Dynamic Algorithm Selection Using Reinforcement Learning

Warren Armstrong* Peter Christen Eric McCreath Alistair P Rendell
Department of Computer Science,
College of Engineering and Computer Science,
The Australian National University,
Canberra ACT 0200, Australia
{first name.surname}@anu.edu.au

Abstract

It is often the case that many algorithms exist to solve a single problem, each possessing different performance characteristics. The usual approach in this situation is to manually select the algorithm which has the best average performance. However, this strategy has drawbacks in cases where the optimal algorithm changes during an invocation of the program, in response to changes in the program's state and the computational environment. This paper presents a prototype tool that uses reinforcement learning to guide algorithm selection at runtime, matching the algorithm used to the current state of the computation. The tool is applied to a simulation similar to those used in some computational chemistry problems. It is shown that the low dimensionality of the problem enables the optimal choice of algorithm to be determined quickly, and that the learning system can react rapidly to phase changes in the target program.

1 Introduction

Recent years have seen much research into adaptive optimisation and dynamic compilation [4] [5] [14], crucial to the fast execution of dynamic languages such as Java. However, this research is not immediately applicable to the large body of computational science code written in C, C++ and Fortran.[1] Optimisation of such code is a static affair, perhaps guided by offline profiling. However, this profiling can only inform optimisations that will generalise to future executions. Moreover, such optimisations will last for the whole execution of the program, they can't be changed on the fly. As a program moves through phases, dynamic hot-swapping of optimisations could well be beneficial.

This paper presents initial research into the development of an execution environment which facilitates the adaptive optimisation of statically compiled binaries. Such an environment would have many different actions available, and many rules guiding the application of these actions. Determining these rules based on some arbitrary benchmark is undesirable since it is costly and leads to non-optimal solutions. Rather the environment should be intelligent: it should learn how to optimise the program which is currently running.

There are many decisions to be made when building such an environment. Should the framework be a virtual machine, or should it be a system which sits apart from its target, working on it through binary modification? What kind of data will it receive, for example, will it use measurements from hardware performance counters [8]? How will it analyse large volumes of data quickly enough that the results are relevant - should it use techniques from data stream mining [9]? After the analysis, what sort of optimisations can be applied? When does the overhead of monitoring and optimising outweigh the benefits? Which learning algorithms should be used?

The focus of this paper is on the final question in an attempt to show the feasibility of reinforcement learning techniques within this domain. Results are presented from a prototype environment, using binary modification to implement a single optimisation technique. This technique involves switching between algorithms which perform the same computation in different ways, and hence perform better at different phases in the execution of a program. The target program is a computational kernel whose phase-changing behaviour echoes that found in many computational science programs.

The rest of this paper is organised as follows. Section 2 describes the computational chemistry based task we use to illustrate the viability of this system, as well as back-

*Corresponding author.

[1]That these languages are common in this domain can be seen by looking at the SPEC floating point benchmarks - the 2006 version contains 17 applications from a variety of scientific disciplines, all written in C, C++ or Fortran. [1]

ground information on binary modification and reinforcement learning, as well as a review of related work. Section 3 describes the design and implementation of our system, at both a high and low level of abstraction. Experimental results are presented and discussed in Section 4, while Section 5 details future work.

2 Background

This section provides background for the rest of the paper. It introduces our sample program and the methods used to optimise it: reinforcement learning and binary modification. It also describes our experimental platform, and reviews related work.

2.1 The Sample Program

The sample program used is a simulation, with characteristics that match many computational science programs. In a computational chemistry simulation of an inert gas, like argon, the force exerted on an individual atom is largely determined by the neighbouring atoms. The effect of distant atoms is negligible and hence may be ignored. During the simulation, events occur which change the distribution of the particles. The target simulation we use incorporates both of these aspects. It works with a three-dimensional distribution of point particles. Execution involves a sequence of steps, each consisting of two consecutive passes: an analysis pass and an action pass. The action pass moves all the particles in the field towards the origin. This step is governed by a contraction factor c, which is used as follows:

$$d(p_i, t+1) = cd(p_i, t)$$

where $d(p_i, t)$ is the Euclidean distance from the origin to the ith particle p_i at time t. When the simulation starts, the particles are distributed with a uniform density. The action pass causes them to coalesce into a densely packed cluster.

The analysis pass consists of finding pairs of particles whose separation is less than a prescribed threshold. More precisely, the program looks for the set of pairings P, where

$$P = \{(p_i, p_j) | i \neq j \wedge distance\_between(p_i, p_j) < \delta\}$$

with the threshold δ being a parameter of the problem.

To carry out the analysis pass, two algorithms have been implemented. The first is a loop over all pairs of particles, and therefore scales as $O(n^2)$ where n is the number of the particles in the system. The second algorithm partitions the domain into a series of disjoint cubic boxes of side length δ, placing each particle in a unique cube, and then checking all pairs (p_i, p_j) where both p_i and p_j are in the same cube, or adjoining cubes. Under certain conditions this algorithm scales as $O(n)$, but incurs a large prefactor from distributing the particles over the cubes.

The performance characteristics of these algorithms differ, and which one is faster depends not only on the number of particles, but also on the density of the particle distribution. For a uniform distribution and on all but the smallest problem sizes, the complex algorithm performs better, because the prefactor is outweighed by the reduction in the number of checks performed. For certain values of the threshold and compactness of the distribution, however, every particle falls within the same few cubes, so every particle needs to be checked and the prefactor associated with sorting particles into cubes is not compensated for. Under these conditions, the simple algorithm is fastest.

In the simulation used here it would be expected that the fastest algorithm would change from the cube based approach to the simple algorithm as the particles contract towards the origin. The exact location of that transition is dependent on the problem case, the simulation software and the actual hardware being used to run the simulation.

2.2 Reinforcement Learning

The optimiser makes use of reinforcement learning to guide its actions. Here we provide a brief overview of relevant concepts from this field. Further details can be found in, for example, [15].

Reinforcement learning involves an agent perceiving and acting within an environment. Sensors supply the agent with perceptions of the environment. The agent takes actions, which affect the state of the environment. The agent also receives rewards, which are a measure of how the agent is succeeding at its task. The agent's goal is to take actions such that it will maximise the reward it receives. A policy is used by the agent for determining actions in a particular situation, effectively it is a mapping from situations onto actions.

A defining feature of reinforcement learning is that it is not given explicit training examples like supervised learning, rather, it will explore different actions to search the space of possible policies. Rewards are scaler values that are relative to each other, hence in general, the agent will not know that an action is 'correct' or 'optimal', as would be expected in supervised learning. Moreover, the rewards may be delayed in which case success must be attributed to previous actions.

It is necessary for the agent to experiment, to explore the range of possible actions. An agent which does not explore runs a high risk of being stuck with a suboptimal policy. However, an agent which explores too much will not get the full benefit if it does happen across an optimal (or near optimal) policy. Thus, the agent must balance *exploration* with *exploitation*.

Our problem fits nicely within the reinforcement learning framework. The actions are choices of the best algorithm to use next, rewards are the negation of the execution time, and the environment is a conglomeration of architecture, program and input.

2.3 Dynamic Binary Modification

The system presented depends on the ability to modify a running binary, both to insert sensors and to apply optimisations. DynInst [6] is a suite of tools which provides this ability. Only a subset of its facilities are used, as detailed below.

The first ability harnessed is a mechanism for inserting code into a running binary. DynInst does this by means of *trampolines*. At the insertion point, α, the old instruction, ι, is overwritten by a jump to a handler function. This function invokes the user-designated code, executes ι, and then jumps back to the address $\alpha+1$. DynInst can only do this at certain points, and not at arbitrary addresses. This facility is used to insert "sensors" into the program.

The second use made of DynInst is to change the target of a function call. In other words, it is possible to find an instruction sequence `call foo`, and modify it to be `call bar`. This has some limitations, obviously: Both `foo` and `bar` must share both signatures and calling conventions. Furthermore, any assumptions the compiler has made about the interaction between calling and called functions must hold for both `foo` and `bar`. For example, if the compiler assumes that `foo` does not modify data in some register, and makes use of that assumption, `bar` must also leave that register alone (or at least, appear to do so, when viewed from the calling function).

2.4 Experimental Platform

All experiments were conducted on a 2.8 GHz Pentium 4, with a 1024 KB cache and 512 MB of RAM. The operating system was a vanilla Linux kernel, version 2.6.17 with SMP enabled, running in single user mode. Binary modification was carried out using DynInst 5.0 [2]. The target program was compiled by the GNU C++ compiler *g++*, version 3.4.6.

The flags used were: *-O2 -march=prescott -mfpmath=sse -fno-optimize-sibling-calls*. The first three flags were chosen in order to optimize the emitted code. The first flag, *-O2*, turns on a variety of standard optimisations. This is the second highest optimisation level in g++. Using the highest level (*-O3*) produced a very slight slowdown. The option *-march=prescott* is used to indicate that emitted code may use instructions not available on earlier x86 chips: it frees the instruction selector from the constraints of backwards compatibility. The *-mfpmath=sse* flag is used to enable Streaming SIMD Extensions (SSE), additions to the instruction set which are used to speed up floating point calculations.

The final flag turns off sibling call optimisation, which is enabled by *-O2*. Using sibling call optimisation prevents DynInst from working on the resulting binary. Disabling this optimisation produced no measurable slowdown.

2.5 Related Work

Several approaches have been taken to using machine learning to guide compilation. Long and O'Boyle used instance based learning to select transformations in a Java compiler [13]. Agakov *et al* used various learning techniques to build models, which they used to focus the search for iterative optimisations [3]. Cavazos and O'Boyle used offline logistic regression to learn heuristics for applying optimisations at the granularity of methods [7].

All these works make use of program features, such as the depth of a loop, or the presence of data-flow dependencies, in order to classify a program. In contrast, this work does not attempt to classify algorithms, merely to observe their performance and react accordingly.

Outside the domain of explicit machine learning, much work has gone into adaptive optimisation systems for languages such as Java. The closest in spirit to our work is that of Lau *et al* [11], who propose and implement a *Performance Auditor*. This is an on-line mechanism for evaluating multiple implementations of a method and choosing the fastest one. Their system uses statistical analysis to deal with the fact that each version of the method might be called with different inputs, or at a different stage of execution. This work focuses on the use of different algorithms, instead of judging the effects of various compiler optimisations. This difference also distinguishes this work from more traditional adaptive optimisation research, which tends to describe more effective ways to target a compiler's time-intensive optimisations [5] [14].

3 System

This section presents the system design, first as a high level overview, and then as a detailed discussion of each component.

3.1 High Level Design

Conceptually, there are four components to the optimisation system: the sensors, the actuators, the adjudicator and the controller. These are each explained in more detail below. Figure 1 shows the interactions between the components, as well as the connection between the optimisation system and the target process.

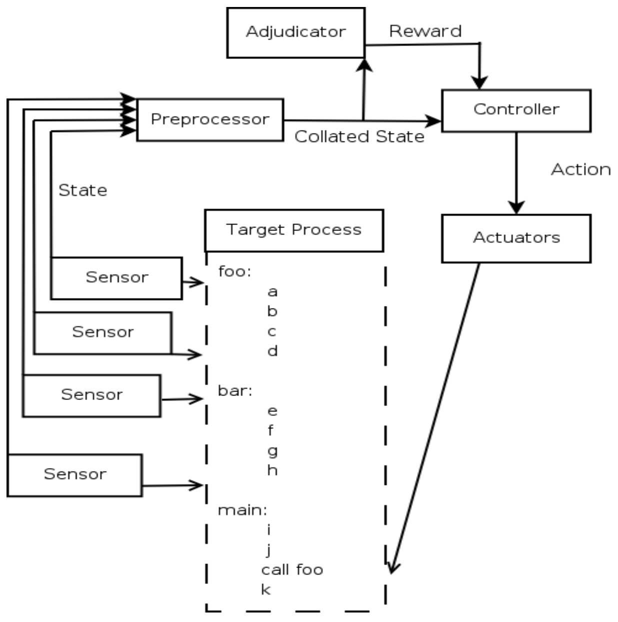

Figure 1. High level design of the optimisation system.

The sensors measure the target program and transmit its state to the controller. In the current system, these sensors are clocks. The actuators are responsible for acting on the agent's environment. Currently, the environment has actuators for inserting function calls, manipulating a call-site, and loading shared libraries into the target's address space. The adjudicator computes the reward signal of the current state, and transmits that to the controller. The controller is responsible for picking the best algorithm for use at a given point in the target program's execution.

3.2 Components

The above framework is very general. In this section, we describe the implementations of each component, as well as their interaction.

The **sensors** are implemented as procedure calls. These calls could be to any procedure, in this paper, they call `gettimeofday`. The calls are made by the target thread, and their results need to be communicated to the optimiser. This is handled by encasing each procedure within a wrapper. The wrapper encodes the result using an agreed protocol, and transmits it to the controller. Currently, the communication is performed using pipes. The wrappers are collected into a shared object library.

All of the **actuators** are built on top of DynInst. The simplest actuator loads a shared object library into the address space of the target. This is used to load the library of sensors. It is implemented as a single call into the DynInst library.

The second actuator inserts procedure calls. This is implemented by searching the process's address space for two procedures, α and β. The search is made by procedure name, and is carried out by DynInst. Once handles to α and β have been found, these handles are passed back to DynInst. DynInst then modifies the code of α to call β as either its first or last instruction - the choice is governed by a flag passed along with the handles. As discussed in Section 2.3, this is carried out using a mechanism known as *trampolines*.

The final actuator is used to change the target of a callsite. The target is a handle to a procedure found by searching, as for the above actuator. DynInst is passed the target and the callsite, and rewrites the call instruction so that it points to the new target.

Communications are provided by a named pipe. Messages are written to the pipe by the sensors, and read by a preprocessor, described next. Each message represents a measurement, and contains the following information:

- the name of the procedure that was measured,
- whether the measurement was at the beginning or the end of the procedure,
- the value that was measured.

The raw data provided over the communications channel is not very useful to the controller. Collating it into an informative representation is done by the **preprocessor**. This task involves matching the starting measurement of invocation i of routine ρ to the ending measurement of the same invocation of the same routine. The summary mechanism uses one stack per routine. Starting measurements are pushed on to the stack, and end measurements are paired with the top of the stack, which is then popped. As each match is made, the difference δ is computed. The preprocessor maintains one list per method, and δ is appended to this list. The set of lists forms the measured state, and is passed on to the adjudicator and the controller.

The **adjudicator** is responsible for computing the reward gained from being in the current state. The reward is computed as the negation of the most recent time difference resulting from using the currently swapped-in algorithm.

In reinforcement learning terms, the **controller** is an ϵ-*greedy on-policy* learning agent, as described in [15]. An ϵ-greedy agent makes a greedy choice in (100 - ϵ) % of selections. The other times, it makes a random choice, in the hope that this will reveal information that brings it closer to an optimum. In other words, it explores in ϵ % of cases, and exploits in the rest.

An on-policy agent is best explained by contrast with its opposite. An *off-policy agent* learns one policy based on actions generated under a different policy. Our system is an on-policy agent because the policy it learns is the same policy which generates the actions learned from.

The job of the controller is to process the data from the sensors and issue commands to the actuators. We have implemented a variety of methods to achieve this, as discussed next.

3.3 Learning Algorithms

In this paper, we have implemented two reinforcement learning algorithms, and evaluated each of them as the controller component. This section describes the algorithms, as well as a theoretical perfect agent by which we will judge them.

In order to compare the speedups achieved by the various algorithms, we consider a perfect algorithm. Such an algorithm will spend no time at all executing fruitless exploration - every time it explores, the option it investigates will be the correct action to choose from that point on.

Our first agent is a simple temporal difference agent, *TD*. It knows of two possible actions, a_{loop} and $a_{partition}$, corresponding to the two algorithms given in Section 2.1. The utilities of these actions are given by $U(a) \leq 0$. The agent works in a series of episodes. In each episode, the agent receives a percept $P = (r, a)$, where r is the reward observed, and a is the action which generated the reward. Note that this may or may not be the same as the last action. On receiving each percept, the agent updates its utility estimate for the relevant action, by using the update rule

$$U(a) = U(a) + \alpha(reward - U(a))$$

In this rule, α is a learning rate parameter, and is usually bounded within the interval $(0, 1]$. The agent then chooses the desired action, in accordance with the ϵ-greedy strategy, where a random action is chosen in $100 - \epsilon\%$ of cases. In all others, the action which maximises U is chosen. The results reported later were obtained with $\epsilon = 0.96$.

Our second agent, referred to as the regression agent, plots the reward signal against time for each option, and uses linear regression via a least squares fit to determine a straight line through each set of points. From this, the agent can predict which algorithm will be best at a given point in the future, and hence which algorithm would be best to run at the current time. Frequency of exploration is thus less important - more important is the length of the exploration phase. The longer the phase, the more datapoints the agent has to work with. This should lead to improved accuracy, but this is not always the case: when the environment undergoes a phase change, the earlier data should be discarded in order to avoid confusing the model. If this data was on the suboptimal algorithm, then this was wasted exploration. It should also be noted that this phase-changing aspect means that prediction does not totally do away with the reliance on exploration frequency. An agent with low exploration frequency could predict with out of date data, and incur a substantial penalty.

4 Results

This section presents and discusses the performance of the optimisation system. The system was tested on two configurations of the simulation. These configurations differed in the number of particles being simulated, the volume over which the particles are initially distributed, and the number of simulation steps. The details of the problems are given in Table 1.

Size	Particles	Side Length[2]	Simulation Steps
Small	8000	26	100
Large	10000	30	120

Table 1. A list of problems, which consist of parameters used for the simulation.

4.1 Static Algorithm Choice

The first set of results are from the simulation running without the benefit of the optimisation system. The simulation consists of a number of runs. The time required by each algorithm to complete one run has been plotted in Figure 2. The horizontal axis of this plot measures simulation steps, while the vertical axis shows CPU time elapsed for a single step.

The crossover point can clearly be seen, and represents the point at which the optimal choice of algorithm changes. In the rest of the paper, the term *phase* is used to denote a group of simulation steps during which the optimal algorithm choice remains constant. Thus, in Figure 2, there are two phases.

The system performs multiple runs of each simulation. Between each run, the system is reset to a uniform distribution of particles. This results in another phase change. Figure 3 shows three runs of the system, and the associated six phase changes.

[2]The particles are distributed over a cubic volume with side length given by this column.

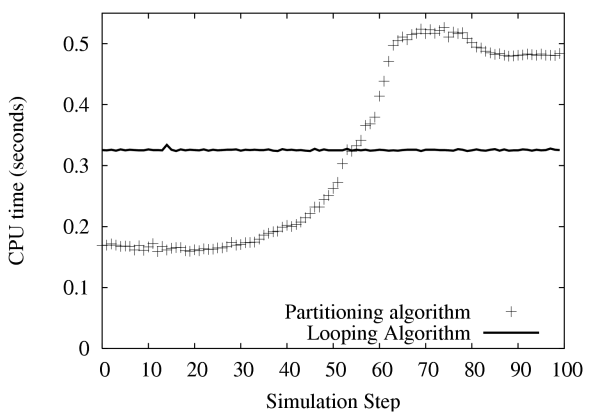

Figure 2. One simulation run, showing a single phase change.

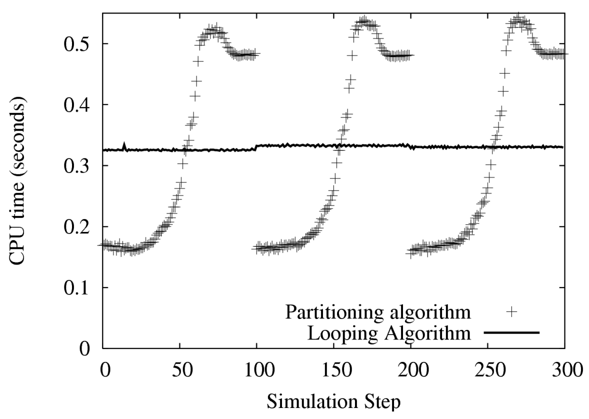

Figure 3. Three simulation runs, showing six phase changes.

4.2 Perfect Agent

This section establishes the metric by which the agents will be measured. This metric is the reduction in execution time achieved by a perfect agent, one which has access to the knowledge of which choice should be made at each point in time. Such an agent does not make suboptimal choices. The performance of this agent is thus unattainable by any of the agents discussed below - it serves as an upper bound on their performance. Table 2 presents the results for using each algorithm exclusively (labelled *Loop* and *Partition*), and for optimisation by a perfect system (labelled *Perfect*). From this data, it can be seen that the perfect agent achieves a reduction of the total CPU time used of between 21% and 22%.

4.3 Agent Performance

The first agent examined is the *TD* agent. Table 2 shows results for this agent run with two different configurations. These are labelled as *TD T = 5* and *TD T = 25*, where T represents the inverse of the degree of exploration. Larger values of T reduce the agent's affinity for exploration. This agent is found to reduce the time taken to execute the program by 6% to 10%, depending on the exact value of T.

The temporal difference agent achieves just over half of the speedup achieved by the perfect agent. There are several factors contributing to this. First, it cannot predict changes - its internal model of the time each algorithm will take is just $time = c$, where c is a constant. This means it must experience changes in order to react to them. Thus, it will very frequently make at least one suboptimal choice per phase change. The second impediment to this agent reaching perfection is that it must make the occasional non-greedy choice, lest it get stuck in a rut. For example, during one of the phases where the loop algorithm is faster, the agent's model might look like: $time_{loop} = 0.8, time_{partition} = 0.9$. A greedy agent would always use the loop algorithm from that point forward, getting better and better estimates of its performance, but not exploring the performance of the partition algorithm, which at some point will improve markedly.

The performance of the regression agent is also sensitive to the value of T. Results, labelled *Regression, T = 5* and *Regression, T = 25*, in Table 2, show that this agent reduces execution time by between eight and ten percent. Despite its predictive ability, this agent does not perform better than the temporal difference agent in all cases. This is due to the way the two algorithms behave for a given problem size over the lifetime of a simulation: their performances can be very accurately modelled by the temporal difference agent's model. The only significant non-constant periods are when the phase-changes take place, and these are too brief to have much effect.

To extract the maximum speedup from each agent, it is necessary to tune several parameters - for these results, only T has been varied. The optimal value of the parameters will, in general, depend not only on the nature of the learner, but on the detailed performance characteristics of the algorithms being swapped, as well as the architecture the system has been deployed on.

4.4 Learning Speed

This section examines how quickly the agents can learn an optimal policy, when that policy does not change over time. A total of 1000 runs were made, each consisting of 50 epochs, an epoch being one cycle of perception, analysis

Algorithm	Time		
	Raw Mean	Standard Error	Normalised Mean
Small Problem			
Loop	164.76	0.02	1.02
Partition	162.30	0.08	1.00
Perfect	126.95	0.04	0.78
TD T = 5	145.91	0.97	0.90
TD T = 25	152.00	1.36	0.94
Regression T = 5	147.56	1.89	0.91
Regression T = 25	149.07	0.71	0.92
Large Problem			
Loop	309.18	0.04	1.00
Partition	320.93	0.24	1.04
Perfect	244.79	0.08	0.79
TD T = 5	278.45	3.12	0.90
TD T = 25	287.20	6.78	0.93
Regression T = 5	279.54	3.24	0.90
Regression T = 25	285.56	6.39	0.92

Table 2. Total CPU time in seconds for all configurations. These data are taken from twenty measurements. The values in the last column are normalised against the fastest static algorithm. The resolution of the timer was 0.01 seconds.

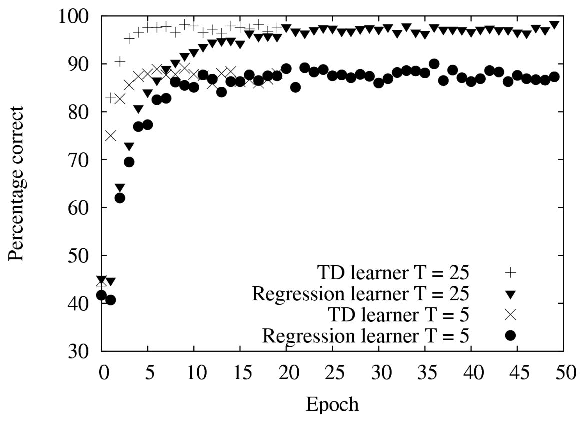

Figure 4. Epochs required to converge to optimal policy, showing the percentage of correct decisions per epoch, averaged over 1000 repetitions.

and action undertaken by the agent. Each run began with a different number and configuration of particles, where the number of particles n was calculated as:

$$n = 1000 + R(2) \times 3000 + 50 - R(101)$$

where, for any integer p, $R(p)$ returns a pseudo-random integer $0 \leq r < p$. This formulation gives sizes in two bands: $n \in [950, 1050]$ and $n \in [3950, 4050]$. These bands were employed so that each algorithm had an equal chance of being the optimal algorithm on any given run: in the first case, the looping algorithm was quicker, while the partitioning algorithm was optimal in the second case. The results are plotted in Figure 4 for two different exploitation periods T. A random choice is made if $R(T) = 0$, thus, higher values of T denote less frequent exploration.

It can be seen that the temporal difference agent converges quickly, taking around five epochs to discover the optimal policy. The regression agent, on the other hand, requires more time, around twenty epochs. In each case, smaller values of T result in faster convergence to the optimal policy, because of the frequency with which alternatives can be checked and discarded. In both cases, the average value is about what would be theoretically predicted: $(100 - \epsilon)\%$, where $\epsilon = \frac{1}{T}$. This rapid convergence is due to the simple nature of the learning task, however, it demonstrates that a reinforcement learning agent in this kind of domain would not have to undertake large amounts of exploration in order to learn good policies.

5 Future Work

The work presented here involves a prototype environment for dynamic algorithm selection; there are many issues to be addressed in future work.

One issue is that the system has so far been applied only to a target program designed to exhibit the kinds of behaviour found in scientific applications. It is a necessary next step to apply future versions of this system to actual scientific codes.

Another issue is the nature of the mechanism which allows performance monitoring and program modification. This is currently done using a binary editing tool [6]. An alternative approach would be to use a virtual machine, such as the Low Level Virtual Machine (LLVM) [10]. This would impose greater runtime overheads on the system, and would require re-compilation into the virtual machine's input language. The benefits of this approach come from the facilities that systems like LLVM offer, specifically, the ability to adaptively recompile code at runtime. While theoretically possible with a system like DynInst, it would be a much tougher task.

Finally, it would be beneficial to exploit the many other opportunities for learning, which fall into two areas: learning parameters like cache blocking factors which affect the target process, and learning values for variables such as the exploration rate, which guide the optimiser. The second area is at least as important as the first, given that the optimal combination of parameters will likely vary with the architecture, the environment and the target program.

6 Conclusion

During the early days of computing execution cycles were at a premium. This was demonstrated by the opposition of von Neumann to the use of a computer to translate assembly language into machine code:

> He [Donald Gillies, a student of von Neumann] took time out to build an assembler, but when von Neumann found out about he was very angry, saying (paraphrased), "It is a waste of a valuable scientific computing instrument to use it to do clerical work."[12]

A paradigm shift has clearly taken place in the way we view the cost and function of a computing device. Our research aims to continue this progression by off-loading the selection of functionally equivalent algorithms from the human operator or programmer to the computer.

Scientists who run large simulations on super-computers spend considerable time configuring the software to produce results efficiently. This configuration is time consuming yet important as it may impact significantly on the total execution time. In addition, programmers will spend considerable time selecting and tuning the algorithms used by a single program. Often, neither approach will produce the best solution. Moreover, there may not be a single optimal solution as the most efficient algorithm may change during execution. Clearly hybrid algorithms could be constructed, however this would be complex to do as it would require an exact model of the problem configuration and execution environment. In this paper we have demonstrated the viability of using simple reinforcement learning approaches to dynamically select the algorithm that performs best given a particular execution environment. The solution we present is robust as it focuses on selecting the algorithm that gives the best execution time during the actual use of the program. Thus the approach does not need to model the complexities of a particular algorithm, executing on a particular architecture, and within a particular problem configuration.

Many reinforcement learning domains require numerous runs for a good policy to be converged upon, however, given the restricted nature of the domain we only require a few runs to find a good policy. This is encouraging as the overhead of introducing such an approach is small in comparison to the possible benefits of reducing human effort along with improved execution performance.

References

[1] http://www.spec.org/cpu2006/CFP2006/.

[2] http://www.dyninst.org/rel5.0/srcDist_v5.0.tar.gz.

[3] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimisation in virtual machines. In *Proceedings of the IEEE*, volume 92, pages 449–466, 2005.

[5] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of java. pages 111–129, 2002.

[6] B. R. Buck and J. K. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14, 2000.

[7] J. Cavazos and M. O'Boyle. Method-specific dynamic compilation using logistic regression. Accepted for OOPSLA 2006.

[8] A. Czezowski and P. Christen. How fast is '-fast'? Performance analysis of KDD applications using hardware performance counters on UltraSPARC-III. In *Proceedings of the Australasian Data Mining Workshop*, December 2002. pp. 117–129.

[9] L. Golab and M. T. Ozsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.

[10] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. 2004. http://www.llvm.org.

[11] J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: using hot optimizations without getting burned. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 239–251, New York, NY, USA, 2006. ACM Press.

[12] J. A. N. Lee. John Louis von Neumann, 2002. http://ei.cs.vt.edu/ history/VonNeumann.html.

[13] S. Long and M. O'Boyle. Adaptive java optimisation using instance-based learning. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 237–246, New York, NY, USA, 2004. ACM Press.

[14] M. Paleczny, C. A. Vick, and C. Click. The java hotspot^{TM} server compiler. In *Java^{TM} Virtual Machine Research and Technology Symposium*. USENIX, 2001.

[15] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.