 7.1 | 3 | Documentation | Summary report outlines | Progress report |
| 7.2 | 7 | Runtime component | yarn\_dep implementation and testing | Source code |
| 7.3 | 2 | Runtime component | yarn front-end implementation and testing | Source code |
| Total: | 67 |  |  |  |

This table presents our estimates for the rest of the project. It should be noted that the duration of one iteration will last for one week and that the tasks for the compiler component are still generic because more research is required to understand exactly how the component will be constructed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Effort | Component | Task/Milestone | Artefacts |
| 8 | 10 | Iteration 8 |  |  |
| 8.1 | 6 | Documentation | Performance report | Test and performance |
| 8.2 | 5 | Runtime component | Performance tests and profiling | Test and performance |
| 9 | 10 | Iteration 9 |  |  |
| 9.1 | 5 | Documentation | Definition of modules and their interaction | Architecture |
| 9.2 | 5 | Compiler component | Conception and research | Algorithms and proofs |
| 10 | 10 | Iteration 10 |  |  |
| 10.1 | 5 | Documentation | Definition of modules and their interaction | Architecture |
| 10.5 | 5 | Compiler component | Conception and research | Algorithms and proofs |
| 11 | 10 | Iteration 11 |  |  |
| 11.1 | 5 | Documentation | Definition of modules and their interaction | Architecture |
| 11.2 | 5 | Compiler component | Conception and research | Algorithms and proofs |
| 12 | 10 | Iteration 12 |  |  |
| 12.1 | 4 | Documentation | Summary report and recommendation | Summary report |
| 12.2 | 6 | Compiler component | Implementation and testing | Source code |
| 13 | 10 | Iteration 13 |  |  |
| 13.1 | 4 | Documentation | Project presentation and demo | Presentation slides |
| 13.2 | 6 | Compiler component | Implementation and testing | Source code |
| 14 | 10 | Iteration 14 |  |  |
| 14.1 | 6 | Documentation | Project presentation and demo | Presentation slides |
| 14.2 | 4 | Compiler component | Implementation and testing | Source code |
| 15 | 10 | Iteration 15 |  |  |
| 15.1 | 8 | Documentation | Summary report and recommendation | Summary report |
| 15.2 | 2 | Compiler component | Implementation and testing | Source code |
| Total: | 80 |  |  |  |

1. http://openmp.org/ [↑](#footnote-ref-1)
2. A speed up represents how much faster the parallel program is compared to the sequential program. A speed up of 2 indicates that the parallel program is twice as fast as the sequential program. [↑](#footnote-ref-2)
3. Git is a distributed version control system (http://git-scm.com/). [↑](#footnote-ref-3)
4. Github provides free hosting for open source projects (https://github.com/RAttab/yarn). [↑](#footnote-ref-4)
5. FreedBSD is a permissive open source license (http://www.freebsd.org/copyright/freebsd-license.html). [↑](#footnote-ref-5)
6. LLVM is a versatile and user-friendly compiler (http://llvm.org/). [↑](#footnote-ref-6)
7. Clang is the C, C++ and Objective-C front-end for the LLVM compiler (http://clang.llvm.org/). [↑](#footnote-ref-7)
8. https://github.com/RAttab/yarn/tree/master/libyarn [↑](#footnote-ref-8)
9. Check is a unit testing framework for C (http://check.sourceforge.net/). [↑](#footnote-ref-9)