**PROPOSITION**

**Projet de fin d’études**

**Département de génie logiciel et des TI**

**Speculative Multithreading System**

**A generic software-only implementation**

**Author**

**Rémi Attab****ATTR05098501**

**Supervising professor**

**Eric** **Paquette**

**Date**

**11 May 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 | 2011-05-11 |  | A | Initial draft | n/a |
|  |  |  |  |  |  |

Table of Contents

[1. Context and Problem Defintion 4](#_Toc292808704)

[1.1 Manual Parallelization 4](#_Toc292808705)

[1.2 Automatic Parallelization 4](#_Toc292808706)

[1.3 Speculative Multithreading 4](#_Toc292808707)

[2. Goals of the Project 5](#_Toc292808708)

[3. Methodology 5](#_Toc292808709)

[4. Deliverables and Planification 5](#_Toc292808710)

[4.1 Artefacts description 5](#_Toc292808711)

[4.2 Planning 6](#_Toc292808712)

[5. Risks 6](#_Toc292808713)

[6. Techniques and Tools 6](#_Toc292808714)

[6.1 Project Management 6](#_Toc292808715)

[6.2 Document Control 7](#_Toc292808716)

[6.3 Code Development 7](#_Toc292808717)

[7. References 7](#_Toc292808718)

[ANNEXE A: Work Plan 8](#_Toc292808719)

# Context and Problem Defintion

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.

## 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 be needed to be introduced. This would negate most, if not all, 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[[2]](#footnote-2). 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 has been developed. Speculative multithreading uses a run-time component as well as a compiler based 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. Most of the research papers also concentrate on specialised hardware assisted schemes which are not usable on commodity hardware[[3]](#footnote-3).

# Goals of the Project

The goal of this project is to provide a robust foundation for a speculative multithreading system which doesn't rely on any specialised hardware. This system will use both a runtime component as well as a compiler component 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. This means that the system will focus on performances by only considering a small subset of the target language.

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 multi-threading is a viable parallelization scheme in a real world scenario.

# Methodology

This section describes the development methodology that will be used during the course of the project.

To ensure the quality and good performance of the system, the components of the modules will be implemented in short 1 week iterations. During these iterations, a small set of features will be selected to be implemented and tested with unit tests. Since performance is a key factor, the system will be benchmarked regularly to ensure that no performance regression occurs. Extra time will also be allocated for researching and learning specific tools and the appropriate algorithm for a given problem.

Since a preliminary design was created during the initial research process, the code will be implemented in parallel with the high level documentation. Our short iteration should still keep the development agile enough to quickly address any problems that might be raised during the creation of the various high level documents. There are however a set of low level documents that depends on the results of the coding process. These will be integrated more tightly into the development process and will be part of the deliverables for the relevant iterations.

# Deliverables and Planification

## Artefacts description

| **Artefacts** | **Description** |
| --- | --- |
| Vision | Description of the problem and the needs being fulfilled. |
| SRS | Description of the requirements and specifications of the system. |
| Architecture | Description of the system modules and their interaction. |
| Test and performance | Description of the concurrent algorithms and data structures accompanied by their proof of correctness. |
| 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. |

## Planning

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

# Risks

This section describes the various risks associated with the project along with a short analysis for each.

|  |  |  |  |
| --- | --- | --- | --- |
| **Risk** | **Impact** | **Probability** | **Mitigation / attenuation** |
| Project objectives | Low | Low | Use a vision document to clearly define the objectives. |
| Hardware constraints | Medium | Low | Request specialized hardware from the school. |
| Reusable components | Medium | Medium | Research data-flow analysis algorithms in case we can't reuse the ones provided by LLVM. |
| Project size | Medium | Medium | Break the implementation in short 1 week iteration with plenty of testing. |
| Requirements stability | High | Low | Use a specification document to clearly define the requirements. |
| Testability | High | Medium | Allocate extra time to isolate and test difficult and critical areas. |
| Implementation difficulty | Medium | High | Fallback to simpler algorithms if necessary. Could negatively affect performances. |
| Performance factors | High | High | Allocate an iteration for performance testing and optimization. |
| External hardware or software interfaces | Low | Medium | Segregate any potential portability issues and test on multiple platforms. |
| Technology experience | High | High | Allocate time to locate and read available documentation. |
| Maturity of Technology | Medium | High | Locate and read research papers available on the subject. |

# Techniques and Tools

This section describes the various tools that will be used to develop and manage the project.

## Project Management

For a project of this size, *Emacs*'[[4]](#footnote-4) *org-mode[[5]](#footnote-5)* will be enough to manage and track the various tasks. It is not necessary to use a more robust tool like *Microsoft Project* because all the features have to be developed in a linear logical sequence. *org-mode*'s files are also easier to manage in versioning software like *git[[6]](#footnote-6)* or *svn*.

## Document Control

*Git* will be used as version control for all the source code and documents. Since the artefacts will be open-sourced under the FreeBSD[[7]](#footnote-7) license, *GitHub[[8]](#footnote-8)* will be used as a remote repository at no extra costs. The relevant repositories can be found below:

1. Documents: *https://github.com/RAttab/yarn-doc*
2. Source code: *https://github.com/RAttab/yarn*

## Code Development

The project will mostly use the standard *Linux* toolchain for the development. This includes *autotools*, *gcc*, *glibc*, *pthreads*, etc. The code will also adhere to the *clean C* and the *C99* standard[[9]](#footnote-9) to keep it as portable as possible. The code injection part of the system will be developed using the *clang[[10]](#footnote-10)* front-end for *LLVM[[11]](#footnote-11)* and should conform to the *LLVM* coding standards.

# References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles Techniques, & Tools* (Second ed.). Boston, MA, USA: Addison Wesley.

Bhowmik, A., & Franklin, M. (2004). A General Compiler Framework for Speculative Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems* , 713-724.

Cintra, M., & Llanos, D. R. (2003). Toward efficient and robust software speculative parallelization on multiprocessors. *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '03)* (pp. 13-24). New York, NY, USA: ACM .

Herlihy, M., & Shavit, N. (2008). *The Art of Multiprocessor Programming.* Burlington, MA, USA: Morgan Kaufmann.

Porter, L., Choi, B., & Tullsen, D. M. (2009). Mapping Out a Path from Hardware Transactional Memory to Speculative Multithreading. *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)* (pp. 313-324). Washington, DC: IEEE Computer Society.

Prabhu, M. K., & Olukotun, K. (2003). Using thread-level speculation to simplify manual parallelization. *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '03)* (pp. 1-12). New York, NY, USA: ACM.

Steele Jr., G. L., & Harbison III, S. P. (2002). *C: A Reference Manual* (Fifth ed.). Upper Saddle River, NJ, USA: Prentice Hall.

Steffan, J. G., Colohan, C. B., Zhai, A., & Mowry, T. C. (2000). A scalable approach to thread-level speculation. *Proceedings of the 27th annual international symposium on Computer architecture (ISCA '00)* (pp. 1-12). New York, NY, USA: ACM.

# ANNEXE A: 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.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| # | Start | End | Effort | Task/Milestone | Artefacts |
| 1 | 1 | 5 | 20 | Analysis |  |
| 1.1 | 1 | 3 | 5 | Scoped project planning | Project Proposal |
| 1.2 | 3 | 5 | 15 | Full system analysis | Vision |
| 2 | 6 | 11 | 40 | Conception |  |
| 2.1 | 6 | 8 | 20 | Elicitation of system specification | SRS |
| 2.2 | 9 | 11 | 20 | Definition of modules and their interaction | Architecture |
| 3. | 1 | 14 | 70 | Development |  |
| 3.1 | 1 | 7 | 35 | Runtime kernel module |  |
| 3.1.1 | 1 | 1 | 5 | Conception | Algorithms and proofs |
| 3.1.2 | 2 | 6 | 15 | Implementation | Source code |
| 3.1.3 | 3 | 7 | 15 | Quality assurance | Test and performance |
| 3.2 | 8 | 14 | 35 | Code injection module |  |
| 3.2.1 | 8 | 11 | 20 | Conception and research | Architecture |
| 3.2.2 | 9 | 14 | 15 | Implementation | Source code |
| 4 | 12 | 15 | 10 | Documentation |  |
| 4.1 | 12 | 14 | 4 | Project presentation and demo | Presentation slides |
| 4.2 | 7 | 15 | 6 | Project summary |  |
| 4.2.1 | 9 | 9 | 1 | Summary report outlines | Progress report |
| 4.2.2 | 13 | 15 | 5 | Summary report and recommendations | Summary report |
| Total: | 1 | 15 | 140 |  |  |

1. http://openmp.org/ [↑](#footnote-ref-1)
2. (Aho, Lam, Sethi, & Ullman, 2007) Chapter 11 - Optimizing for Parallelism and Locality [↑](#footnote-ref-2)
3. (Porter, Choi, & Tullsen, 2009) and (Prabhu & Olukotun, 2003) [↑](#footnote-ref-3)
4. http://www.gnu.org/software/emacs/ [↑](#footnote-ref-4)
5. http://orgmode.org/ [↑](#footnote-ref-5)
6. http://git-scm.com/ [↑](#footnote-ref-6)
7. http://www.freebsd.org/copyright/freebsd-license.html [↑](#footnote-ref-7)
8. https://github.com/ [↑](#footnote-ref-8)
9. Both the *clean C* and *C99* standards are described in (Steele Jr. & Harbison III, 2002). [↑](#footnote-ref-9)
10. http://clang.llvm.org/ [↑](#footnote-ref-10)
11. http://llvm.org/ [↑](#footnote-ref-11)