An Adaptive Optimisation System for Thread Level Speculation using Reinforcement Learning.

Andy Onyung

A Background Report

School of Computer Science:
The University of Manchester

May 14, 2010
Abstract

Multicore computers offer increased opportunity for program speedup via parallel execution. However, writing parallel programs is hard. Auto-parallelizing compilers often fail to fully parallelize many classes of programs as thread independence cannot always be proven at compile time. Using thread level speculation, threads can be created out of portions of the program that have not been proven to be independent. The dependencies are then resolved at runtime and offending threads are rolled back or squashed.

Unfortunately, finding the optimal decomposition of the program into threads is an NP-complete problem. Furthermore, there is a need to balance sometimes conflicting objectives such as resource utilization, power efficiency, squash prevention and increased speedup. Several approaches have been proposed in the literature to address these issues with varying degrees of success.

We provide a first mapping of reinforcement learning to thread level speculation. We propose a model for learning the best threads to create adaptively. Starting from a sequential program, we segment the program into states and define the action space as the combination of threads to create. We then apply temporal difference learning algorithms to learn the best threads to create based on defined system objectives.

Based on a simple proof of concept implementation on a two core system, the approach shows promise. We propose to extend this to an arbitrary number of cores. Also, several optimizations have been identified which may give improved results. Using a trace based thread level speculation simulator, the implementations will be evaluated on industry standard benchmarks selected from the SPECJVM 98 and DaCapo benchmark suites.
4.4.3 Serial Sections ......................................................... 22
4.5 Evaluation and Testing ............................................... 23
  4.5.1 Experiment One .................................................. 23
  4.5.2 Experiment Two .................................................. 23
  4.5.3 Experiment Three ................................................ 24
  4.5.4 Benchmarks ...................................................... 24
4.6 Work Plan ............................................................. 24

Appendix:

A Trace File Specification ............................................. 26
  A.1 Document Type Definition ....................................... 26
  A.2 Proof of Concept Trace File ..................................... 27

Bibliography .............................................................. 29
List of Figures

3.1 Illustration of Out of Order Speculation. ...................................................... 12
4.1 Illustration of the Multi-core problem. ............................................................ 21
List of Tables

4.1 Number of episodes to convergence for different learning rates. . . . . . . . . . . . . . . . . 20
4.2 Description of the implementations to be evaluated. . . . . . . . . . . . . . . . . . . . . . 23
4.3 Benchmarks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.4 Work Plan. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
List of Algorithms

1. Sarsa Algorithm ......................................................... 10
2. Algorithm to reject superfluous actions .......................... 19
3. Q-Learning algorithm applied in proof of concept implementation ............................................ 20
4. $Q(\lambda)$: Q-Learning with eligibility trace ..................... 22
Chapter 1

Introduction

1.1 Project Overview

Most computer machines produced today are multi-core. Unfortunately, legacy applications are mostly serial and would be costly to rewrite. Furthermore, writing parallel programs to exploit multi-core systems is hard. Auto-parallelizing compilers attempt to parallelize independent portions of the program. However, for most classes of programs, it is difficult to prove that there are no runtime dependencies. In this case, auto-parallelizing compilers cannot create threads because they have to guarantee correctness of the parallel program. This limits the amount of parallelism that can be exploited to improve performance.

Thread level speculation (TLS) is a technique which attempts to extract parallelism from sequential programs by speculatively executing sections of the program in parallel. These sections of the program need not be proven to be independent. A thread level speculation system monitors memory locations for runtime dependencies and squashes the offending threads. The key problem is therefore how to select the most profitable threads to create. It is important to avoid squashing threads as this represents wasted work and ties up valuable system resources. In fact, poorly chosen threads may result in worse performance than the sequential program. Furthermore, we often need to balance increased program speedup against other objectives such as power efficiency. However, selecting the best threads to spawn is an NP-complete problem. Several techniques[3, 5, 6, 11, 13, 14, 22, 27, 25, 28, 29] have been applied to the problem of selecting the best threads to spawn with varying degrees of success. Some TLS compilers[17, 13, 10, 11] select threads by considering several heuristic values. A common heuristic used is the estimated thread run-time. However, the estimate may be inaccurate and result in sub-optimal threads being chosen. Furthermore, assumptions are often made about the nature of overheads and cache behaviour.

Anecdotal evidence suggests that most production systems only run a limited set of programs during their lifetime. This means a per-program approach to optimization has some merit. We propose a best effort approach which adaptively learns the best policy for thread selection over the lifetime of the system using reinforcement learning. The benefits of this approach are threefold.

- it can be implemented entirely in software. This means that current chip multiprocessor based systems can benefit from this approach,

- programs can be optimized from their binary without the need to analyse the source code and

- it can be used in conjunction with other techniques allowing for advances in thread level speculation research to be easily incorporated.

Reinforcement learning is a technique that models the learning behaviour of an agent in an environment. By leveraging the theoretical framework of Markov Decision Processes (MDP), we can learn the
optimal action to taken given the state of the environment. Reinforcement learning techniques have been applied to several optimization problems [7, 16, 15]. We provide a first mapping of reinforcement learning to thread level speculation. We segment the sequential program into reinforcement learning states. We then define the actions in each state as the combination of spawn points that are spawned into threads. By applying temporal difference learning algorithms, we attempt to find the best threads to spawn.

1.2 Aim and Objectives

The aim of this project is to build an adaptive thread level speculation system that learns the best threads to spawn. To achieve this, the following objectives will be met:

- investigate a mapping of reinforcement learning to thread level speculation,
- design a model for learning which threads to spawn in order to improve performance,
- implement the designed model in a TLS simulator, and
- evaluate the performance of the proposed system.

1.3 Scope

Although the end game for this approach is to implement such a system as described above in a runtime system, this project is limited only to a TLS simulator. Also, a per-program optimization scheme has been proposed. The results of this study may however provide insights that allow generalisation to a group of programs.

1.4 Structure of the Report

The report consists of four chapters and an appendix. Chapter 1 provides an introduction to the project. It outlines the aim and objectives and sets the scope for the work to be done. We provide an introduction to reinforcement learning in Chapter 2. We define some of the key elements of reinforcement learning. The key learning methods as well as the specific algorithms used in this project are described. Chapter 3 contains an overview of thread level speculation. We also provide a review of some the important techniques in the literature for thread selection. Chapter 4 contains the research methodology followed for the project. It begins with a description our proposed solution. We also describe a proof of concept implementation that has been done as well as the preliminary results obtained. The rest of the chapter outlines extensions to be carried out, several optimisations that will be implemented and the work plan for the project. The sample trace file used for the proof of concept as well as a full data definition document of the trace file is included in the Appendix.
Chapter 2

Introduction to Reinforcement Learning

This chapter gives an introduction to reinforcement learning. We give a brief overview of reinforcement learning in Section 2.1. Section 2.2 introduces the key elements in reinforcement learning. The rest of the chapter discusses the different learning methods as well as the key issues in reinforcement learning.

2.1 Overview

Machine learning approaches can be grouped in three groups: supervised, unsupervised and reinforcement learning. In supervised learning, we are provided with marked test cases i.e. we are provided with examples as well as their target classes. The problem is then to predict the correct class of previously unseen example. On the other hand, in unsupervised learning, we only have the input data without the target classes. We then attempt to find patterns within the data. However, we sometimes want to learn the best combination of steps to achieve a given goal.

For example, let’s consider an a game of chess. We cannot usually tell if a single move is a good one or not. We simply know that a move that leads to a win is a good move. On the other hand, a move that leads to a loss is a bad move. Additionally, we may be playing against an intelligent opponent that does not play the same way every time. Therefore, simply remembering the moves that led to a victory the last time is not an appropriate solution. We need to learn how to respond to different opponents to achieve a winning objective. This is the forte of reinforcement learning. In reinforcement learning, we have an intelligent actor ( the chess player in our example) that observes the environment and takes an action. The decision of which action to take depends on the state of the environment at that point in time.

2.2 Key Elements of Reinforcement Learning

We define the key elements of reinforcement learning as follows.

Agent The agent is the learning decision maker. The agent observes the environment and makes decisions.

Model The system is modelled as a markov decision process. A markov decision process is a framework for modelling decision making. It can be completely specified a set of states, actions, transition probabilities and reward distribution. It relies on the markov property to make decisions i.e. the next state depends on the state immediately preceding it. The agent begins at the initial state $s_0$ and transitions to a next state at each time step. The step transitioned to depends on the transition probabilities
between states.

**State** The state refers to information or signal available to the agent from the environment. This is what the decision is based on. We define the state \( s_t \in S \) as the state of the agent at time step \( t \).

**Action** The action refers to a decision taken by the agent in a particular state. We define the action \( a_t \in A \) as the action taken in time step \( t \).

**Policy** The policy defines the behaviour of the agent. It defines the actions that are to be taken by the agent in each state. We define the policy \( \pi \) as the mapping of states to actions i.e.

\[
\forall s_t \in S, \pi : S \rightarrow A. \tag{2.1}
\]

**Reward** This is an immediately obtainable signal from the environment indicating the desirability of the most recently taken action. We define reward \( r_{t+1} \in R \) as the immediate reward obtained from taking action \( a_t \) in state \( s_t \).

**Episode** This is a complete navigation of the agent from the initial state, \( s_0 \), to a terminal state, \( s_T \). At the end of each episode, the agent is reset to the initial state.

**Quality** The quality, \( Q(s_t, a_t) \), is the accumulated desirability of a particular state action pair. It is usually updated each time the state action pair is visited.

### 2.3 Learning Methods

Several methods exist for learning the optimal policy. Dynamic programming methods compute the optimal value function. From this, the optimal policy is derived. It is guaranteed to find the optimal policy in polynomial time. However, dynamic programming methods are inefficient and may not be practical for very large reinforcement learning problems. Also, dynamic programming methods require that we have full knowledge of the model.

Another class of methods known as Monte Carlo methods can work without full knowledge of the model. These methods only require sample sequences of states, actions and rewards. The required model parameters are estimated as averages from the previous visits. With enough visits, the estimates converge to their optimal values. However, some state action pairs may not be visited. This results in a problem since we require the action values to choose among available actions. This can be avoided by forcing each episode to start with a state-action pair; with each state action pair having a non-zero probability of being selected.

Temporal difference methods combine properties of both dynamic programming and Monte Carlo methods. Like Monte Carlo methods, they learn from raw experience without the need for a model. Like dynamic programming, the estimates are based in part on other estimates. We explore two temporal difference algorithms, Q-Learning and Sarsa, in Sections 2.3.1 and 2.3.2 respectively.

#### 2.3.1 Q-Learning

The Q-learning algorithm [26] learns the Q values directly from the trial episodes. It is an off-policy method. This means that the value of the next best action is used irrespective of the policy used in selecting actions. Given a probability distribution of rewards, \( p(r_{t+1}|s_t, a_t) \), and a probability distribution for the next state, \( P(s_{t+1}|s_t, a_t) \), we can estimate the Q value for the state action pair, \( (s_t, a_t) \), as
\[ Q(s_t, a_t) = E[r_{t+1} + \gamma \sum P(s_{t+1}|s_t, a_t)Q(s_{t+1}, a_{t+1})] \]  
\[(2.2)\]

However, we may receive different rewards on each visit to the same state action pair. Using the Q-learning algorithm, we keep a running average. Equation 2.2 then becomes

\[ \hat{Q} \leftarrow \hat{Q}(s_t, a_t) + \eta \left( r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right) \]  
\[(2.3)\]

This repeatedly reduces the difference between the backed up estimate \( \hat{Q} \) and the current Q value.

### 2.3.2 Sarsa

The Sarsa algorithm is an on-policy variant of the Q-learning algorithm. In this case, instead of using the maximum value over \( a_{t+1} \), we use the value of the action selected using the policy to compute the temporal difference. This allows us to estimate the value of the policy while using it to take actions. The pseudo-code for the Sarsa algorithm is shown in Algorithm 1.

**Algorithm 1 Sarsa Algorithm**

1: Initialize \( Q(s_t, a_t) \) arbitrarily  
2: for all episodes do  
3:    Initialise \( s_t \)  
4:    Choose \( a_t \) using policy derived from Q, for example, \( \epsilon \)-greedy  
5:    while \( t < T \) do  
6:      Update \( Q(s_t, a_t) \):  
7:      \[ Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta (r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \]  
8:      \( t = t+1 \)  
9:    end while  
10: end for

### 2.4 Exploration and Exploitation

A key problem in reinforcement learning is deciding whether to choose our best action or to try a different action. If we do not try new actions, then we may never find the best solution. On the other hand, if we only try new actions, we may not achieve our goal quickly enough. Achieving a balance between exploration and exploitation is therefore the subject of much research. A popular approach is the \( \epsilon \)-greedy approach. We choose an action randomly, among all possible actions, with probability \( \epsilon \) and with probability \( 1 - \epsilon \), we choose the best action so far. We begin with a high \( \epsilon \) value and gradually reduce it. This means that we explore frequently early and exploit more later.

It is possible for a learning algorithm to explicitly handle the exploration and exploitation problem. One of such algorithm is the E3 algorithm described in [12]. The algorithm guarantees optimal performance in polynomial time for a general case MDP. The algorithm starts off using a technique called balanced wandering. When the agent reaches an unvisited state, it takes an arbitrary action. If the state has been previously visited, the agent selects the action which has been tried the fewest number of times. It uses the notion of known states i.e. states that have been visited so many times that the reward and transition probability estimates are close to their true values. The algorithm can determine if the optimal policy achieves its high return using the currently known states. If it does, then we exploit. Otherwise, we explore further.
Chapter 3

Thread Level Speculation

In this chapter, we discuss some of the important techniques on thread level speculation in the literature. We begin with a general overview of thread level speculation in Section 3.1. A key problem in thread level speculation is how to select profitable threads. We discuss different approaches for selecting profitable threads in Section 3.2.

3.1 Overview of Thread Level Speculation

Thread level speculation is a technique which attempts to extract parallelism from sequential programs by speculatively executing sections of the program in parallel. These sections of the program need not be proven to be independent. A thread level speculation system monitors memory locations for runtime dependencies and squashes the offending threads. Several studies [5, 6, 11, 13, 14, 22, 28, 29] have studied creation of threads from loop iterations and methods.

3.1.1 Loop-Level Speculation

Loops are usually a good source of parallelism. Some loops exhibit data parallelism with each iteration processing a different part of a data block. Most parallel programming languages provide constructs that allow easy parallelization of loops. In sequential programs, we can speculatively execute loop iterations to extract parallelism. Any loop in which the results of the loop do not depend on the order in which the iterations are executed can be executed speculatively. Threads are extracted from loops by spawning threads to execute each iteration. Where there is not enough work in each iteration, larger chunks of iterations may be packed within a single thread. Also, nested loops may be parallelized by speculatively executing iterations at different levels. Finding the right loop level to parallelize is the subject of research with solutions proposed in [14, 29].

3.1.2 Method-Level Speculation

Methods are also a good candidate for speculative execution. This is because methods often represent a logical separation of tasks. Method level speculation has been shown to provide significant speedups in single threaded, general purpose Java programs [6]. However, the instructions immediately following the method completion often require values returned from the method. Therefore, method level speculation is often coupled with return value prediction. Using return value prediction, newer threads are given the predicted return value of the method. This allows progress to be made on the newer threads. Upon completion, the predicted value is validated. If the value was wrongly predicted, then the newer threads are rolled back or squashed.
3.1.3 In-Order Speculation

In-Order speculation requires that newer threads are more speculative than previous threads. Given a method to be parallelized speculatively, we spawn a speculative thread to execute instructions following the method completion. The non-speculative thread executes the instructions within the method body. If any nested methods are encountered within the method body, they cannot be spawned. This is because they would have to be spawned at a less speculative order than the previous thread. This limits the amount of parallelism that can be exploited. Out of order speculation offers more potential for thread level speculation.

3.1.4 Out-of-Order Speculation

As shown in [19], out-of-order speculation offers significantly greater speedup than in-order speculation. Figure 3.1 illustrates threads being created out of order. When the non-speculative thread approaches the Method1 call, a new thread is created to execute the method continuation. The non-speculative thread continues into the method body. At the Method2 call, a new thread is also created to execute instructions following the method. Despite being newer than the previously created Thread 1, Thread 2 is less speculative.

![Figure 3.1: Illustration of Out of Order Speculation.](image)

Out-of-order execution is hard to manage due to the irregular sequence of thread creation and completion. Care must be taken that the threads commit their work in the right order to ensure correctness of the program. Also, it may be necessary to forward values from less speculative threads to more speculative threads.

3.2 Thread Selection

It is important to avoid squashing threads as this represents wasted work and ties up valuable system resources. Although, it has been suggested that squashed threads often have pre-fetching benefits[13, 25]. However, identifying profitable threads is a difficult problem. In fact, finding the optimal combination
of threads has been shown to be NP-complete[20]. We have to balance sometimes conflicting objectives such as minimising thread overheads, fully utilizing system resources and preventing squashes. We discuss some of the techniques proposed in the literature for selecting profitable threads.

### 3.2.1 Simple Heuristics Based Techniques

Several heuristics have been presented in the literature for selecting profitable threads[21, 24, 27]. Some of the heuristics proposed for selecting methods are average run length and the number of dynamic stores[27]. When selecting methods to speculate on, we want methods that contain a considerable amount of work. The overhead of thread creation and completion means that speculating on small methods may result in worse performance that serial execution. We can discard small methods by requiring that methods to be speculated upon have a run length that is greater than a threshold [27]. The threshold is chosen to be greater than the overhead. However, longer methods are more likely to result in a squash. Hence a upper limit is also set on method run length. We often do not have prior information on the run length of methods. This means that we have to use the average run length from previous executions [27] or predict whether the run length exceeds the threshold. The run length prediction scheme described in [24] uses a bit to determine if the last execution of method exceeded the threshold. However, there is only a small correlation between the run length and the likelihood of a squash. Therefore methods whose run lengths lie between the upper and lower thresholds may still result in a squash.

Addition techniques proposed by [27] include using single-pass and multi-pass heuristics. Using single-pass heuristics, methods are ranked depending on either the speedup obtainable or the number of cycles that can be saved by speculating there. This was shown to give the best performance improvements as it strikes a fine balance between over-speculation and under-speculation. With multi-pass heuristics, interactions between nested speculative methods are explored. This attempts to limit over-speculation that may result from double counting the benefits of both parent and child methods. However, this may in poor performance due to under-speculation.

An interesting approach to squash prediction is to apply machine learning algorithms to static and dynamic program features to produce a general predictor. Several nano-patterns that may be useful for classifying Java methods were proposed in [21]. Using clustering techniques, it was shown that different method types form natural clusters. These nano-patterns as well as other program features were combined to form a predictor in [22]. By applying the C4.5 algorithm[18], a decision tree was generated to predict the likelihood of squashing. The threads are then created using a greedy strategy. This means threads that are predicted not to squash are spawned as soon as the associated spawn points are reached and execution units are available. The technique shows promise but the performance gains are limited by under-speculation.

### 3.2.2 Thread Level Speculation Compilers

Traditional compilers are limited by having to ensure that there are no dependencies between threads. TLS compilers are not bound by any such constraint. Several compiler systems[17, 13, 10, 11] have proposed different techniques for selecting profitable threads for speculative execution.

The Mitosis Compiler [17] minimises inter-thread dependencies using pre-computation slices to predict machine state before executing new threads. These pre-computational slices are derived from the original code and are inserted at the beginning of every speculative thread. The pre-computational slices are built by traversing the instructions between the spawn point and the start of the speculative thread looking for read after write dependencies. The instructions that produce thread live-ins, i.e values consumed but not produced by the thread, are added to the pre-computation slice. To reduce the amount of overhead introduced, the ratio between the length of the pre-computation slice and the estimated length
of the speculative thread must be lower than a threshold. The scheme achieves significant speedup with optimisations such as memory dependence speculation and dependence pruning (infrequent data dependencies are ignored). However, the Mitosis compiler uses an exhaustive search process to identify profitable spawn points. This is a lengthy process and is not practical for many systems.

The Posh compiler [13] uses a simple profiling pass to discard unprofitable threads. The compiler creates threads out of tasks. These tasks are selected from programmer-defined structures such as method calls, loop iterations, method continuations and loop continuations. Before execution, the tasks are hoisted as possible, i.e., set to start execution as soon as possible to increase overlap. However, the tasks cannot be hoisted higher than the definition of any variable that is used in the task unless the value can be predicted. A profiler is then used to collect information about the selected tasks. The information collected is then used to estimate the likelihood of a squash occurring. The amount of parallelism obtainable as well as pre-fetching benefits are then estimated. During the profiling run, timestamps are recorded for stores to memory locations. This information is used to detect data dependencies and violating tasks are squashed. The benefit of squashed tasks are quantified as the sum of the task overlap remaining after re-execution and pre-fetching benefits. Tasks are then eliminated based on three criteria. Firstly, tasks that are smaller than a threshold are eliminated. The hoisting distance is then used to eliminate other tasks. Tasks whose hoisting distance do not fall between the upper and lower threshold distances are eliminated. This is because a small hoisting distance indicate limited overlap while a large hoisting distance might result in a large number of data dependencies. Finally, the average number of squashes per task commit for each task is compared against a threshold value. Tasks with a higher average number if squashes than the threshold value are eliminated. The POSH compiler achieves an average speedup of 32% for applications in the SPEC CPU2000 benchmark suite on a simulated 4 core superscalar processor.

A minimum cut approach to thread selection is described in [10]. Using a control flow graph with weighted edges, a balanced minimum cut algorithm was applied to decompose the sequential program into threads. The control flow graph consists of basic blocks of the program as vertices. The graph is annotated with static information about instructions in each block. Profiling information such as branch frequencies, average cycles per function call and dependencies were also included. The edges of the graph are weighted with the estimated number of cycles lost to mis-speculation. This is used as the cut metric. Given that a minimum cut represents a cut with the minimum sum of edges, this minimises the amount of mis-speculation. However, minimising mis-speculation may result in zero parallelism. Therefore, a balancing metric is used to encourage speculation. The balancing metric comprises of two components. It includes the cut metric as well as the estimated overall runtime and cycles lost to load imbalance. By using the overall runtime as the termination criteria, all overheads are equally considered. The cut algorithm begins with the sequential program and recursively performs minimum cuts on threads until the overall performance metric cannot be improved upon. The technique is able to achieve an average speedup of 74% on floating point applications and 23% on integer applications in the SPEC CPU2000 benchmark suite using a four-core superscalar processor.

A different technique, proposed in [11] performs a profile time search for profitable threads. Static techniques rely on runtime estimates which may be inaccurate. On the other hand, techniques that use dynamic runtime information to select threads at runtime often suffer from increased overhead. Using the proposed technique, an empirical search is performed using actual runtime information during the profiling phase. The selected threads are then compiled into the production binary. This avoids the overhead of doing runtime threads selection despite using more reliable runtime information. The compiler begins with the assumption that all loop iterations and method calls can be speculatively executed. It then searches through the space to determine if serializing each loop iteration or method call improves performance. This technique obtains an average speedup of 87% on selected floating point applications from the SPEC CPU2000 benchmark suite on a four core CMP simulator. Unfortunately, no timing information is given on the duration of the search during profiling.
3.2.3 Adaptive Optimisation Techniques

In some cases, source and object code may not be available to extract static features from. A technique for automatic parallelization of Java programs was proposed in [3]. In this approach, paths of execution known as traces are considered as units of speculation. These paths may comprise of multiple methods and loops. They are collected by keeping track of program execution. The traces are then packaged into methods are are executed in parallel. The performance was shown to be comparable to hand-parallelized versions of the same programs. However, these tests were performed on programs that exhibit data level parallelism.

The Jrpm system [5] uses a hardware profiler to capture information on which loops are the best to parallelize. It uses the TEST tracer[4] to profile sequential program execution. Dependency timing information as well as thread length and speculative buffer usage estimates are collected and used to select loops to speculative on. Load dependency analysis, which compares timestamps on previous memory stores to thread start timestamps, is used to determine if dependencies exist. This dependencies limit the amount of parallelism between threads. Also, speculative state flow analysis is performed to ensure that the speculative state for a loop can fit within the level one caches and store buffers. This prevents speculative state overflow. The accumulated results of the analysis are used to predict performance of speculative loop threads. Results are accumulated until either 1000 iterations have been executed or threads created from outer loops are predicted to always overflow speculative state. Generally, the system is able to achieve speedups of at least 1.5 on most benchmarks running on four cores. Greater speedup of between 3 and 4 is achieved on floating point applications. However, the system requires some intervention by the programmer to expose parallelism in some cases.

Several performance tuning policies were explored in [14] to identify efficient speculative threads. Using the simple policy, the innermost level of a loop nest is speculatively parallelized several times. The execution time is then compared against the serial execution time. If the speculative performance is worse than serial execution more times than a threshold, the loop is serialised. Speculation then continues at a level above the current level. The qualitative policy uses information on the number of cycles saved to weight the performance counts used in simple. Under this policy, the loop is only serialised if it satisfies the criteria under simple and the weighted number of cycles becomes negative. The simple and quantitative policies both search for the best level from the innermost level. This means that the best level for speculation may never be reached. The Quantitative+Static policy uses compiler annotations to suggest a level to begin searching from. If the level is better than sequential execution, the search is stopped. Otherwise, the suggested level is serialised and the search resumes from the innermost loop.

The JaTLS scheme proposed in [29] uses a runtime feedback to drive adaptive optimisation to search for the best way to parallelise loops. It uses the collapse rate, defined as the ratio of re-executions to the number of speculative threads created, as the major feedback parameter. Several optimisations are introduced for chunk size, number of threads and loop level selection. The appropriate loop level is selected using a technique similar to that described in [14]. In this case, the search is done starting from the outermost loop. If the collapse rate of the loop level is less than the threshold, the inner sub-loops are evaluated. For some loops, a single iteration may not contain enough work to offset the thread creation and completion overheads. The optimisation for chunk size searches linearly from 1 to the maximum number of iterations that can be executed in a single speculative thread.

A different approach using hardware mechanisms for preventing thread squashes was proposed in [8]. The framework proposed extends the cache coherence protocol of a directory-based CC-NUMA architecture. Hardware counters are used to track memory accesses per cache line. This means that access to different words on the same line cannot be disambiguated. This may result in false dependencies. Several mechanism are introduced to prevent squashing. These are Delay\&Disambiguate (potential violations are assumed to be false dependencies), ValuePredict (a predicted value is given to the consumer thread on
request), Stall&Release (consumer thread is stalled till the producer commits) and Stall&Wait (consumer thread is stalled till it becomes non-speculative). The technique shows an average overall speedup of 36.8% on selected applications from SPEC CPU2000 benchmark suite. However, it is architecture dependent and cannot be applied to processors not based on the directory based CC-NUMA architecture.

3.3 The Problem Statement

This project attempts to answer the question: how can we adaptively learn the best combination of threads to achieve predefined system objectives?

Several techniques have been proposed which attempt to find profitable threads. But these often settle for a selection of threads that offer a good speedup. It may be possible to learn the best combination of threads as defined by system objectives. These objectives may be good speedup, power efficiency, etc. We provide a novel application of reinforcement learning to thread level speculation. This allows us to learn complex spawn structures without resorting to exhaustive search. The proposed solution is described fully in Section 4.1.
Chapter 4

Research Methodology

In this chapter, we present the methodology followed for this project. Firstly, we present the proposed solution to the problem. We discuss a simple proof of concept implementation of the proposed solution. This includes some discussion on preliminary results obtained. We then extend the proof of concept to systems with more than two cores. Based on the preliminary results, we propose several optimisations for improved convergence. Finally, we describe our testing and evaluation framework.

4.1 Proposed Solution

We model the system as a Markov Decision Process (MDP). Let $S$ be a set of numbered and non-overlapping segments of sequential program execution, $A_s$ be a set of actions for each segment and $a \in A_s$ be a combination of threads that are spawned in segment $s$. We define the quality of action $a_t$ taken in state $s_t$ as Equation 4.1.

$$Q(s_t, a_t) = E[r_{t+1} + \gamma \sum_{a_{t+1}} P(s_{t+1}|s_t, a_t)Q(s_{t+1}, a_{t+1})].$$  \hspace{1cm} (4.1)

This is known as Bellman’s equation [1]. The immediate reward $r_{t+1}$ is defined as the weighted sum of the number of cycles saved if action $a_t$ is taken in state $s_t$. This is defined fully in Section 4.1.2. We select our best policy $\pi$ as the maximum possible $Q(s_t, a_t)$ in each state, i.e.

$$a^\pi_t = \arg \max_{a_t} Q(s_t, a_t).$$  \hspace{1cm} (4.2)

Given the sequential order of the program execution, we can guarantee that we always move from state $s_t$ to $s_{t+1}$. Hence, $P(s_{t+1}|s_t, a_t) = 1$ for the next state as specified by the program order and 0 for all other state. This reduces Equation 4.1 to

$$Q(s_t, a_t) = E[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})].$$  \hspace{1cm} (4.3)

However, programs often exhibit data dependent behaviour that result in varying rewards between executions. To overcome this, we keep a running average of $Q(s_t, a_t)$ using the update function

$$Q(s_t, a_t) = Q(s_t, a_t) + \eta(r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)).$$  \hspace{1cm} (4.4)

We use a lookup table to store quality values. This is a rather inefficient approach which may be untenable for very large programs. Function approximators which estimate the reward for different segments and actions may be explored.

17
4.1.1 Sequential Program Segmentation

The choice of states is critical to the performance of the system. We may divide the program into fixed sized segments. This is straightforward approach that is easy to implement but may not give the best results. This is because the fixed sized segments do not take program structure into account. Using variable sized segments, we may exploit program structure to obtain better results.

We segment the sequential program execution around sections where the processing resources are fully utilised. We define a variable sized segment as a sequence of instructions beginning from the earliest spawn point in a fully utilized system to end of our most speculative thread.

Any segmentation scheme will encounter a common problem. Spawn points may appear anywhere within the section. Spawn points appearing near the end of the segment or spawn points corresponding to long methods may result in speculation beyond segment boundaries. This violates one of the premises of MDPs that the actor can only be in one state at a time. There are two approaches to solving this problem. A straightforward approach is to prevent speculation beyond section boundaries. However, this reduces the amount of overlap and would result in poor performance. We may allow speculation beyond section boundaries by relaxing the time step constraint of the MDP. We call our states that relax the time step constraint, virtual states. This allows us to speculate beyond the section boundary. Using virtual states, we can reward action $a_t$ in state $s_t$ for speculatively executed instructions that lie beyond the section boundary. The proposed reward function in described in Section 4.1.2.

4.1.2 Reward Function

We define the reward function $r$ as

$$r = \alpha r_p + \beta r_c + \phi r_o + kC_{ms}. \quad (4.5)$$

where

- $r_p$ is the number of successful speculative cycles executed from this state as a result of an action taken in the previous state,
- $r_c$ is the number of successful speculative cycles executed from this state as a result of an action taken in this state,
- $r_o$ is the number of successful speculative cycles executed from the following state as a result of an action taken in this state, and
- $C_{ms}$ is the number of unsuccessful speculative cycles executed as a result of an action taken in this state.

We can define the priorities of the system and control how credit is assigned by adjusting the parameters $[\alpha, \beta, \phi, k]$. For example, on a low power system, we can set $k$ to a large negative value to discourage mis-speculation. Furthermore, given the ability to speculate beyond state boundaries, taking an action in state $s_t$ may affect which actions are available to be taken in state $s_{t+1}$. This raises two issues. Firstly, we must assign some credit of the successful cross boundary speculation to the actions that are still possible in the following state. This prevents us from choosing a policy that is not feasible. The $r_p$ term in our reward function addresses this problem. Additionally, the $\gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$ term in our update function (Equation 4.4) assigns a fraction of the reward from future actions to the present state. With exclusive states, this is valid. However, using virtual states, the best action in the next state may not be feasible because of an action taken in the present state. To avoid assigning credit incorrectly, we modify Equation 4.4 such that
\[ Q(s_t, a_t) = Q(s_t, a_t) + \eta(r_{t+1} + \gamma \max_{a'_{t+1}} Q(s_{t+1}, a'_{t+1}) - Q(s_t, a_t)), \]  

(4.6)

where \( a'_{t+1} \) is the set of actions that are feasible in state \( s_{t+1} \) given \( (s_t, a_t) \).

### 4.1.3 Action Space Definition

We define action \( a_t \) in state \( s_t \) as a set of spawn points that are executed speculatively. There exists a null action for each state which corresponds to no speculation being done in the state. If there exists \( n \) spawn points in state \( s_t \), ordinarily we have \( 2^n \) actions in that state. However, given the number of available cores and positions of the spawn points, some actions are superfluous. We trim the action space by rejecting superfluous actions using the following algorithm.

**Algorithm 2** Algorithm to reject superfluous actions

1. for all actions in segment do
2. Given \( p \) cores, \( n \) spawn points in action (with each spawn point beginning at startTime and committing at commitTime), where \( n > p \).
3. Sort spawn points by sequential order
4. for first \( p - 1 \) spawn points do
5. Compute run length
6. end for
7. select earliest committing thread
8. earliestCommit = commitTime
9. for spawn points \( p \) to \( n \) do
10. if \( startTime \leq \) earliestCommit then
11. Reject action
12. end if
13. end for
14. end for

### 4.2 Proof of Concept Implementation

To test the feasibility of the proposed solution, a simple proof of concept implementation was done. It emulates a 2 core TLS system using a sequential program trace. For the sake of simplicity, we ignore cost of thread spawns and commits. We adopt the same trace file format used in [22]. The trace file records significant execution events such as method calls and completions, memory reads and writes. A complete description of the trace file format can be found in Appendix A.1. We evaluate the implementation on a constructed program trace. The trace file, shown in Appendix A.2, consists of 4 methods of varying lengths. It is engineered such that there are no data dependencies between the methods. Furthermore, the methods calls are placed close to each other so a choice has to made on which method to speculate on. The emulator takes this sequential trace as input and spawns methods as specified by the learning policy. Upon convergence, the emulator prints out the recommended selection of spawn points where threads should be created.
4.2.1 Application of Q-Learning

We adopt an episodic model where we execute the entire program trace repeatedly. We apply a slightly modified version of the Q-learning algorithm which takes into account the feasible actions based on previously executed actions. This is shown in Algorithm 3. We select the reward function parameters \([\alpha, \beta, \phi, k] = [0,1,1,0]\). This means that we equally value speculation in the current state as well as future states. We also do not penalise mis-speculation.

Algorithm 3 Q-Learning algorithm applied in proof of concept implementation

1: Initialize all \(Q(s,a) = 0\)
2: for all episodes do
3: while \(s_t \neq s_T\) do
4: Choose \(a_t\) from possible actions
5: \(r_{t+1} = \alpha r_p + \beta r_c + \phi r_o + kC_{ms}\)
6: Update \(Q(s_t, a_t) = Q(s_t, a_t) + \eta(r_{t+1} + \gamma \max_{a'_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))\)
7: \(t = t + 1\)
8: end while
9: end for

To track which actions are feasible in any state, we maintain a void list for each action. The void list contains the identifiers of all the spawn points encountered during the speculative execution that could not be created. A global void list is also maintained which contains all the voided spawn points in the current episode. This information is used both for action selection and credit assignment.

4.2.2 Preliminary Results

We explore the effect of the learning rate, \(\eta\), on the number of episodes required for convergence. We use an aggressive exploratory value of \(\epsilon = 1\). Table 4.1 shows the number of episodes required for different values of \(\eta\). The episode values are the average of five runs. In our reference example, the reward values are stationary, so we can achieve convergence using a single \(\eta\) value. As expected, we achieve convergence faster with higher learning rates. For non-stationary rewards, we will start with a high \(\eta\) and gradually reduce it to achieve convergence.

<table>
<thead>
<tr>
<th>Learning Rate ((\eta))</th>
<th>Episodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>88</td>
</tr>
<tr>
<td>0.8</td>
<td>131</td>
</tr>
<tr>
<td>0.7</td>
<td>241</td>
</tr>
<tr>
<td>0.6</td>
<td>311</td>
</tr>
<tr>
<td>0.5</td>
<td>423</td>
</tr>
</tbody>
</table>

Table 4.1: Number of episodes to convergence for different learning rates.

To put the results in perspective, we compare the performance to a naive exhaustive search. Using exhaustive search, we visit every possible combination of spawn points. Due to the stationary rewards, we only need to visit each combination once. Therefore, an exhaustive search algorithm can find the optimal solution in \(2^n\) episodes; where \(n = 4\) is the number of spawn points. However, if we have non-stationary rewards, the exhaustive search algorithm may never find the optimal solution. In section 4.4 we discuss
several optimisations to our default implementation that will improve the number of episodes required for convergence.

4.3 Extension to Many Cores

The proposed solution shows promise on a two core system. On a two core system, only a single speculative thread can be executed. Therefore, any spawn points encountered when we cross state boundaries cannot be created and are added to the void list. However, where there are more cores available, a decision has to be made: to spawn or not to spawn.

Consider the sample program shown in Figure 4.1. On a two core system, we have three actions in state 1; the null action, spawn at Method2 or spawn at Method3. With an additional core, we can spawn Method2 from the speculative thread from state 0 or from a non-speculative thread in state 1. This can result in incorrect reward calculations under certain conditions e.g the more speculative spawn of Method2 results in a squash. The implication of this on our learning algorithm is that the action to spawn Method2 is penalised unfairly relative to Method3.

To avoid this unfair penalty, we can limit speculation to only spawn points that are reached in the current state. However, this limits the amount of parallelism that can be exploited. A more robust solution is to create actions for executing the method at higher speculative orders. To keep our implementation simple, we use a boolean variable to mark speculative execution of the spawn points in the action.
4.4 Convergence Optimisations

4.4.1 Optimistic Q-values

The rate of convergence depends on the initial Q values chosen. For the proof of concept implementation, the Q values were initialised to 0 with $\epsilon = 1$. By taking exploratory actions only, we sidestepped the need to select initial Q values. However, to balance exploration and exploitation, we will start with a high $\epsilon$ value and gradually reduce to increase exploitation. A slightly different technique, which is shown to approach optimal actions faster [23], is to use optimistic initial values with $\epsilon = 0$. Combined with the greedy policy, early actions are exploratory. Unprofitable actions are quickly marked down while good actions are selected more frequently resulting in the improved performance.

4.4.2 Eligibility Trace

In the default implementation, the update equation (Equation 4.4) only updates the previous Q value of the state-action pair. However, we may improve convergence by assigning credit temporally i.e by assigning some credit to recently executed state-action pairs. To implement eligibility trace, we store an eligibility value, $e(s,a)$, for each state-action pair. Algorithm 4 shows a modified Q learning algorithm with eligibility trace, $Q(\lambda)$. The $e(s,a)$ values are all initialised to 0. When we take action $a_t$ in state $s_t$, we set $e(s_t,a_t) = 1$. We then decay the eligibilities of all other state-action pairs by $\gamma \lambda$; where $0 \leq \lambda \leq 1$ is the trace decay parameter. It should be noted that additional computation is required to update the eligible items. However, due to decay, only a few $e(s_t,a_t)$ values will be non-zero and need to be updated.

**Algorithm 4** $Q(\lambda)$ : Q-Learning with eligibility trace

1: Initialize all $Q(s,a) = 0$, $e(s,a) = 0$
2: for all episodes do
3: while $s_t \neq s_T$ do
4: Choose $a_t$ from possible actions
5: $r_{t+1} = axp + br_o + kC_{ms}$
6: $e(s_t,a_t) = 1$
7: for all s,a: do
8: $Q(s,a) = Q(s,a) + \eta \delta e(s,a)$
9: $e(s,a) = \gamma \lambda e(s,a)$
10: end for
11: $t = t + 1$
12: end while
13: end for

The eligibility trace implementation shown in algorithm 4 is the naive $Q(\lambda)$ described in [23]. The Sarsa algorithm with eligibility trace, $Sarsa(\lambda)$, will also be implemented.

4.4.3 Serial Sections

We define a serial section as a segment of code that can only be executed non-speculatively with all the other cores in an idle state. Let us define a state that contains only the sequence of instructions in the serial section $S_{ser}$. The only action that can be taken in $S_{ser}$ is the null action. Hence, the action taken in $S_{ser}$ does not depend on the previous state. Similarly, the actions taken in the following state do not depend on the action in $S_{ser}$. This represents a break in the markov decision process that we can exploit. Given that that the rate of convergence is directly related to the number of states, it is important that we
<table>
<thead>
<tr>
<th>Name</th>
<th>Learning Algorithm</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>Q-Learning</td>
<td>uses optimistic initial values</td>
</tr>
<tr>
<td>Sarsa(λ)</td>
<td>Sarsa with eligibility trace</td>
<td>uses optimistic initial values</td>
</tr>
<tr>
<td>Q(λ)</td>
<td>Q-Learning with eligibility trace</td>
<td>uses optimistic initial values</td>
</tr>
<tr>
<td>BaseS</td>
<td>Q-Learning</td>
<td>includes serial section optimization</td>
</tr>
<tr>
<td>SarsaS(λ)</td>
<td>Sarsa with eligibility trace</td>
<td>includes serial section optimization</td>
</tr>
<tr>
<td>Qs(λ)</td>
<td>Q-Learning with eligibility trace</td>
<td>includes serial section optimization</td>
</tr>
<tr>
<td>Greedy</td>
<td>None</td>
<td>spawns threads at spawn points as long as there is an available core</td>
</tr>
<tr>
<td>JaTLS</td>
<td>None</td>
<td>search based approach described in [29]</td>
</tr>
</tbody>
</table>

Table 4.2: Description of the implementations to be evaluated.

reduce the number of states. We can exploit the serial portions of the program to decompose the program into several smaller reinforcement learning problems. The number of episodes required for convergence is then limited by the slowest converging segment. Hence the optimised version will perform no worse than the non-optimised version.

4.5 Evaluation and Testing

We evaluate the performance of the various implementations using trace based simulation. The TLS simulator takes as input trace files for a sequential execution of the selected benchmarks. The simulation is done by taking the spawn points specified in the trace file and executing them in parallel. The read and write sets of the parallel threads are then checked to detect dependencies. If a dependence is detected, the speculative thread is squashed. We instruct the simulator on which threads to create using the policy generated by our learning algorithms. In addition to the reinforcement learning based implementations, we implement two other TLS schemes to evaluate against. The Greedy implementation simply spawns threads at spawn points as long as there is an available core. We also implement the search based approach described in [29]. We have chosen these schemes as they are reasonably straightforward to implement. Table 4.2 shows the implementations to be evaluated. We describe the experiments to be executed in Sections 4.5.1, 4.5.2 and 4.5.3. Finally, we describe the benchmarks and datasets used in Section 4.5.4.

4.5.1 Experiment One

The aim of this experiment is to evaluate the rate of convergence. We will simulate as many episodes as required for convergence using each of the implementations described in table 4.2. The obtained measurements will allow us to evaluate the effectiveness of the proposed optimisations.

4.5.2 Experiment Two

The aim of this experiment is to evaluate the progress of the learning algorithm towards obtaining an optimal solution. We measure the number of cycles saved via speculation after fixed intervals of episodes. This allows us to determine which implementations quickly approach a good speedup. We can also observe if there are any oscillations in the measured speedup.
4.5.3 Experiment Three

The aim of this experiment is to evaluate the performance of our implementation on a full-featured TLS simulator. We select spawn points by simulating until convergence as in experiment one. We configure the TLS simulator used in [22] to create threads at the selected spawn points. We measure the speedup obtained and evaluate the performance of the various techniques.

4.5.4 Benchmarks

We will use the same set of benchmarks used in [22]. The benchmarks were selected from SpecJVM98 [9] and DaCapo [2] benchmark suites. We choose to use these benchmarks as they are industry-standard benchmarks that are widely used for evaluating thread level speculation schemes. Also, the required traces have been captured and made available for evaluation. Table 4.3 describes the selected benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_202_jess</td>
<td>JESS is the Java Expert Shell System is based on NASA’s CLIPS expert shell system. In simplest terms, an expert shell system continuously applies a set of if-then statements, called rules, to a set of data, called the fact list. The benchmark workload solves a set of puzzles.</td>
</tr>
<tr>
<td>_205_raytrace</td>
<td>A ray tracer that works on a scene depicting a dinosaur</td>
</tr>
<tr>
<td>_213_javac</td>
<td>This is the Java compiler from the JDK 1.0.2.</td>
</tr>
<tr>
<td>_222_mpegaudio</td>
<td>This is an application that decompresses audio files that conform to the ISO MPEG Layer-3 audio specification.</td>
</tr>
<tr>
<td>_228_jack</td>
<td>A Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS).</td>
</tr>
<tr>
<td>antlr</td>
<td>Parses one or more grammar files and generates a parser and lexical analyser for each.</td>
</tr>
<tr>
<td>fop</td>
<td>Takes an XSL-FO file, parses it and formats it, generating a PDF file.</td>
</tr>
<tr>
<td>pmd</td>
<td>Analyses a set of Java classes for a range of source code problems</td>
</tr>
</tbody>
</table>

Table 4.3: Benchmarks.

4.6 Work Plan

The work plan followed for this project is shown in Table 4.4. We have divided the work to be done for this project into manageable tasks. These tasks are described in this section.

**Problem Conceptualisation and Literature Review** This involved reviewing the body of work on thread level speculation and reinforcement learning. A problem to be solved was then identified and a solution proposed.

**Proof of Concept Implementation** This involved implementing a simple case version of the proposed solution. The implementation was then used to test new ideas and identify optimisations.

**Background Report** This involves the preparation and submission of this report. The report contains the scope of work, literature review, methodology and the evaluation and testing methods.
Multi-core Implementation This is the second implementation task. It involves extending the proof of concept implementation to support emulation on more than two cores. All further implementation depend on this task.

Learning Algorithms Implementation This task involves the implementation of the learning algorithms and optimisations identified in Chapter 4. This is the final implementation task.

Testing and Evaluation We test the implementations and evaluate performance using the specified benchmarks.

Final Report This involves preparation of the thesis. The thesis contains a complete report of the research carried out as part of this project. The results of the evaluation as well as descriptions of the implementations done will be included. Any insights gained from the project will also be discussed. Finally, ideas that can be explored as future work will be identified. The submission of the thesis marks the end of the project.

<table>
<thead>
<tr>
<th>Name</th>
<th>Start</th>
<th>Finish</th>
<th>Duration</th>
<th>Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Conceptualisation and Literature Review</td>
<td>Feb 2</td>
<td>Apr 23</td>
<td>59d</td>
<td>100%</td>
</tr>
<tr>
<td>Proof of Concept Implementation</td>
<td>Mar 29</td>
<td>Apr 23</td>
<td>20d</td>
<td>100%</td>
</tr>
<tr>
<td>Background Report</td>
<td>Apr 26</td>
<td>May 14</td>
<td>15d</td>
<td>100%</td>
</tr>
<tr>
<td>Multicore Implementation</td>
<td>May 31</td>
<td>Jun 11</td>
<td>10d</td>
<td>0%</td>
</tr>
<tr>
<td>Learning Algorithms Implementation</td>
<td>Jun 14</td>
<td>Aug 6</td>
<td>40d</td>
<td>0%</td>
</tr>
<tr>
<td>Testing and Evaluation</td>
<td>Aug 9</td>
<td>Aug 27</td>
<td>15d</td>
<td>0%</td>
</tr>
<tr>
<td>Final Report</td>
<td>Aug 2d</td>
<td>Sep 3</td>
<td>25d</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 4.4: Work Plan.
Appendix A

Trace File Specification

A.1 Document Type Definition

```xml
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE trace [ 
  <!ELEMENT trace ( DI)*> 
  <!ATTLIST trace 
    id CDATA #IMPLIED > 
] 
<!ELEMENT DI ( rv|StopC|WS|RS|DI|StartC)*> 
<!ATTLIST DI 
  kind CDATA #IMPLIED 
  id CDATA #IMPLIED > 
<!ELEMENT StartC EMPTY> 
<!ATTLIST StartC 
  cyc CDATA #IMPLIED > 
<!ELEMENT RS (mo)*> 
<!ELEMENT WS (mo)*> 
<!ELEMENT StopC EMPTY> 
```

<!-- DI denotes a spawn point. It may be a method call or a loop iteration. -->

<!-- StartC specifies the start cycle of the method call or loop iteration. -->

<!-- RS specifies the read set of the method call or loop iteration. It contains a set of memory operation elements. -->

<!-- WS specifies the read set of the method call or loop iteration. It contains a set of memory operation elements. -->

<!-- StopC specifies the stop cycle of the method call or loop iteration. -->
A.2 Proof of Concept Trace File

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--
Document : reference.xml
Created on : March 30, 2010, 10:51 PM
Author : onyunga
Description:
 Proof of concept trace file.
  Specifies the execution of a simple program
  with 4 method calls.
-->
<trace id="0">
  <DI id="0" kind="0">
    <StartC cyc="0"></StartC>
  </DI>
  <DI id="1" kind="0">
    <StartC cyc="1"></StartC>
  </DI>
  <RS>
  </RS>
  <WS></WS>
  <StopC cyc="6"></StopC>
 </DI>
  <DI id="2" kind="0">
    <StartC cyc="7"></StartC>
  </DI>
  <RS>
  </RS>
  <WS>
  </WS>
  <StopC cyc="14"></StopC>
 </DI>
  <DI id="3" kind="0">
    <StartC cyc="15"></StartC>
  </DI>
  <RS>
  </RS>
  <WS>
  </WS>
  <StopC cyc="23"></StopC>
 </DI>
  <DI id="4" kind="0">
```
<StartC cyc="24"></StartC>
<RS></RS>
<WS></WS>
<StopC cyc="2C"></StopC>
</DI>
<RS></RS>
<WS></WS>
<StopC cyc="3A"></StopC>
</DI>
</trace>
Bibliography


