

Swift: A flexible framework for runtime performance tuning

Warren Armstrong

August 2011

A thesis submitted for the degree of Doctor of Philosophy
of the Australian National University



Declaration

The work in this thesis is my own except where otherwise stated. I enjoyed many interesting discussions with members of the Computer Systems Research Group, past and present, particularly with Joseph Antony, Arlio Daler, Daniel Fraipont, Eric McCraith, Pete James, Josh Milthorpe and Alistair Rendell.

Jim Cui, Pete James, Josh Milthorpe and Jin Wong read parts of this thesis and provided useful feedback, for which they have my thanks. I also wish to thank the staff of the Technical Services Group for replacing computers when I broke them out, and the administrative staff for their assistance over the years.

My supervisors, Alistair Rendell, Eric McCraith and Peter Christen have provided much help and guidance over the years, for which I am extremely grateful. Alistair also introduced me to Quintessence Labs, the company for which I have been working over the final period of my candidature. It is a fantastic environment and I would like to thank my colleagues: Jim Cui, David Chan, Kent Chong, Brazen Kuchro, Andrew Lamm, Michael McQuinn, Michael Reharier, Clark O'Neil, Scott Raschleigh, Wendy Reed, Michael Stevens and Jin Wong, along with the departed Neil Daler. I would also like to thank Quintessence Labs for providing the Xeon-based servers used in some experiments.

Finally, I wish to thank my family for their endless love and support, without which I would have had no hope of completing this thesis.

W. Armstrong
Warren Armstrong

Acknowledgements

My time at ANU has been enriched by many people. I have enjoyed many interesting discussions with members of the Computer Systems Research Group, past and present, particularly with Joseph Antony, Arrin Daley, Daniel Frampton, Eric McCreath, Pete Janes, Josh Milthorpe and Alistair Rendell.

Jie Cai, Pete Janes, Josh Milthorpe and Jin Wong read parts of this thesis and provided useful feedback, for which they have my thanks. I also wish to thank the staff of the Technical Services Group for replacing computers when I burn them out, and the administrative staff for their assistance over the years.

My supervisors, Alistair Rendell, Eric McCreath and Peter Christen have provided much help and guidance over the years, for which I am extremely grateful. Alistair also introduced me to QuintessenceLabs, the company for which I have been working over the final period of my candidature. It is a fantastic environment and I would like to thank my colleagues Jason Chapman, Raymond Chan, Kenli Chong, Russel Koehne, Andrew Lance, John Leiseboer, Milind Neharker, Chris O'Neil, Scott Rashleigh, Wendy Read, Vikram Sharma, Michael Stevens and Jin Wong, along with the departed Neil Baker. I would also like to thank QuintessenceLabs for providing the Xeon-based servers used in some experiments.

Finally, I wish to thank my family for their endless love and support, without which I would have had no hope of completing this thesis.

Abstract

Many computational kernels require extensive tuning to achieve optimal performance. The tuning process for some kernels must take into account the architecture on which they are executing, the input data that they are processing and the changing levels of contention for limited system resources. Maintaining performance in the face of such fluctuating influences requires kernels to continuously adapt.

Swift is a software tool that performs this adaptation. It can be applied to many different target applications. Such an approach is more efficient than developing application-specific code for continuous tuning. Swift performs controlled experiments to gauge the performance of the target application. Results from these experiments are used to guide the execution of the target application. Swift performs periodic re-evaluations of the application and updates the application if environmental conditions or the internal state of the application have caused performance to degrade. The frequency of evaluation is scaled with its likely necessity - Swift performs few evaluations until it detects a potential performance degradation, at which point more detailed assessments are conducted.

Swift is constructed using the DynInst library to modify and tune the executing kernel. The effectiveness of Swift depends on the computational expense of utilising this library. A suite of micro-benchmarks was developed to measure this expense. These benchmarks are not specific to Swift, and could guide the design of future DynInst-enabled applications.

Swift was applied to tune sparse matrix-vector multiplication kernels. Tuning such kernels requires selecting a matrix storage format and the associated multiplication algorithm. The choice of format depends on the characteristics of the matrix being multiplied as well as on prevailing system conditions and the number of multiplications being conducted.

Swift was evaluated using both simulated environments and physical hardware. Simulated evaluation demonstrated that Swift could correctly select the

Contents

Acknowledgements	v
Abstract	vii
1 Introduction	1
1.1 The case for improved performance	2
1.2 The motivation for runtime tuning	3
1.2.1 Applying reinforcement learning to runtime tuning	4
1.3 Motivation for using binary modification	5
1.3.1 Tools for runtime binary modification	6
1.4 The importance of tuning sparse matrix-vector multiplication . .	8
1.5 Publications	8
1.6 Contribution of this work	8
1.7 Swift: A system for runtime tuning	9
1.8 Thesis Structure	10
2 Background Information and Related Work	13
2.1 Notation and terminology	13
2.1.1 Pseudocode	14
2.2 Reinforcement Learning	15
2.2.1 Action selection	16
2.2.2 Generalisation	18
2.2.3 Q-learning: an overview	19
2.3 DynInst	21
2.3.1 The Ptrace System Call	22
2.3.2 Inserting and Removing Code	22
2.3.3 Reading and Writing Variables	27
2.3.4 Procedure Switching	27
2.4 Sparse Matrix Background	27

2.4.1	The utility of sparse matrices	27
2.4.2	Review of different sparse matrix storage formats	29
2.4.3	Influences on Sparse Matrix-Vector Multiplication Speed	34
2.5	Related work	35
2.5.1	Automatic Tuning	35
2.5.1.1	The effectiveness of automatic tuning	37
2.5.1.2	Generality of autotuning	40
2.5.1.3	The timing of tuning	41
2.5.1.4	Tuning mechanism	43
2.5.2	DynInst	45
3	The Learning Agent	47
3.1	Introduction	47
3.2	Design of the learning algorithm	48
3.2.1	A framework for discussion	48
3.2.2	Overview	49
3.2.3	Generalisation	49
3.2.4	Handling action requests	51
3.2.5	Experiment Scheduler	53
3.2.6	Iteration count estimation	55
3.2.6.1	Global estimation	55
3.2.6.2	Windowed estimation	59
3.2.7	Parameter tuning	59
3.2.7.1	Weight parameters $w_{initial}$ and w_{min}	60
3.2.7.2	The reset threshold $min_{\mu weight}$	63
3.3	Evaluation Methodology	64
3.3.1	Simulator Databases	66
3.3.2	Tuning simulator	66
3.3.3	Static simulator	67
3.4	Results and Discussion	67
3.4.1	Test problem	71
3.4.2	Experimental platforms	72
3.4.3	Results	72
3.4.3.1	A beacon of utility	72
3.4.3.2	Varying iteration counts	73
3.4.3.3	The necessity of careful matrix selection	76
3.4.3.4	Perceptual aliasing as a dynamic environment	80

3.5	Conclusions	84
4	Performance Characterisation of the DynInst Library	85
4.1	Experimental Methods	87
4.1.1	Gathering Data	87
4.1.1.1	Description of the measurement interface	87
4.1.1.2	Costs of using the measurement interface	88
4.1.2	Processing the Data	89
4.1.3	Experimental Setup	90
4.2	Benchmark Design	91
4.2.1	Overhead of inserted code	91
4.2.2	Inserting and Removing code	97
4.2.3	Variable Access	97
4.2.4	Switching Procedures	103
4.2.5	Ptrace	105
4.3	Results and Discussion	105
4.3.1	Overhead of inserted code	107
4.3.2	Cost of Inserting Code	110
4.3.3	Variable Access	111
4.3.4	Function Swapping	117
4.3.5	Ptrace	121
4.4	Conclusion	121
5	The tuning driver and full system evaluation	125
5.1	The design of the tuning driver	126
5.1.1	Configuration	126
5.1.1.1	The original function	126
5.1.1.2	The auxiliary libraries	128
5.1.1.3	Binder specifications	128
5.1.1.4	Adaptor specifications	128
5.1.1.5	Lookup specifications	129
5.1.1.6	Allocation specifications	129
5.1.1.7	Composed variable specifications	129
5.1.1.8	Conversion patterns	130
5.1.2	Locating optimisation sites	130
5.1.3	Injected code: structure	133
5.1.3.1	Pre-loop block	136

5.1.3.2	Pre-call block	136
5.1.3.3	Post-loop block	138
5.1.4	Injected code: behaviour	138
5.2	Auxiliary Libraries	141
5.2.1	Sparse Matrix Format Libraries	141
5.2.2	Remote agent library	145
5.2.3	Synchronisation library	145
5.2.4	Matrix Analysis Library	146
5.3	Experimental Evaluation	148
5.3.1	Limitations	149
5.3.2	A synthetic benchmark	149
5.3.3	Sparse Matrix Format Selection	156
5.4	Conclusions	156
6	Conclusions and Future Work	161
6.1	The architecture of Swift	162
6.2	The learning agent	162
6.3	Future Work	163
A	Performance Comparison of Binary Modification Tools	167
B	Source code availability of scientific programs	173
C	Experimental Platforms	177
D	Communication costs between the agent and the tuned application	181
E	Experimental Parameters	185
	Bibliography	192

List of Tables

3.1	Overview of experimental platforms	72
3.2	Parameters for the first experiment	73
3.3	Characteristics of the matrices used in the first experiment.	75
3.4	The effect of μ_{iter}	76
3.5	Number of matrices for which different formats were optimal.	78
3.6	Aliased matrix sets.	81
3.7	Agent configuration for simulated perceptual aliasing.	82
4.1	Models which predict the total interface overhead.	90
4.2	Benchmarks that measure the overhead of executing inserted code.	96
4.3	Overhead of inserted code on x86	108
4.4	Overhead of inserted code x86_64	109
4.5	The cost of inserting code on the x86 platform.	112
4.6	The cost of inserting code on the x86 platform.	113
4.7	The cost of inserting code on x86_64	115
4.8	The cost of inserting code on x86_64	116
4.9	Reading and writing to global variables on the x86 platform.	118
4.10	Reading and writing to global variables on the x86_64 platform.	119
4.11	Parameters for a linear model.	120
4.12	Single swap cost between on x86 and x86_64	122
4.13	The cost of <code>ptrace</code> reading or writing one word of data.	123
5.1	Properties of the matrices used in Section 5.3.3.	156
A.1	Execution time (measured in seconds) for the Core2 platform.	170
B.1	Software with available source code.	175
B.2	Software without available source code.	176
D.1	Overhead of communication.	183

List of Figures

- 1.1 An example of binary cost tuning. 7
- 1.2 The components of the Swift system, its target and environment. 10

- 2.1 A basic reinforcement-learning setup. 16
- 2.2 The requirement for exploration 17
- 2.3 An example of tile-coding for a single attribute. 19
- 2.4 How DynInst fits into a complete optimisation system. 21
- 2.5 Internal organisation of DynInst’s trampoline mechanism. 23
- 2.6 The manipulation of global variables. 28
- 2.7 Representing an example matrix in different storage formats. . . . 30

- 3.1 The communications pattern when tuning an application. 50
- 3.2 Request flow in the learning agent 52
- 3.3 The steps undertaken by the experiment scheduler. 56
- 3.4 Different models for loop iteration counts. 57
- 3.5 Applying global estimation to stratified data. 58
- 3.6 The dependence on the chance of exploration on $w_{initial}$ 61
- 3.7 Reaction speed 62
- 3.8 Reaction time influenced by action count 63
- 3.9 Critical values for the scaling constant $c = \frac{\psi}{(\psi-1)+\rho}$ 65
- 3.10 Dynamic tuning outperforming static tuning 74
- 3.11 Amortisation of conversion costs. 77
- 3.12 The variance of dynamically tuned performance 79
- 3.13 Performance of Swift on matrices from *Set 1*. 82
- 3.14 Performance of Swift on matrices from *Set 2*. 83
- 3.15 Performance of Swift on matrices from *Set 1* and *Set 2*. 83

- 4.1 Performance counters for code insertion 114
- 4.2 Variable manipulation costs 117

4.3	The costs incurred by the ptrace thread.	120
5.1	The conversion of matrices to graph sets for region detection. . . .	144
5.2	Per-iteration costs on the synthetic problem.	153
5.3	Comparison of cumulative execution time on the synthetic problem.	154
5.4	Worst ratio of cumulative execution time.	155
5.5	Overhead imposed by Swift while tuning the synthetic problem. .	155
5.6	Cost per phase of Algorithm 5.14.	157
5.7		158
A.1	Performance comparison of binary modification tools	171

List of Algorithms

2.1	Sparse matrix-vector product for matrices in coordinate format. . .	31
2.2	Sparse matrix-vector product for matrices stored in CSR format. .	32
2.3	Sparse matrix-vector product for matrices stored in BCSR format.	33
2.4	The <i>push</i> style of parametrised dispatch routine	44
2.5	The <i>pull</i> style of parameterised dispatch routine	44
2.6	An example computational loop	45
3.1	Example of the target program class.	48
3.2	Example of the target program class.	49
3.3	Algorithm for the tuning simulator.	68
3.4	The simulate operation, invoked by the tuning simulator.	69
3.5	Algorithm for the static simulator.	70
4.1	Placement of performance counters to measure the ptrace thread.	89
4.2	Algorithm to determine measurement cost variance	89
4.3	Example code for measuring benchmark mechanism overhead. . . .	90
4.4	<i>Target_0</i> , used to measure the overhead of inserted code.	92
4.5	The application <i>mod_0</i> , which modifies the program <i>target_0</i>	93
4.6	Deducting the cost of functions called from within snippets. . . .	95
4.7	<i>Target_1</i> , used to benchmark the cost of inserting or removing code.	98
4.8	<i>Mod_1a</i>	99
4.9	<i>Mod_1b</i>	100
4.10	<i>Target_2</i> , used to measure the cost of variable access.	101
4.11	The application <i>mod_2</i>	102
4.12	<i>Target_3</i> , which is used to measure the cost of function swapping. .	103
4.13	<i>Mod_3</i> which redirects callsites in a process running <i>target_3</i>	104
4.14	Benchmark used to measure ptrace	106
5.1	Logic of the tuning driver.	127

5.2	Algorithm for detecting optimisation sites.	131
5.3	Algorithm to find the loop around an optimisation site.	132
5.4	The main loop of the target application.	135
5.5	An example stub function	136
5.6	Pre-loop block.	136
5.7	Pre-call block.	139
5.8	Post-loop cleanup.	140
5.9	Extra BCSR logic located before the multiplication	142
5.10	Extra BCSR logic located after the multiplication	143
5.11	Analysis data structure setup.	147
5.12	Mean neighbour count analysis.	148
5.13	The target application of the synthetic benchmark	150
5.14	Sparse matrix-vector benchmark.	157
A.1	The matrix-vector multiplication benchmarks <i>csr</i> and <i>bcsr</i>	169
A.2	The rgen benchmark.	169

Chapter 1

Introduction

There exist important computational kernels that achieve peak performance only when adapted to the properties of the executing architecture. Several systems (such as ATLAS [1, 2], FFTW [3] and SPIRAL [4]) have been developed to automatically tune these kernels to suit different architectures. More challenging are kernels whose performance varies with the data they are processing and with the behaviour of other programs executing on the same system. Projects such as SPARSITY [5], OSKI [6, 7] and AcCLES [8] have the ability to tune such kernels. This thesis describes another such project, a system named Swift. Swift extends previous efforts by learning tuning rules online rather than offline, and by not requiring source code changes in the kernel being tuned.

Most tuning systems aim to improve the execution speed of a certain class of programs. For many systems (including Swift) this class consists of kernels used to answer scientific questions. Considerable intellectual effort is invested into developing such tuning systems, to the benefit of those who use these kernels. For scientific kernels, improved performance brings new problems into the realm of the feasible, allows for more accurate investigation of old problems, reduces the financial cost of employing computer simulation to aid scientific discovery, *et cetera*. These points are further developed in Section 1.1, which justifies expending effort on improving the performance of scientific kernels.

Achieving the broad goal of improving performance requires making specific choices. When designing and applying the Swift system, decisions were made to pursue runtime optimisation using binary modification and to apply the results to selecting the storage formats of sparse matrices. These decisions are discussed in subsequent sections (Sections 1.2, 1.3 and 1.4).

1.1 The case for improved performance

The Swift system is designed to reduce the execution time of certain scientific programs on existing commodity hardware platforms. Speeding up scientific codes leads to several desirable consequences including:

- The scope of computational investigations can be expanded.
- More alternatives can be investigated in the old scope.
- Computational resources can be more widely afforded.

The scope of a computational simulation can be expanded in several ways: by simulating a greater number of timesteps, by investigating the simulated system with greater accuracy, and by simulating larger systems.

The useful length of a simulation depends on the system being studied. For a molecular dynamics simulation of the atoms in a protein and its immediate environment, many interesting activities occur on a timescale of milliseconds [9]. Specialised hardware can achieve a simulation rate of $16.4\mu\text{s}$ per day, while software on commodity hardware has only reached 400ns per day [9]. A millisecond would take just over sixty days to simulate in the first system and almost seven years on the second. Developing dedicated hardware is a time-consuming process, so making software fast enough to reach millisecond timescales on commodity hardware would be of great benefit.

One way of speeding up software is to introduce approximations. To extend the previous example, explicit simulation of the atoms in the environment around the protein can be avoided by approximating their effect on the protein. For some applications, there is evident that such approximations produce incorrect results [10]. If computation can be made fast enough, it becomes possible to discard these approximations and thereby improve the accuracy of the simulation.

Increasing the speed of a calculation allows for larger problems to be addressed. An example of this was presented by Ikegami *et al.* [11], who describe how new algorithms reduced the time required to simulate 20,000 atoms from 2×10^5 years to three days. The need to address larger problems occurs across disciplines: for example, Cantalupo *et al.* [12] forecast that data sets used to map cosmic background radiation will grow by three orders of magnitude over the next 15 years, as astronomers attempt to detect fainter signals.

Even without expanding the scope of a computational investigation, faster software can yield benefits. One example is software used for *ensemble forecasting*.

Ensemble forecasting is used to increase predictive accuracy by making a suite of predictions [13]. Every element in this suite comes from a single simulation run, for which either the starting conditions, the model being used, or the parameters of a particular model are varied. Analysis of the statistical distribution of the results can show the sensitivity of the simulated system to these changes, and can enable more precise predictions. If every element of the ensemble requires less time to execute, then larger ensembles can be used to explore a wider range of possibilities.

Finally, reducing running times also has a financial benefit. Computing consumes financial resources, and large-scale shared computer facilities charge for computational resources consumed. Reducing the running time of a program and hence reducing the resources consumed will reduce the financial cost of using such a facility. This allows researchers already using simulation to increase their usage for the same monetary outlay. Researchers who currently have to choose between experiment and simulation can do both. The additional crosscheck of simulation with experiment could result in better simulations.

1.2 The motivation for runtime tuning

Runtime tuning, also known as *feedback-directed optimisation* or *dynamic optimisation*, is a broad term covering a wide variety of techniques, and it has been applied in a variety of situations [14, 15, 16, 17]. Typically, the motivation for tuning at runtime is the presence of information not available when the code is compiled and linked.

Libraries are one area in which runtime tuning is useful for the statically compiled languages (such as C, C++ and FORTRAN) which are used for many scientific applications. As an example, consider a library for sparse matrix operations. In order to be generally quick, the library needs to provide a variety of different storage formats to cater for different matrix structures - this point is developed in Section 2.4. When used in a shared computational environment, the library will be used with many different programs, each processing matrices with different structures. Storage format selection can only occur at runtime, when the structure of each matrix is discovered.

Further motivation for runtime tuning comes from the literature on program phase analysis [18, 19, 20]. It has been repeatedly demonstrated that program execution goes through distinct phases, each of which may exhibit different perfor-

mance characteristics and use hardware resources in different ways. This creates an opportunity for runtime tuning in other applications sharing the same system. If one of those applications can compute its result using either network communication or redundant computation, and if a second application enters a phase where it is competing for network bandwidth, being able to adapt and use the redundant computation method would be beneficial.

1.2.1 Applying reinforcement learning to runtime tuning

The rules governing a runtime tuning system may be derived statically or dynamically. Static derivation occurs prior to tuning, usually based on timing results from a suite of benchmark programs. At runtime, such a system has a fixed set of rules, each of which is applied as conditions dictate. The rules may be learnt *offline* - prior to use. A tuning system utilising statically acquired rules need not overcome the costs of their acquisition, merely of their application.

In contrast to statically acquired rules, dynamically acquired rules are formulated as part of the process of adapting a target application. This imposes an additional cost on the learner and reduces its scope for improvement: any benefits it brings must be large enough to offset the overhead of rule acquisition. Such penalties are worth paying when the agent is faced with an unpredictable, dynamic environment. An agent that formulates tuning rules on the fly can adapt to novel situations.

A tuning system that derives its rules dynamically must accomplish certain tasks, including:

- perceiving the state of the application being tuned,
- acting to change the application being tuned,
- observing the consequences of its actions, as measured by changes in the performance of the tuned application,
- assembling a mapping from observed states to optimal actions, and
- recognising when changes to the mapping are required, and act to implement the required changes.

These tasks are characteristic of the *reinforcement learning task*, which is the task of learning from experience [21]. A number of techniques have been developed to accomplish this task. Construction of the mapping from states to

actions (known as the *policy*) can be achieved using model-free methods such as *Q-learning* [21]. Techniques such as *ϵ -greedy action selection* and *softmax action selection* enable a tuning system to adapt itself to changing environmental conditions. Methods such as *tile coding* have been developed to compress environmental perceptions, and to aid the process of generalisation. The work reported in this thesis applies methods such as these to the problem of runtime performance tuning. Section 2.2 provides more detail on reinforcement learning methods.

1.3 Motivation for using binary modification

Binary modification is the process of changing the executable image of a program without needing to recompile the program. Swift uses binary modification to implement runtime tuning. Runtime binary modification is not the only possible choice - the alternative is to modify the application's source code to request guidance from an external agent. The alternative approach has been investigated by others as described in Section 2.5.1. There is no runtime tuning action which cannot, in principle, be implemented in such a manner.

Any tuning action can be viewed as a function mapping programs onto programs. Given an input program, there is a finite number of results from the application of tuning. These results can be written out into a library of possibilities, through which the flow of computation can be diverted as runtime circumstances dictate. However, the practicalities of this approach imbue runtime binary modification with some advantages.

While the size of a tuning function's range is finite, it can be extremely large. As an example, Bronson *et al.* [14] report on the runtime tuning of barriers within Java programs. Each barrier protects one field, and they have a suite of different barrier implementations. An implementation is chosen based on a field's access patterns. Bronson *et al.* also report on the number of fields whose barriers are tuned, for different benchmark programs, and these numbers range from 300 through to 892. Assuming only two barrier implementations there are at least $2^{300} \approx 10^{90}$ possible mappings of fields to implementations. Assuming each implementation choice is stored as a single bit, then $\frac{2^{300}}{2^{33}} = 2^{267}$ gigabytes of space is required to store these mappings. This is clearly not feasible with current technology. A binary rewriting approach, on the other hand, does not have to store such vast quantities of data, because it only needs one mapping at a time.

The second advantage of binary modification is that it can implement *binary cost tuning*. This phrase, introduced here for brevity, refers to the ability to completely remove the overhead of tuning for portions of program execution. Figure 1.1 shows an example where tuning overheads are confined to phases, each lasting for one second. Outside these phases, the application runs unmodified. This is not possible with source code modification, although it can be approximated, in some cases quite well.

The third advantage of binary modification is that it can work in the absence of source code. The source of many scientific kernels is available, but the situation is not universal. Appendix B shows a survey of scientific programs provided by the National Computational Infrastructure (NCI) National Facility, located at the ANU Supercomputing Facility. The survey reveals that almost 24% of the available programs do not provide their source code. While not even close to a majority, there is no way for a third party to tune these codes other than through using binary modification.

1.3.1 Tools for runtime binary modification

Runtime binary modification is a complex task, for which Swift employs a specialised third-party tool. Many such tools exist, most notably Pin [22], DynamoRIO [23], Valgrind [24], DynInst [25], Etch [26], Vulcan [27] and ATOM [28]. Swift is being developed for operation in a Linux environment on x86 and x86_64 architectures. These constraints rule out Etch and Vulcan (which require Microsoft Windows) and ATOM (which requires Alpha binaries).

The four remaining binary modification tools were benchmarked to assess the overhead they impose on the tuned application - see Appendix A for details. The benchmark results showed that DynInst was the fastest tool once its relatively large startup cost has been recouped. These results do not consider factors such as ease of programming, the quality of documentation, the availability of source code or the extent and nature of each tool's community of users. This project is only concerned with speed of execution, and thus those additional factors have been ignored. DynInst, as the fastest evaluated tool, has been chosen as the technology on top of which Swift will be constructed.

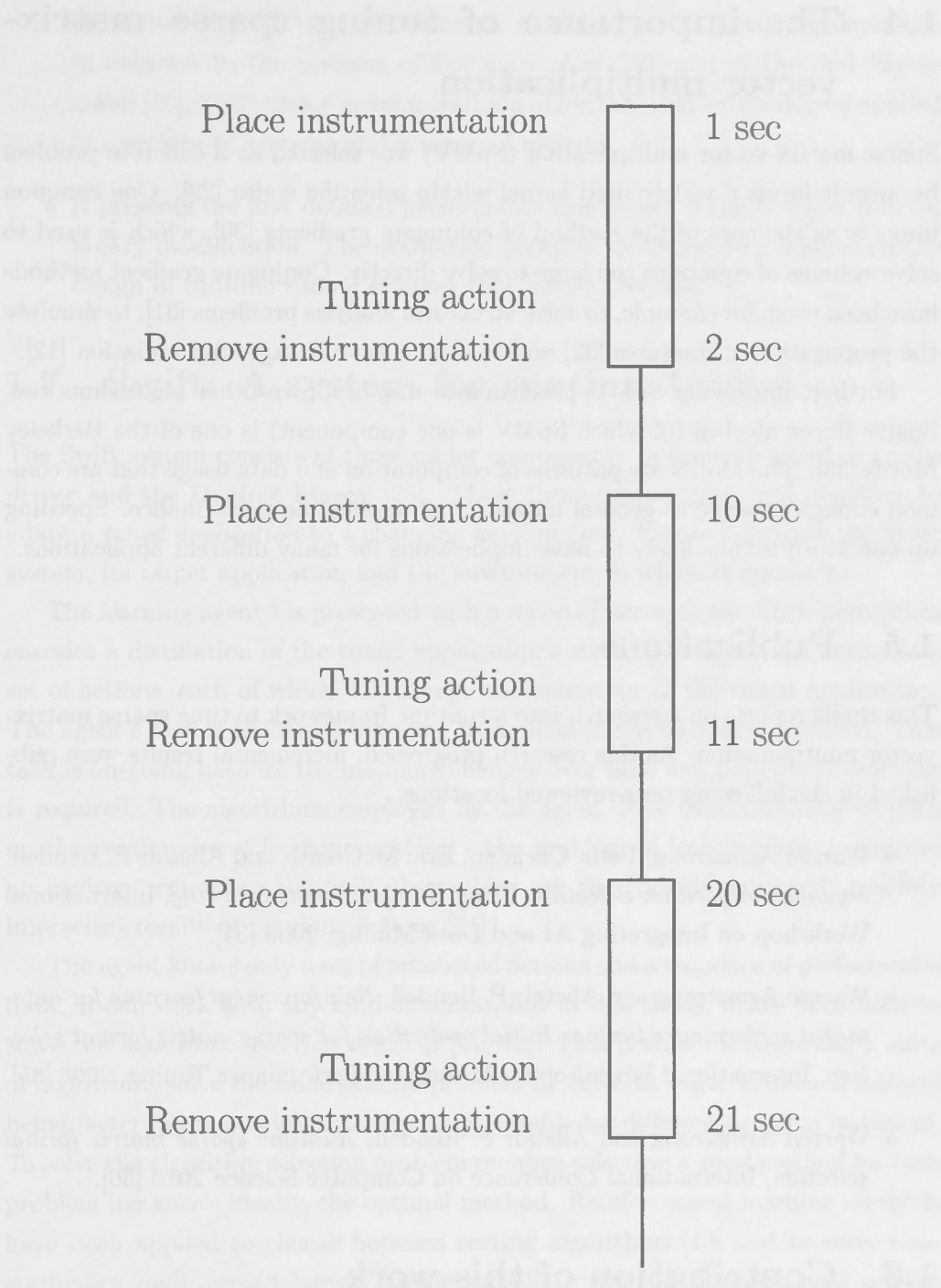


Figure 1.1: An example of binary cost tuning.

1.4 The importance of tuning sparse matrix-vector multiplication

Sparse matrix-vector multiplication (SpMV) was selected as a concrete problem because it forms a widely used kernel within scientific codes [29]. One common usage is as the core of the method of conjugate gradients [30], which is used to solve systems of equations too large to solve directly. Conjugate gradient methods have been used, for example, to solve structural analysis problems [31], to simulate the propagation of fractures [32] and to map cosmic background radiation [12].

Further, improving SpMV performance may improve other algorithms too. Sparse linear algebra (of which SpMV is one component) is one of the Berkeley Motifs [33]. The Motifs are patterns of computation and data usage that are common enough to serve as general indicators of application performance. Speeding up one Motif is thus likely to have implications for many different applications.

1.5 Publications

This thesis reports on a research into a runtime framework to tune sparse matrix-vector multiplication. As this research progressed, incremental results were published in the following peer-reviewed locations:

- Warren Armstrong, Peter Christen, Eric McCreath and Alistair P. Rendell, *Dynamic Algorithm Selection Using Reinforcement Learning*, International Workshop on Integrating AI and Data Mining, 2006 [34].
- Warren Armstrong and Alistair P. Rendell, *Reinforcement learning for automated performance tuning: Initial evaluation for sparse matrix format selection*, International Workshop on Automated Performance Tuning, 2008 [35].
- Warren Armstrong and Alistair P. Rendell, *Runtime sparse matrix format selection*, International Conference on Computer Science 2010 [36].

1.6 Contribution of this work

The work reported in this thesis makes the following novel contributions:

- It presents the runtime tuning system Swift, which learns tuning rules at runtime. In contrast to the systems of Suda [37] and Kajiyama *et al.* [38],

Swift does not utilise a performance model to make its tuning decisions. In contrast to the systems of Seshagiri *et al.* [39] and of Lee and Eigenmann [40], Swift uses a general learning algorithm that can easily be applied to a variety of systems and a range of metrics.

- It presents the first detailed performance evaluation of the DynInst tool for binary modification. The evaluation provides quantitative guidance on the design of runtime tuning systems that utilise DynInst.

1.7 Swift: A system for runtime tuning

The Swift system consists of three major components: a *learning agent*, a *tuning driver* and the DynInst library [25]. These three components work together to adapt a *tuned application* to a changing environment. Figure 1.2 shows the Swift system, its target application and the environment in which it operates.

The learning agent * is presented with a series of perceptions. Each perception encodes a distillation of the tuned application's state. The agent has available a set of actions, each of which will change the behaviour of the tuned application. The agent's task is to learn a mapping from perceptions to optimal actions. This task is on-going because the mapping changes over time and periodic re-learning is required. The algorithms employed by the agent draw from the body of work on the *reinforcement learning problem* - the problem of learning how to control an environment when the only observables are the (possibly delayed, possibly imprecise) results of previous actions [21].

The agent knows only a set of numbered actions and a sequence of performance data. It can work with any kind of action, but in this thesis, it has been used to solve the *algorithm selection problem* [41, 42]. This problem arises when a suite of algorithms solve the same general problem in different ways, with each method being faster, more precise or otherwise preferable for different problem instances. To solve the algorithm selection problem requires selecting a good method for each problem instance - ideally, the optimal method. Reinforcement learning methods have been applied to choose between sorting algorithms [43] and to solve non-stationary multi-armed bandit problems [44], which are mathematical models of the algorithm selection problem. This thesis applies them to select different sparse matrix-vector (SpMV) storage formats and their associated multiplication algorithms.

*Hereafter referred to simply as the agent.

When used for algorithm selection, the actions available to the agent have the following form: *at address a , replace the invocation of the function f with an invocation of function g* . The replacement is achieved using the DynInst [25] library for runtime binary modification. This library also enables the injection of code to gather performance data. These data are provided to the agent, enabling it to discover the effects of its actions. .

Between the agent and the DynInst library lies the tuning driver. This component translates the actions chosen by the agent into sequences of calls to DynInst. These sequences are then inserted into the tuned application. They transform the parameters of f into the parameters of g , gather statistics and transmit them to the agent and rearrange data structures within the tuned application.

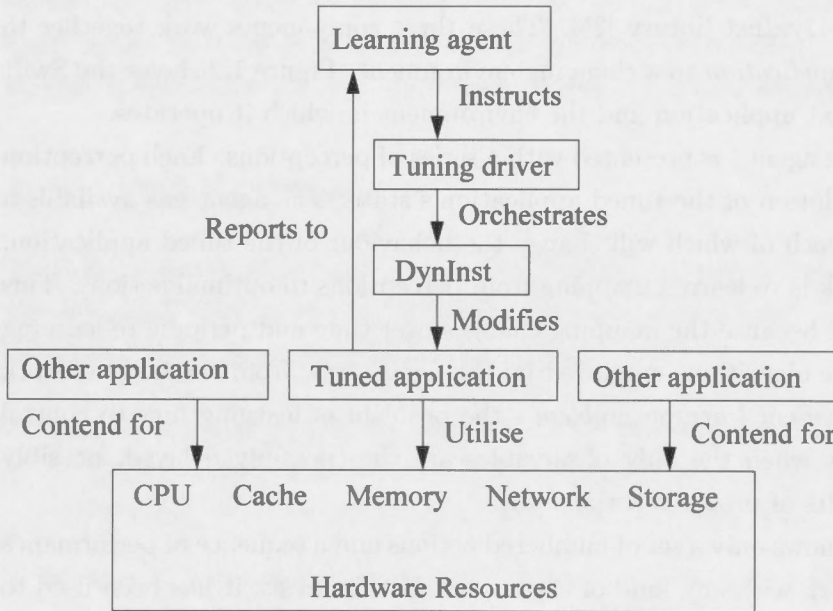


Figure 1.2: The components of the Swift system, its target and environment.

1.8 Thesis Structure

Chapter 2 presents background information on reinforcement learning, DynInst and sparse matrices. It also discusses related work on automatic performance tuning and on evaluating the performance of DynInst. Chapters 3, 4 and 5 form the core of this thesis. They describe and measure the main components of Swift. Chapter 3 describes the learning algorithm and evaluates it within a simulated

environment. Chapter 4 benchmarks a subset of the functionality of DynInst, the binary modification tool used to implement Swift. Chapter 5 describes the tuning driver and measures its effect on a computational kernel. Chapter 6 presents the conclusions and discusses directions for future research. Additional information is available in the appendices.

Chapter 2

Background Information and Related Work

The first section of this chapter presents some notation used in the rest of the thesis. The next three sections of this chapter present background information. Section 2.2 contains a brief introduction to reinforcement learning as it is applied in this thesis. Section 2.3 describes the processes by which DynInst implements its own operations. Section 2.4 shows a number of system machines. It describes their design and describes three different storage formats and associated algorithms.

Various constants are used throughout this chapter, and they are described in Section 2.1. Section 2.1 surveys related work on software auto-tuning.

2.1 Notation and terminology

This section describes the notational conventions and nomenclature used throughout this thesis. Notations and terms used in only one section of the thesis will be defined on their first use.

- $\vec{V}$ represents a vector
- $|V|$ denotes the number of elements in the vector V
- $[a, b]$ represents a closed interval from a to b inclusive - that is all numbers x such that $a \leq x$ and $x \leq b$
- (a, b) represents a half-open interval from a to b inclusive of a - that is all numbers x such that $a \leq x$ and $x < b$

Chapter 2

Background Information and Related Work

The first section of this chapter presents some notation used in the rest of this thesis. The next three sections of this chapter present background information. Section 2.2 contains a brief introduction to reinforcement learning as it is applied in this thesis. Section 2.3 describes the processes by which DynInst implements its core operations. Section 2.4 forms a primer on sparse matrices - it motivates their usage and describes three different storage formats and associated algorithms.

Various notations are used throughout this thesis, and they are described in Section 2.1. Section 2.5.1 surveys related work on software automatic tuning.

2.1 Notation and terminology

This section describes the notational conventions and nomenclature used throughout this thesis. Notations and terms used in only one section of the thesis will be defined on their first use.

- $\bar{V}$ represents a vector
- $\|\bar{V}\|$ denotes the number of elements in the vector $\bar{V}$.
- $[a,b]$ represents a closed interval from a to b inclusive - that is all numbers i such that $i \geq a$ and $i \leq b$.
- $[a,b)$ represents a half-open interval from a to b inclusive of a - that is all numbers i such that $i \geq a$ and $i < b$.

- $(a, b]$ represents a half-open interval from a to b inclusive of b - that is all numbers i such that $i > a$ and $i \leq b$.
- $\bar{V}_i$ represents the i^{th} element of the vector $\bar{V}$. Indexing is 0-based and valid indexes lie on the interval $[0, \|\bar{V}\|)$.
- $\{ f(i) : I \}$ represents the unordered set of items resulting from evaluating the function f on every element i within the interval I . For example, if $f(x) = 2x$ and I is the integer interval $[2, 6]$, then $f(i) : I$ is the set $\{4, 6, 8, 10, 12\}$.
- $Rows(A)$ denotes the number of rows in a matrix A .
- $Cols(A)$ denotes the number of columns in a matrix A .
- $NNZ(A)$ denotes the number of non-zero elements in a matrix A .
- $r \times c$ -BCSR is shorthand for the BCSR format with a block size of r rows and c columns.
- *optimal set*: An optimal set arises when discussing statistical performance results. In order to ensure that performance numbers are representative, experiments are repeated several times and the set of results are interpreted as a sample from a larger population. When comparing two options, however, what is of interest is whether the mean values of the *population* of timing results for each option differ. This is answered in a probabilistic manner by using statistical measures of the sample. Let $P(x, y)$ be a function that returns 1 if the probability that the mean of x is less than mean of y is greater than some threshold. If this condition does not hold, $P(x, y) = 0$. Then, from a set of n alternatives $a_1, a_2, a_3, a_4, \dots, a_n$, the optimal set consists of the alternatives a_i where $\forall_{j \in [1, n]} P(a_j, a_i) = 0$.

2.1.1 Pseudocode

Algorithms in this thesis are presented using pseudocode - structured English combined with specialised notations. The following conventions are used:

Array indexing $A[i]$ denotes the i^{th} element of the array A . Array indices start at 0.

Array initialisation $A = \{a, b, c\}$ denotes the creation of an array A with three elements a , b and c .

Assignment $a \leftarrow b$ sets the value of variable a to that of variable b .

Equality $a = b$ denotes a test of the equality of the values a and b - if either or both of these elements is a variable, the test uses the value of the variable.

First function When describing an entire program, many functions will be grouped together. In these cases, the function named *main* will be the first one executed and control will flow from there.

Function definition The function f is defined using the notation **function** $f \dots \text{end}$. The terms *function*, *procedure*, *routine* and *method* are treated as synonyms.

Function invocation $f(a, b, c)$ denotes the invocation of function f using arguments a , b and c . All arguments are passed by value.

Function replication $f[n - p]$ is a shorthand for the set of functions $f_n, f_{n+1}, \dots, f_p$. For example $f[1 - 4]$ refers to the functions f_1, f_2, f_3 and f_4 .

No operation The **nop** instruction has no effect, it is employed as a placeholder.

Outlined functions are functions which the compiler is forbidden to inline. An outlined function f is written as **outline** f .

Variables are usually not declared in advance. They are introduced as needed and their types can be inferred from their usage. There arises only one situation where there is more to a variable than a name and a type: a variable may also be *aligned*. An aligned variable has restrictions on the memory locations used to represent it: n -byte alignment requires the variable's address be an integer multiple of n . Alignment of variable v to a multiple of n bytes is conveyed using the notation **let** v **be aligned on** n .

2.2 Reinforcement Learning

Reinforcement learning is the process of learning from experience. Figure 2.1 shows a typical setup for reinforcement learning. The primary division is between an *agent* and an *environment*. The agent is an entity which senses the environment, acts to change the environment, and learns which actions will shape the

environment into a beneficial form. Within the agent are several components. *Sensors* measure the environment and pass their perceptions into a *perceptual filter*. This filter processes raw perceptions and passes a single input - the *state* - to the learning algorithm, which selects an action to undertake. This action is translated into *commands* which are sent to *actuators*, which carry out the action.

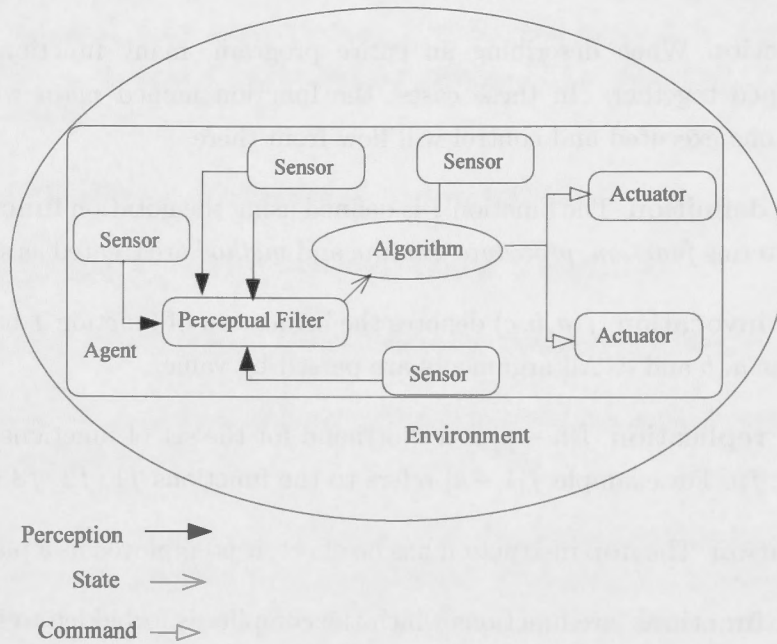


Figure 2.1: A basic reinforcement-learning setup.

2.2.1 Action selection

As it learns, the agent will construct a *policy* which maps states onto desirable actions. The term *action selection* refers to the manner in which the agent uses this policy. The action selection method needs to choose between two types of action. The agent may *exploit* its knowledge by taking the action recommended by its policy, or it may *explore* by acting in contravention of its policy. Exploration is the means by which the agent constructs a policy, and the means by which the policy is adapted to changing circumstances.

As an example, consider an agent with two possible actions *A* and *B*, whose rewards over time are plotted in Figure 2.2. Exploration of the unknown would quickly ascertain that action *A* was the best option. However, there comes a point

(T in the figure) where a change in the agent's environment causes the utility of action B to increase. If the agent continued to pick action A under the belief that it was the best action, the agent would begin making sub-optimal choices. The only way the agent can act correctly after the point T is by selecting action B, despite the agent's accumulated experience.

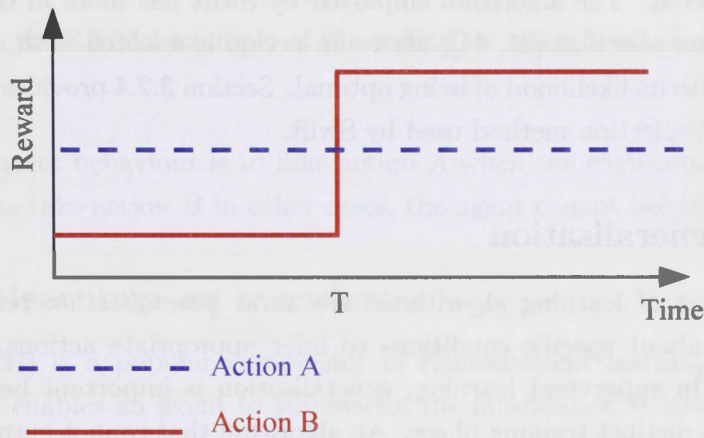


Figure 2.2: An example of the need for an agent to occasionally act in contravention of its accumulated knowledge.

The question then arises: when should exploration occur? When faced with the environment in Figure 2.2, the best solution is explore once at the point T . However, achieving this solution requires that the learning agent be specific to one particular environment in which the behaviour of the reward signal is always the same. To apply the agent to different environments, or to environments where the reward signal is not fixed in advanced, the algorithm must have a more general notion of when to explore. A common approach is to explore periodically - either with a fixed or changing frequency.

One example of such a policy is known as ϵ -greedy action selection. Under this scheme, every time an action must be taken, a number r is selected randomly * from the interval $[0, 1)$. The agent will explore if and only if $r < \epsilon$. Variants of this approach set the exploration rate to be a function of time: $\epsilon(t)$ - this function is usually constructed so as to decay towards zero. This allows a high rate of exploration initially, and then reduces the rate once the agent has had a chance to learn. Such constructions have led to various theoretical results on the performance of different algorithms [45]. However, they fail to adapt to a

*The selection is usually only pseudorandom, not truly random.

dynamic environment. Such an approach is similar to offline classification, in that a policy is learned once and then blindly applied. To be able to adapt to changing environments, exploration must continue for the lifetime of the agent, but its level must be selected with care.

Under ϵ -greedy action selection, each non-optimal action has an equal chance of being selected. The algorithm employed by Swift has more in common with *softmax* action selection [21, 44], where an action is selected with a probability proportional to its likelihood of being optimal. Section 3.2.4 provides more details on the action selection method used by Swift.

2.2.2 Generalisation

In the context of learning algorithms, the term *generalisation* refers to using information about specific conditions to infer appropriate actions under novel conditions. In supervised learning, generalisation is important because of the presence of a distinct training phase. An algorithm that cannot extrapolate from its training data will have limited utility when faced with novel data. Reinforcement learning lacks this separate training phase, but generalisation is still important because it reduces learning time - a learner that can judge states A and B through testing A alone can learn faster than one that must also test B . Similarly, the learner need only store information about how to react to state A , reducing memory requirements.

In this thesis, generalisation is achieved through the use of *tile coding*, a method described by Sutton and Barto [21] and employed in several reinforcement learning systems, such as those of Sutton [46], Ipek *et al.* [47] and Kalyanakrishnan and Stone [48]. In tile coding, each attribute a_i is associated with a set of n tiles. When a specific instance of a_i is perceived, the value of that instance is mapped onto an element of the set of tiles. An example is shown in Figure 2.3. The figure shows the interval $[0, 20]$ split into nine tiles of equal size. Two inputs are perceived, with values 12.3 and 13.42, and each is mapped to the appropriate tile. In this case, both inputs are mapped to the same tile, and so the learning algorithm perceives them as having the same value. This means that any lessons learned about these inputs can be applied directly to future inputs with values 12.75, 13, 12.01 and so on.

The compression of multiple distinct environmental states into a smaller set of tiles gives rise to *perceptual aliasing* [49]. The agent loses the ability to distinguish between certain external states, which may impair its ability to learn. In Figure

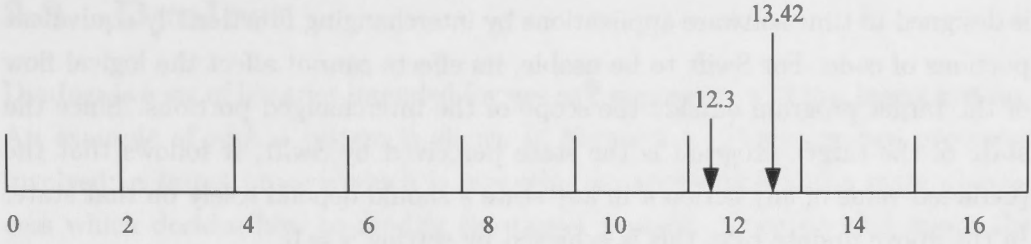


Figure 2.3: An example of tile-coding for a single attribute.

2.3, if the correct behaviour is to take action A when the environmental state is ≤ 12.7 and to take action B in other cases, the agent cannot behave correctly.

2.2.3 Q-learning: an overview

Q-learning [21] is a popular component of reinforcement learning algorithms. The method enables an agent to summarise the information it has gathered by exploring its environment. It is based on a function $Q(s, a)$ that maps a state s and action a onto a numerical reward.

The function Q begins as an arbitrary mapping. Rewards received by the agent are incorporated into Q , moving the mapping towards one reflecting the agent's accumulated experience. At every time step t the agent perceives a state s_t and takes an action a_t - the choice of action is based on the current values of Q . Once executed the action returns a reward r_t , along with a new state s_{t+1} . The reward is used to update the function Q :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t - Q(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a)]$$

The update can be broken down into the following components:

- the difference between expected reward and received reward: $r_t - Q(s_t, a_t)$,
- the best possible outcome from choosing action a_t in state s_t , which is the highest reward obtainable in the successor state s_{t+1} : $\max_a Q(s_{t+1}, a)$,
- the degree to which long term outcomes should be favoured over short-term results: γ , and
- how much weight should be accorded each observation: α .

Swift is designed for situations in which the selection of actions does not influence the sequence of states perceived to the agent. To be more precise, Swift

is designed to tune software applications by interchanging functionally-equivalent portions of code. For Swift to be usable, its effects *cannot* affect the logical flow of the target program outside the scope of the interchanged portions. Since the state of the target program is the state perceived by Swift, it follows that the predicted value of any action a in any state s should depend solely on that state. In the above update rule, this is achieved by setting $\gamma = 0$.

In the description so far the reward has been a scalar value r_t . Obtaining this value, however, brings up some interesting issues when the algorithm is used with an incomplete description of the environment. This can occur when the environment is generalised as described in Section 2.2.2. As a motivating example, consider the following situation:

- There are two environmental states x and y .
- The learner perceives both x and y as generalised state z .
- There are two possible actions a and b .

- State-action pairs have the following rewards:

	x	y
a	5	15
b	3	10

Action a is optimal in this scenario. If the agent tries to determine the reward of action a in state z when the true environmental state is x , the reward will be measured as 5. If the agent then tries to measure action b when the true environmental state is y , the agent will receive a reward of 10. Based on these rewards, the agent will incorrectly conclude that action b is better than action a .

One way around this problem is to define the reward of a state-action pair as

$$R(s, a) = \begin{cases} 1 & \text{if } a \text{ is in the optimal set of } s \\ 0 & \text{otherwise} \end{cases}$$

Thus, the learning agent is no longer trying to predict numerical rewards but rather to predict which action is the best. However, this leads to the problem of figuring out which action is best in order to provide an implementation of R . This requires constructing the optimal set on the fly, in a manner that avoids the problems just discussed with the composite nature of state z . The manner in which this is achieved is described in Chapter 3.

2.3 DynInst

DynInst is a set of libraries intended for use as components within a larger system. An example of such a system is shown in Figure 2.4. There are two processes involved: a *target* process which is executing an application and a *mutator* process which decides how to modify the target process. Once decided upon, the modifications are carried out using tools supplied by DynInst.

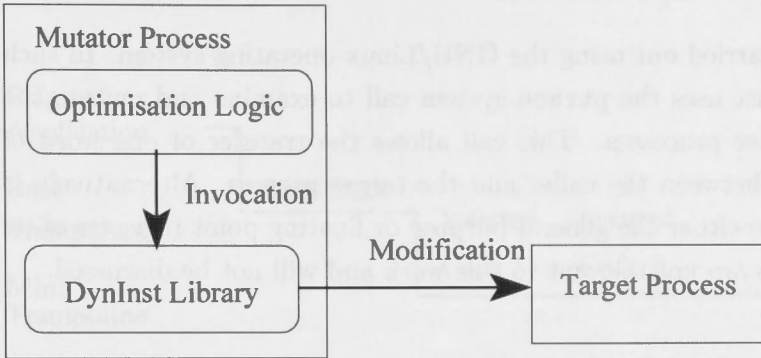


Figure 2.4: How DynInst fits into a complete optimisation system.

This discussion concentrates on three of the facilities provided by DynInst: code emplacement, access to variables, and procedure call redirection. *Code emplacement* is a mechanism for inserting arbitrary code into a target process. As an example, this facility can be used to insert performance profiling code and then to remove the code (and its overheads) once profiling has been carried out. Code can be inserted almost anywhere using a range of techniques. Each technique makes a different tradeoff between generality and the speed at which it executes. General techniques are relatively slow, while those that omit certain precautions are faster but not always safe.

Variable access involves reading and writing portions of the address space of the target process. This allows the mutator to discover or set the values of variables in the target process. Making use of this facility requires the mutator to understand the algorithms employed by the target, reducing the generality of the mutator.

Procedure call redirection refers to changing the target address of a subroutine invocation. This can be useful in many situations, such as the implementation of dynamic algorithm selection [43, 34]. Dynamic algorithm selection supposes that a particular problem can be solved by more than one algorithm and each algorithm implies a different tradeoff: some might be faster on small, sorted

datasets, while others are better on large, unordered inputs. If a program deals with both types of data, the optimiser can locate the points at which the routines implementing these algorithms are called and substitute the desired one.

The following subsections provide more detail on the means by which DynInst carries out each of these tasks.

2.3.1 The Ptrace System Call

The work here was carried out using the GNU/Linux operating system. In such environments, DynInst uses the `ptrace` system call to examine and mutate the address space of other processes. This call allows the transfer of one word of data per invocation between the caller and the target process. Alternatively it can read and write to either the general purpose or floating point registers of its target. Its other uses are not relevant to this work and will not be discussed.

2.3.2 Inserting and Removing Code

This section describes how DynInst inserts code into a target process. The basic mechanism - known as a *trampoline* - diverts the target program's control flow. There are two main kinds of trampoline: *base trampolines* and *mini trampolines*. Figure 2.5 shows how these trampolines interact with each other and with the target application. The base trampolines are responsible for saving and restoring the state of the CPU, while the mini trampolines contain the inserted code.

Only mini trampolines are explicitly inserted by the user of DynInst. Base trampolines are emplaced automatically. Inserting the mini trampolines requires four major steps. These steps are initiated by a procedure which requires two inputs: a handle to the instrumentation, and the point at which the instrumentation is to be applied. From this point on, the instrumentation referenced by the handle will be called a *snippet* and the point at which it will be applied will be termed the *target point*.

The first step in the insertion procedure is to carry out a safety check. This check requires traversing the call stack within the target process and inspecting the program counter at each step. If any such counter is within a trampoline associated with the target point, the snippet insertion is aborted. This is necessary as the act of insertion can change the manner in which mini-trampolines are linked together. This would leave the program counter pointing to an incorrect instruction. The time spent on such checks will increase with the depth of the

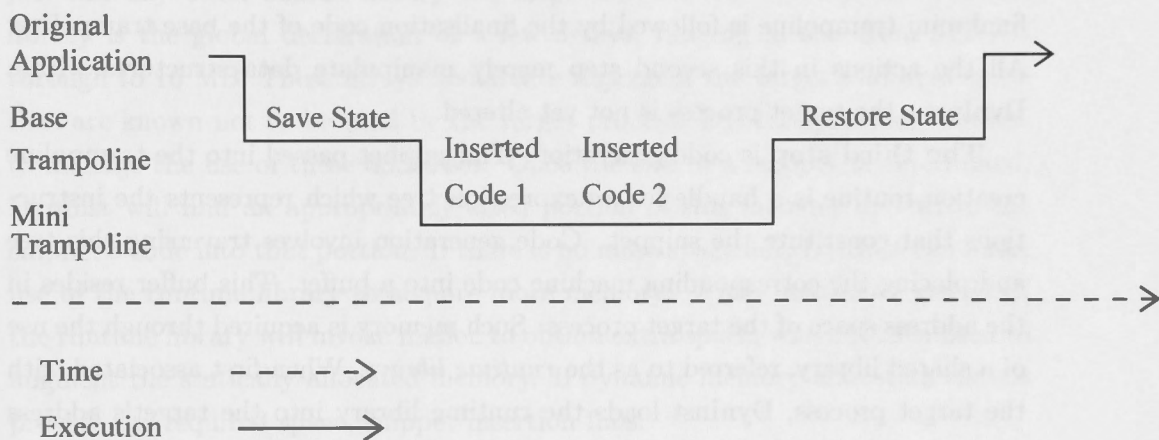


Figure 2.5: Internal organisation of the trampoline mechanism employed by DynInst. Branch instructions are inserted into the application at the desired points, causing control flow to divert through a base trampoline. The base trampoline saves the state of the CPU (registers, etc) and then branches to the instrumentation which is stored in a mini trampoline. There can be several such mini trampolines for every base trampoline. Once the last mini trampoline has finished, control passes back to the base trampoline, which restores the previously saved state and branches back into the original application.

call stack.

The second step associates the target point with a base trampoline. If the target point is not already linked to a base trampoline, one is created. Otherwise, the new mini trampoline may need to share the base trampoline with pre-existing mini trampolines. This sharing takes one of two forms, termed here *inlined* and *outlined*. If a base trampoline contains outlined mini trampolines, then each mini trampoline has an explicit jump instruction linking it to its successor. In contrast, inlined mini trampolines are merged together so that the final instruction in a non-terminal mini trampoline is followed by the first instruction of its successor. The final mini trampoline is followed by the finalisation code of the base trampoline. All the actions in this second step merely manipulate data structures within DynInst - the target process is not yet altered.

The third step is code generation. The snippet passed into the trampoline creation routine is a handle for an expression tree which represents the instructions that constitute the snippet. Code generation involves traversing this tree and placing the corresponding machine code into a buffer. This buffer resides in the address space of the target process. Such memory is acquired through the use of a shared library, referred to as the *runtime library*. When first associated with the target process, DynInst loads the runtime library into the target's address space. It achieves this through the following process:

1. Locate and designate with F one of the functions `main`, `_start`, `_init` or `__libc_start_main`, which are the first functions involved in executing a C or C++ program.
2. Locate the C library function `_do_dlopen`, used to load shared libraries.
3. Create machine code (s bytes in size) denoted by I . This code will invoke `_do_dlopen` passing the runtime library as the library to be loaded and then execute a trap instruction.
4. Read and store the first s bytes after the start of F .
5. Write the s generated bytes after the start of F .
6. Modify the program counter so that the next instruction to execute is the first instruction of I ,
7. Wait until the trap instruction at the end of I is executed. This signals that the library has been loaded.

After loading, the library must be initialised. DynInst can do this in one of two ways. The first way relies on the linker to ensure the initialisation routine is called when the library is loaded [†]. The runtime library requires some information from the mutator-linked library to initialise itself. This is sometimes not present when the linker-driven initialisation is carried out. Under such circumstances, DynInst reverts to the second way: calling the initialisation routine itself through a remote procedure call (RPC) [‡].

The runtime library is now present in the address space of the target process, just like any other shared library the target has loaded. Part of the runtime library is the global declaration of a few arrays, ranging in size from 512 KB through to 16 MB. These arrays demarcate regions of the target's address space that are known not to be used by the target process. DynInst provides routines to manage the use of these addresses. Once the size of a snippet is determined, DynInst will find an appropriately sized portion of this memory and write the snippet's code into that portion. If there is no more space left, DynInst can make use of the runtime library to acquire more memory. When instructed via RPC, the runtime library will invoke `malloc` to obtain extra space, which is then used to augment the statically allocated memory. If dynamic memory allocation cannot provide the required space, snippet insertion fails.

The fourth step of snippet insertion links the snippet to the target point. This is done by replacing the code at the target point with a branch instruction. The replaced code is incorporated into the base trampoline. In some cases the replaced code is larger than a single instruction, for example when inserting a snippet into a loop header, the whole header is replaced. In these cases, *trap* instructions are inserted after the branch to fill up the replaced code region. These instructions cause a trap to be delivered to the operating system. This results in the target process receiving a signal, which is intercepted by DynInst. DynInst maintains a table which maps every replaced code region to the target of the branch that replaced it. On receiving a trap signal, DynInst can use `ptrace` to change the program counter of the target so that when it resumes, it executes

[†]This is done on Linux through the use of the `-init` argument to `ld`, as well as the use of the `constructor` attribute if compiled with `gcc`.

[‡]The DynInst RPC mechanism is built on top of `ptrace` - the code to invoke the desired procedure is written into one of the runtime library's memory regions (statically allocated) and the program counter of the mutated process is pointed to the start of the region. The procedure call is followed by a trap instruction which will return control to the mutator process, signalling the completion of the RPC.

from the target of that branch. Use of traps is exceedingly slow and is only rarely required: it is purely a safety measure.

Code insertion is a complicated process. Removing the inserted code is much simpler. The actual effect of such a removal depends on the nature and the number of mini trampolines at the instrumentation point. If the mini trampoline being removed is the only one that remains then the following actions take place:

1. The portion of the target process's memory containing the trampolines is marked as available for future snippets,
2. The instructions which branched to the trampoline are overwritten with the original instructions that they replaced

With the version of DynInst being tested, trying to remove a subset of mini trampolines does not work - it is an all or nothing proposition. This is a bug [50] in DynInst.

The discussion so far has concerned the manipulation of trampolines, but what of their structure? The base trampoline is responsible for saving and restoring the state of the processor. At a minimum, this state consists of all general purpose registers and flags. If the inserted code is going to perform floating point computations it is necessary that floating point state be saved and restored as well. There is a further issue to consider, one which is best explained by example: consider a snippet consisting of a call to the C output function `printf`. What happens if this snippet is inserted into the function `printf` itself? Without safety measures, this will result in an infinite recursion. DynInst provides guard constructs that enable such situations to be negotiated safely. These guards are Boolean flags which initially have the value false. Part of a base trampoline's initial execution involves reading the flag and examining its value: if the flag is false, it is set to true and execution proceeds. If the flag is already true, then the mini trampolines are not executed. After the execution of the mini trampolines, the flag is set to false. The flag is read and set by reading and writing to a memory location. The guard constructs are not atomic and cannot be used to safely encapsulate thread-unsafe instrumentation. The guards may be disabled if the programmer knows that there is no danger of infinite recursion happening. Since the guards involve only a few instructions, disabling them is not likely to be a major optimisation.

2.3.3 Reading and Writing Variables

DynInst provides a facility for reading and writing global variables belonging to the target process. Such variables may be basic types such as integers or composite types like C `structs`. Internally, DynInst follows the process laid out in Figure 2.6. The request is formulated as a `ptrace` operation and passed to a dedicated handler thread.[§] Once the operation has been completed, any resulting data are passed back up the unwinding call chain and the previously stopped thread is restarted.

2.3.4 Procedure Switching

DynInst provides a way to rewrite procedure calls within a target process. The underlying mechanism is the same as that for writing a variable in the target, the only difference being that values are written to the process's code rather than to its data. The value in question is the address of a different function and it is written over the address of the function being replaced. Thus, a machine instruction denoting `call procedure A` is transformed into one denoting `call procedure B`.

2.4 Sparse Matrix Background

This section introduces sparse matrices. It begins with a motivation for their use, and follows with a review of how sparse matrices are stored in memory. There are many such schemes, and Section 2.4.2 will discuss only those used in the rest of this thesis.

2.4.1 The utility of sparse matrices

A *sparse* matrix is a matrix that has a large percentage of “useless” elements. These elements are useless because they don't contribute to the result of computations involving the matrix. A concrete example was provided by Millam and Scuseria [51] for certain kinds of quantum chemistry calculations. These calculations involve three different matrices, and these matrices can be processed faster if matrix elements below a certain threshold are omitted. Millam and Scuseria

[§]In theory, this would allow asynchronous reading and writing of variables. However, this is not provided for in the API version which was tested. It would be possible to achieve such operations through other means, if required.

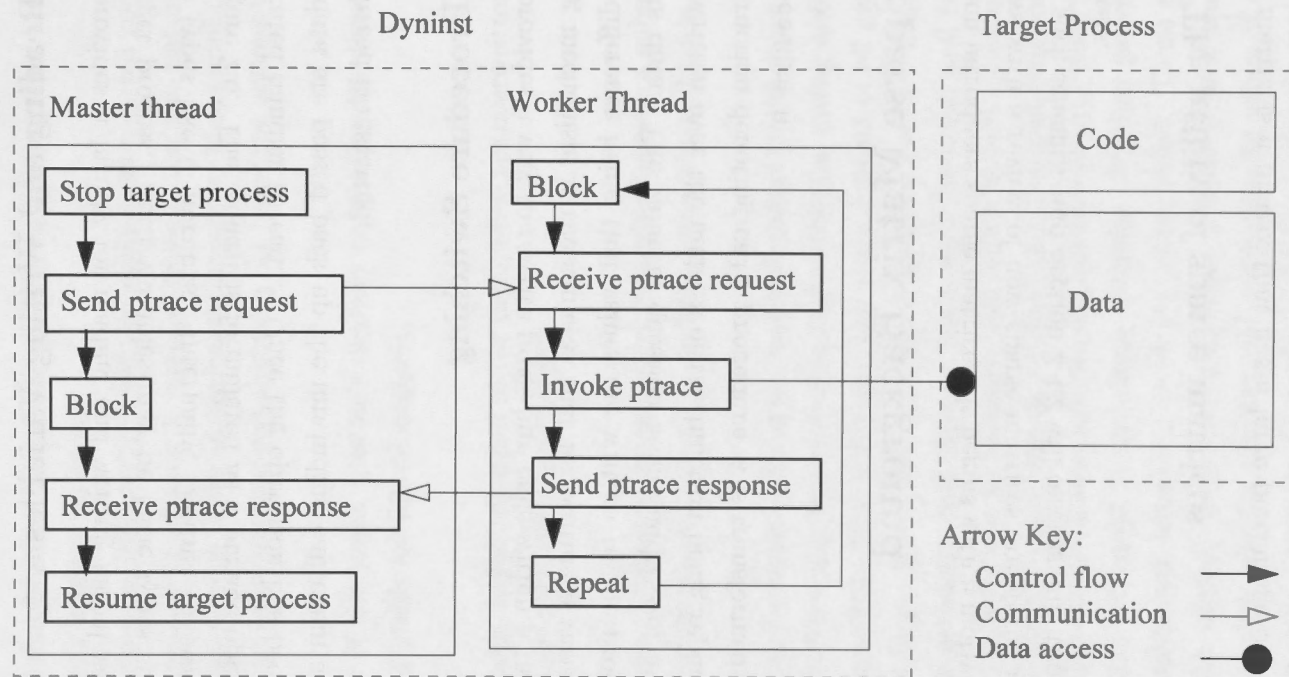


Figure 2.6: An outline of the process by which DynInst manipulates the global variables of a process.

show that, depending on the problem being solved, more than 90% of elements can be culled in this fashion. Consider such a system with n elements before the cull. In the coordinate format (described below), the space required to store the post-cull system is $24 \times 0.1 \times n$ - this is less than a third of the space required to store the matrix in a non-sparse format. The calculation assumes 64-bit indices and the most storage-intensive sparse format, and the gains are greater if 32-bit indices suffice, or if a more compact storage format was used.

As well as the reduction in storage space, the sparse matrix representation requires fewer floating point operations to process. There is an increase in auxiliary operations required to support the remaining floating point computations, but a sufficiently sparse system can achieve a net speedup by moving to sparse storage formats. Returning to the previous example, Millam and Scuseria showed (for a different chemical system and computation) that using sparse matrix representations reduces the CPU time required for processing. For one example when 31.6% of elements were culled, the measured operation required approximately 85% less CPU time to complete. Sparse matrices are not always faster than their dense counterparts. If sufficient elements are not dropped, then the extra indexing operations required by sparse matrix formats will overwhelm the reduction in floating point cost.

2.4.2 Review of different sparse matrix storage formats

Many different formats have been developed for storing sparse matrices. This section will describe the formats employed in the rest of this thesis. There are three basic formats employed: *coordinate* (COO), *compressed sparse row* (CSR) and *blocked compressed sparse row* (BCSR). Each will be discussed in the context of a sample matrix. Figure 2.7 shows this sample matrix as well as its representation in the three different formats, the details and purpose of which are discussed in the following sections.

Throughout the rest of this section, the term κ is used to denote the number of bytes required to store a matrix index - common cases are $\kappa = 4$ and $\kappa = 8$. All values are assumed to be stored as 64-bit floating point quantities. Finally, the discussion of each format will include an algorithm showing how to use that format for matrix-vector multiplication. In each of these example algorithms, it is assumed that the output vector has been initialised before the algorithm is called.

The **coordinate** (COO) format is the simplest sparse matrix storage format.

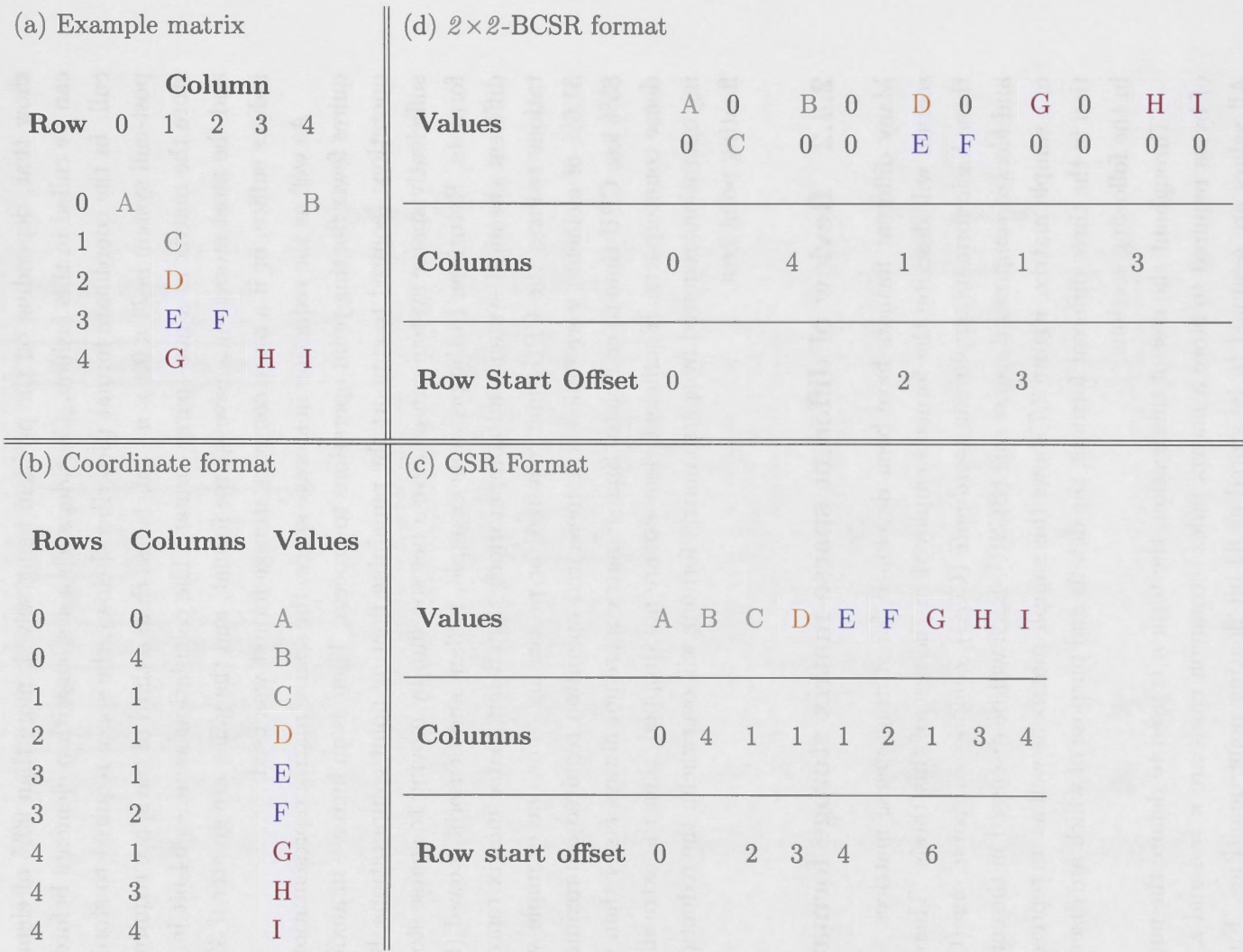


Figure 2.7: An example matrix and its representation in different storage formats. Note the reduction in the number of index values along the anti-clockwise progression (b), (c), (d), and the explicit zero padding in (d).

Along with each non-zero element are stored its row and column indices, for a total size of $n(8 + 2\kappa)$ bytes. Algorithm 2.1 shows how matrix-vector multiplication is carried out under this scheme.

Algorithm 2.1: Sparse matrix-vector product for matrices stored in coordinate format.

Data: rows, columns, values, input, output, n

```

1 for  $i \in [0, n)$  do
2    $c \leftarrow \text{columns}[i]$ 
3    $r \leftarrow \text{rows}[i]$ 
4    $v \leftarrow \text{values}[i]$ 
5    $\text{output}[r] \leftarrow \text{output}[r] + v \times \text{input}[c]$ 
6 end
```

For every floating point operation needed to compute the result, there are two extra memory loads (lines 2 and 3) and two extra integer arithmetic instructions[¶].

Silva and Wait [52] aimed to increase the spatial locality of the matrix data, and they proposed two different ways in which this might be achieved. The first method was to convert indices to the same data type as values and store everything in one array. The second was to use a single array of structs, and store indices without type conversion. Results were presented showing improved performance, but these were far from convincing. The first problem is that only execution time results were shown - no data were presented on cache miss rates, the reduction of which was the aim of these new formats. While the execution time did show (on single-precision matrices) a performance boost for the first suggested alternative relative to traditional coordinate format, it is not clear that this was due to improved locality. The comparison between the two suggested innovations showed the struct-based solution achieved shorter execution times. Whether this is due to decreased cache miss rates is not clear, as again that data was not shown. Further, the struct-based solution was never compared directly to traditional coordinate format, but only indirectly through comparison (on different matrices on a different machine) to the other suggested alternative. While it was compared directly to blocked compressed sparse row, this comparison was made on matrices with randomly distributed non-zero elements, which is a scenario diametrically opposed to the non-zero layout BCSR is designed to handle.

The **compressed sparse row** (CSR) format is designed to reduce the number

[¶]If the indirect loads are implemented as $*(\text{base} + i)$ then the extra arithmetic is the summation to get from the base to the offset. If it is implemented as $*\text{base}$, then there needs to be an increment $++\text{base}$ on each iteration of the loop, which is again extra overhead.

of indirect memory accesses and the associated offset calculations. The format is designed for matrices having many rows with more than one non-zero value. Values and column indices are stored as for coordinate format, but row indices are not stored. They are replaced by the offsets at which each row's data are located in the value and column arrays. The algorithm for multiplying matrices stored in this format is shown in Algorithm 2.2.

Algorithm 2.2: Sparse matrix-vector product for matrices stored in CSR format.

Data: rows, columns, values, input, output, row_count

```

1  $i \leftarrow 0$ 
2 for  $r \in \text{row\_count}$  do
3    $\text{next} \leftarrow \text{rows}[r + 1]$ 
4   while  $i \neq \text{next}$  do
5      $c \leftarrow \text{columns}[i]$ 
6      $v \leftarrow \text{values}[i]$ 
7      $\text{output}[r] \leftarrow \text{output}[r] + v \times \text{input}[c]$ 
8      $i \leftarrow i + 1$ 
9   end
10   $i \leftarrow \text{next}$ 
11 end

```

The total storage size of a CSR-format matrix with r rows is $(r+1)\kappa + n(8+\kappa)$ bytes. To facilitate the floating point calculations this algorithm incurs an excess cost of one memory load and one integer arithmetic instruction per row, and the same per non-zero. Whether this is less than coordinate format depends on the ratio of $\frac{n}{r}$. The more this ratio grows away from one, the less CSR costs relative to coordinate format. Apart from reducing the overhead of COO, CSR also reduces pressure on the data cache within the inner loop, since the array of row indices no longer needs to be accessed therein.

Given that the overhead of CSR is less than or equal to that of coordinate format, why bother with the latter? The reason is that coordinate format is a common format for the distribution and on-disk storage of matrices - for example, the University of Florida Sparse Matrix Collection is stored in the Matrix Market format, which uses coordinate representation. It is thus important to understand what coordinate format is, even if it is not used for actual matrix processing.

The **blocked compressed sparse row** (BCSR) format aims to reduce indexing overhead further than CSR. This is achieved by making an assumption about how the non-zero matrix elements are laid out. The assumption is that

the elements are clustered, which implies that a storage format need store only one index per cluster and treat every cluster as a dense submatrix. The BCSR format is parameterised by the size of the assumed block - the block is deemed to have br rows and bc columns. The data stored for the BCSR format is the same as for the CSR format, except that column indices refer to the left-most column of each block. Algorithm 2.3 shows how a matrix stored in this format is processed:

Algorithm 2.3: Sparse matrix-vector product for matrices stored in BCSR format.

Data: rows, columns, values, input, output, row_count, br, bc

```

1  $i \leftarrow 0$ 
2  $vi \leftarrow 0$ 
3  $r \leftarrow 0$ 
4 while  $r \neq \text{row\_count}$  do
5    $\text{next} \leftarrow \text{rows}[r + 1]$ 
6   while  $i \neq \text{next}$  do
7      $c \leftarrow \text{columns}[i]$ 
8     for  $j \in [0, br)$  do
9       for  $k \in [0, bc)$  do
10         $\text{output}[r + j] \leftarrow \text{values}[vi + j \times bc + k] \times \text{input}[c + k]$ 
11      end
12    end
13     $vi \leftarrow br \times bc$ 
14     $i \leftarrow i + 1$ 
15  end
16   $r \leftarrow r + br$ 
17 end
```

When judging the space requirements or overhead of a BCSR implementation, the position of the nonzeros within the matrix is crucial. If the matrix is composed exclusively of $br \times bc$ blocks, then the space requirement is $8n + \kappa((N + 1) + \frac{n}{br \times bc})$ bytes. At the other extreme, if the non-zero elements are scattered such that there is only one per block, then the matrix requires $8n + \kappa((N + 1) + n \times br \times bc)$ bytes of storage, since every block must be padded out with zero values.

When compared to the coordinate or CSR formats, the BCSR format has an additional source of overhead: the processing of explicit zero elements. These elements occur when the matrix is not perfectly blocked, and must be padded to fit the assumptions of BCSR. This dependence on matrix structure also affects

the indexing overhead, because it determines how many blocks are required to store the matrix, and indexing costs scale with the number of blocks.

The algorithm as presented has been written for explanatory brevity, not quick execution. In an optimised version, the double inner loop that implements the dense submatrix-vector product would be at least partially unrolled. Further, various index computations could be simplified, especially if it could be assumed that the input matrix was sorted by rows.

2.4.3 Influences on Sparse Matrix-Vector Multiplication Speed

Determining the storage format which will yield the fastest matrix-vector multiplication requires assessing a number of factors. One of these factors is the hardware architecture employed: Im *et al.* [5] show that the optimal format for a single matrix varies in a non-intuitive way across four different architectures. Armstrong and Rendell [36] demonstrate variation on two additional architectures. Two other important factors in deciding optimality are the non-zero structure of the matrix and resource contention within the host system. These points are demonstrated by the following experiment.

The experiment was carried out on a 64-bit 8-core 3.4GHz Intel Core i7 processor running a Linux 3.2.2 kernel in single-user mode. The benchmark code was compiled using gcc 4.6.0 with the flags `-march=native -O3`. In the first phase of the experiment, a set of 141 matrices was converted sequentially into 65 different storage formats - BCSR with blocking factors of 1x1 through 8x8 inclusive and CSR. While represented in each format, the matrix was used for 1000 matrix-vector multiplications. All multiplications were carried out in a process bound to a single processor core, with only background operating-system activity running on the other seven cores.

In the experiment's second phase the same process was followed, except that the seven cores not engaged in matrix-vector multiplications were endlessly looping through two 10 MB buffers.

The results of the experiment are reported in Table 2.1. Each format is represented in the table as a 2x2 matrix. The top left element of the matrix reports the *baseline peak* - the number of matrices for which the format in question was fastest under baseline conditions. The top right element reports the *baseline 10%* - the number of matrices for which the format in question was no more than 10% slower than the optimal format, in the absence of contention. The bottom left

element reports the *contested peak* - the number of matrices for which the format was optimal in the face of resource contention. The bottom right element reports the *contested 10%* - the number of matrices for which the format's execution time was within 10% of that of the optimal format, on a system experiencing resource contention.

The baseline conditions show that the CSR format is optimal for the majority of matrices, although there do exist matrices best processed in other formats - for 15 matrices CSR is more than 10% slower than the optimal format. The picture remains roughly the same when contention is introduced, although some changes have occurred, particularly when only peak times are considered.

These results (as well as similar results on other platforms reported in Section 3.4.3.3) demonstrate that the choice of the best matrix storage format for matrix-vector multiplication is determined partly by the matrix itself. This echoes the findings of earlier studies into sparse matrix format selection [5]. The results also demonstrate that a fixed matrix and a fixed architecture are not enough to determine the optimal storage format - resource contention on the computing platform must also be taken into account.

2.5 Related work

This section presents related work in two areas. Section 2.5.1 discusses the literature on the automatic tuning of software by other software. Section 2.5.2 describes previous work on DynInst, with an emphasis on the measurement of costs.

2.5.1 Automatic Tuning

Automatic tuning, or *autotuning*, is defined as *tuning without human intervention, where the selection of transformations used depends on measurements of the tuning environment*.

There are many different aspects to an autotuning system. This review assesses such systems on how they address the following criteria.

- Effectiveness of tuning system
- Breadth of applicability
- Timing of tuning

	1	2	3	4	5	6	7	8
1	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -
2	- 1 - 1	- 3 - 1	- 4 - 1	- 3 - -	- 2 - 1	- 2 - -	- - - -	- 3 - -
3	- 1 - -	- 1 - -	1 2 - -	- 1 - -	- 1 - 1	- 1 - -	- - - -	- 2 - -
4	- 2 9 1	- 7 5 2	- 11 - 4	4 11 3 8	- 9 - 11	- 5 - 5	- 1 - 5	2 2 - 6
5	- - - -	- 3 - 2	- 3 - 4	- 2 - 3	3 6 2 3	- 4 - 2	- - - -	- - - 1
6	- 1 - -	- 16 - 4	3 16 - 10	1 15 - 8	- 19 - 11	3 14 3 8	- 5 - 5	1 4 - 5
7	- - - -	- - - -	- - - -	- 2 - 1	- 1 - 2	- - - -	- - - 1	- - - -
8	- 1 - -	- 7 - 3	1 8 1 6	1 7 1 8	- 8 - 7	- 2 - 3	1 2 1 4	- 2 - 4
CSR	120 126 129 131	Key	Baseline peak Baseline 10% Contested peak Contested 10%					

Table 2.1: The number of matrices (from a corpus of 141) for which different matrix storage formats satisfy certain optimality criteria. Optimality is judged on the mean execution time for 1000 multiplications.

- Implementation techniques

2.5.1.1 The effectiveness of automatic tuning

There are two interesting ways to look at the performance of an automatic tuning system. One method is to compare the performance of the tuned system to the performance of an untuned baseline. This shows whether or not the tuning is having any effect, and whether this effect is positive. The second method is to compare the performance of the tuned system to that of a benchmark implementation. This second method usually involves comparison with a hand-optimised implementation (such as mathematics libraries provided by Intel or AMD), and the goal is to show that the performance gap is acceptably small. If achieved, this means that there is a way to achieve similar performance to manual tuning with a reduced cost in time and effort.

FFTW [3] is a library that provides tuned implementations of Fast Fourier transforms. The paper describing FFTW [3] does not report explicitly on tuned versus untuned performance, but it does examine the effect of tuning effort on the final execution time. FFTW can take various shortcuts when tuning, in order to reduce overhead. For small problems, these shortcuts produce code that performs identically to the full tuning process. However, taking shortcuts on larger problems reduces performance by approximately three quarters.

SPIRAL [4] generates implementations of digital signal processing codes. The performance of the tuning process has been measured in two ways - by selectively enabling optimisations, and by a combination of search space characterisation and comparison to other libraries. The first measure compares the code generated by SPIRAL with and without vectorisation. Puschel *et al.* [4] report that the vectorised code is a factor of three faster than the scalar code, and a factor of two faster than automatic vectorisation by the compiler. This would indicate that useful work is being done when tuning the vectorised implementations. The second measure is an indirect one. To begin, Puschel *et al.* [4] measure the performance difference over a random sampling of 10^4 out of approximately 1.6×10^9 possible implementations for a particular algorithm. Even within this minuscule sample, the fastest implementation was approximately twice as quick as the slowest. In addition, slow implementations were more common than fast implementations. Therefore, achieving good performance requires non-random selection from a large state space, which requires an effective tuning process. Puschel *et al.* [4] then go on to compare SPIRAL code against a variety of vendor tuned

implementations, and show that the performance gap between SPIRAL code and the fastest hand-tuned vendor implementation is capped at $\approx 15\%$. This would indicate that the tuning process is selecting the right implementations from the vast space of possibilities.

Like FFTW, the UHFFT system [53] aims to optimise fast Fourier transforms. It follows a similar philosophy to FFTW, namely that it generates small fast baseline implementations and decomposes larger problems into these implementations. Its reported performance is competitive with FFTW [53] - it is slower on some data sizes, faster on others. It is never faster than Intel's hand-tuned library (MKL), but for large problem sizes ($2^{22}, 2^{23}, 2^{24}$) the performance is indistinguishable.

Perhaps the most well-known example of automatically tuned software is ATLAS [1, 2]. ATLAS provides a tuned linear algebra library, implementing the BLAS * API. Whaley *et al.* [1] measure ATLAS on a wide range of platforms, and compare it with the performance of the platform vendors' own BLAS implementations. For the core matrix multiplication kernel, the performance of ATLAS is always close to and sometimes exceeds that of the vendor implementations.

Other researchers have also used ATLAS as a research platform. Yotov *et al.* [54] compared it to vendor-supplied BLAS libraries as well as to their own model-driven ATLAS variant. The benchmark ATLAS implementation ran between 20% and 70% slower than vendor's BLAS implementation. Cooper and Waterman [55] use ATLAS as a benchmark for automatically tuning a compiler for use on linear algebra kernels. They demonstrate performance close to that of ATLAS.

Stephenson *et al.* [56] achieved speedups of between 3% and 54% depending on their optimisation algorithm and test methodology. Moss *et al.* [57] used supervised learning methods to derive heuristics to schedule basic blocks, and those methods that proved beneficial reduced simulated execution time by approximately 1% compared to the output of scheduling algorithm of the vendor's compiler.

The PHiPAC project [58] optimised matrix multiplication, using a combination of code generation and search. Comparison with vendor implementations showed competitive performance, and performance that vastly exceeded untuned implementations.

Im *et al.* [5] developed the SPARSITY system, which applies blocking to sparse matrices, guided by heuristics. They compared the performance of the

*BLAS stands for Basic Linear Algebra Subprograms

heuristic-tuned code to code tuned by exhaustive search, and also to untuned code. Depending on the matrix and architecture used, the speed of the method determined by heuristics was between 65% and 100% of that selected after an exhaustive search. When compared to an untuned implementation, the heuristic was between 0 and 4 $\times$ faster. When different optimisations were used, they achieved speedups over baseline code of between 1 $\times$ and 2.2 $\times$.

Buttari *et al.* [8] implement the AcCLES system for automatically blocking sparse matrices, and compare their results to SPARSITY. Relative to SPARSITY, AcCLES achieved speedups between 0.94 $\times$ and 2.04 $\times$.

OSKI [6, 7] is another system for tuning sparse linear algebra operations. Published performance data on OSKI is difficult to locate. Nishtala *et al.* [59] use OSKI to evaluate performance models of SpMV operations and show that exhaustive search among some of OSKI's possible tuning actions delivers speedups of 2.93 $\times$, compared to unoptimised code. However, this work does not evaluate OSKI's mechanism for selecting implementations.

Williams *et al.* [60] implement an automatic tuning framework that tunes applications on multicore architectures. They report speedups ranging from 1.9 $\times$ to over 130 $\times$ for various problems, architectures and baseline implementations. Kamil *et al.* [61] implement a system that optimises stencil codes by combining auto-parallelisation with automatic tuning. They report speedups of up to 22 $\times$ over scalar code, and up to 4 $\times$ over untuned parallel code.

Seshagairi *et al.* [39] see autotuning leading to a 50% drop in execution time over an untuned implementation. Sosonkina [62] showed a drop of almost 92% in execution time when autotuning is used. Ustemirov *et al.* [63] report performance improvements that peak at approximately 54% over an untuned implementation, depending on the degree of disk I/O contention in the system being tuned. Lee and Eigenmann [40] report execution time reductions of approximately 30% on average, peaking at almost 70%. Thomas *et al.* [64] use runtime algorithm selection to beat static algorithm selection by approximately 15% on parallel sorting codes, and by around 4% on parallel matrix multiplication.

Stephenson and Amarasinghe [65] used supervised learning to achieve performance 1% better than a hand-optimised heuristic, for a fraction of the effort.

Vuduc *et al.* [66] automatically tune a sparse triangular solution routine by selecting blocking factors for the sparse matrix and by selecting the point at which the matrix is split into sparse and dense submatrices. The report performance increases of up to 1.8 times over a baseline implementation.

In summary, much of the published work on automatic tuning reports signif-

icant speedups. It is worth asking why this state of affairs should hold. Only Yotov *et al.* [54] report a slowdown. One possible reason is that most automatic tuning works are applied to a small set of domains, wherein tuning is already known to be advantageous. This is not a pejorative statement: when a domain is of broad interest, then in-depth investigation into extracting maximum performance on the domain is merited. However, a narrow application of automatic tuning systems might explain why there are few reported failures. As discussed further in Section 2.5.1.2, many of the systems referenced herein are not fully general, being aimed at specific applications or specific computational kernels. Looking at the problem areas involved, there is a heavy bias towards linear algebra [1, 2, 58, 6, 7, 64, 40]. Signal processing is also a common target [3, 4, 53], as is the numerical solution of systems of equations [62, 67, 68].

2.5.1.2 Generality of autotuning

The generality of an autotuning system is a measure of the range of programs to which it can be applied without extra development. When looked at in this manner, there is a spectrum of generality. At one end lie systems that tune single programs [69, 39, 63, 62, 67, 70]. At the middle of the spectrum are systems that tune libraries, or common computational kernels [1, 2, 58, 6, 7, 3, 53, 60, 37, 71, 72, 38, 68, 73, 61, 66, 5]. At the second end of the spectrum sit systems that tune compilers [55, 56, 57, 65, 74].

While general applicability is a desirable attribute of a tuning system, specialised tuning systems are an important area of research. Such systems, while capable of tuning many programs, rely on being customised for each target. This customisation could involve specifying a list of tunable parameters provided by the target, bounding the search space of possible tuning actions using target-specific heuristics, or using knowledge of the target's source code such as the names of various subroutines.

There are many applications that have been subjected to specialised tuning of this nature. Some examples are the chemistry package GAMESS [69, 39, 63] and five applications examined by Imai *et al.* [70]: the weather prediction programs ARPS and MM5, the molecular dynamics codes GROMACS and LAMMPS and the fluid analysis program FrontFlow Blue.

Automatically tuned libraries are more generally applicable than systems which tune specific programs. Once a library and its tuning mechanism have been written, they can be deployed and linked against many different programs with-

out further development. The development costs of that library can be amortised to a greater extent than can the costs of program-specific applications. However, generality comes with a penalty: the library will define a set of assumptions of how it can be used, embodied in its API. In order to use the library with any specific application, the application must place on the library a set of demands that conform to these expectations. If the application does not satisfy this constraint, then it must be changed. Depending on the structure of the application, the development of a program-specific tuning system may require less effort. When a widely used, general API such as the BLAS is used as the interface to a tuning system, this situation is lessened, but it is never eliminated completely. For users of a specific problematic application, the fact that a given system works for many other programs is irrelevant - they only care that it doesn't work for theirs.

At the most general end of the spectrum are systems that target compilers and runtime systems. The tuning of compilers usually aims to improve the quality of compiled code, or to shorten compilation times without markedly reducing the performance of generated code. This is often implemented by changing the heuristic functions used within a compiler to guide its actions. In such cases, the changed compiler can be applied to any program in its source language, although the effectiveness of the change may vary depending on how the change was constructed. Target-agnostic runtime systems can also improve performance. Garbage collectors can improve the locality of data structure placement in memory [75], and memory migration on non-uniform memory architecture (NUMA) systems can also aid performance [76].

2.5.1.3 The timing of tuning

Timing is an important attribute of an autotuning system. Interest lies in the timing of two related but distinct events: the invocation of tuning actions, and the acquisition of the rules governing such invocations.

Tuning actions can be invoked at many stages within a program's life-cycle. Some tuning systems operate on models of the target system and invoke their actions during the design of the tuned system [77]. While useful for new programs, this does not help when tuning existing software. A more common approach is to invoke different tuning actions while installing the tuned system - various possible configurations are tried out experimentally, and the best configuration is installed. This approach has no overhead during the execution of the system, but there are limits to what it can tune for: it cannot measure circumstances

that do not exist on the target system during install time. This is fine when tuning to architectural parameters such as the cache size, but it does not take into account changing load on a non-dedicated system, nor execution on a virtual machine subject to migration, where architectural attributes - such as cache sizes - can change underfoot. The offline tuning approach has been implemented by SPIRAL [4] and others [73, 72, 70, 55, 56, 57, 65, 5].

A different approach is to defer tuning until the latest possible moment - indeed, to continue tuning throughout the life of the tuned program. This approach allows the tuning algorithm to adapt to changing conditions, but it carries the drawback of increased overhead, both for the tuning mechanism itself and for the measurement systems that gather the input data on which the tuning system bases its decisions. Systems that perform runtime tuning include OSKI [6, 7] among others [71, 37, 38, 62, 69, 39, 63, 40, 64, 8].

Conceptually, all tuning systems are composed of rules of the form *when circumstances satisfy condition X, perform action Y*. Runtime automatic tuning systems differ in when these rules are laid down. Some systems determine their rules at installation time after running a set of benchmarks. The *application* of the rules may occur at runtime based on the data being processed, but the rules themselves are immutable once formed. This approach is taken by OSKI [6, 7] and other systems [71, 62, 78, 69, 63, 64, 65, 8].

Such systems are, in theory, all that are needed - with a large enough set of rules, any conceivable circumstance can form the *X* component of a rule. However, there are two practical problems with such an approach. The first is that such a system must experience, during its offline learning phase, enough circumstances that it can construct its rules. It need not experience every possible circumstance if it has a method of extrapolating rules, but it needs both this method and a body of knowledge large enough that its extrapolations are accurate. The second problem with such an approach is the number of rules. A balance needs to be struck between the resolution of the variable *X* and the space required to store the rules. Storage requirements can be reduced by employing coarse generalisations, a strategy that also lessens the potential of the learner. Letting the learner discriminate at a higher resolution potentially improves its performance, but increases the storage requirements. The storage requirements arise in two forms: long time storage on permanent media and short term storage in RAM. This gives rise to another consideration: large numbers of rules will reduce the size of the memory cache available to the application being tuned, and (depending on the application) may increase the number of conflict and capacity

cache misses experienced by the application.

Instead of a static ruleset, it is possible to use a dynamic set of rules that vary during the lifetime of the tuning agent. Such a set contains only those rules that apply to the current state of the system. Dynamic sets of rules also face the resolution versus space trade off present in static systems. In addition, tuning systems employing dynamic rulesets must not only be able to discover a good set of rules (for example, by using a search algorithm), they must also be able to detect the obsolescence of the current ruleset and to build it anew. Well-known methods (such as ϵ -greedy action selection as described in Section 2.2.1) can achieve this goal, but they impose additional costs on such a tuning system. Existing systems that update their action selection rules at runtime include the proposal by Suda [37], the SILC system [38], the tuning of GAMESS [39] and Lee and Eigenmann's distributed system [40].

2.5.1.4 Tuning mechanism

The choice of tuning mechanism depends on when tuning takes place. If the tuning occurs before the system is running, then the tuning mechanism is typically a parametrised code generator. The parameters are selected by a search algorithm, and code is generated, compiled, executed and evaluated. Once an implementation has been selected, tuning halts. This is the method employed by ATLAS [1, 2], SPIRAL [4], SPARSITY [5] as well as other systems [73]. Another method is to make use of tuning actions already present within the target application, when tuning actions are the selection of command-line switches. This is the approach taken by various parameter-tuning systems [70, 55]. Stephenson *et al.* [56] modified a compiler to read in heuristic functions as expression trees from an external source, and then used machine learning to select good expression trees. Stephenson and Amarasinghe [65] modified a compiler to query a tuned classifier during its compilation work.

Another method is to have a parametrised dispatch routine as the interface to the selection of different implementations. This dispatch routine invokes a particular implementation based on its parameters, and these parameters can be found by the calling code through communication with an agent that controls the tuning process. There are two possible ways for this mechanism to work, which are here termed the *push* and *pull* methods. The *push* method is shown in Algorithm 2.4. In this method, the computational kernel does not query the tuning agent to determine the appropriate action. Instead, it provides a way

for the tuning agent to asynchronously set a new parameter, which will be used on the next loop iteration. This method is used by the NICAN framework [79], which is used in several automatic tuning systems [69, 39, 63, 62].

In contrast, the *pull* mechanism from Algorithm 2.5 queries the tuning agent whenever it decides a new tunable action may be required - this is not necessarily on every iteration. This approach has been used by Thomas *et al.* [64]. Lee and Eigenmann [40] also use a method of this type, although they lack a distinct tuning agent. They use MPI calls to gather raw data from their system and compute new data partitions at the start of tuned loops.

The pull method is simpler to implement than the push method, but harder to tune - selecting the correct frequency at which to poll the tuning agent may be hard, especially since this frequency is likely to vary as the program goes through different phases of execution. In contrast, the *push* method incurs synchronisation costs on every loop iteration, but tuning is easier because the learning agent can limit communication costs to those times when it decides a tuning action is necessary. It is in a better position to make that decision than the computational loop. Note that in both cases, it may well be necessary for the computational loop to send performance measurements (such as timing data) to the tuning agent.

Algorithm 2.4: The *push* style of parametrised dispatch routine

```

1 while computing do
2   lock()
3   local_copy = current_action
4   unlock()
5   dispatch_tunable_action( current_action )
6 end

```

Algorithm 2.5: The *pull* style of parameterised dispatch routine

```

1 while computing do
2   parameters = query_tuning_agent()
3   dispatch_tunable_action( parameters )
4 end

```

Another method for implementing tuning is to change the address of the currently invoked tunable function. As an example, consider the sequence in Algorithm 2.6. This sequence is just a loop which invokes a function that carries out a tunable action. When the tuning agent decides to change the action, the agent modifies the instruction stream of the sequence so that the instruction that used to jump to `tunable_action_A` now jumps to `tunable_action_B`. This approach

is similar to the *push* style discussed above, but it doesn't require modification to the source code of the tuned program. It carries the secondary advantage that tuning can be done periodically, and when it is not required, there is zero additional overhead within the tuned program. This is not possible if the source code has been modified, although depending on the nature of the modifications, the overhead in such cases can be reduced to only a few instructions. This may or may not be significant depending on the programs involved.

Algorithm 2.6: An example computational loop, which is tuned through runtime binary modification by an external agent.

```

1 while computing do
2   tunable_action_A( parameters )
3 end
```

2.5.2 DynInst

DynInst has been around for over ten years. In that time, it has been applied to solve a wide range of problems. These problems include the analysis of application caching behaviour [80, 81, 82, 83, 84, 85], memory page placement and migration in multi-processor environments [86, 87, 88], detecting strided memory accesses [89], dynamic recompilation for energy efficiency [90] and measuring statement coverage [91, 92]. It has been incorporated into a variety of higher-level instrumentation tools, including DynTG [93] and DPCL [94].

Knowing the execution time overheads imposed by DynInst is useful when designing a novel DynInst-using system (such as Swift). This knowledge influences the choices made when designing the new system. Many previous works have not been concerned with the costs of using DynInst, as they use DynInst to gather profiling information for offline analysis and application [80, 84, 95, 81, 82, 91, 92, 89, 85, 83], or with the description of higher-level tools rather than with the application of those tools [94]. Other works have utilised DynInst to tune a program while it is running, but the timing results of these papers cover each system in its entirety [88, 86, 87, 90, 93]. Such data is not useful when assessing the manner in which DynInst can be best utilised in a new system.

The work by Zhang and Jacobsen [96] did include measurements of the costs of DynInst, exclusive of a surrounding system. The work evaluated the cost of using DynInst to insert code into a process as a function of the size of the inserted code. Also reported was the increase in execution time resulting from using code which has been inserted by DynInst compared to using the same code inserted

at compile-time.

Although interesting, these measurements have several drawbacks. The only data shown are two graphs, which plot time against increasing number of benchmarked operations. It is not clear what the time quantity is measuring, wall-clock time or CPU time. In addition, only one of the graphs includes the units in which time is represented. The only information provided about individual operation costs is what can be read off the graphs. In addition, the paper fails to mention several facts that are important when trying to apply existing performance results to a proposed new system. Such facts include the compiler used, the version of the compiler, operating system and DynInst and the size of the caches.

Chapter 3

The Learning Agent

3.1 Introduction

This section details the learning algorithm developed for Swift and evaluates its performance in a range of different scenarios. The work herein is based on two papers by Armstrong and Rendell [35, 36]. Both papers investigate sparse matrix format selection. The first paper presented a simple algorithm solving a simplified problem. The second paper enhanced the algorithm and added complexity to the problem. The work in this chapter extends the papers in the following ways:

1. In order to work within a non-simulated environment, the sparse matrix-vector multiplication algorithms require additional logic. The timings used in these experiments incorporate this additional logic.
2. Where Armstrong and Rendell [36] investigate tuning for changes in architectural properties such as cache size, this chapter applies the tuning algorithm to adapt to changing runtime conditions on a static architecture.
3. The tuning algorithm has been enhanced to facilitate its use in non-simulated environments. Experimental results in this chapter were obtained using this enhanced algorithm.

The remainder of this chapter is organised in the following manner: Section 3.2 describes the tuning algorithm used. Section 3.3 details how simulated evaluations are used to evaluate the learning agent. A series of experiments are described and discussed in Section 3.4. The chapter is summarised in Section 3.5.

3.2 Design of the learning algorithm

This section describes the algorithm used to adapt an application to changing runtime conditions. It provides a general outline of the algorithm's logic, with subsections providing more detail on specific aspects.

3.2.1 A framework for discussion

The subject of this chapter is a learning algorithm designed to tune a specific class of application. Programs in the class are defined by two characteristics:

1. They possess regions with iterative structure, and
2. They contain functions which are interchangeable in that they solve the same problem, albeit in different ways - for example, functions implementing different sorting algorithms.

Algorithm 3.1 shows a specimen of this class. The function `tunable()` is interchangeable with a suite of other functions that achieve the same logical effect in a different manner. The agent's goal is to select the functions which minimise execution time.

Algorithm 3.1: Example of the target program class.

```
1 for  $i \in [1, n]$  do
2   tunable()
3 end
```

The rest of this section discusses the agent in the context of a generic program of the target class. In some places, however, the discussion is made concrete with illustrative examples. These examples are drawn from the program in Algorithm 3.2. This program executes several matrix-vector multiplications. If the matrix is *sparse*, then there exist many different in-memory arrangements of the matrix, along with an associated set of algorithms for performing the multiplication *. The goal of the learning algorithm when tuning such a program is to select the quickest representation.

*See Section 2.4 for more details on sparse matrices

Algorithm 3.2: Example of the target program class.

```

1  $A \leftarrow$  a matrix
2  $x \leftarrow$  an initial vector
3 for  $i \in [1, n]$  do
4    $y \leftarrow y + Ax$ 
5 end

```

3.2.2 Overview

The algorithm works as part of a larger system, shown at a high level in Figure 1.2. The goal of the algorithm is to tune a specific application (*tuned application* in the figure) which requires hardware resources to execute. There is contention for these resources from other applications. Tuning is carried out by a *tuning driver*, which contains the code required to modify the target application. The tuning driver requests instructions from the *learning agent*[†], which also receives feedback from the (modified) tuned application. The communication pattern between the three main components is shown in Figure 3.1. Messages in this system are of three types: *requests*, *responses* and *reports*. The tuning driver makes requests of the learning agent and each request encodes the current state of the tuned application as a set of attributes. The learning agent responds with the tuning action to be undertaken and the number of iterations for which this action must be applied. The tuned application sends reports to the learning agent, each report containing information about the application's performance. The code to generate and transmit these reports is injected into the tuned application by the tuning driver.

The rest of this section provides more detail on different aspects of the learning agent. These aspects include generalisation from observations to react correctly to novel situations (Section 3.2.3), handling requests from the tuning driver (Section 3.2.4), the acquisition of new knowledge (Section 3.2.5) and estimation of loop counts within the tuned application (Section 3.2.6). Section 3.2.7 investigates the effect of some important parameters on the algorithm's behaviour.

3.2.3 Generalisation

Swift generalises over its inputs by using *tile coding*, a method introduced in Section 2.2.2. As will be discussed in Section 3.4.1, Swift has been evaluated on

[†]Also referred to simply as the *agent*.

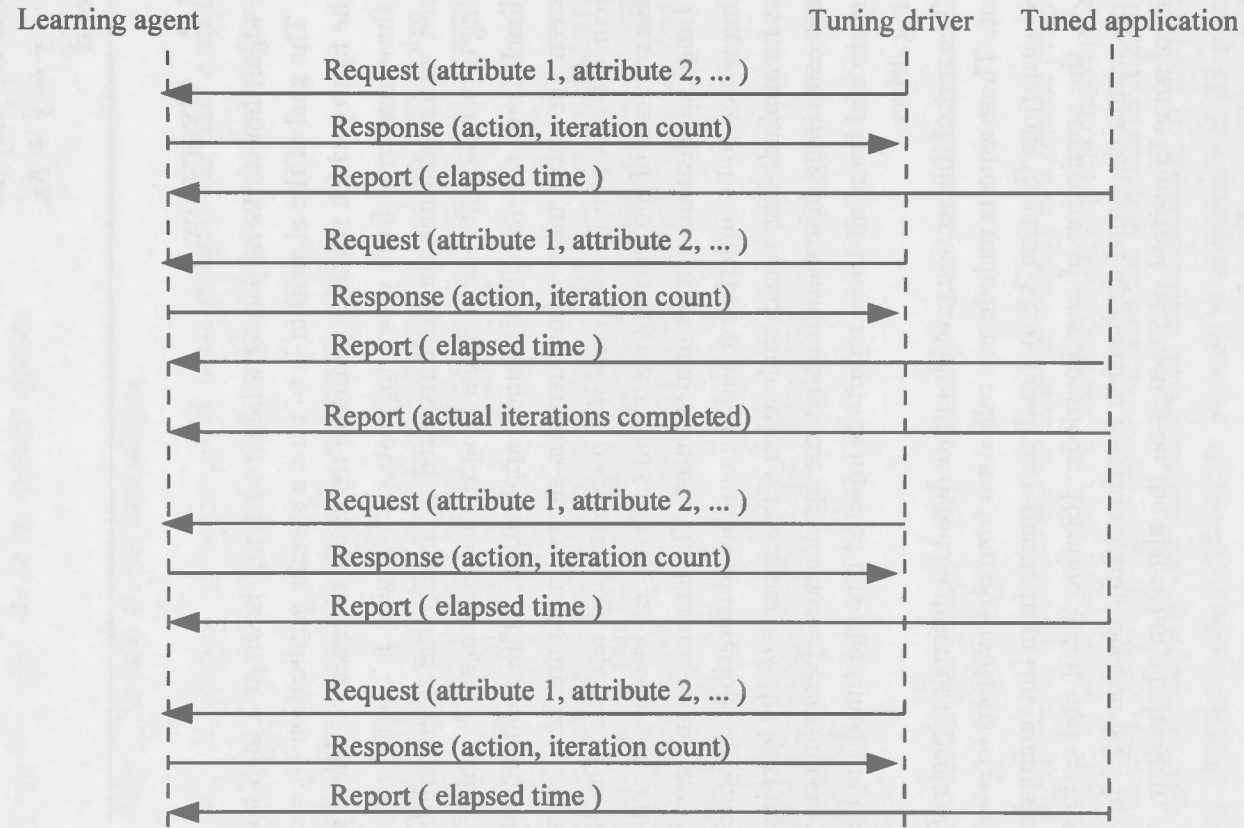


Figure 3.1: The communications pattern when tuning an application.

an example problem with five attributes. Each attribute has its own set of tiles, so that any perceived input is mapped onto a five-dimensional grid. The number of tiles in the grid of attribute a_i is determined by the *granularity* g_i , along with the range of values a_i may take on. Each of these parameters is hard-coded by the algorithm's user.

3.2.4 Handling action requests

This section describes how the agent processes requests for action recommendations. A high-level view of the agent's logic is presented in Figure 3.2. The figure refers to an *experiment scheduler*, which is the means by which the agent acquires new knowledge. The scheduler is described in Section 3.2.5. The rest of Figure 3.2 refers to the selection of actions based on attribute requests.

Action selection in the agent is a probabilistic process. Each action is associated with a weight and an action is selected with a probability proportional to its weight. More formally, consider a set of n positive weights $w_1, w_2, \dots, w_n$ with sum T . Let S_k denote the partial sum $\sum_{i=1}^k \frac{w_i}{T}$. Let r be a random value drawn from a uniform distribution on the interval $[0, 1)$. Define m so that $S_m \leq r < S_{m+1}$. Then the m^{th} action is selected.

To understand how the value of a weight changes, consider a specific weight w_i , associated with the i^{th} action a_i . When the algorithm begins, a parameter provides an initial weight value: $w_i = w_{\text{initial}}$. After a period of time, the algorithm may observe that a_i is not an optimal action. When this occurs, the value of w_i is updated to be $w_i \leftarrow w_{\min}$, where $w_{\min}$ is an algorithm parameter providing a lower bound on a weight's value. More details on selecting values for $w_{\min}$ and w_{initial} are presented in Section 3.2.7.

The lower bound $w_{\min}$ ensures that exploration occurs. *Exploration* refers to a classic trade-off in reinforcement learning systems, typically phrased as *exploitation versus exploration*. This was discussed in Section 2.2.1. Swift's method for balancing exploration and exploitation is based on the weights associated with different actions. Recall that weights are reduced to $w_{\min}$, a specific non-zero floor. This ensures that even weights with low probability are still selected occasionally - thus enforcing exploration. Exploration is also tied to the level of uncertainty in the agent's knowledge - an uncertain agent has more weights with greater values and the associated actions will be selected with a higher frequency. Controlling exploration in this way also provides a more granular approach: there is a different exploration probability for each set of weights and since each grid

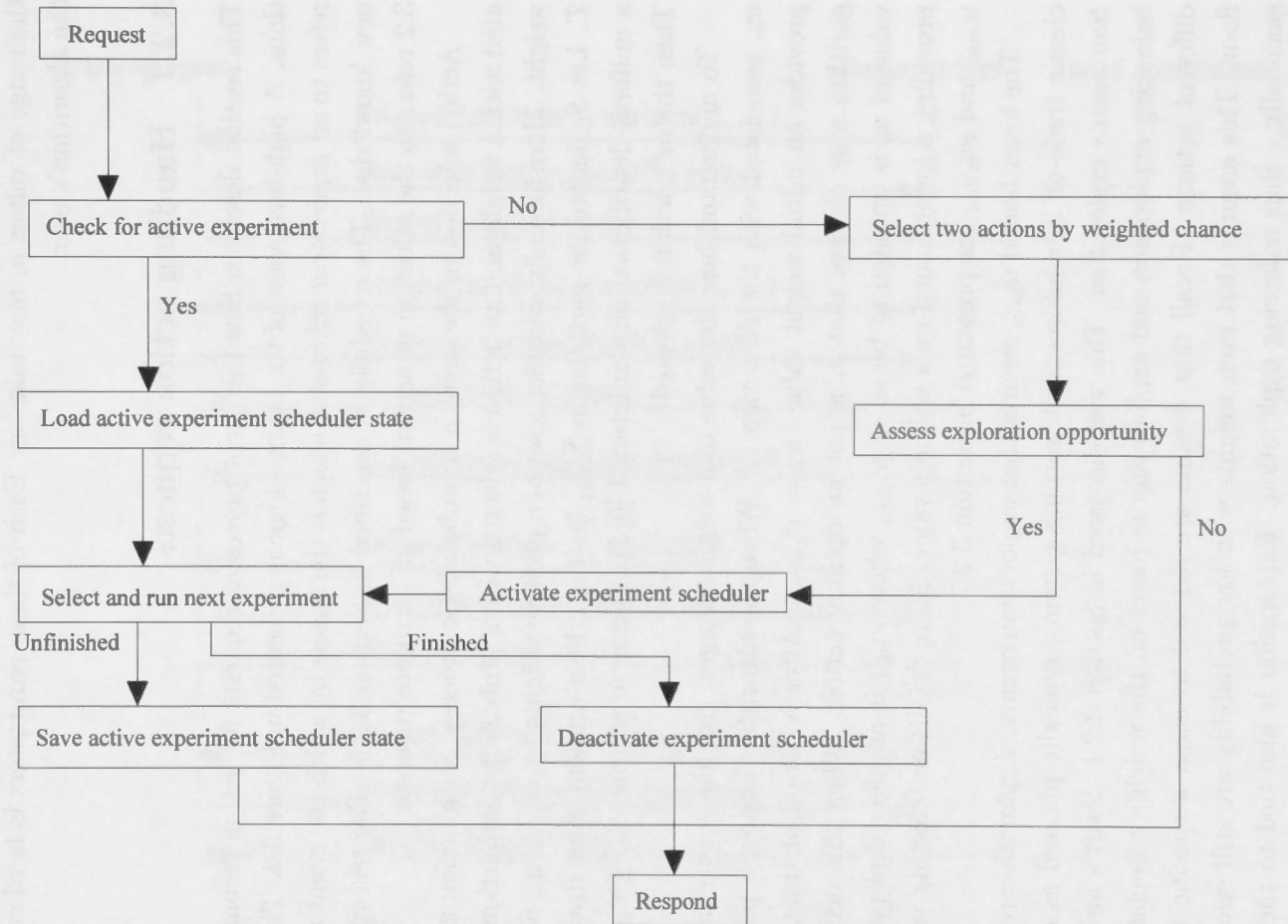


Figure 3.2: The high level procedure followed by the tuning agent when responding to requests for action recommendations.

element has its own set of weights, this gives each grid element an independent rate of exploration. This helps to focus exploration on those areas where it is most needed - if particular grid elements have actions whose difference is hard to discern, exploration can continue in that element while other elements reduce their exploration chance and avoid expensive exploratory actions.

3.2.5 Experiment Scheduler

As shown in Figure 3.1, the agent sends out response messages of the form *take action A and persist with it for t iterations*. The feedback from this action consists of a single data point: the execution time required to perform the action. To estimate the true cost of an action, multiple measurements are needed. Gathering these requires multiple feedback messages for the same action. Comparison of two actions requires further response messages to acquire two sets of data points. There is a further complication: what is best value of t ? The ideal value is small enough to provide quick feedback and to minimise the execution time when A is not the optimal action, but large enough to be accurately measured given the resolution of the timer being employed. The best value of t will differ for different matrices and so setting it to a fixed value (such as an algorithm parameter) is overly crude.

The *experiment scheduler* is a component of the agent that addresses these issues. Figure 3.3 shows the logic driving the scheduler.

In this figure, the notation (A, t) means *deliver a response using action A for t iterations*. The figure is complicated by the transitions needed to handle certain exceptional conditions. These corner cases will be discussed later, but to begin with, they will be ignored to better illuminate the core logic behind the state machine.

The state machine starts by triggering any required conversion to format A . The feedback from this action is discarded, as the algorithm doesn't know whether a conversion took place and hence it doesn't know what the feedback measurement is measuring. Once any required conversion has taken place, the state machine proceeds by selecting t_A . This is achieved by trying successive values in the sequence $t_A = 1, 2, 4, 8, \dots$ until the feedback execution time is greater than the algorithm parameter threshold *min.time*. Once a sufficiently large value of t_A has been discovered, the algorithm returns the response (A, t_A) until sufficient responses have been gathered - the required number is the algorithm parameter *samples_per_action*. The time reported in each response is divided by t_A to de-

termine the arithmetic mean. These actions are then repeated for format B - conversion, selection of t_B and accumulating the required number of responses. After these steps the algorithm has two sample distributions, each defined by *samples_per_action* arithmetic means. By the Central Limit Theorem, these distributions are normal [97]. This allows the use of a two-sided t -test to compare the means (μ_A and μ_B) of the two distributions. This test computes the probability that:

1. The two sample distributions are drawn from populations with the same distribution, and
2. The observed difference $\mu_A - \mu_B$ arises purely by chance.

If this probability is less than another predefined parameter α , the difference is treated as statistically significant. In these cases, the action with the greater mean has its weight reduced to w_{min} .

The final steps of the state machine test the distribution of weights, which is done by calculating the mean weight μ_{weight} and comparing it to a threshold $min_{\mu_{weight}}$ which is calculated based on the number of actions available (ψ), the lower bound on action weights (w_{min}) and the initial action weight ($w_{initial}$). Given these, the arithmetic mean of a set of weights with at least one possibly optimal action is $k = \frac{(\psi-1)w_{min} + w_{initial}}{\psi}$. The reset threshold is then defined as $min_{\mu_{weight}} = ck$, where c is a constant less than 1. Further constraints on the value of c are discussed in Section 3.2.7.2.

If $\mu_{weight} < min_{\mu_{weight}}$, then *every* action has been observed to be slower than some other action. This implies that the environmental conditions under which timing data were gathered have changed, so that the format that once was optimal is no longer so. There are two causes of such a change:

1. External causes including changes in system load or the degree of contention for resources
2. Internal causes arising from failures of generalisation.

The agent is designed to handle external changes. To this end, once sufficiently low μ_{weight} has been observed, all weights are reset to $w_{initial}$ and the algorithm begins searching for optimal actions under the new conditions. Generalisation failures represent cases where two different problems are perceived as being the same and yet have different optimal actions. The perception of similarity arises from one of two causes: either the attributes used to measure the

problem are insufficient to capture performance-determining distinctions, or the grid onto which attribute values are mapped has too large a granularity. The former cause requires the human operator to rethink the attributes being used. The latter cause can be handled as an external cause: the weights are reset to $w_{initial}$ and learning continues.

3.2.6 Iteration count estimation

The algorithm presented above relies on knowing how many iterations a given problem will require. This enables it to inform its interrogator of how many iterations to execute with the currently recommended action.

As one of the algorithm's inputs is a count of iterations elapsed, the algorithm can estimate the number of iterations left if it can estimate the total number of iterations. The number of elapsed iterations is easy to gather from within the target application - it requires only the addition of a counter. Gathering enough data to estimate the iteration count could require more complex logic, especially if the iteration count is not constant but varies with different problem instances. For this reason, estimation of the total iteration count is handled by the agent, rather than being required as an additional input.

There are two estimation methods currently available to Swift. Each is designed to address a particular class of target application. The *global estimator* is applicable when the iteration count of the target application's tuned loop is confined within a single narrow range of values. The *windowed estimator* is designed for target applications whose loop iteration counts are stratified - they can be confined to one broad region or several narrow regions. Figure 3.4 shows an example of the two kinds of loop iteration count dynamics. The crucial difference between the two estimation methods is that the windowed estimator discards old data - it generates estimates based on a sliding window of the most recent samples. The need for this is illustrated by Figure 3.5(a), which shows the estimates produced by the global estimator for stratified iteration counts. It is evident that the first method does not perform well in such cases - Figure 3.5(b) shows the error in the estimates it produces. When Swift is applied to tune loops with small iteration counts, these errors become significant.

3.2.6.1 Global estimation

The *global estimator* assumes that iteration counts can be modelled by a Gaussian distribution, with mean μ and variance σ^2 . A running estimate of the mean and

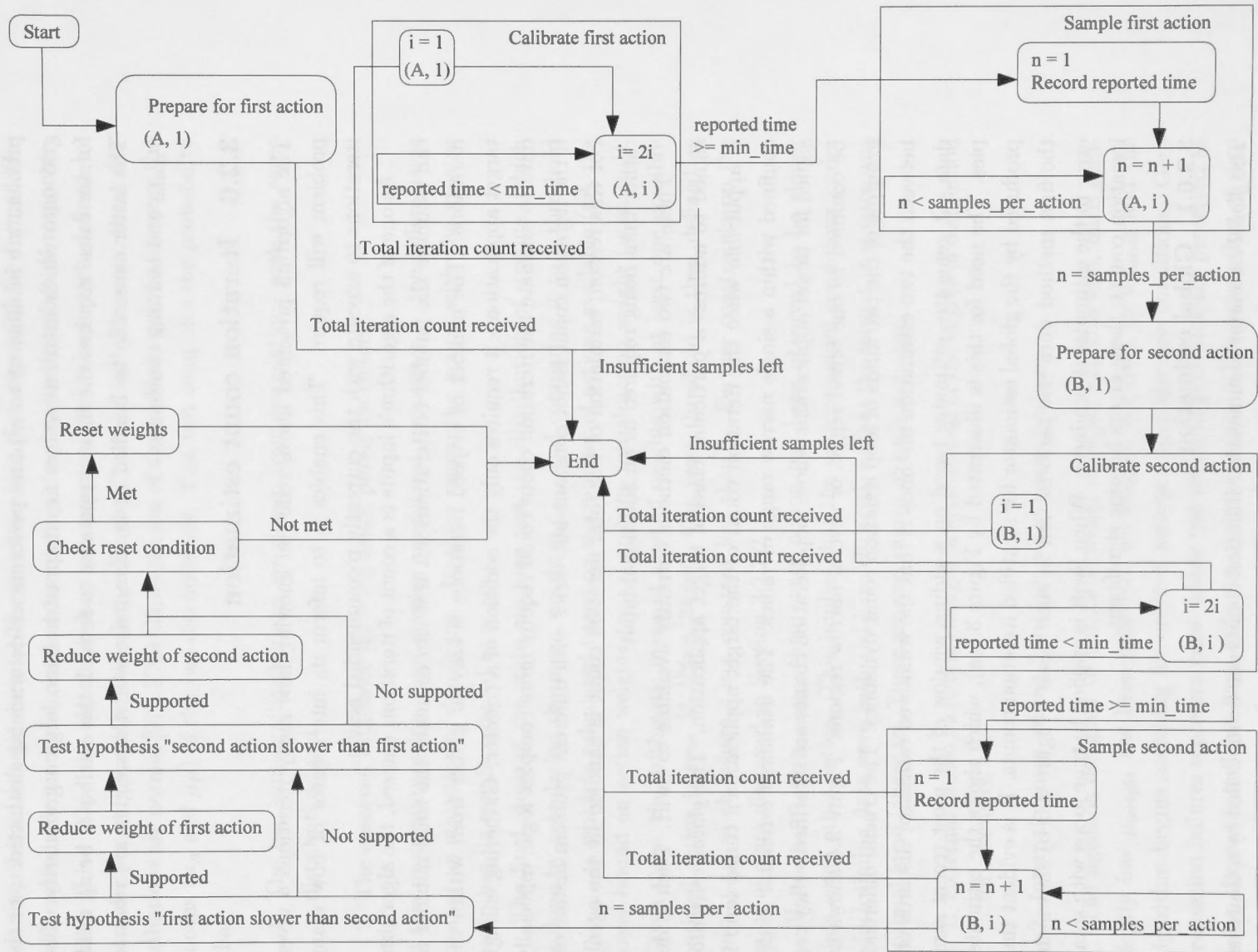


Figure 3.3: The steps undertaken by the experiment scheduler. The notation (A, t) denotes a response sent to the tuning driver that instructs action A to be implemented for t iterations. Execution of the state machine pauses after each such message, resuming only when the next request is received from the tuning driver.

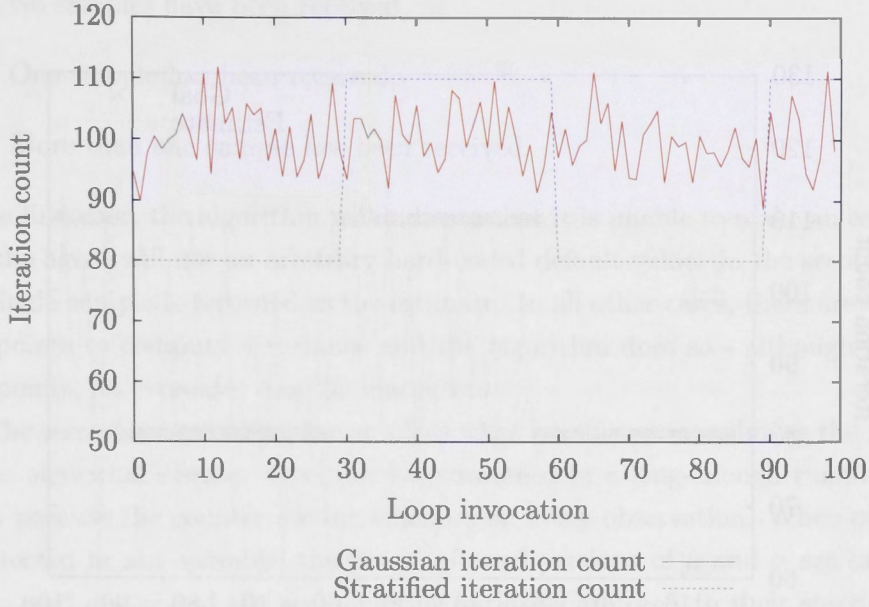


Figure 3.4: Different models for loop iteration counts.

standard deviation of the iteration count is maintained. Given input x_i as the i^{th} observed iteration count, the following updates occur:

$$T_i = T_{i-1} + x_i \quad (3.1)$$

$$\mu_i = \frac{T_i}{i} \quad (3.2)$$

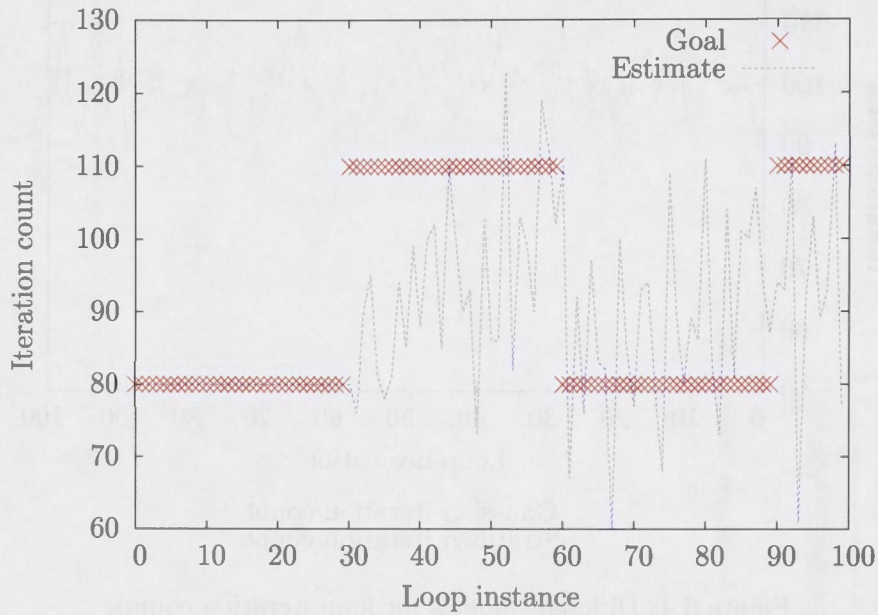
$$a_i = a_{i-1} + \frac{x_i - a_{i-1}}{i} \quad (3.3)$$

$$\sigma_i = \sigma_{i-1} + (x_i - a_{i-1})(x_i - a_i) \quad (3.4)$$

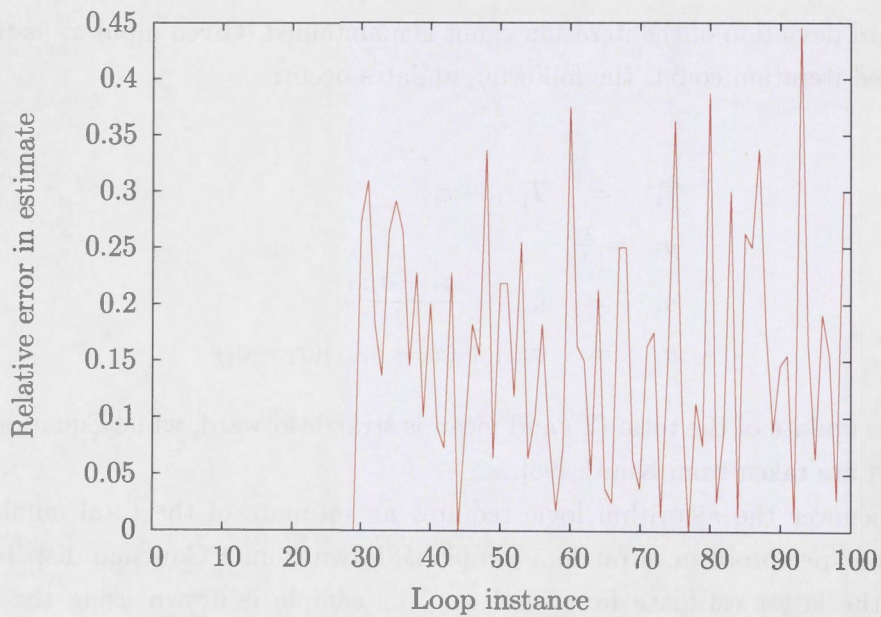
The update of the total (T) and mean is straightforward, while Equations 3.3 and 3.4 are taken from Knuth [98].

Whenever the algorithm logic requires an estimate of the total number of iterations per problem, a random sample is drawn from a Gaussian distribution using the latest estimate for μ and σ . The sample is drawn using the GNU Scientific Library (GSL) routines for pseudo-random number generation (PRNG).

There are two complicating factors in using this estimation procedure. First, what happens when an iteration count is required, but insufficient data has been observed? The current algorithm recognises three distinct volumes of previously acquired data:



(a) Global estimation of stratified iteration counts



(b) Error of global estimation of stratified iteration counts

Figure 3.5: Applying global estimation to stratified data.

1. No samples have been received,
2. One sample has been received,
3. More than one sample has been received.

In the first case, the algorithm will indicate that it is unable to make an estimate and the agent will use an arbitrary hard-coded default value. In the second case, the single sample is returned as the estimate. In all other cases, there are enough datapoints to compute a variance and the algorithm does so - although for few datapoints, its estimates may be inaccurate.

The second complicating factor arises when overflow occurs during the update of the algorithm's state. Overflow is guaranteed in a long-enough running program because the counter i is incremented on every observation. When overflow is detected in any variable, the pre-overflowed versions of μ and σ are saved as μ_{legacy} and σ_{legacy} and the accumulating variables are reset to their starting values. For the next handful of iterations, requests for iteration count estimates will be satisfied using μ_{legacy} and σ_{legacy} - this is done to avoid providing estimates based on small numbers of samples, which could be misleading. The use of μ_{legacy} and σ_{legacy} can also mislead if the iteration count changes rapidly, although this problem can be addressed by tuning the period for which the legacy distribution is used. In this thesis, the period is arbitrarily defined as 1,000 samples.

3.2.6.2 Windowed estimation

When the iteration count varies between multiple widely-separated values the accuracy of estimation can be improved by basing estimates only on the most recent data. This is the approach taken by the *windowed estimator*. This estimator is parameterised by a window size W and stores the most recent W samples. When an estimate is required it is calculated as the arithmetic mean of this data set. This estimation method uses more space than the global estimator because it must store each sample in the window. The value of W must be selected by the user to reflect the expected behaviour of the target application.

3.2.7 Parameter tuning

The algorithm presented above has several parameters that control its execution. This section investigates the effects of some important parameters on the algorithm's behaviour, in order to guide the selection of appropriate values.

3.2.7.1 Weight parameters $w_{initial}$ and w_{min}

As described in Section 3.2.4, actions are selected based on learned weights and exploration occurs when two different actions are selected. By tuning the weights set by the algorithm, an operator can control the degree to which the algorithm exploits its knowledge, trading this off against the speed of response to changing circumstances. To see the effect of different weights, consider the algorithm executing with ψ possible actions, of which $\psi - 1$ have had their weight reduced to the minimum threshold. This scenario represents a temporary converged state in the sparse matrix format selection problem which will be used in later experiments. The convergence is temporary because it relies on environmental conditions not changing and the experimental setup is designed to change them periodically.

Weight values are bounded by two parameters, the initial weight ($w_{initial}$) and the minimum weight (w_{min}). In the converged state, the chance of selecting the optimal action is $p = \frac{w_{initial}}{w_{initial} + (\psi - 1)w_{min}}$ [†] and the chance of selecting a specific non-optimal action is $L = \frac{w_{min}}{w_{initial} + (\psi - 1)w_{min}}$. An exploration takes place when two successive action selections return different actions. The chance of this happening is $1 - (p^2 + (\psi - 1)L^2)$.

Figure 3.6 shows how the chance of exploring at each opportunity is determined by different values of $w_{initial}$ and different values of ψ . Only one w_{min} is shown, as the values of p and L depend on the ratio of $w_{initial}$ to w_{min} and not on the absolute value of the two weight parameters. The plot shows that increasing the number of actions increases the exploration chance for a given $w_{initial}$, but does not change the scaling behaviour. Exploration chance is robust against changes in $w_{initial}$ when the ratio of $w_{initial}$ to w_{min} is markedly high or low, but the exploration is sensitive to changes in the middle range of $w_{initial}$ values.

In most cases, a low exploration rate in the converged state is desirable as every exploration incurs the cost of acting non-optimally. In a dynamic environment where continued exploration is required, a low exploration rate makes the agent less able to react to environmental change. To illustrate, take the previous example setting and assume that the single action with weight $w_{initial}$ is no longer optimal - another action with weight w_{min} is now the best choice. The next time an action selection is requested of the agent, the ideal response is to explore with the newly optimal action. The probability of selecting this action followed by a

[†]If this expression appears strange, recall that weights are not probabilities because they do not sum to 1 - this expression scales weights to turn them into probabilities.

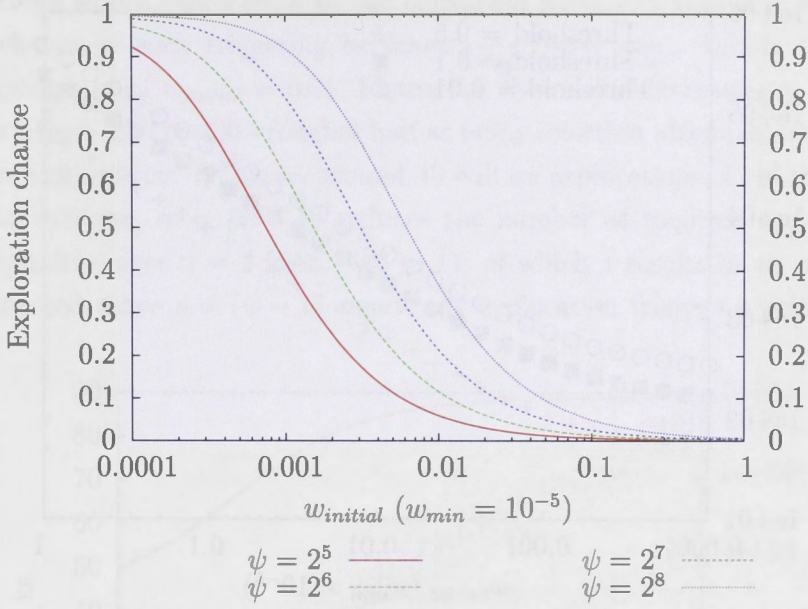


Figure 3.6: The dependence on the chance of exploration on $w_{initial}$.

different action [§] is

$$z = \frac{w_{min}}{w_{initial} + (\psi - 1)w_{min}} * \left(1 - \frac{w_{min}}{w_{initial} + (\psi - 1)w_{min}}\right)$$

The probability of having *not* undertaken the desired exploration after k opportunities is $(1 - z)^k$. The question then arises: how does $w_{initial}$ influence this probability? In particular, given some threshold t , how does the choice of $w_{initial}$ determine the value of k required such that $(1 - z)^k = t$? Figure 3.7 plots $k = \frac{\log_{10}(t)}{\log_{10}(1-z)}$ for different values of $w_{initial}$. Data are plotted for three values of t . The red crosses show $t = 0.5$, the point at which the desired exploration has an equal chance of having been performed or not. The solid green squares show $t = 0.1$, when the chance of the desired action being chosen is no less than 90%. The blue circles show $t = 0.01$, when there is no more than a one in one hundred chance that the desired action has not been selected. In all cases, the values of k (shown on the vertical axis) are large, even for $w_{initial} = 10^{-4}$, just under 50 action selections must be made to achieve the break-even point.

Comparing Figure 3.6 and Figure 3.7 shows that making the learning agent more reactive (that is, reducing the value of k) requires either changing the learn-

[§]Recall that exploration only occurs when successive action selections result in different actions

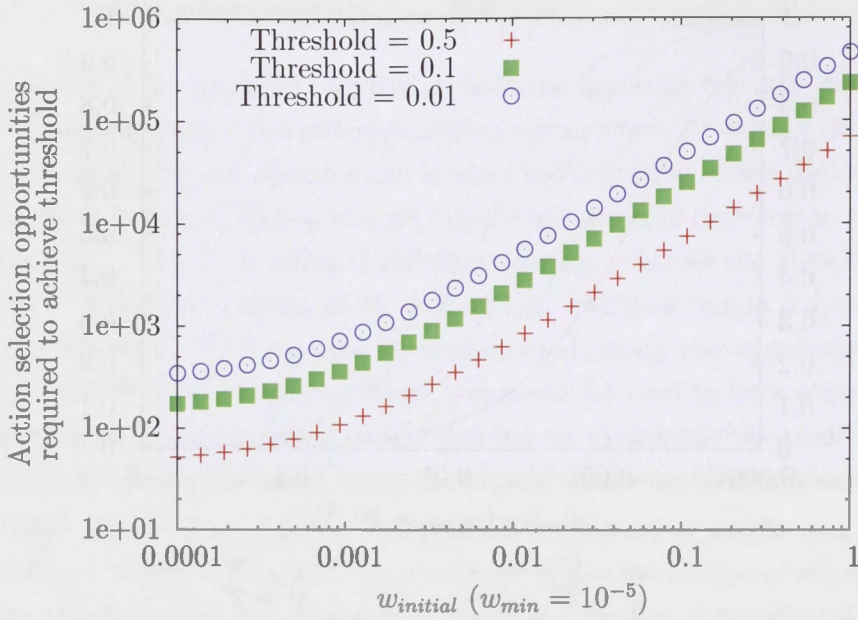


Figure 3.7: The speed with which the learning agent can react to changing circumstances ($\psi = 65$).

ing environment (by changing the number of actions) or reducing the parameter $w_{initial}$. Optimising the agent for a specific learning task precludes the first approach. When applying the second approach, the user must achieve a balance between reducing k and the increased cost of exploration - as shown in Figure 3.6, lower values of $w_{initial}$ lead to increased levels of exploration, which hurt performance when explorations are not required. The more exploration hurts, however, the less important are quick reactions: exploration hurts when there are long periods when the optimal action remains constant, and in such environments, there is more room to amortise the relatively long period of suboptimal behaviour resulting from slow reflexes.

The discussion to this point has been kept simple to illustrate the points involved, but the situation can become more complex in certain environments. One factor to consider is that, in a converged state, there may be more than one action whose selection will trigger re-exploration. As an extreme example, if an environmental change causes an optimal action to become the worst possible selection, then any exploration involving this action will drop its weight and trigger a re-exploration. This is illustrated by Figure 3.8, which shows how the parameter k is influenced by a new variable n . The variable n denotes the number

of actions whose exploration in the converged state will trigger a weight reset. The chance of such triggering becomes $\frac{nw_{min}}{w_{initial} + (\psi - 1)w_{min}}$. Taking the previous example point of $w_{initial} = 0.01$, Figure 3.8 shows that increasing n from 1 to 2 drops k from 850 to 430 - roughly half as many selection attempts are required to trigger exploration. Of these, around 43 will be explorations, 42 of them useless. At the extreme, $n = (\psi - 1)$ reduces the number of required action selection opportunities over $n = 1$ from 850 to 11, of which 1 results in an exploration - as expected, since $n = (\psi - 1)$ means any exploration triggers a weight reset.

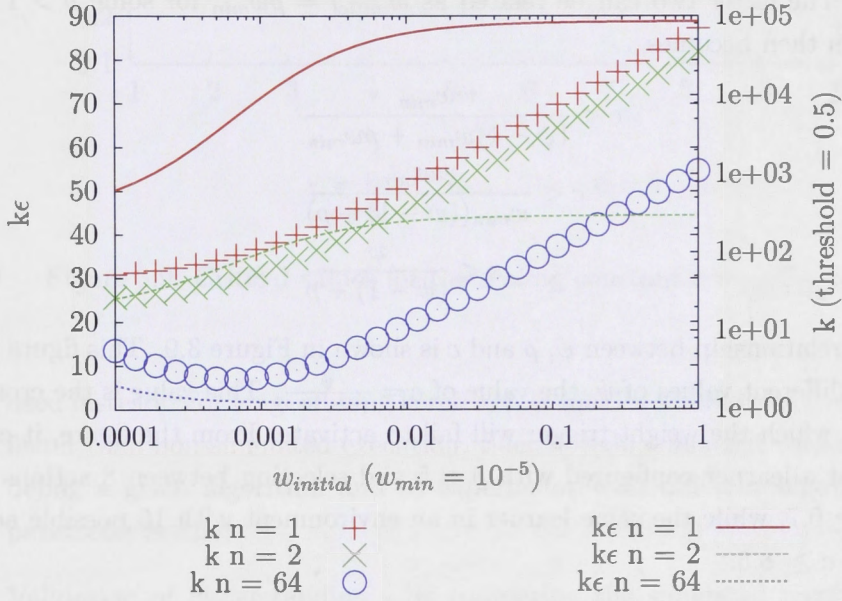


Figure 3.8: The effect of the number of reset-triggering actions on the agent's reaction time.

3.2.7.2 The reset threshold $\min_{\mu_{weight}}$

The parameter $\min_{\mu_{weight}}$ is used to trigger the resetting of weights. It is calculated from the values of $w_{initial}$, w_{min} , the number of actions ψ and a scaling constant $c < 1$. This section discusses constraints on the lower bound of possible values of c .

If c is too small the agent can incorrectly decide against resetting weights. A reset should be triggered when there is no action remaining with weight $w_{initial}$ - in other words, the mean of the weights is $\mu_{weight} = w_{min}$. A mean with this

value will fail to trigger a reset when $w_{min} > ck$, where $k = \frac{(\psi-1)w_{min}+w_{initial}}{\psi}$. Expanding this inequality gives:

$$w_{min} > ck \quad (3.5)$$

$$w_{min} > c \frac{(\psi-1)w_{min} + w_{initial}}{\psi} \quad (3.6)$$

$$\frac{\psi w_{min}}{(\psi-1)w_{min} + w_{initial}} > c \quad (3.7)$$

This condition depends on the number of actions and the values of w_{min} and $w_{initial}$. The latter two can be related as $w_{initial} = \rho w_{min}$ for some $\rho > 1$. The condition then becomes

$$c < \frac{\psi w_{min}}{(\psi-1)w_{min} + \rho w_{min}} \quad (3.8)$$

$$c < \frac{\psi w_{min}}{w_{min}((\psi-1) + \rho)} \quad (3.9)$$

$$c < \frac{\psi}{(\psi-1) + \rho} \quad (3.10)$$

The relationship between ψ , ρ and c is shown in Figure 3.9. This figure plots, for two different values of ψ , the value of $c = \frac{\psi}{(\psi-1)+\rho}$. This value is the crossover point at which the weight trigger will fail to activate. From the figure, it can be seen that a learner configured with $\rho = 5$ and selecting between 8 actions must have $c \geq 0.7$, while the same learner in an environment with 16 possible actions requires $c \geq 8.5$.

3.3 Evaluation Methodology

To evaluate the runtime tuning algorithms, two simulators were developed. Using simulators for evaluation brings the following advantages:

- Exact reproducibility - because the simulator is deterministic, a given input can be run multiple times and it will produce the same results[¶]. This is useful when trying to debug the algorithm or its implementation.
- Rapid execution - the design of the simulator is such that the computational kernel being adapted is never evaluated - pre-computed execution times are

[¶]Unless the evaluated algorithm has non-deterministic properties. The algorithm evaluated in this chapter can be made deterministic by fixing the seeds used to initialise its random number generation logic.

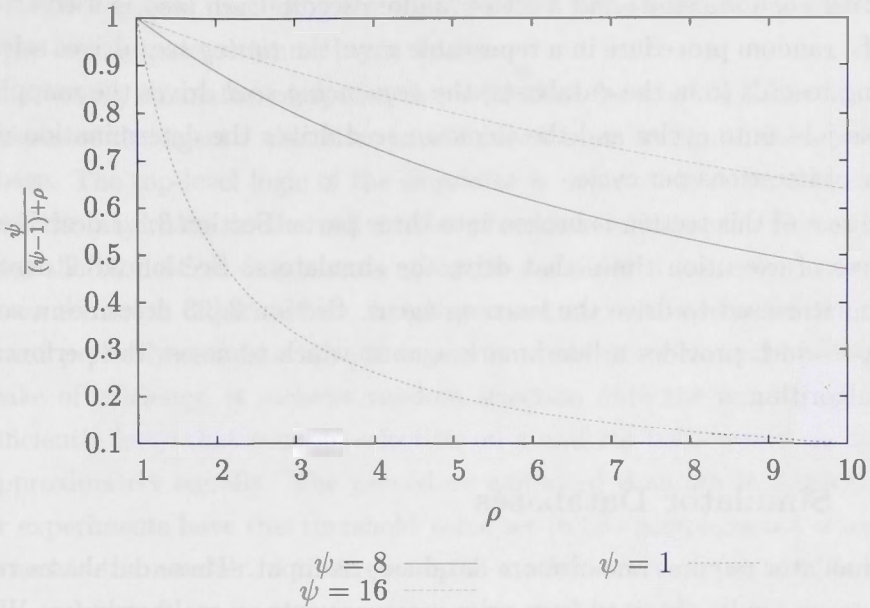


Figure 3.9: Critical values for the scaling constant $c = \frac{\psi}{(\psi-1)+\rho}$.

used instead. For long-running kernels, this enables simulations to complete faster than non-simulated execution. Such speedups make it easier both to debug a given algorithm and to experiment with different algorithms or parameter settings.

- Validation of understanding - by comparing the simulated results to results obtained on actual hardware, it is possible to check the implementer's understanding of the algorithm's behaviour.

However, using a simulator also comes with a large caveat: on its own, simulation is not enough to demonstrate the benefits of adapting programs. In order to provide the benefits listed above, the simulation presents a simplified model of program behaviour to the learning agent. A simulation's designer attempts to capture all pertinent behaviours in this model, but success in this endeavour must be determined experimentally.

The simulators in this thesis all use certain common parameters. These parameters are: a list of databases and their associated formats, the number of operations to simulate for each database (*cycles*), the mean and standard deviation of a Gaussian distribution which determines the number of iterations carried out for each cycle (μ_{iter} and σ_{iter} respectively), the number of actions available to

the learner (*action_count*) and a set of random seeds. Each seed is used to drive a specific random procedure in a repeatable way: the *timing seed* drives selection of timing records from the databases, the *sequencing seed* drives the mapping of available jobs onto cycles and the *iteration seed* drives the determination of the number of iterations per cycle.

The rest of this section is broken into three parts: Section 3.3.1 describes the databases of execution times that drive the simulators. Section 3.3.2 describes the simulator used to drive the learning agent. Section 3.3.3 describes a second simulator which provides a benchmark against which to assess the performance of the algorithm.

3.3.1 Simulator Databases

Each simulator requires one or more databases as input. These databases record performance results obtained from prior measurements on real hardware. Within each simulator, a database is accessed through an API that provides operations for selecting a random matrix from the databases and generating running times for various operations. Different databases can implement the API in different ways. The experiments in this thesis use a *model-free database*, which contains a list of execution times and pseudo-randomly selects from this list when an execution time is required. A different style of database would use a model of the execution time - for example, a Gaussian distribution parametrised by experimental results. The benefit of a model-based database is that a small number of parameters (such as a mean and variance) can produce a large set of simulated running times. The drawback is that a model database introduces assumptions into the simulator. If the underlying hardware doesn't meet these assumptions then what is being simulated is not reality. This drawback motivated the choice of a model-free database for this thesis.

3.3.2 Tuning simulator

In the field of computer science, the term *simulator* often refers to a program that emulates the behaviour of some hardware platform and which is used to study said platform. This simulator is not that kind of simulator - it is a simulated environment, which appears real to the tuning agent. The simulated environment enables controlled changes in the agent's perceptions, unbuffered by the perturbations of a real system. The simulated environment is also easier to implement

than the full system, including the tuning driver - this enables experimentation with the learning algorithm to begin sooner.

The tuning simulator computes the total time incurred in following the instructions of the agent - this time is simulated time, as stored in a model-free database. The top-level logic of the simulator is depicted in Algorithm 3.3. The core of the algorithm is the call to the procedure `simulate`. This procedure utilities the model-free database to compute a simulated execution time for the given combination of job, action, iteration count and conversion requirement. It does not perfectly simulate the process as it would occur on real hardware: for the sake of efficiency, it eschews random selection once the number of samples is sufficiently large that random selection on a uniform basis would choose them all approximately equally. The procedure employed is shown in Algorithm 3.4. Later experiments have this threshold value set to $50 \times \text{sample_count}$, where *sample_count* is the number of timing samples available for a given combination of matrix and format.

3.3.3 Static simulator

This section describes a simulator that evaluates static tuning. Under static tuning, a single action is picked at the beginning of the tuned application's lifetime and used for the duration of that lifetime. This provides a benchmark against which to measure dynamic tuning. It has certain inherent advantages: there is no need for a tuning driver and the associated runtime overhead, nor is there a need to convert data between storage formats. For dynamic tuning to beat the static tuning benchmark, it must provide performance gains that outweigh these additional costs. The logic used by the static simulator is shown in Algorithm 3.5.

3.4 Results and Discussion

This section will begin by demonstrating that dynamic tuning outperforms static tuning given the right circumstances. It will then investigate the constraints required to achieve these circumstances. All experiments focus on the problem of sparse matrix-vector format selection, which is described in Section 3.4.1. The experiments were carried out using timing databases acquired on various hardware platforms - these platforms are described in Section 3.4.2. The experiments aimed to measure the benefits of dynamic tuning under varying system conditions. The

Algorithm 3.3: Algorithm for the tuning simulator.

```

1 tot  $\leftarrow$  0.0
2 for  $d \in \text{databases}$  do
3   for  $j \in [1, \text{cycles}]$  do
4     matrix  $\leftarrow$  select_random_matrix( $d$ )
5     iter  $\leftarrow$  gen_gaussian_random( $\mu_{\text{iter}}, \sigma_{\text{iter}}$ )
6     old_fmt  $\leftarrow$  -1
7     while iter > 0 do
8       (new_fmt, count)  $\leftarrow$  query_agent( matrix )
9       conversion_required  $\leftarrow$  False
10      if new_fmt  $\neq$  old_fmt then
11        conversion_required  $\leftarrow$  True
12        old_fmt  $\leftarrow$  new_fmt
13      end
14      t  $\leftarrow$  simulate( matrix, conversion_required, new_fmt, count )
15      tot  $\leftarrow$  tot + t
16      iter  $\leftarrow$  iter - count
17    end
18  end
19 end
20 print tot

```

Algorithm 3.4: The **simulate** operation, invoked by the tuning simulator.

Input: m

```

1  matrix, conversion_required, format, iteration_count
2  tot  $\leftarrow$  0.0
3  if conversion_required = True then
4      tot  $\leftarrow$  tot + select_random_conversion_time( matrix, format )
5  end
6  if iteration_count > threshold then
7      bin_size  $\leftarrow$   $\lfloor \frac{\text{iteration\_count}}{\text{sample\_count}} \rfloor$ 
8      surplus  $\leftarrow$  iteration_count - (bin_size  $\times$  sample_count)
9      for sample  $\in$  iteration_timing_samples do
10         tot  $\leftarrow$  tot + sample  $\times$  bin_size
11     end
12     for  $i \in [0, \text{surplus})$  do
13         tot  $\leftarrow$  tot + the  $i^{\text{th}}$  iteration sample
14     end
15 end
16 else
17     for  $i \in [0, \text{iteration\_count})$  do
18         tot  $\leftarrow$  tot + select_random_iteration_time( matrix, format )
19     end
20 end
21 return tot

```

Algorithm 3.5: Algorithm for the static simulator.

```

1  for  $i \in [1, nact]$  do
2    tot[i]  $\leftarrow$  0.0
3  end
4  for  $d \in databases$  do
5    for  $j \in [1, cycles]$  do
6      matrix  $\leftarrow$  select_random_matrix(d)
7      iter  $\leftarrow$  gen_gaussian_random(  $\mu_{iter}, \sigma_{iter}$  )
8      for  $f$  in  $[1, nact]$  do
9        for  $i$  in  $[1, iter]$  do
10         t  $\leftarrow$  select_random_runtime( matrix, f )
11         tot[f]  $\leftarrow$  tot[f] + t
12       end
13     end
14   end
15 end
16 i  $\leftarrow$  argmini(tot[i])
17 print tot[i]
```

system conditions were varied through the introduction of contention for hardware resources. There are two different scenarios: under *baseline* conditions, the tuned application runs alone, with only background services active. Under *contention* conditions, the SpMV kernel competes with a *memory hog* - a small program which sits in an infinite loop, reading continuously through a region of memory at least twice the size of the architecture's last level of cache. The memory hog and the tuned application are bound to the same core.

3.4.1 Test problem

The experiments in this section focus on the problem of sparse matrix-vector format selection. In this problem, there exist a number of different matrix storage formats. For a given matrix, selecting the format that yields the shortest processing time requires accounting for the matrix size and structure, the hardware platform in

use and prevailing system conditions. Further detail on sparse matrices and their storage formats is available in Chapter 2.4. For this chapter, the problem is made concrete as follows: there are 65 possible storage formats, comprised of the Compressed Sparse Row format (CSR) and 64 variants of Blocked Compressed Sparse Row (BCSR) format - these variants consist of all blocking factors from 1×1 through to 8×8 inclusive. The matrices used come from the University of Florida Sparse Matrix Collection [99] (UFSMC) - this is a publicly available set of sparse matrices from a wide range of application domains, including engineering, chemistry and atmospheric modelling.

Each matrix is characterised by five attributes:

1. The number of rows in the matrix
2. The number of columns in the matrix
3. The density of the matrix: $\frac{\text{Number of non-zero matrix elements}}{\text{row count} \times \text{column count}}$
4. The standard deviation in the number of non-zero elements in each matrix row - this accounts for some kinds of irregularity in the matrix's internal structure.
5. The mean count of neighbouring elements - the mean neighbour count of a matrix element at row r and column c is computed as $\frac{1}{9} \sum_{i=r-1}^{r+1} \sum_{j=c-1}^{c+1} B(i, j)$, where the function $B(i, j)$ has the value 0 if the element at row i and column j is also zero and 1 in all other cases. This attribute indicates the

presence of groups of non-zero elements. This is useful information because grouped elements offer the opportunity to reduce indexing costs by utilising a blocked storage format.

3.4.2 Experimental platforms

The experiments in this chapter make use of data gathered on multiple platforms. For brevity, each platform has been given a tag by which it can be referenced. Table 3.1 gives an overview of the different platforms, and more details are provided in Appendix C.

Tag	Overview
<i>xeon-old</i>	8 Xeon cores clocked at 2 GHz. Caches: 32KB L1, 6MB L2
<i>xeon-new</i>	8 Xeon cores clocked at 2.4 GHz. Caches: 32KB L1, 256KB L2, 12MB L3
<i>phenom</i>	4 Phenom cores clocked at 3GHz. Caches: 64 KB L1 512 KB L2 6MB L3
<i>core2</i>	2 Core2 Duo cores clocked at 2.26 GHz Caches: 32 KB L1, 3 MB L2

Table 3.1: Overview of experimental platforms. Full details are in Appendix C.

3.4.3 Results

The experimental results start by demonstrating that the learning algorithm achieves a performance gain, given favourable circumstances (Section 3.4.3.1). Further experiments vary these circumstances to show the limitations of the learning algorithm. Section 3.4.3.2 shows the effect of reducing the mean iteration count μ_{iter} . Section 3.4.3.3 investigates changes in the set of matrices used for evaluation. In Section 3.4.3.4, generalisation parameters are changed, enforcing perceptual aliasing. It is demonstrated that, under certain conditions, this presents no barrier to the performance of Swift, as it is merely another set of environmental dynamics.

3.4.3.1 A beacon of utility

Figure 3.10(a) shows the ratio $\frac{\text{Kernel time with dynamic tuning}}{\text{Kernel time with static tuning}}$ for a simple kernel which performs sparse matrix-vector multiplications. The graph was generated using the parameters in Table 3.2. It can be seen that the initial exploration phase was over rapidly - this is highlighted by Figure 3.10(b) which shows only the

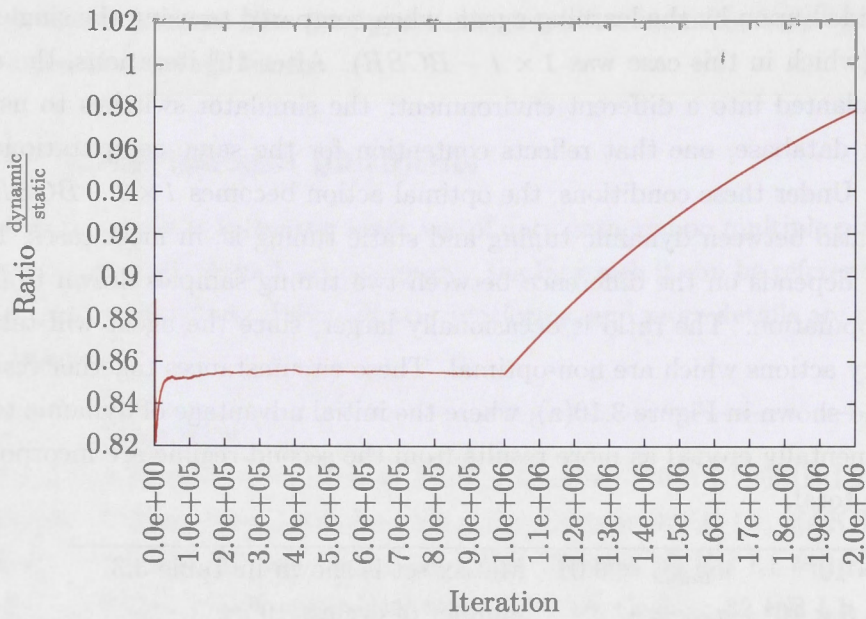
early iterations. There is a steady performance gain of 14% from using the format decided upon by the learning agent, when compared to using the single best format (which in this case was $1 \times 1 - BCSR$). After 10^6 iterations, the agent is transplanted into a different environment: the simulator switches to using a different database, one that reflects contention for the same computational resources. Under these conditions, the optimal action becomes $1 \times 1 - BCSR$ and so the ratio between dynamic tuning and static tuning is, in most cases, $1 + \eta$, where η depends on the difference between two timing samples drawn from the same population. The ratio is occasionally larger, since the agent will take exploratory actions which are non-optimal. These circumstances together result in the trend shown in Figure 3.10(a), where the initial advantage of dynamic tuning is incrementally eroded as more results from the second regime are incorporated into the total.

$w_{min} = 10^{-5}$	$w_{initial} = 0.01$	Matrix set is shown in Table 3.3	
$\mu_{iter} = 5 \times 10^4$	$\sigma_{iter} = 1$	number of cycles: 10^6	
Ordering: baseline, cache			
Available formats: CSR, 1×1 -BCSR through 8×8 -BCSR inclusive			
Random seeds			
<i>timing</i>	<i>sequencing</i>	<i>iteration</i>	<i>agent</i>
12,343	7,121	9,812	9,812
Attribute grid parameters			
<i>Attribute</i>	<i>Minimum</i>	<i>Maximum</i>	<i>Granularity</i>
Row count	1	500	4.99
Column count	1	500	4.99
Density	10^{-3}	1	9.9×10^{-3}
Non-zero deviation	0	7	7×10^{-2}
Mean neighbour count	1	9	0.8

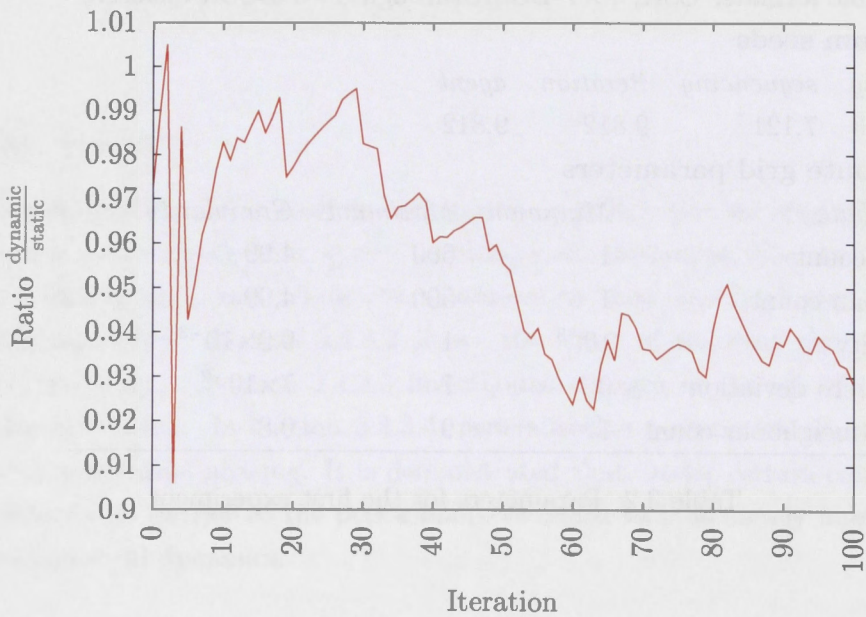
Table 3.2: Parameters for the first experiment

3.4.3.2 Varying iteration counts

This experiment investigated the effect of reducing μ_{iter} . Smaller values of μ_{iter} are desirable because they render the system applicable to smaller problems - that is, to loops with lower iteration counts. The challenge is that lower values of μ_{iter} reduce the scope for amortising format conversion costs. As will be explained



(a) Full experimental run



(b) Initial tuning phase

Figure 3.10: Demonstration of circumstances such that dynamic tuning outperforms static tuning.

Name	Rows	Cols	Density	npr	mnc	Best format	
						Baseline	Contention
GD06_theory	101	101	4×10^{-2}	5	2.12	7×8 -BCSR	1×1 -BCSR
GD98_c	112	112	3×10^{-2}	0	1.20	7×8 -BCSR	1×1 -BCSR
ash219	219	85	2×10^{-2}	0.0	3.23	1×7 -BCSR	1×1 -BCSR
bcsstm06	420	420	2×10^{-3}	0	3	1×4 -BCSR	CSR
bcsstm20	485	485	2×10^{-3}	0	3	8×6 -BCSR	CSR
flower_4_1	121	129	2×10^{-2}	1.53	1.48	8×7 -BCSR	1×1 -BCSR
gre_115	115	115	3×10^{-2}	0.98	2.64	1×8 -BCSR	1×1 -BCSR
wheel_7_1	114	113	3×10^{-2}	1.64	2.05	8×8 -BCSR	1×1 -BCSR

Table 3.3: Characteristics of the matrices used in the first experiment. The column headed *npr* shows the standard deviation in the number of non-zeros per row. The column labelled *mnc* reports the mean neighbour count. The column labelled *Cols* records the number of columns in each matrix.

shortly, the matrices used to generate these results and those in Section 3.4.3.1 were carefully selected to provide the best scope for dynamic format selection. A consequence of this selection is that the dynamic performance tuning beats static performance tuning under baseline conditions, but is slightly worse when contention is introduced. Thus, the most favourable relative performance arrives at the end of the baseline conditions - that is, after 10^6 matrices have been processed. The ratios at this point for different values of μ_{iter} are shown in Table 3.4. It is evident that the value of μ_{iter} has little effect as long as it is no less than 1000. At some point between $\mu_{iter} = 100$ and $\mu_{iter} = 1,000$, the effect of reducing μ_{iter} becomes significantly detrimental. A note of caution is appropriate here, though: the minimum useful value of μ_{iter} will likely increase for larger matrix sets.

An additional experiment ^{||} sheds more light on the mechanics behind these results. The dynamic simulator was modified to record *conversion-free iteration costs* - that is, to measure the costs incurred when conversion between matrix formats is instantaneous. Subsequently the k^{th} input matrix in a simulation run generates two costs: i_k and c_k . The former represents the simulated execution

^{||}The experiment used the parameters from Table ??, except that μ_{iter} ranged from 100 to 2000 in steps of 100 and a variety of different random seeds were used - see Appendix E for details.

μ_{iter}	<u>Dynamically tuned simulated time</u> <u>Statically tuned simulated time</u>
100	1.14
1000	0.88
5000	0.90
10000	0.86
50000	0.85

Table 3.4: Relative time incurred under different optimisation schemes at the point where the simulation environment evolves.

time when conversion costs are excluded. The latter represents simulated execution time including conversion costs. The ratio $r_k = \frac{i_k}{c_k}$ represents the relative impact of conversion costs on simulated execution time. As discussed above, the most favourable point of comparison is approximately ** at the point where the environment changes from baseline conditions to contended conditions. Let *best* be the value of k at this point for a given simulation.

For all simulations with the same μ_{iter} the set of r_{best} values forms a statistical distribution of the impact of format conversion costs. These values are plotted in Figure 3.11 for a range of different values of μ_{iter} . The plot shows an overall positive trend - larger μ_{iter} have i_{best} drawing closer to c_{best} . Experiments in this chapter have assumed that data format conversion costs are inevitable and must thus be accounted for when assessing the impact of the learning algorithm. This implies that the learning algorithm cannot provide improved performance until conversion costs no longer dominate the total running time.

3.4.3.3 The necessity of careful matrix selection

The results in Figure 3.10 were obtained using a small set of carefully selected matrices. The matrices came from a subset of the UFSMC. For each matrix m in the subset and for each format f , the simulated execution time of 10^6 SpMV operations was computed under both baseline and contention conditions. Each matrix had an optimal format, which minimised this execution time. The number of matrices for which each format was optimal is shown in Table 3.5. The matrices used to generate Figure 3.10 are those whose optimal format under baseline conditions is something other than CSR. To further illuminate the limitations of

**This point is approximate because the final matrix evaluated under baseline conditions may incur exploration costs.

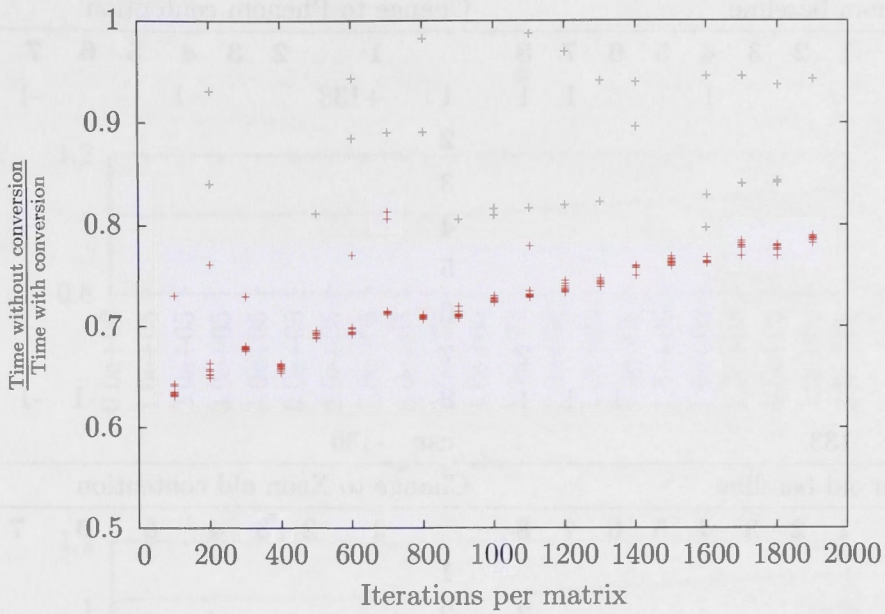


Figure 3.11: Amortisation of conversion costs.

the dynamic tuning system, the first step is to select additional matrices. Three sets of 18 matrices were generated, each set consisting of the original eight matrices and ten others randomly selected from the rest of the original UFSMC subset. The simulation was executed again, using $\mu_{iter} = 10,000$ and all other parameters as in Table 3.2. The results are shown in Figure 3.12 and they show that the dynamic tuning approach did not pay off in this situation.

Phenom baseline										Change to Phenom contention									
	1	2	3	4	5	6	7	8		1	2	3	4	5	6	7	8		
1				1			1	1	1	+138			-1			-1	-1		
2									2										
3									3										
4									4										
5									5										
6									6										
7								2	7								-2		
8						1	1	1	8						-1	-1	-1		
csr	133									csr	-130								
Xeon old baseline										Change to Xeon old contention									
	1	2	3	4	5	6	7	8		1	2	3	4	5	6	7	8		
1									1										
2								3	2										
3									3										
4	2			4					4	+1			-1						
5					2				5					-1					
6	2			1	6	5	2	1	6	-2				+1	+1				
7						2	1		7							-1			
8				2	1	2	3	4	8				+2	-1	-1	-1			
csr	98									csr	+3								
Xeon new baseline										Change to Xeon new contention									
	1	2	3	4	5	6	7	8		1	2	3	4	5	6	7	8		
1									1										
2		2	5	2	1				2				-1						
3									3										
4				2					4										
5					1				5					+1					
6	1	2	1	2	2	1		1	6	+1	-1	-1							
7									7										
8	7		2				1	1	8										
csr	107									csr	+1								

Table 3.5: Number of matrices for which different formats were optimal under different conditions. The grids on the left denote format optimality counts under baseline conditions. The grids on the right show how these counts must be changed to arrive at the counts when contention is present.

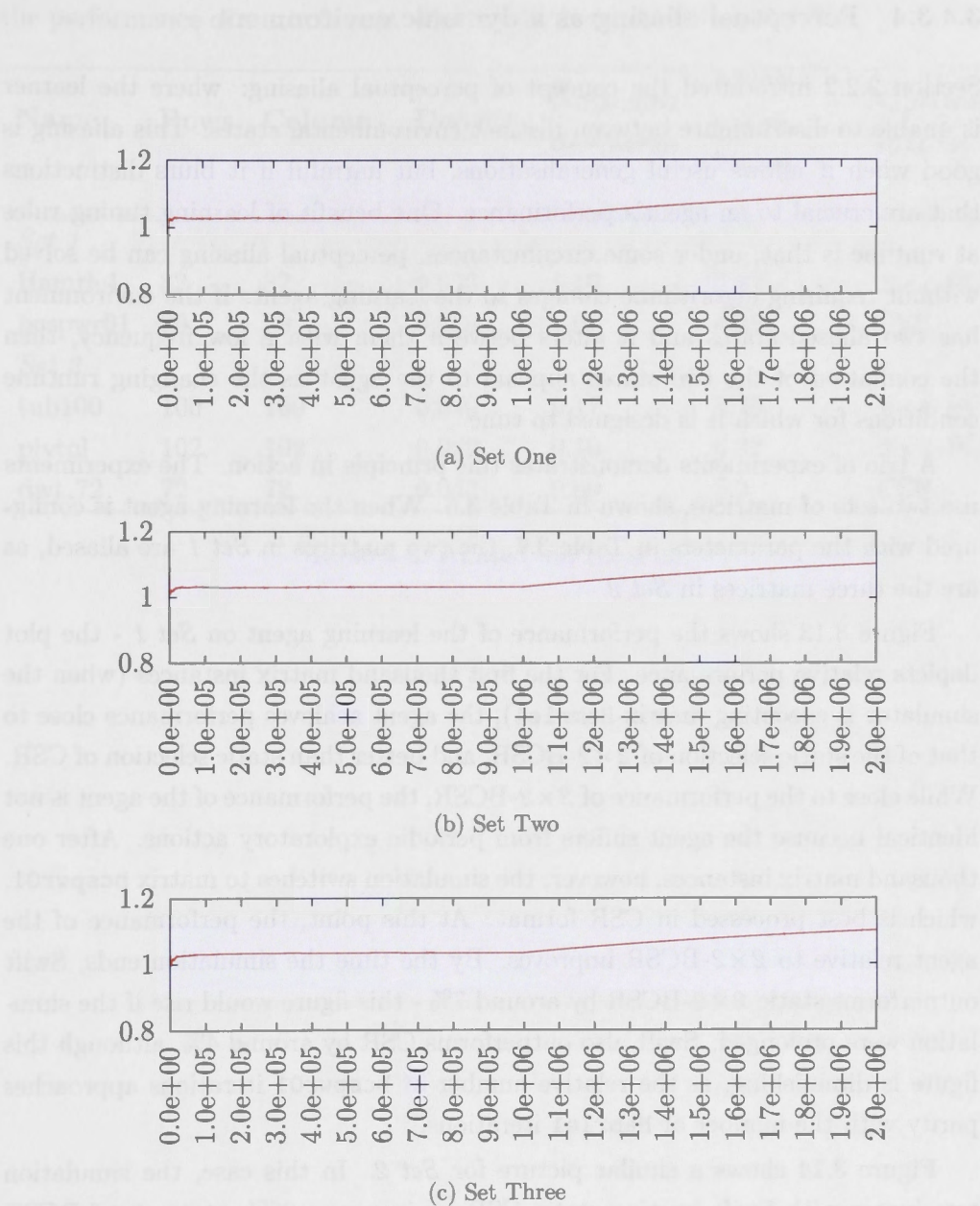


Figure 3.12: Plots of $\frac{\text{Dynamically tuned simulated time}}{\text{Statically tuned simulated time}}$ for different randomly augmented sets on the Phenom machine. The vertical axis shows the ratio of times, the horizontal axis records the number of matrices processed.

3.4.3.4 Perceptual aliasing as a dynamic environment

Section 2.2.2 introduced the concept of perceptual aliasing: where the learner is unable to discriminate between distinct environmental states. This aliasing is good when it allows useful generalisations, but harmful if it blurs distinctions that are crucial to an agent's performance. One benefit of learning tuning rules at runtime is that, under some circumstances, perceptual aliasing can be solved without requiring algorithmic changes to the learning agent. If the environment has two aliased states and it alters between them with a low frequency, then the conflation of the two states appears to the agent as the changing runtime conditions for which it is designed to tune.

A trio of experiments demonstrates this principle in action. The experiments use two sets of matrices, shown in Table 3.6. When the learning agent is configured with the parameters in Table 3.7, the two matrices in *Set 1* are aliased, as are the three matrices in *Set 2*.

Figure 3.13 shows the performance of the learning agent on *Set 1* - the plot depicts relative performance. For the first thousand matrix instances (when the simulator is executing matrix **Hamr1e1**), the agent achieves performance close to that of the static selection of 2×2 -BCSR, and better than static selection of CSR. While close to the performance of 2×2 -BCSR, the performance of the agent is not identical because the agent suffers from periodic exploratory actions. After one thousand matrix instances, however, the simulation switches to matrix **bcspwr01**, which is best processed in CSR format. At this point, the performance of the agent relative to 2×2 -BCSR improves. By the time the simulation ends, Swift outperforms static 2×2 -BCSR by around 7% - this figure would rise if the simulation were prolonged. Swift also outperforms CSR by around 4%, although this figure is diminishing, as the relative number of **bcspwr01** iterations approaches parity with the number of **Hamr1e1** iterations.

Figure 3.14 shows a similar picture for *Set 2*. In this case, the simulation terminates with Swift beating static CSR by just over 20%, static 2×6 -BCSR by just over 15%, and static 2×4 -BCSR by just under 15%. These figures will depend on the length of the simulation - as the simulation length is specified in terms of the number of matrices, this second experiment runs longer than the first, which contributes to the greater gains made by Swift under this scenario.

Figure 3.15 provides a counter example: it reports on a simulation that runs for longer than that shown by 3.14. The data set used for Figure 3.15 is the union of *Set 1* and *Set 2*. When the simulation halts, Swift has improved upon

the performance of static format selection by between 10% and 20%.

Name	Rows	Columns	Density	Non-zero deviation	Mean neighbours	Optimal format
<i>Set 1</i>						
Hamrle1	32	32	0.096	1.16	3.87	2×2 -BCSR
bcpwr01	39	39	0.086	1.00	4.09	CSR
<i>Set 2</i>						
tub100	100	100	0.040	0.17	5.76	2×6 -BCSR
pivtol	102	102	0.029	0.10	6.33	2×4 -BCSR
dwt_72	72	72	0.043	0.60	5.2	CSR

Table 3.6: Aliased matrix sets.

$w_{min} = 10^{-5}$	$w_{initial} = 0.01$	Matrix set is shown in Table 3.6	
$\mu_{iter} = 5 \times 10^5$	$\sigma_{iter} = 1$	number of jobs: 10^3	
Ordering on <i>Set 1</i> : Hamrle1, bcspr01			
Ordering on <i>Set 2</i> : tub100, pivtol, dwt.72			
Ordering on <i>Set 3</i> : Hamrle1, bcspr01, tub100, pivtol, dwt.72			
Available formats: CSR, 2×2 -BCSR, 2×4 -BCSR, 2×6 -BCSR, 6×1 -BCSR, 7×1 -BCSR			
Random seeds			
<i>timing</i>	<i>sequencing</i>	<i>iteration</i>	<i>agent</i>
12,343	7,121	9,812	9,812
Attribute grid parameters			
<i>Attribute</i>	<i>Minimum</i>	<i>Maximum</i>	<i>Granularity</i>
Row count	11	485	47.4
Column count	11	485	47.4
Density	0.002	0.628	0.0626
Non-zero deviation	0	6.125	0.6125
Mean neighbour count	1.204	7.623	1.284

Table 3.7: Agent configuration for simulated perceptual aliasing.

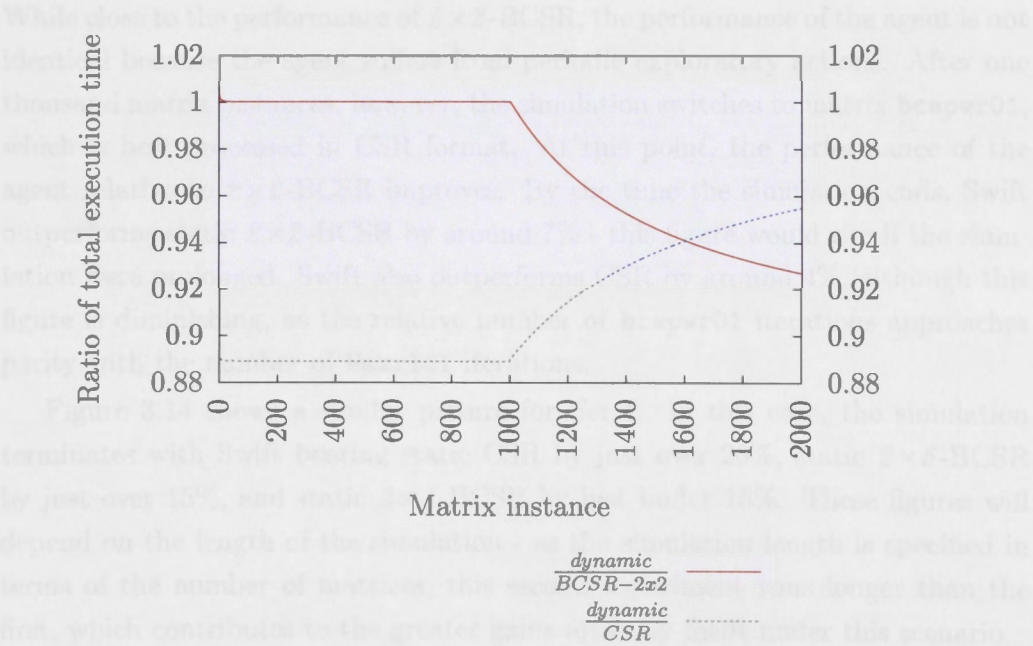
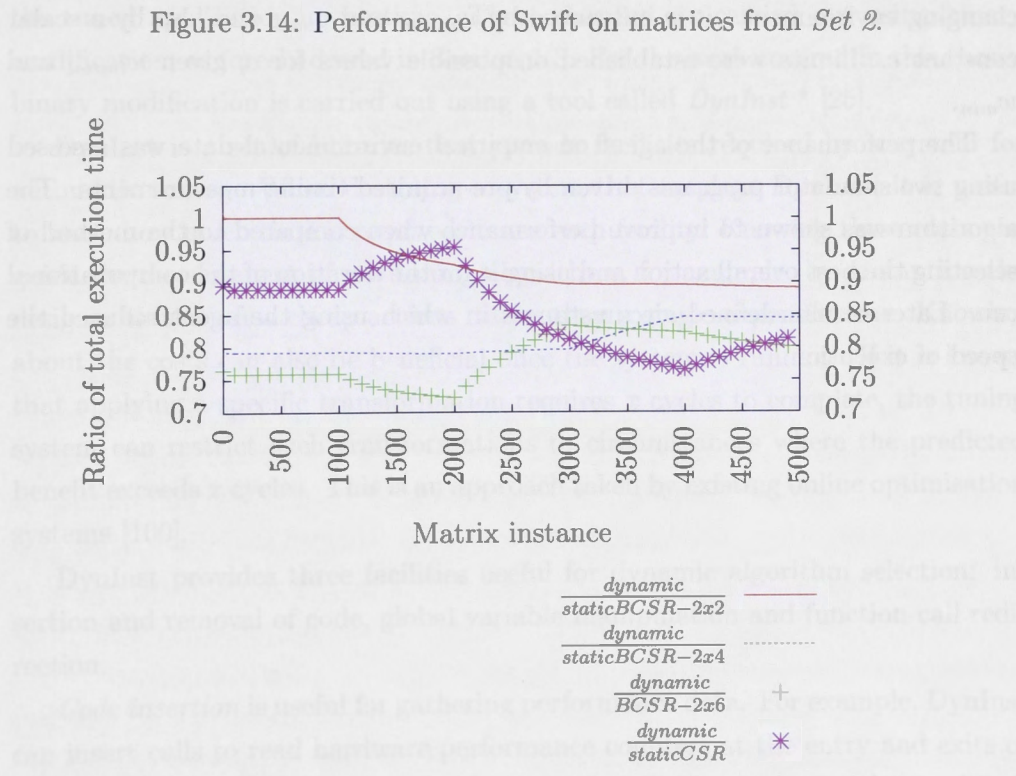
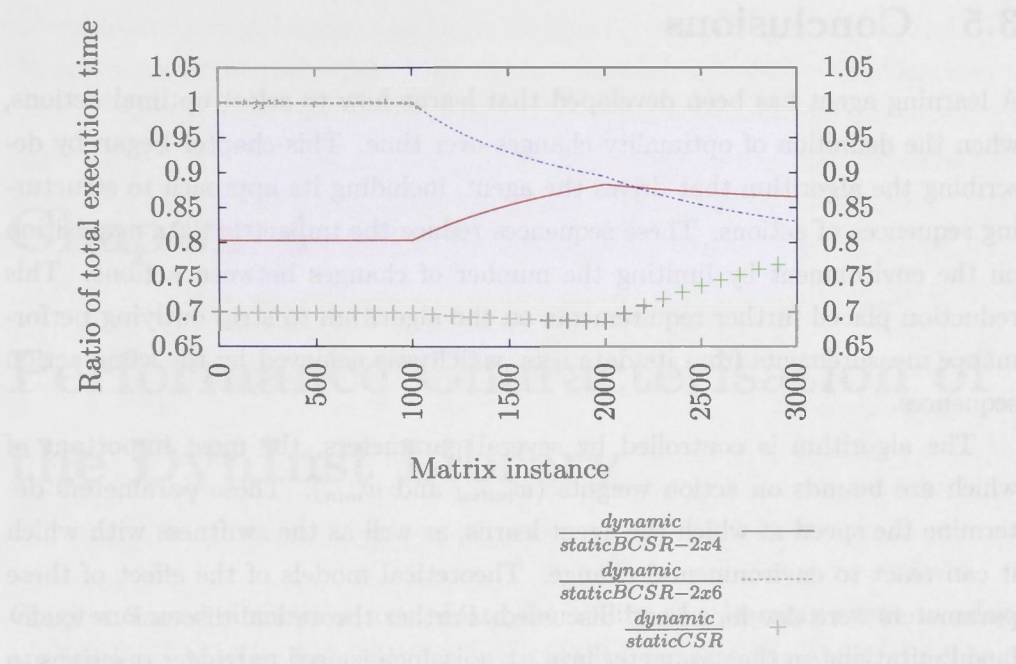


Figure 3.13: Performance of Swift on matrices from *Set 1*.



3.5 Conclusions

A learning agent has been developed that learns how to select optimal actions, when the definition of optimality changes over time. This chapter began by describing the algorithm that drives the agent, including its approach to structuring sequences of actions. These sequences reduce the impact of data acquisition on the environment by limiting the number of changes between actions. This reduction placed further requirements on the algorithm to strip outlying performance measurements from its data sets, which was achieved by the same action sequences.

The algorithm is controlled by several parameters, the most important of which are bounds on action weights ($w_{initial}$ and w_{min}). These parameters determine the speed at which the agent learns, as well as the swiftness with which it can react to environmental change. Theoretical models of the effect of these parameters were developed and discussed. Further theoretical discussions examined limitations on the parameter $min_{\mu weight}$, which is used to trigger reactions to changing environment - it is influenced by $w_{initial}$ and w_{min} and also by a scalar constant c . Limits were established on possible values for c , given $w_{initial}$ and w_{min} .

The performance of the agent on empirical environmental data was assessed using two simulator programs driven by pre-acquired timing measurements. The algorithm was shown to improve performance when compared to the method of selecting the best overall action and using it for the duration of the computational run. Later results defined circumstances in which using the agent reduced the speed of execution.

Chapter 4

Performance Characterisation of the DynInst Library

Chapter 3 used simulation to evaluate the ability of a learning agent to adapt application. Moving from simulation to reality requires the ability to change the tuned application at runtime. These changes are carried out using binary modification, as foreshadowed in Section 1.3. For the work covered in this thesis, binary modification is carried out using a tool called *DynInst* * [25].

DynInst is a set of libraries that provide facilities required by a system for automatic tuning. When designing a system that employs DynInst, it is useful to know how much time is required to use each facility. Knowing these costs can guide the development of the tuning algorithms, for example, by prompting the designer to use a few cheap actions rather than a single expensive one. Knowing about the costs can also be beneficial once the system is running. If it is known that applying a specific transformation requires x cycles to complete, the tuning system can restrict such transformations to circumstances where the predicted benefit exceeds x cycles. This is an approach taken by existing online optimisation systems [100].

DynInst provides three facilities useful for dynamic algorithm selection: insertion and removal of code, global variable manipulation and function call redirection.

Code insertion is useful for gathering performance data. For example, DynInst can insert calls to read hardware performance counters at the entry and exits of a specific routine, without the application source code needing to be modified. Once sufficient data have been gathered, DynInst can remove the performance

*<http://www.dyninst.org>

monitoring code and its associated overhead. Sections 4.2.1 and 4.2.2 develop microbenchmarks to measure the process, and the measurements are reported in Sections 4.3.1 and 4.3.2.

Using performance counters and similar tools is one way of characterising application performance. When measuring a program's use of memory, such tools commonly look at addresses. They provide a means of answering questions such as:

- How many addresses are accessed every second? This quantity is the *memory bandwidth*.
- What is the delay between knowing that the contents of a memory address is required and that contents being available for computation? This metric is known as the *memory latency*.
- What is the largest set of addresses a program accesses while performing its core computations? This measure is termed the *working set size*.
- What is the relationship between addresses accessed over time? This relationship determines the *miss rates* of the cache and Translation Lookaside Buffer (TLB).

Addresses are usually [†] irrelevant to the logical function of modern programs. *Values* matter, not addresses. Knowing the values on which programs are operating can be useful when trying to tune performance. An example arises when selecting a sorting algorithm based on the size of the data to be sorted [43] - that size will be stored in the target application's data structures. DynInst provides facilities that allow a tuning application to read and write the variables of a target process. Section 4.2.3 develops microbenchmarks to measure the performance of variable modification. The benchmark results are reported in Section 4.3.3.

Measurement alone is insufficient for a system like Tedium - the ability to modify the target application is crucial. Code insertion and variable modification go some way to allowing this, but an important extra facility is *function call redirection*. This facility allows a tuning program to find and redirect function invocations within the target application. The redirection transforms the function call instruction from `call address A` into `call address B`. Microbenchmarks

[†]Excluding those aspects of programs explicitly concerned with memory manipulation, such as garbage collectors.

to measure the cost of this process are developed in Section 4.2.4 and their results are presented in Section 4.3.4.

The rest of this chapter is structured as follows. Section 4.1 describes the experimental environment used to gather performance data. This environment informed the design of the microbenchmarks, which is described in Section 4.2. Results are presented and discussed in Section 4.3.

4.1 Experimental Methods

This section describes in detail the methods used to gather data about the benchmarks described in Section 4.2. It also demonstrates how that data is processed into the form presented in Section 4.3. Section 3.4.2 and Appendix C describe the architectures on which the benchmarks were run.

4.1.1 Gathering Data

The data reported in this thesis are derived from one of two sources: the system call `gettimeofday` and hardware performance counters. The former reports the current calendar time. The performance counters must be initialised before use, and are used here measure the number of cycles and the number of instructions that have elapsed since initialisation. Access to each source is hidden behind a common interface, which benchmarks may access by including the appropriate header file. The following two sections describe this interface (Section 4.1.1.1) and quantify its costs (Section 4.1.1.2).

4.1.1.1 Description of the measurement interface

The core of the measurement interface is the pair of methods which start and stop measurements. The first method reads its metric source (`gettimeofday` or hardware performance counters) and stores the result. Stopping the measurement requires taking a second reading and recording the difference between the two readings. The final reported measurement is the sum of the recorded differences. Care must be taken to ensure that neither the compiler nor the processor reorders the measurement-taking instructions. To prevent reordering by the compiler or the CPU, timing measurements were enclosed by barriers of the form `__asm__ __volatile__ ("mfence" : : "memory")`. The effectiveness

of this instruction was verified by inspecting the disassembly of the compiled executable.

In addition to the two core methods, there exist auxiliary methods to initialise, finalise, enable and disable measurement. The first two methods allocate and deallocate memory, initialise any required libraries, etc. The second two methods are required since not all executions of the code containing the start and stop method are of interest. For example, the DynInst code used for accessing variables is also invoked while DynInst is initialising itself, and these invocations are not of interest when trying to measure the costs of variable access by a benchmark program.

The performance counters are accessed using the PAPI (Performance API) library [101]. PAPI has some limitations which must be handled. The first is that it is not possible to use PAPI to obtain measurements on groups of processes: each measurement is obtained for a single process[‡]. As already mentioned, DynInst uses `ptrace` extensively, and this use is encapsulated within a dedicated thread. It is thus necessary to measure such uses in two phases. The first handles the user interface thread, which runs benchmark codes making use of DynInst's user interface. The second phase measures the execution of the `ptrace` thread. The two measurements are then summed.

This method is not perfect, since a small amount of thread-switching code is excluded from both measurements. Such exclusion is a consequence of the way the thread is structured, and hence how the measurement procedure calls must be placed. This placement is shown in Algorithm 4.1. The thread itself is structured as the server component of a client-server relationship. It executes an infinite loop within which it awaits requests for `ptrace` invocation, executes `ptrace` as instructed by its clients and returns the results before resuming its vigil. To measure a single `ptrace` request it is necessary to measure from just after the thread awakens until just before it blocks. This means the code to awaken and block the thread is exempt from measurement.

4.1.1.2 Costs of using the measurement interface

Use of the measurement interface incurs costs. These costs come in two classes: variable and constant, corresponding to two usage scenarios. The scenarios occur because the benchmarks in Section 4.2 are run with varying workloads. Variable costs are suffered when the number of measurements taken depends on the work-

[‡]It is possible to obtain a global measurement, but that is likely to lack precision.

Algorithm 4.1: Placement of performance counters to measure the `ptrace` thread.

```

1 while True do
2   end  $\leftarrow$  take_second_measurement();
3   block_and_await_ptrace_request();
4   start  $\leftarrow$  take_first_measurement();
5   handle_ptrace_request();
6 end

```

load. Constant costs are incurred when there is no such dependence. They are easy to correct for using the techniques developed in Section 4.1.2.

Variable costs are harder to account for. It is necessary to predict them in order to remove them. This prediction was done through the use of a linear model $f(x) = mx + b$, where x denotes the number of measurements taken and m and b are real-valued constants. To obtain values for m and b , empirical data were gathered for a variety of different values of x . These data came from the benchmark code shown in Algorithm 4.2. Least-squares fitting was used to fit $f(x)$ to the data. Table 4.1 reports the model parameters for each different metric. These parameters will be used in Section 4.3 to remove overhead from the benchmark results.

Algorithm 4.2: Algorithm for measuring variance of measurement costs with respect to the number of measurements x .

```

1 cost  $\leftarrow$  0;
2 for  $i \in [0, x]$  do
3   A  $\leftarrow$  take_first_measurement();
4   B  $\leftarrow$  take_second_measurement();
5   cost  $\leftarrow$  cost + (B - A);
6 end

```

4.1.2 Processing the Data

The result of running each benchmark is a set of readings from a single measurement source. These readings are reported in Section 4.3 as summaries. Each summary contains two pieces of information: the arithmetic mean and the standard error. This section describes how these quantities were obtained.

	Unoptimised	Optimised
Time (μsec)	$0.48 x + 0.76$ ($r^2 = 1.00$)	$0.46 x - 0.52$ ($r^2 = 1.00$)
Cycles	$884.84 x + 1076.79$ ($r^2 = 1.00$)	$906.54 x + 162.35$ ($r^2 = 1.00$)
Instructions	$296.02 x + 388.12$ ($r^2 = 1.00$)	$293.03 x + 299.70$ ($r^2 = 1.00$)

Table 4.1: Models which predict the total interface overhead. The figure r^2 is the coefficient of correlation - it shows the fraction of observed variance in timings explained by the model.

Each reported benchmark measurement is based on a population of N samples, $x_1 \dots x_N$. Each sample contains overhead from the measurement code. This overhead is removed using the model developed in Section 4.1.1.2.

Some benchmarks suffer a second source of overhead, which arises from the code (such as loops) necessary to execute the measured operation several times. Repeated execution is necessary when the benchmarked operation executes too fast for one of the metrics to measure. It also happens when trying to empirically determine how an operation scales with the number of invocations. Such overhead can be measured by disabling the measured code. For example, removing the call to `do_work` in Algorithm 4.3 will result in $B - A$ measuring the total overhead. Removing measurement overhead using the data from Section 4.1.1.2 leaves the cost of the benchmark mechanism - in this case, N iterations of the loop header.

Algorithm 4.3: Example code for measuring benchmark mechanism overhead.

```

1 A = take_first_measurement()
2 for i = 0; i < N; i = i + 1 do
3   do_work()
4 end
5 B = take_second_measurement()
```

4.1.3 Experimental Setup

The results in this chapter were obtained using DynInst 5.2 on two architectures. The architectures in question are an Intel Pentium 4 and an AMD Opteron. Details are provided in Appendix C

Two variants of DynInst were needed: the *direct* and *indirect* versions. The direct version was the standard version downloaded from the DynInst website. The indirect version was patched to insert profiling code around key loops in the style of Algorithm 4.1. The libraries are installed into variant-specific locations and selected by setting environment variables. Each variant was configured using the default configuration options, except for customising the installation directory and directing the configuration script to certain libraries and header files.

4.2 Benchmark Design

This section examines each benchmark in detail, explaining exactly what it consists of and what is being measured. The benchmarks are grouped according to the feature of DynInst they are designed to measure. These features are:

- The overhead of executing inserted code.
- The insertion and removal of code.
- The cost of accessing variables in the target process.
- Rewriting a function invocation instruction.
- The `ptrace` system call

4.2.1 Overhead of inserted code

The cost of executing inserted code can be decomposed into two components: the costs of using the base trampoline and the cost of using the mini trampoline. The first component accounts for unchanging overhead associated with any possible mini trampoline. The second component measures the cost of the inserted code. Although this will vary from program to program, it has been measured here in order to gauge the slowdown imposed by not inlining the mini trampolines.

The benchmark consists of the two applications *target_0* and *mod_0* shown in Algorithms 4.4 and 4.5. The application *target_0* is built around a loop which undertakes N iterations[§]. The loop is enclosed by two functions: `take_first_measurement()` and `take_second_measurement()`. These show where measurements are taken while abstracting away the details of the measurement mechanisms, which were explained in Section 4.1.1.

[§]Various values of N were used - see Section 4.3.1 for details

Algorithm 4.4: *Target_0*, used to measure the overhead of inserted code.

```

1  function trampoline_target
2      nop
3  end
4  function f
5      for  $i \in [1, 10]$  do
6           $x_4 \leftarrow 3.6 \times x_4 (1.0 - x_4)$ 
7      end
8  end
9  function loop_container
10     for  $i \in [1, N]$  do
11          $x_0 \leftarrow x_0 + 1$ 
12          $x_1 \leftarrow x_1 + 1$ 
13          $x_2 \leftarrow x_2 + 1$ 
14          $x_3 \leftarrow x_3 + 1$ 
15         f()
16     end
17 end
18 function main
19      $x_0 \leftarrow 0$ 
20      $x_1 \leftarrow 0$ 
21      $x_2 \leftarrow 0$ 
22      $x_3 \leftarrow 0$ 
23      $x_4 \leftarrow 0.0$ 
24     start  $\leftarrow$  take_first_measurement()
25     loop_container( N )
26     end  $\leftarrow$  take_second_measurement()
27 end

```

Algorithm 4.5: The application *mod_0*, which modifies the program *target_0*.

```

1 int main
    Input: target_path
2
    Input: style
3
    Input: location
4
    Input: safety
5
    Input: inlining
6
7 configure_base trampolines( safety, inlining )
8 Process ← create_paused_process( target_path )
9 Function ← find_function_in_process( Process, "trampoline.target" )
10 if style = "null" then
11     Instructions ← create_null_instruction()
12 else
13     Instructions ← create_function_call( Function )
14 end
15 if location = "start of function" then
16     Tmp ← lookup_function_in_process( Process, "f" )
17     Point ← find_first_instruction_of_function( Tmp )
18 else
19     Tmp ← lookup_function_in_process( Process, "main" )
20     Tmp_Loop ← get_first_loop_in_function( Tmp )
21     Point ← find_header_of_loop( Tmp_Loop )
22 end
23 insert_into_process( Process, Instructions, Point )
24 resume_process( Process )
25 wait_for_process_to_finish( Process )
26 end

```

The code for *target_0* also contains a routine `trampoline_target` which appears to be unused. This routine is provided for the benefit of the *mod_0* application, which uses DynInst to modify *target_0*. The modification involves placing a call to the function `trampoline_target` at some point within the application. The point at which the call is linked to the code of *target_0* and the nature of the linking mechanism are controlled by command-line arguments passed into *mod_0*. These arguments form a tuple: (*style*, *location*, *safety*, *inlining*).

The element *style* determines the nature of the code that is inserted into the target (excluding for the moment the trampolines used to invoke it). There are two possible values: a value of *call* directs *mod_0* to emplace code that invokes the function `trampoline_target`, while a value of *null* directs *mod_0* to insert a *null snippet*. A null snippet is a snippet without any code: the base trampolines are inserted but no mini trampolines are present. Using the *call* style is necessary in order to measure the cost of recursion safeguards - if the inserted code does not contain a function call, DynInst will eliminate the safeguards even if they have been explicitly requested. Using a function call as the inserted code requires deciding how to apportion changes in the observed metrics between the trampolines and the invocation and execution of the inserted function. Using the *null snippet* for trampoline styles where it is viable gives a baseline from which the cost of the *call* code can be deduced. This cost can then be removed from the overhead incurred by other trampoline styles to obtain an adjusted time T , as reported in Section 4.3.1. The deduction is carried out by Algorithm 4.6.

The tuple element *location* controls whether code is inserted into the termination test of the target's loop or as the first instruction in the function `f`. The nature of DynInst's modifications differs in these two cases. The element *safety* is used to specify how many safeguards are employed by the base trampoline. There are four possible values for this element: *none*, *recursion*, *floating point* and *recursion + floating point*. The final element *inlining* is a boolean flag which controls whether or not the mini trampolines are inlined.

The data are gathered by executing the command `mod_0 target_0, arg_1, arg_2, arg_3, arg_4` fifty times, where *arg_1* through *arg_4* correspond to the benchmark configurations from Table 4.2. In addition, fifty invocations of the application *target_0* provided baseline measurements of code untouched by DynInst.

4.2.2 Inserting and Removing code

This section describes the benchmarks used to measure the cost of inserting and removing code using Dynalot. There are three experiments, each with a target and a baseline. The first experiment is for inserting code, the second for removing code, and the third for inserting and removing code. The benchmarks are designed to measure the cost of inserting and removing code in a program that has been compiled to a target architecture. The benchmarks are designed to measure the cost of inserting and removing code in a program that has been compiled to a target architecture. The benchmarks are designed to measure the cost of inserting and removing code in a program that has been compiled to a target architecture.

Algorithm 4.6: Algorithm for deducting the cost of functions called from within snippets.

```
1 foreach metric do
2   foreach location do
3     foreach inlining do
4       foreach choice do
5         N ← the value of the metric style null with no safety
              measures
6         C ← the value of the metric for style call with no safety
              measures
7          $\delta \leftarrow C - N$ 
8         foreach type of safety measure do
9           B ← the cost of the metric for style call using the safety
                measure
10          T ← B -  $\delta$ 
11        end
12      end
13    end
14  end
15 end
```

4.2.3 Variable Access

The goal of this benchmark is to measure the cost of accessing a global variable belonging to a target system. The target system is represented by the program target, which is a program that has been compiled to a target architecture. The benchmark is designed to measure the cost of accessing a global variable target, which is a global variable that has been declared in the target program.

Benchmark Number	<i>style</i>	<i>location</i>	<i>safety</i>	<i>inlining</i>
1	null	loop	none	true
2				false
3		function		true
4				false
5	call	loop	none	true
6				false
7			recursion	true
8				false
9			floating point	true
10				false
11			floating point + recursion	true
12				false
13		function	none	true
14				false
15			recursion	true
16				false
17			floating point	true
18				false
19			floating point + recursion	true
20				false

Table 4.2: Benchmarks used to measure the overhead of executing inserted code. The first four benchmarks establish baselines used to measure the cost of the code inserted in the other benchmarks.

4.2.2 Inserting and Removing code

This section describes the benchmarks used to measure the cost of inserting and removing code using DynInst. There are three applications of interest: *target_1*, *mod_1a* and *mod_1b* (respectively Algorithms 4.7, 4.8 and 4.9). *Target_1* defines many identical functions, each of which is modified by *mod_1a* and *mod_1b*. This approach was taken in order that successive modifications begin from the same target state and thus carry out the same tasks. If modifications were applied incrementally to a single function, later modifications would need to parse, understand and modify the effects of earlier modifications. The costs incurred by the later modifications could not then be treated together with the costs of earlier modifications.

The modification applications perform either 19 or 20 modifications. Each modification is the insertion or the removal of code from *target_1*. The nature of said code, the location of the insertion and the effects of inlining and safety measure are the same as those described in Section 4.2.1. One subtlety in these programs is the nature of the functions `take_first_measurement()` and `take_second_measurement()`. Since the actions of *mod_1a* and *mod_1b* require the invocation of `ptrace`, these measurements need to be taken independently across the thread running the modification program and the thread running `ptrace` and then summed. The process was explained in Section 4.1.1.1.

Several benchmark configurations (i.e: sets of arguments to *mod_1*) were used. The combinations listed in Table 4.2 were used as a starting point. Each configuration listed therein was augmented in four ways: once with *action* = “insert” and *repeats* = 19, once with *action* = “insert” and *repeats* = 20, once with *action* = “remove” and *repeats* = 19 and once with *action* = “remove” and *repeats* = 20. This gives a total of 80 configurations. The raw cost for each action in each configuration was then computed as the difference between the times measured for *repeats* = 20 and *repeats* = 19. Overhead is calculated by following the above process with the insertion and removal calls commented out. Subtracting the overhead from the raw cost gives the final cost.

4.2.3 Variable Access

The goal of this benchmark is to measure the cost of reading or writing a global variable belonging to a target process. The target process is instantiated from the program *target_2* presented in Algorithm 4.10. The program contains a single global variable `target` which is aligned on a four-byte boundary. This alignment

Algorithm 4.7: *Target_1*, used to benchmark the cost of inserting or removing code.

```

1  function trampoline_target
2    nop
3  end
4  function f[0-19]
5    j ← 0.0
6    for i ∈ [1,10] do
7      j ← j + i × 12.2
8    end
9  end
10 function loop_container[0-19]
11   for i ∈ [1,N] do
12     x0 ← x0 + 1
13     x1 ← x1 + 1
14     x2 ← x2 + 1
15     x3 ← x3 + 1
16     f[0-19]()
17   end
18 end
19 function main
20   x0 ← 0
21   x1 ← 0
22   x2 ← 0
23   x3 ← 0
24   x4 ← 0.0
25   loop_container[0-19]( N )
26 end

```

Algorithm 4.8: *Mod_1a*

```

1 function main
   Input: target_path, style, location, safety, inlining
2
3   configure_base_trampoline( safety, inlining )
4   Process  $\leftarrow$  create_paused_process( target_path )
5   if location = "loop" then
6     for  $i \in [0,19]$  do
7       Handles[i]  $\leftarrow$  find_loop_header_in_function( "loop.container" + i
8         )
9     end
10    else
11      for  $i \in [0,19]$  do
12        Handles[i]  $\leftarrow$  find_function( "f" + i )
13      end
14    end
15    if style = "null" then
16      Payload  $\leftarrow$  create_null_instruction()
17    else
18      Payload  $\leftarrow$  create_function_call_to( "trampoline.target" )
19    end
20    start  $\leftarrow$  take_first_measurement()
21    if repeats = 20 then
22      insert_into_process(Process, Handles[19], Payload )
23    end
24    for  $i \in [0,18]$  do
25      insert_into_process(Process, Handles[i], Payload )
26    end
27    end = take_second_measurement()
28    terminate_process( Process )
29 end

```

Algorithm 4.9: *Mod_1b*

```

1 function main
   Input: target_path, style, location, safety, inlining
2
3   configure_base_trampolines( safety, inlining )
4   Process  $\leftarrow$  create_paused_process( target_path )
5   if location = "loop" then
6     for  $i \in [0,19]$  do
7       Handles[i]  $\leftarrow$  find_loop_header_in_function( "loop_container" + i
8         )
9     end
10  else
11    for  $i \in [0,19]$  do
12      Handles[i]  $\leftarrow$  find_function( "f" + i )
13    end
14  if style = "null" then
15    Payload  $\leftarrow$  create_null_instruction()
16  else
17    Payload  $\leftarrow$  create_function_call_to( "trampoline_target" )
18  end
19  if repeats = 20 then
20    insert_into_process(Process, Handles[19], Payload )
21  end
22  for  $i \in [0,18]$  do
23    insert_into_process(Process, Handles[i], Payload )
24  end
25  start  $\leftarrow$  take_first_measurement()
26  if repeats = 20 then
27    remove_from_process(Process, Handles[19], Payload )
28  end
29  for  $i \in [0,18]$  do
30    remove_from_process(Process, Handles[i], Payload )
31  end
32  end  $\leftarrow$  take_second_measurement()
33  terminate_process( Process )
34 end

```

is required because `ptrace` assumes 4-byte alignment of the addresses it accesses. If the variable is not so aligned, `DynInst` must read and manipulate extra data to obtain the unaligned portions of the variable. *Target_2* also contains the function `target_func` which the compiler is prohibited from inlining. This prohibition is required as the presence of a clear anchor point in the target program makes the modification program easier to write.

The variable `target` is read and written by a separate process running the program *mod_2*, shown as Algorithm 4.11. By changing the size of the variable's type, it is possible to measure the time required to access different volumes of data. The modification application *mod_2* takes three arguments: the path of the target binary to execute, the action to take (reading or writing), and the number of reads or writes to perform. For reported times, it is necessary to eliminate the overheads incurred by the measurement mechanism and to account for the overhead of the looping construct. This is done using the algorithms described in Section 4.1.2.

Algorithm 4.10: *Target_2*, used to measure the cost of variable access.

```

1 let target be aligned on 4
2 outline function target_func
   Input: N
3
4   sum  $\leftarrow$  0.0
5   for  $i \in [1, N]$  do
6     sum  $\leftarrow$  sum  $\times$   $3.6 \times (1.0 - \text{sum})$ 
7   end
8   return sum
9 end
10 function main
   Input: N
11
12   zero_memory(target)
13   target_func( N )
14 end

```

Algorithm 4.11: The application *mod.2* which uses DynInst to read or write global variables in a target process running *target.2*.

```

1  function main
   Input: target_path, action, iterations
2
3  zero_memory( local_copy )
4  Process ← create_paused_process( target_path )
5  Variable ← lookup_variable_in_process( Process, "target" )
6  if action = "read" then
7      start ← take_first_measurement()
8      for i ∈ [1, iterations] do
9          local_copy ← read_variable( Variable )
10         end
11         end ← take_second_measurement()
12     else
13         start ← take_first_measurement()
14         for i ∈ [1, iterations] do
15             write_variable( Variable, local_copy )
16         end
17         end ← take_second_measurement()
18     end
19     terminate_process( Process )
20 end

```

4.2.4 Switching Procedures

The benchmark for finding a procedure operates on a process which is executing the program *target_3*, as outlined in Algorithm 4.12. The three key routines in the program are denoted **caller**, **new** and **old**. Initially, **caller** is invoking **old** while **new** is unused. *target_3* is modified by the program *mod_3* which is depicted in Algorithm 4.13. This program is used to redirect the procedure call within **caller**, alternately directing it to **new** and **old**.

The measurement points shown for *mod_3* will include the overhead of the loop itself. To account for this, baseline measurements are taken by commenting out the two calls to **redirect_callsite** and rerunning the benchmark, while ensuring that the compiler does not remove the loop entirely. For this reason, this benchmark is compiled without optimisation. Such an approach is justified because the reported results will have the baseline measurements removed from them. The results will thus contain only the costs of the code in the DynInst library, which is compiled with the same optimisations as all the other benchmarks. In other words, the cost of non-optimised code will be factored out.

Algorithm 4.12: *Target_3*, which is used to measure the cost of function swapping.

```

1 int global ← 0
2 function old
3   global ← 42
4 end
5 function new
6   global ← 24
7 end
8 function caller
9   old()
10 end
11 function main
12   caller()
13 end

```

Algorithm 4.13: *Mod_3* which redirects callsites in a process running *target_3*.

```

1  function main
   Input: target_path, exchanges
2
3  Process ← create_paused_process( target_path )
4  Old ← lookup_function_in_process( Process, "old" )
5  New ← lookup_function_in_process( Process, "new" )
6  Caller ← lookup_function_in_process( Process, "caller" )
7  Site ← lookup_invocation_of( Old, Caller )
8  start ← take_first_measurement()
9  for  $i \in [1, exchanges]$  do
10     redirect_callsite( Site, New )
11     redirect_callsite( Site, Old )
12  end
13  end ← take_second_measurement()
14  terminate_process( Process )
15 end

```

4.2.5 Ptrace

Every time DynInst modifies a program, it does so using the system call `ptrace`. The performance of this system call will influence the performance of DynInst as a whole. Although several of the previous benchmarks have included components that make use of `ptrace`, it is not easy to separate out the cost of `ptrace` from the cost of the many levels of DynInst that lie between it and the top level of DynInst-using code. The performance of `ptrace` is thus measured by a separate benchmark, presented as Algorithm 4.14. The benchmark consists of two processes: a parent and a child. The child executes an infinite loop and the parent reads a number of words from the child's address space. The benchmark reports results as the elapsed difference (`end-start`). Executing the benchmark fifty times with a single set of arguments provides one sample set, from which the mean and standard error can be calculated. However, this number contains additional overhead, namely that of the timing mechanism as well as that incurred by the loops. To address this, the benchmark is run fifty times using each of two distinct values for `runs`, r_0 and r_1 . From these data are obtained means and standard errors $R_0 \pm E_0$ and $R_1 \pm E_1$. That portion of the overhead which exists outside the measured loop can then be removed by computing the intermediate quantity ψ :

$$(\psi \pm \psi_e) = (R_1 - R_0) \pm (E_1 + E_0)$$

In order to remove the overhead imposed by the mechanics of iteration, the benchmark is modified by commenting out the `ptrace` calls from the measured loop. Another two sets of 50 runs are undertaken with the same arguments as before. This will result in the quantities $L_0 \pm e_0$ and $L_1 \pm e_1$. The non-loop overhead is again removed by computing the intermediate quantity λ :

$$\lambda \pm \lambda_e = (L_1 - L_0) \pm (e_1 + e_0)$$

The final result F can then be computed by removing iteration overhead and averaging out the incurred costs equally over every iteration:

$$F = \frac{(\psi - \lambda) \pm (\psi_e + \lambda_e)}{r_1 - r_0}$$

4.3 Results and Discussion

This section presents and discusses the results obtained by running the benchmarks described in Section 4.2 on the hardware reported in Section 4.1. Each

Algorithm 4.14: Benchmark used to measure ptrace.

```

1 function workload
2   x ← 0.1
3   while True do
4     x ← x × 3.6 × ( 1.0 - x )
5   end
6 end
7 word_t words[101]
8 function main
   Input: runs, action
9
10  Child_ID ← create_process_executing( workload )
11  if action = "read" then
12    start ← take_first_measurement()
13    for i ∈ [1,runs] do
14      ptrace( Child_ID, Read, words + i )
15    end
16    end ← take_second_measurement()
17  else
18    start ← take_first_measurement()
19    for i ∈ [1,runs] do
20      ptrace( Child_ID, Write, words + i )
21    end
22    end ← take_second_measurement()
23  end
24  terminate_process( Child_ID )
25 end
```

result is reported as the arithmetic mean and standard error from a sample of 50 results, calculated as described in Section 4.1.

4.3.1 Overhead of inserted code

The first results considered are those relating to the overhead of executing inserted code. The data from the relevant benchmarks (those in Section 4.2.1) are contained in Table 4.3 (for the x86 platform).

Turning first to the x86 platform, the data in Table 4.3 show that the execution overhead of the trampoline mechanism would be negligible outside the tightest of loops. The most notable cost is adding floating point safety to code inserted into loop headers. This fact should guide placement strategies when inserting floating point using code. Such code would be best inserted at the start of a function.

The data do not show a consistent bias with respect to trampoline inlining. This is likely due to the nature of the benchmarks: inlining would only show up when multiple insertions were carried out at a single program location, and even then it would only incur a few instructions extra overhead. Aside from floating point safe trampolines, there is also no consistent benefit to placing trampolines in loop headers or in function preambles. When deciding on the design and placement of inserted code snippets, these factors can thus be ignored.

On the x86_64 platform, the data show the same broad patterns: the overhead of a trampoline is measured in tens of cycles, and would have a negligible overhead in all but the tightest of loops. The exception remains floating-point sensitive code placed within a loop header - the overhead of saving and restoring floating point state causes this cost to balloon, although the effect is less severe than on the x86 .

When comparing the two platforms, it can be seen that the x86_64 platform requires more instructions to be executed to invoke the trampoline. Despite this, and despite the slower clock rate, the x86_64 platform is competitive with the x86 one, and sometimes faster. This occurs because the code used on the x86_64 processor is able to execute multiple instructions per cycle, something that not true of the x86 code. Specifically, the x86 processor never executes more than 0.88 instructions per cycle, and sometimes many fewer. The x86_64 , on the other hand, can manage 1.90 instructions per cycle on some benchmarks.

Baseline				
Wall time (nanoseconds)	119± < 1			
CPU cycles	356 ± 3			
Instructions	139 ± 0			
Costs in excess of the baseline				
	Safety checks			
	None	Recursion	Floating point	Recursion and floating point
<i>inline trampolines located in loop header</i>				
Wall time (nanoseconds)	8 ± 2	10 ± 2	390 ± 32	395 ± 29
CPU cycles	25 ± 9	31 ± 9	1173 ± 97	1187 ± 83
Instructions	9 ± 0	24 ± 0	15 ± 0	30 ± 0
<i>inline trampolines located in called function</i>				
Wall time (nanoseconds)	8 ± 3	12 ± 6	31 ± 2	38 ± 5
CPU cycles	25 ± 3	33 ± 4	94 ± 3	126 ± 3
Instructions	10 ± 0	25 ± 0	12 ± 0	27 ± 0
<i>outline trampolines located in loop header</i>				
Wall time (nanoseconds)	10 ± 2	12 ± 2	371 ± 64	382 ± 67
CPU cycles	31 ± 8	73 ± 7	1195 ± 188	1126 ± 188
Instructions	12 ± 0	21 ± 0	16 ± 0	25 ± 0
<i>outline trampolines located in called function</i>				
Wall time (nanoseconds)	12 ± 6	13 ± 6	12 ± 9	13 ± 7
CPU cycles	37 ± 66	25 ± 48	30 ± 70	25 ± 44
Instructions	13 ± 0	22 ± 0	13 ± 0	22 ± 0

Table 4.3: Benchmark results gathered as discussed in Section 4.2.1, running on the x86 platform. These data record the time, cycles and instructions required to execute each iteration of the routine `loop_container` in Algorithm 4.4, with N set to 1×10^8 .

Baseline				
Wall time (nanoseconds)	95.6 ± 0.2			
CPU cycles	204 ± 0.02			
Instructions	138 ± 0			
Additional Overhead				
	Safety checks			
	None	Recursion	Floating point	Recursion and floating point
<i>inline trampolines located in loop header</i>				
Wall time (nanoseconds)	15.1 ± 0.8	14.6 ± 0.8	22.4 ± 0.8	18.3 ± 0.8
CPU cycles	27 ± 0.09	28 ± 0.09	41 ± 0.09	41 ± 0.09
Instructions	35 ± 0	46 ± 0	41 ± 0	52 ± 0
<i>inline trampolines located in called function</i>				
Wall time (nanoseconds)	9.2 ± 1.0	11.9 ± 1.0	21.5 ± 1.0	21.5 ± 1.0
CPU cycles	16 ± 0.1	21 ± 0.1	44 ± 0.1	43 ± 0.1
Instructions	24 ± 0	35 ± 0	36 ± 0	47 ± 0
<i>outline trampolines located in loop header</i>				
Wall time (nanoseconds)	10.5 ± 0.8	11.4 ± 0.8	77.3 ± 0.8	78.6 ± 0.8
CPU cycles	35 ± 0.09	36 ± 0.09	180 ± 0.10	183 ± 0.10
Instructions	37 ± 0	49 ± 0	45 ± 0	57 ± 0
<i>outline trampolines located in called function</i>				
Wall time (nanoseconds)	8.3 ± 1.0	8.3 ± 1.0	10.1 ± 1.0	8.3 ± 1.0
CPU cycles	16 ± 0.1	20 ± 0.1	16 ± 0.1	20 ± 0.1
Instructions	26 ± 0	38 ± 0	26 ± 0	38 ± 0

Table 4.4: Benchmark results gathered as discussed in Section 4.2.1, running on the x86.64 platform. These data record the time, cycles and instructions required to execute each iteration of the routine `loop_container` in Algorithm 4.4, with N set to 1×10^8 .

4.3.2 Cost of Inserting Code

This section reports the cost of inserting code into another process. These costs were measured using the benchmark *mod_1*, detailed in Section 4.2.2. The description which follows uses terms and concepts introduced in that section.

The data below aims to show how cost varies based on safety measures, for a variety of different configurations. To this end, eight different sample groups were formed as set of all permutations of the tuple (*link*, *location*, *action*) where *link* denotes the mini trampoline linkage method (inline or outline), *location* is the insertion location (loop header or function prelude) and *action* is either insertion or deletion. Each sample group was represented by 750 measurement runs. These runs each modified the sample group using one of ten methods. Each method was defined by the tuple (*style*, *safety*, *measure*). The first two elements are described in Section 4.2.1. Note that style of *null* only applies when *safety* is none. The style *call* applies to all safety settings. The element *measure* specifies which thread is being measured, the user interface thread or the *ptrace* thread.

Modification methods were assigned to measurement runs randomly, subject to the constraint that each treatment was measured 50 times. This was done to avoid introducing trends into the results. Each measurement was obtained by using the benchmarks *mod_1* from Algorithms 4.8 and 4.9.

The raw measurements from running *mod_1* were stripped of instrumentation overhead as described in Section 4.1.2. For each benchmark configuration (trampoline style, trampoline insertion location, safety measures and inlining decision), a per-repeat cost was obtained as half of the difference between four repeats and two repeats. The overhead of the inserted procedure call was determined by comparing the per-repeat cost of the null trampoline with that of the procedure call trampoline. If they differ by less than their cumulative standard error, the overhead is assumed to be zero. This overhead is then subtracted from the individual per-repeat costs, resulting in the numbers which are reported in Tables 4.5, 4.6, 4.7 and 4.8.

The act of inserting code into a process is the most expensive considered in this chapter. This should not come as a surprise given the complexity outlined in Section 4.2.2. For both platforms, it can be seen that the costs of all configurations overlap.

The two platforms display significant differences in the numbers of instructions required for trampoline insertion. Under all configuration options, the x86 platform requires fewer instructions. This does not show up when comparing the

cycles required - a fact due to the large width of the confidence intervals around the cycle measurements.

For ease of comparison, the data are plotted graphically in Figure 4.1. The comparison of instruction counts shows that the difference between safety measures is not significant. Placement within a loop header or a function call does make a difference, although this choice is usually constrained by what the instrumentation is supposed to measure. The choice between inline and outline trampolines is not so constrained, and this choice can make a detectable difference to the instruction count. These distinctions disappear, however, when the cycle counts are examined. The relatively high variability of the cycle counts and consequent width of the confidence intervals shows that, as far as running time goes, each configuration can be regarded as equally costly.

4.3.3 Variable Access

This section describes the cost of using DynInst to modify a variable in a remote process. Table 4.9 reports the results from the x86 platform, and Table 4.10 reports the numbers from x86_64. Each table reports data for two different sizes: 1 byte and 1024 bytes. For each data size, three different metrics measure reading and writing performance. The first metric is elapsed wall time, reported as a single number per size and operation. The second and third metrics are performance counter measurements of elapsed cycles and instruction counts. These are reported as three numbers for each size and operation. The first number measures the user interface thread. The second number measures the `ptrace` thread, and the third number is the sum across both threads. The reason for this split was covered in Section 4.1.1.1.

Looking at the total costs first, and excluding the costs of stopping and restarting the target process, such modifications require at least 7×10^4 cycles, with more cycles required for larger variable types. On a cross-platform comparison, the x86 results are better than those obtained on the x86_64. The x86 chip has a faster clock speed, but this is not the whole story. The x86_64 architecture requires between 1.0 and 1.5 cycles per instruction, while the x86 chip requires between 2.8 and 3.6 cycles per instruction. However, the fact that the x86_64 chip requires substantially more instructions than the x86 chip sees it require more time overall.

Figures 4.2 and 4.3 show the costs of variable access for different variable sizes. It is apparent that the user interface thread has a fixed cost, and that changing the

	Safety checks			Recursion & Floating Point
	None	Recursion	Floating point	

Cycle counts $\times 10^5$				
<i>inline trampolines located in loop header</i>				
UI	18.3 $\pm$ 13.1	19.5 $\pm$ 13.0	18.8 $\pm$ 13.5	19.7 $\pm$ 11.8
Ptrace	5.2 $\pm$ 3.1	5.4 $\pm$ 2.8	5.3 $\pm$ 2.7	5.4 $\pm$ 3.1
Total	23.5 $\pm$ 16.2	25.0 $\pm$ 15.8	24.0 $\pm$ 16.1	25.0 $\pm$ 14.9
<i>inline trampolines located in called function</i>				
UI	22.5 $\pm$ 11.8	23.4 $\pm$ 12.4	22.1 $\pm$ 13.9	24.6 $\pm$ 10.2
Ptrace	5.3 $\pm$ 2.8	5.3 $\pm$ 2.7	5.3 $\pm$ 2.6	5.5 $\pm$ 2.9
Total	27.8 $\pm$ 16.8	28.7 $\pm$ 15.1	27.4 $\pm$ 16.5	30.1 $\pm$ 13.1
<i>outline trampolines located in loop header</i>				
UI	21.0 $\pm$ 13.0	20.4 $\pm$ 14.9	21.6 $\pm$ 13.7	21.8 $\pm$ 11.9
Ptrace	7.40 $\pm$ 3.9	6.92 $\pm$ 3.5	7.13 $\pm$ 3.4	7.08 $\pm$ 3.9
Total	28.5 $\pm$ 16.8	27.3 $\pm$ 18.4	28.7 $\pm$ 17.0	28.9 $\pm$ 15.8
<i>outline trampolines located in called function</i>				
UI	24.8 $\pm$ 14.4	25.2 $\pm$ 14.2	24.3 $\pm$ 14.0	25.0 $\pm$ 11.1
Ptrace	7.2 $\pm$ 3.7	7.1 $\pm$ 3.8	7.0 $\pm$ 3.4	6.70 $\pm$ 3.1
Total	32.1 $\pm$ 18.1	32.4 $\pm$ 17.9	31.3 $\pm$ 17.4	32.0 $\pm$ 14.2

Table 4.5: Performance counter data on the cost of inserting code on the x86 platform. The performance counter data was gathered in two disjoint portions, as outlined in Section 4.1.1.1. This table reports this data for the user interface thread (*UI*), the thread which manages `ptrace` invocations (*Ptrace*), and their sum (*Total*). Each value is reported as $\mu \pm \epsilon$ where μ is the arithmetic mean of 50 trials, and ϵ is half of the width of the 95% confidence interval around μ .

	Safety checks			Recursion
	None	Recursion	Floating point	& Float- ing Point
Instruction counts $\times 10^5$				
<i>inline trampolines located in loop header</i>				
UI	6.4 ± 0.2	7.0 ± 0.2	6.5 ± 0.3	7.0 ± 0.2
Ptrace	1.7 ± 0.2	1.7 ± 0.02	1.7 ± 0.02	1.7 ± 0.02
Total	8.1 ± 0.4	8.7 ± 0.2	8.2 ± 0.3	8.7 ± 0.2
<i>inline trampolines located in called function</i>				
UI	8.3 ± 0.2	8.8 ± 0.2	8.3 ± 0.3	8.8 ± 0.2
Ptrace	1.7 ± 0.2	1.7 ± 0.01	1.7 ± 0.01	1.7 ± 0.01
Total	10 ± 0.4	10.5 ± 0.2	10.1 ± 0.3	10.5 ± 0.2
<i>outline trampolines located in loop header</i>				
UI	7.2 ± 0.3	7.2 ± 0.3	7.2 ± 0.2	7.2 ± 0.3
Ptrace	2.3 ± 0.09	2.3 ± 0.1	2.3 ± 0.03	2.3 ± 0.03
Total	9.5 ± 0.4	9.5 ± 0.4	9.5 ± 0.2	9.5 ± 0.3
<i>outline trampolines located in called function</i>				
UI	9.0 ± 0.4	9.0 ± 0.3	9.0 ± 0.3	9.0 ± 0.3
Ptrace	2.3 ± 0.03	2.3 ± 0.6	2.3 ± 0.04	2.3 ± 0.03
Total	11.3 ± 0.4	11.3 ± 0.9	11.3 ± 0.3	11.3 ± 0.3

Table 4.6: Performance counter data on the cost of inserting code on the x86 platform. The performance counter data was gathered in two disjoint portions, as outlined in Section 4.1.1.1. This table reports this data for the user interface thread (*UI*), the thread which manages `ptrace` invocations (*Ptrace*), and their sum (*Total*). Each value is reported as $\mu \pm \epsilon$ where μ is the arithmetic mean of 50 trials, and ϵ is half of the width of the 95% confidence interval around μ .

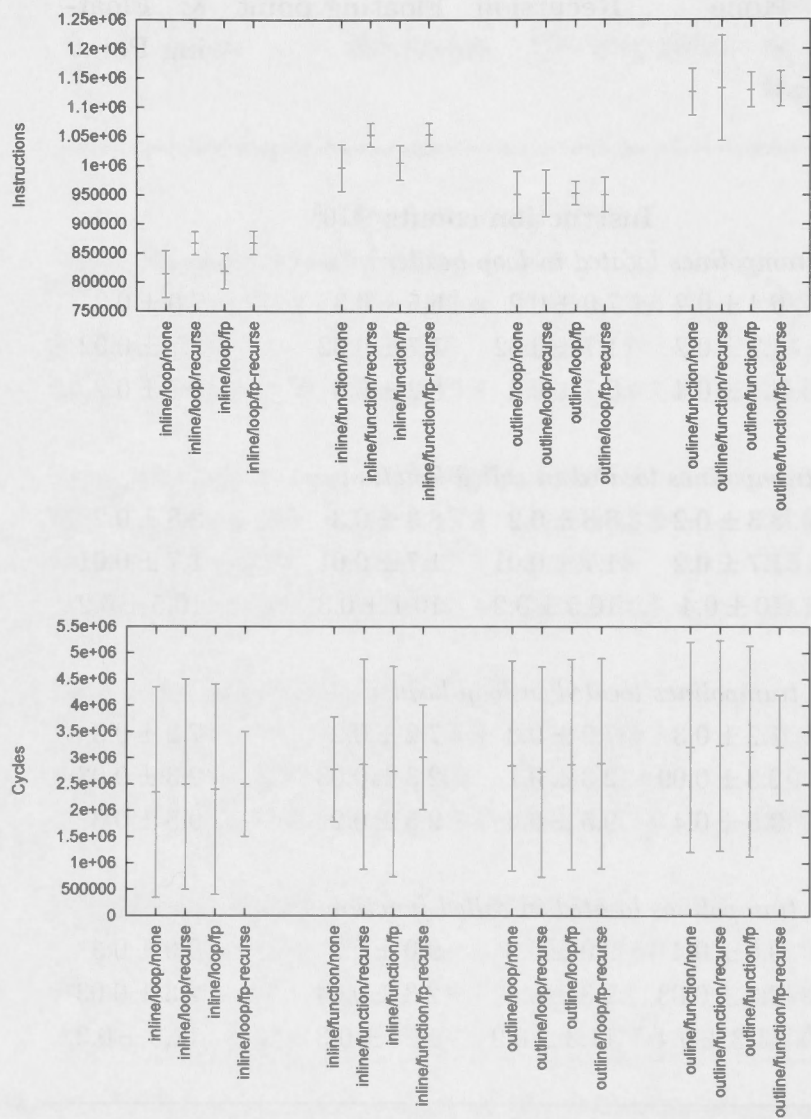


Figure 4.1: Instruction counts (top) and cycle counts (bottom) for DynInst-enabled code insertion on the x86 platform, under differing constraints.

	Safety checks			
	None	Recursion	Floating point	Recursion & Floating point
Cycle counts $\times 10^6$				
<i>inline trampolines located in loop header</i>				
UI	8.9 $\pm$ 25.9	9.1 $\pm$ 26.8	8.0 $\pm$ 26.6	8.9 $\pm$ 27.9
Ptrace	7.9 $\pm$ 9.7	8.6 $\pm$ 6.5	8.3 $\pm$ 6.1	8.0 $\pm$ 8.7
Total	16.9 $\pm$ 35.6	17.7 $\pm$ 33.3	16.3 $\pm$ 32.7	16.9 $\pm$ 36.7
<i>inline trampolines located in called function</i>				
UI	9.0 $\pm$ 28.1	7.6 $\pm$ 27.1	7.9 $\pm$ 26.7	9.1 $\pm$ 26.9
Ptrace	7.8 $\pm$ 5.8	7.8 $\pm$ 10.3	9.1 $\pm$ 8	73.1 $\pm$ 10.8
Total	16.8 $\pm$ 33.9	15.4 $\pm$ 37.4	17.0 $\pm$ 34.8	16.4 $\pm$ 37.7
<i>outline trampolines located in loop header</i>				
UI	9.0 $\pm$ 27.8	7.8 $\pm$ 27.4	8.1 $\pm$ 26.5	6.5 $\pm$ 27.0
Ptrace	8.4 $\pm$ 9.8	8.3 $\pm$ 5.8	9.2 $\pm$ 9.0	9.0 $\pm$ 9.4
Total	17.4 $\pm$ 37.6	16.1 $\pm$ 33.2	17.4 $\pm$ 35.4	15.5 $\pm$ 36.4
<i>outline trampolines located in called function</i>				
UI	7.5 $\pm$ 25.2	9.0 $\pm$ 26.5	8.9 $\pm$ 26.2	8.9 $\pm$ 28.0
Ptrace	8.5 $\pm$ 7.7	8.9 $\pm$ 8.4	8.1 $\pm$ 11.3	8.3 $\pm$ 10.7
Total	16.0 $\pm$ 33.0	17.9 $\pm$ 35.0	16.9 $\pm$ 37.5	17.3 $\pm$ 38.6

Table 4.7: Performance counter data on the cost of inserting code on the x86_64 platform. The performance counter data was gathered in two disjoint portions, as outlined in Section 4.1.1.1. This table reports this data for the user interface thread (*UI*), the thread which manages *ptrace* invocations (*Ptrace*), and their sum (*Total*). Each value is reported as $\mu \pm \epsilon$ where μ is the arithmetic mean of 50 trials, and ϵ is half of the width of the 95% confidence interval around μ .

	Safety checks			
	None	Recursion	Floating point	Recursion & Floating point
Instruction counts $\times 10^6$				
<i>inline trampolines located in loop header</i>				
UI	7.7 ± 1.4	7.8 ± 1.1	7.7 ± 1.6	7.9 ± 0.9
Ptrace	8.2 ± 0.2	8.2 ± 0.1	8.2 ± 0.1	8.2 ± 0.2
Total	15.9 ± 1.6	16.0 ± 1.2	15.9 ± 1.7	16.1 ± 1.0
<i>inline trampolines located in called function</i>				
UI	8.1 ± 0.4	8.1 ± 0.5	8.1 ± 1.0	8.2 ± 1.1
Ptrace	8.2 ± 0.1	8.2 ± 0.1	8.2 ± 0.5	8.2 ± 0.1
Total	16.3 ± 0.5	16.3 ± 0.7	16.3 ± 1.0	16.3 ± 1.3
<i>outline trampolines located in loop header</i>				
UI	8.0 ± 1.3	8.1 ± 1.5	8.1 ± 1.0	8.0 ± 1.0
Ptrace	8.3 ± 0.07	8.3 ± 0.04	8.3 ± 0.1	8.3 ± 0.09
Total	16.3 ± 1.4	16.4 ± 1.5	16.5 ± 1.1	16.3 ± 1.1
<i>outline trampolines located in called function</i>				
UI	8.2 ± 0.8	8.2 ± 1.6	8.3 ± 1.6	8.3 ± 1.1
Ptrace	8.3 ± 0.09	8.3 ± 0.04	8.3 ± 0.1	8.3 ± 0.08
Total	16.5 ± 0.9	16.5 ± 1.6	16.6 ± 1.7	16.6 ± 1.2

Table 4.8: Performance counter data on the cost of inserting code on the x86_64 platform. The performance counter data was gathered in two disjoint portions, as outlined in Section 4.1.1.1. This table reports this data for the user interface thread (*UI*), the thread which manages *ptrace* invocations (*Ptrace*), and their sum (*Total*). Each value is reported as $\mu \pm \epsilon$ where μ is the arithmetic mean of 50 trials, and ϵ is half of the width of the 95% confidence interval around μ .

size of the accessed variable only affects the performance of the ptrace thread. The performance of this thread seems to have a linear dependence on the variable size, which suggests that the number of instructions required to copy a data structure could be modelled by a linear function of the form $f(b) = u + s + ib$. In this model, u represents the constant cost of the user interface thread, s denotes the size-independent cost of the ptrace thread, i is the per-byte cost of the ptrace thread, and b denotes a variable size in bytes. The measured data yielded the models in Table 4.11.

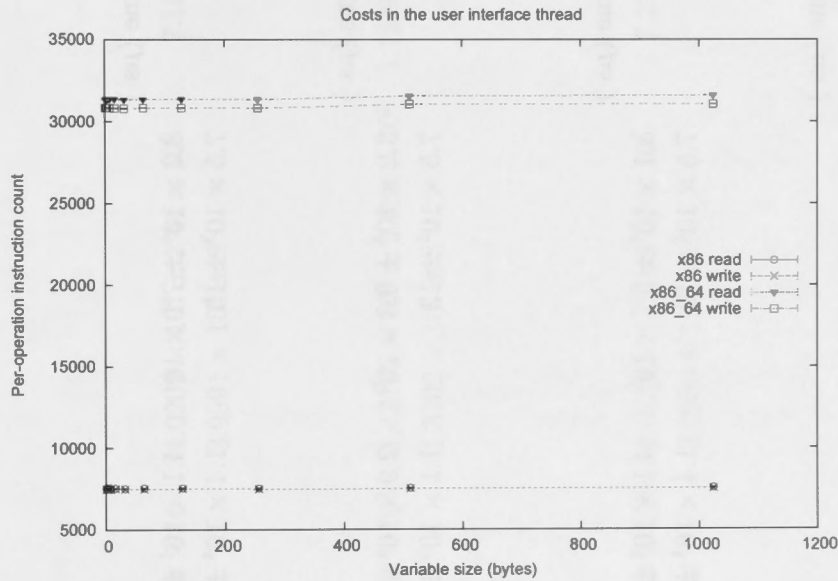


Figure 4.2: This graph shows the costs incurred by the user interface thread, for various sizes of variable.

4.3.4 Function Swapping

This section reports results from the benchmarks described in Section 4.2.4. These benchmarks are designed to measure the cost to redirect a procedure call site. The raw measurements from the benchmark were adjusted to remove the overhead of the measuring code. The results for the ptrace thread were further adjusted to remove the cost of looping, which was obtained by measuring the benchmark with the function swapping code removed, and adjusting for measurement-gathering overhead.

Table 4.12 shows these costs for the two platforms. The table reports per-thread costs for varying numbers of function swaps. The data show costs on

	UI	Ptrace	Total
Reading			
<i>1 byte</i>			
Elapsed wall time (μ s)			39 ± 3
CPU cycles	$3.2 \times 10^4 \pm 7.1 \times 10^2$	$4.1 \times 10^4 \pm 4.6 \times 10^2$	$7.3 \times 10^4 \pm 1.2 \times 10^3$
Instructions	$7.5 \times 10^3 \pm 13$	$1.4 \times 10^4 \pm 0$	$2.2 \times 10^4 \pm 13$
<i>1024 bytes</i>			
Elapsed wall time (μ s)			$5.66 \times 10^2 \pm 9$
CPU cycles	$3.2 \times 10^4 \pm 6.3 \times 10^2$	$2.9 \times 10^5 \pm 7.95 \times 10^2$	$3.2 \times 10^5 \pm 1.4 \times 10^3$
Instructions	$7.5 \times 10^3 \pm 12$	$1.1 \times 10^5 \pm 0$	$1.1 \times 10^5 \pm 12$
Writing			
<i>1 byte</i>			
Elapsed wall time (μ s)			45 ± 5
CPU cycles	$3.1 \times 10^4 \pm 5.7 \times 10^2$	$4.1 \times 10^4 \pm 2.8 \times 10^2$	$7.2 \times 10^4 \pm 8.5 \times 10^2$
Instructions	$7.5 \times 10^3 \pm 12$	$1.4 \times 10^4 \pm 0$	$2.2 \times 10^4 \pm 12$
<i>1024 bytes</i>			
Elapsed wall time (μ s)			$7.02 \times 10^2 \pm 8$
CPU cycles	$3.2 \times 10^4 \pm 7.79 \times 10^2$	$3.8 \times 10^5 \pm 2.3 \times 10^3$	$4.1 \times 10^5 \pm 3.1 \times 10^3$
Instructions	$7.5 \times 10^3 \pm 10$	$1.1 \times 10^5 \pm 1$	$1.1 \times 10^5 \pm 11$

Table 4.9: Performance data for reading and writing to global variables on the x86 platform. The *UI* column reports the costs of the user interface thread. The column labelled *Ptrace* reports costs incurred by the ptrace-handling thread. The sum of these costs is reported in the column labelled *Total*.

	UI	Ptrace	Total
Reading			
<i>1 byte</i>			
Elapsed wall time (μs)			40 ± 1
CPU cycles	$3.1601 \times 10^4 \pm 6.08 \times 10^2$	$4.0202 \times 10^4 \pm 2.41 \times 10^2$	$7.1802 \times 10^4 \pm 8.49 \times 10^2$
Instructions	$3.1366 \times 10^4 \pm 2$	$3.9991 \times 10^4 \pm 8$	$7.1356 \times 10^4 \pm 24$
<i>1024 bytes</i>			
Elapsed wall time (μs)			$8.47 \times 10^2 \pm 12$
CPU cycles	$3.3071 \times 10^4 \pm 6.34 \times 10^2$	$1.42105 \times 10^5 \pm 2.630 \times 10^3$	$1.75176 \times 10^5 \pm 3.264 \times 10^3$
Instructions	$3.1604 \times 10^4 \pm 16$	$1.3202 \times 10^5 \pm 3$	$1.63620 \times 10^5 \pm 20$
Writing			
<i>1 byte</i>			
Elapsed wall time (μs)			40 ± 1
CPU cycles	$3.1687 \times 10^4 \pm 5.95 \times 10^2$	$4.0638 \times 10^4 \pm 3.00 \times 10^2$	$7.2325 \times 10^4 \pm 8.95 \times 10^2$
Instructions	$3.0833 \times 10^4 \pm 12$	$3.9986 \times 10^4 \pm 16$	$7.0819 \times 10^4 \pm 28$
<i>1024 bytes</i>			
Elapsed wall time (μs)			$1.049 \times 10^3 \pm 21$
CPU cycles	$3.2713 \times 10^4 \pm 6.59 \times 10^2$	$2.63398 \times 10^5 \pm 2.397 \times 10^3$	$2.96111 \times 10^5 \pm 3.056 \times 10^3$
Instructions	$3.1073 \times 10^4 \pm 2$	$1.62715 \times 10^5 \pm 7$	$1.93787 \times 10^5 \pm 23$

Table 4.10: Performance data for reading and writing to global variables on the x86_64 platform. The *UI* column reports the costs of the user interface thread. The column labelled *Ptrace* report costs incurred by the ptrace-handling thread. The sum of these costs is reported in the column labelled *Total*.

	u	s	i
read & write on x86	7.5×10^3	1.4×10^4	90

Table 4.11: Parameters for a linear model to predict the number of instructions required to copy a data structure between processes as a function of the data structure’s size.

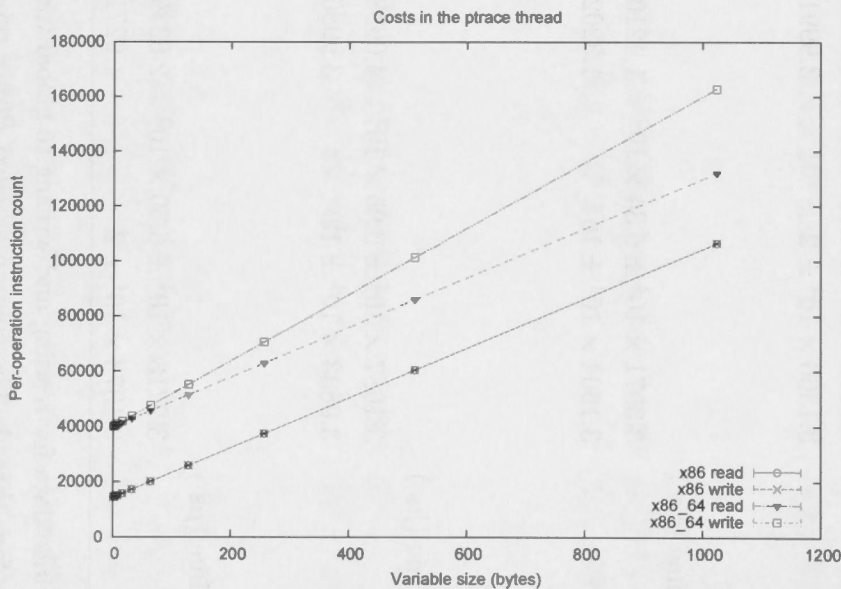


Figure 4.3: This graph shows the costs incurred by the ptrace thread, for various sizes of variable.

the order of 10^4 cycles. The metrics in the table decrease as the number of function swap operations rises. This is because the table reports the metrics on a per function swap basis. There appears to be a large, once-off cost when function swapping is first invoked. As increasing numbers of function swaps are performed, this cost becomes less influential. An additional point of note is that the process which underwent function swapping was paused at the time the benchmark was run - the pause costs are not included in the reported times.

4.3.5 Ptrace

The results in this section are computed using the method described in Section 4.2.5, using $r_0 = 8$ and $r_1 = 16$. Table 4.13 shows how the **ptrace** system call performs on each platform. It can be seen that, due to the variation in the measurements, neither platform is clearly optimal. In each case, however, the cost of working with data via **ptrace** is not prohibitive. In addition, there is a discrepancy between the number of cycles recorded and the total wall time measured - the latter is several times larger than the former would suggest. This is likely caused by the thread-specific nature of the performance counters, and any execution of the target thread (or indeed other unrelated threads) could contribute to the mismatch.

4.4 Conclusion

DynInst is a library that enables the runtime modification of remote processes. It can be used to build systems to carry out runtime tuning of long-running programs. For such tuning to be effective, it is necessary to know the costs of using DynInst. This chapter has presented a set of four microbenchmarks to measure these costs. The benchmarks measured the cost of inserting profiling code into a process, the overhead imposed by the profiling code, the cost of retargeting a function call and the cost of reading and writing variables from the target process's address space.

The cost imposed by the profiling code was found to be a few tens of instructions, while the cost of performing the insertion was on the order of 10^6 cycles. The exact instruction count varied based on the number and type of safeguards used. Variation across architectures was also observed. For runtime function selection, costs were of the order of 10^4 instructions, with more required on the x86_64 platform than on the x86 platform. The cost of remote data access varied

Exchanges	Instruction Count	CPU Cycles	Predicted time (μsec)	Measured time (μsec)
x86				
<i>UI thread</i>				
1	$3.0 \times 10^4 \pm 8.8 \times 10^1$	$1.51 \times 10^5 \pm 6.75 \times 10^3$	50 ± 2.3	78 ± 3
10	$1.4 \times 10^4 \pm 1.1 \times 10^2$	$6.4 \times 10^4 \pm 1.03 \times 10^3$	21 ± 0.3	49 ± 2
100	$1.3 \times 10^4 \pm 1.9 \times 10^2$	$5.5 \times 10^4 \pm 9.6 \times 10^2$	18 ± 0.3	44 ± 1
1000	$1.2 \times 10^4 \pm 9.4 \times 10^1$	$5.3 \times 10^4 \pm 9.5 \times 10^2$	18 ± 0.3	43 ± 1
<i>ptrace thread</i>				
1	$1.5 \times 10^4 \pm 5.6 \times 10^1$	$4.9 \times 10^4 \pm 1.19 \times 10^3$	16 ± 0.4	26 ± 2
10	$1.5 \times 10^4 \pm < 1$	$4.5 \times 10^4 \pm 5.0 \times 10^2$	15 ± 0.2	21 ± 0.4
100	$1.5 \times 10^4 \pm < 1$	$4.5 \times 10^4 \pm 1.17 \times 10^3$	15 ± 0.4	21 ± 0.4
1000	$1.5 \times 10^4 \pm 9$	$4.3 \times 10^4 \pm 2.02 \times 10^3$	14 ± 0.7	21 ± 1
x86_64				
<i>UI thread</i>				
1	$6.4 \times 10^4 \pm 7.4 \times 10^1$	$2.7 \times 10^5 \pm 6.01 \times 10^3$	124 ± 2.7	136 ± 8
10	$3.9 \times 10^4 \pm 9.3 \times 10^1$	$6.6 \times 10^4 \pm 1.34 \times 10^3$	30 ± 0.6	71 ± 3
100	$3.7 \times 10^4 \pm 1.6 \times 10^2$	$4.4 \times 10^4 \pm 1.16 \times 10^3$	20 ± 0.5	63 ± 3
1000	$3.7 \times 10^4 \pm 0.9 \times 10^{-2}$	$4.2 \times 10^4 \pm 5.0 \times 10^2$	19 ± 0.2	62 ± 3
<i>ptrace thread</i>				
1	$4.0 \times 10^4 \pm 5.6 \times 10^1$	$4.8 \times 10^4 \pm 9.8 \times 10^2$	22 ± 0.4	35 ± 1
10	$4.0 \times 10^4 \pm 4.5 \times 10^1$	$4.3 \times 10^4 \pm 6.5 \times 10^2$	20 ± 0.3	28 ± 1
100	$4.0 \times 10^4 \pm 9$	$4.2 \times 10^4 \pm 1.17 \times 10^3$	19 ± 0.5	25 ± 1
1000	$4.0 \times 10^4 \pm 1.0 \times 10^1$	$4.3 \times 10^4 \pm 9.2 \times 10^2$	19 ± 0.4	24 ± 1

Table 4.12: The cost of a single swap between between two functions on the two platforms. Measured time is reported per function swap operation.

	Reading	Writing
x86		
Wall time (μ sec)	0.51 ± 2.06	0.53 ± 2.06
Cycles	$1.84 \times 10^2 \pm 1.166 \times 10^3$	$1.29 \times 10^2 \pm 0.43 \times 10^2$
Instructions	57 ± 0	54 ± 0
x86_64		
Wall time (μ sec)	0.77 ± 2.13	0.99 ± 6.03
Cycles	$2.13 \times 10^2 \pm 6.27 \times 10^2$	$5.4 \times 10^1 \pm 7.46 \times 10^2$
Instructions	57 ± 13	49 ± 7

Table 4.13: Metric values gathered while `ptrace` reads or writes one word of data. Each value is given as $\mu \pm c$ where μ denotes the arithmetic mean and c denotes half of the width of the 95% confidence interval around μ .

with the size of the data structure being accessed. One-byte structures require on the order of 10^4 cycles to access. Linear models were derived to predict the number of instructions required for arbitrary data sizes.

The relative timings of different operations have implications for different usage patterns of DynInst. One method of using DynInst is to remove instrumentation between profiling phases. Such a strategy would need to have an interval between profiling stages of more than 10^5 cycles in order that the reduction in overhead is not outweighed by the cost of insertions. Given a modern gigahertz processor, this equates to having profiling phases separated by more than 10^{-4} seconds. This should not present problems in practice, as such short phases won't greatly impact the long-running programs which are worth optimising.

Chapter 5

The tuning driver and full system evaluation

This chapter develops the tuning driver, the third of Swift's three major components. With this component in place the full Swift system was evaluated on non-simulated hardware.

This chapter is organised as follows. Section 5.1 discusses the tuning driver, which contains the logic for invoking DynInst to change the target application. The section begins by describing ways in which the tuning driver may be configured. This discussion highlights the degree to which the driver is adaptable to different algorithm and data format selection problems. Also described are two assumptions made by the tuning driver about the target application.

Section 5.1.2 describes how the tuning driver searches the tuned application for sites amenable to tuning. The process of tuning requires the injection of code around these sites, which is subsequently executed by the tuned application. The structure of the injected code is described in 5.1.3 and its effects when executed are discussed in Section 5.1.4.

The tuning driver relies on shared libraries to provide the alternative implementations selected from, the data format conversion routines to condition input for these different implementations and the required control logic within the tuning application's address space. The libraries used are detailed in Section 5.2. The first of these libraries contains routines for sparse matrix-vector multiplication using a variety of matrix storage formats. While the basic algorithms are described in Section 2.4.2, Section 5.2.1 describes some additional scaffolding required to allow the routines to be injected into an oblivious target application.

Section 5.2.2 describes a library used to send measurements of the target

application to the learning algorithm and to receive responses from it. Section 5.2.3 presents a synchronisation library that facilitates communication between the target application and the mutator. This library is used to ensure that changes to the algorithm and data structures occur only when they will not cause incorrect execution of the target program.

Section 5.3 presents the results of a full system evaluation of Swift. It begins with a simple benchmark to measure the baseline overhead imposed by Swift when tuning any problem. Swift is then applied to the problem of sparse matrix format selection.

A note on terminology: This section defines the terms *original function*, *tunable function* and *optimisation site*. These terms arise when discussing the algorithm selection problem, which is the problem that Swift is designed to address. Swift supposes that the *original function* is present in the application it has been instructed to tune. Every invocation of the *original function* is located at a particular address within the target application - these addresses are *optimisation sites*. Swift has available to it a set of *tunable functions*. Swift's job is to replace the function calls at optimisation sites with those particular tunable functions that minimise the total execution time of the target application.

5.1 The design of the tuning driver

The tuning driver is responsible for using DynInst to manipulate the tuned application. An overview of its actions is contained in Algorithm 5.1.

5.1.1 Configuration

The tuning driver is driven by a configuration file containing information about a specific tuned application. This grants the driver a degree of generality, making it applicable to more than a single tuned application.

5.1.1.1 The original function

Configuration starts by specifying the original function. The driver makes certain assumptions about how this function is used. The first assumption is that the original function is located within a loop. This assumption stems from a requirement that any function tuned by Swift must be executed repeatedly, because the tuning method employed makes decisions based on measurements of function per-

Algorithm 5.1: Logic of the tuning driver.

```
1 read configuration
2 pause target application
3 inject libraries
4 locate optimisation sites
5 modify optimisation sites
6 resume target application
7 while target is running do
8     await conversion message
9     decode conversion message
10    retarget function invoked at optimisation site
11    trigger conversion in the target
12    send completion message
13 end
```

formance. A function executed once can be measured but tuning actions taken based on those measurements are pointless. Other forms of repetition exist, but it was judged that the effort required to support these cases would be better employed elsewhere.

There are several common situations where the original function lies within a loop. One example is when a system of equations is being solved iteratively. In this example, the original function implements one step of the iteration and is located within a loop that checks for convergence between successive approximate solutions. A second example is the application of a function to multiple data points, such as a function that calculates the forces on a particle in a molecular dynamics program.

To introduce the second assumption, recall that tuning is implemented by choosing between different functions, solving the same problem in different ways. Swift is not limited to functions that use the same parameters the original function. They can require additional information, or the same information stored in a different format. One example is where the original function takes a list of values stored as an array, while a tunable function takes the same set of values stored in a linked list. This is supported by two mechanisms (*binders* and *adaptors*) described later. These mechanisms rely on the fact that some data structures are unchanged within the loop already assumed. These structures can be copied and reformatted as required. The driver does not attempt to deduce

which variables are read-only within the loop - it makes its second assumption, that the person who configured it knew what they were doing. If they specify a variable is to be copied, then that variable is safe to copy.

5.1.1.2 The auxiliary libraries

After the original function, the next attributes configured are a set of auxiliary libraries. These libraries will be loaded into the address space of the target application and contain the tunable functions and support routines for data format conversion. These libraries are termed *auxiliary* because the driver has a set of libraries that it always loads into the target. These libraries provide support for its tuning mechanisms. Both types of library are described in Section 5.2.

5.1.1.3 Binder specifications

Binders provide a way to reference the original function's parameters, so that they can be used as arguments for the tuning functions. While DynInst will provide a list of a function's parameters, the DynInst API does not provide a way to use these parameters in function invocation expressions. There is a good reason for this: parameters are local variables, which are just names for offsets within the program stack. These offsets only make sense when the function in question is executing, which may not be the case when the expression is executed. However, given the specific way Swift uses DynInst, this problem does not arise - the parameters of the original function will always be valid to use in expressions. Such use requires them to be converted into global variables. This is done by a mechanism described in Section 5.1.3, but the mechanism is guided by configuration data. The configuration element `bind_some_variable=another_variable` is interpreted to mean: *whenever a reference to **some_variable** appears in the parameter list of a function, call that function using the global variable **another_variable** as the corresponding argument.* There is a variant: the configuration element `bind_a_pointer=another_variable*` causes references to `a_pointer` to be replaced by the address of `another_variable`.

5.1.1.4 Adaptor specifications

Closely related to binders are *adaptors*. Using adaptors, a tunable function can use parameters or data not explicitly passed to the original function. An adaptor is a function and is specified by name along with the names of its parameters.

These parameters must be variables previously declared as part of a binder specification, or the results of previously invoked adaptors. The return value of the function is available for use as a parameter in a tunable function. An example adaptor specification is `adaptor_max_A=find_max A size_A`. This will cause the function `find_max` to be invoked with parameters `A` and `size_A`, with the result passed to any tunable function that has a parameter called `max_A`.

5.1.1.5 Lookup specifications

A *lookup specification* encodes an association between a name and one of the tuning driver's internal variables. The association may be made by value or by reference, depending on the syntax of the specification. Having such associations allows for the tuning driver's internal variables to be used in binders or adaptors.

5.1.1.6 Allocation specifications

An *allocation specification* in the configuration file causes Swift to allocate a new global variable in the address space of the target application. The allocation specification has the form `allocate_name = type,initialiser`, where the type must be defined by the target application and is used to deduce the size of the allocation. Once allocated, the variable's value is set to that of the variable specified as *initialiser*. The allocated variable may be referred to elsewhere by the provided name. For example, the specification `allocate_max_row = int,find_max_row` creates a new variable with enough space to hold an integer and initialises it to the result of calling the adaptor function `find_max_row`.

5.1.1.7 Composed variable specifications

Composed variable specifications were created to work around a limitation of DynInst. DynInst does not work reliably on certain platforms when used to insert calls to functions with many parameters. To avoid this situation, the function in question was rewritten to take its required inputs as two aggregate variables (C `structs`). However, such a structure needs to be initialised prior to the function being called - the structure contains data that will vary over the course of a program's execution. Swift will generate the code to configure the structure based on a composed variable specification.

The specification is written using the syntax `compose_name = field_A->source_A, field_B->source_B, ...`. This causes the field named `field_A` of variable

`name` to be assigned the value of variable `source_A`, the field named `field_B` to be assigned the value of variable `source_B` and so on. There is a refinement to the specification: a field can be specified as being set from a variable name followed by an asterisk, as in `field->source*`. This results in the field being initialised to the address of the variable `source` rather than its value.

5.1.1.8 Conversion patterns

Tuning actions are configured using two regular expressions (a *target pattern* and a *conversion identification pattern*) and two strings (a *conversion prefix* and a *conversion suffix*). Any function within the target or auxiliary libraries whose name matches the target pattern is assumed to be a tunable function. Each tunable function can be automatically mapped to a conversion function, which is invoked prior to the first invocation of a newly chosen tunable function and is responsible for reformatting input data for use by that function. This mapping is specified by the conversion prefix, the conversion suffix and conversion identification pattern. The idea behind this is that part of every tunable function name must be unique within the set of all tunable function names. It is assumed that this part of the name is common to both the tunable function and its preparatory conversion routine. The conversion routine's name may have an additional prefix and suffix, common to all conversion routine names. As an example, assume four tunable functions `spmv_csr`, `spmv_bcsr_32`, `spmv_bcsr_41` and `spmv_bcsr_81`. Each has a conversion routine: `convert_csr`, `convert_bcsr_32`, `convert_bcsr_41` and `convert_bcsr_81`. The target pattern is configured as `spmv_b?csr.*`, which matches all four actions. By specifying `convert_` as the conversion prefix, the conversion suffix as empty and `csr | bcsr_[1-8][1-8]` as the pattern unique to each pairing of tunable and conversion functions, `spmv_csr` is mapped to `convert_csr`, `spmv_bcsr_32` is mapped to `convert_bcsr_32`, `spmv_bcsr_41` is mapped to `convert_bcsr_41` and `spmv_bcsr_81` is mapped to `convert_bcsr_81`.

5.1.2 Locating optimisation sites

The driver works with optimisation sites. As defined earlier, these are addresses of function call instructions that invoke the original function. They are detected using Algorithm 5.2 *

*As for Chapter 4, the function names used in these algorithms are not the names defined by DynInst. These names are chosen to be more informative.

Algorithm 5.2: Algorithm for detecting optimisation sites.

```

1 site_index  $\leftarrow$  0
2  $F \leftarrow \text{get\_all\_functions\_in\_target}()$ 
3 for  $f \in F$  do
4    $C \leftarrow \text{find\_all\_function\_calls\_in}(f)$ 
5   for  $c \in C$  do
6      $n \leftarrow \text{get\_name\_of\_function\_called\_by}(c)$ 
7     if  $n = \text{original\_function\_name}$  then
8        $k \leftarrow \text{find\_enclosing\_loop}(f, c)$ 
9        $\text{bind\_to\_site}(c)$ 
10       $\text{build\_instrumentation\_for\_site}(c)$ 
11       $\text{record\_site}(c, k, \text{site\_index})$ 
12       $\text{site\_index} \leftarrow \text{site\_index} + 1$ 
13    end
14  end
15 end

```

The algorithm uses a call to DynInst to find all functions in the target application. For every function in this list, DynInst is used to find all points at which that function invokes another function. If the invoked function matches the name of the original function specified in the configuration file, then an optimisation site has been detected. Each site is associated with a unique index for future identification.

This algorithm is not efficient - it would be better to find the single target function and then work backwards to locate all invoking instructions. However, the version of DynInst being used does not support this method of operation. In practice, this code is executed only once when initially attaching to the target application and so the cost is not a major impediment.

The function `find_enclosing_loop` is used to find the loop that, by the first assumption in Section 5.1.1, is enclosing the invocation. The logic of this function is shown in Algorithm 5.3.

The function uses DynInst to find all loops within the function that contains the invocation and then checks each loop to find the one that contains the function call. The specific method used to check for containment will detect immediately containing loops - if the invocation is located within a set of nested loops, the inner-most loop is the one detected and returned by this function.

Algorithm 5.3: Algorithm to find the loop around an optimisation site.

Input: c : instruction invoking the target function**Input:** f : function containing c

```

1 address  $\leftarrow$  get_address_of(  $c$  )
2 cfg  $\leftarrow$  get_control_flow_graph(  $f$  )
3  $L \leftarrow$  get_loops_in(  $cfg$  )
4 for  $k \in L$  do
5   if loop_contains(  $k$ , address ) then
6     return  $k$ 
7   end
8 end

```

Once the loop has been detected and stored for future use, the next step is to bind variable specifications to the call site. This is the process whereby the binders and adaptors specified in the configuration file are initialised. The routine begins by allocating space within the target application's address space for hard-coded variables, which are used by the inserted snippets. The variables are the integers *counter*, *current iterations*, *new iterations*, *current action*, *new action*, *conversion required*, *cleanup required* and *site index* and the variable *analysis* (with type `analysis_t`, which is expected to be defined by one of the supporting libraries). Once these variables are allocated and references to them stored, the routine goes through five more stages: it processes binder definitions, lookup definitions, allocation definitions, adaptor definitions and composed variable definitions. The syntax for these definitions is presented in Section 5.1.1.

Binder definitions are processed by searching the target application for a global variable with the name specified by the definition. Once found, a reference to the variable is stored into one of two data structures, indexed by the key used in the binder definition. The choice of data structure depends on whether the variable refers to a variable or a pointer to a variable. The two cases are treated differently when retrieving the stored variable reference.

Lookup definitions are used to let a snippet make use of variables the driver allocates within the target application, such as the variables *site index* and *analysis* mentioned earlier. Processing the lookup definitions requires storing handles to those variables that are the targets these definitions. These handles are then made available to adaptors and tuned functions. The same process is used to process allocation definitions: these definitions refer to global variables that Swift has

allocated within the target application, so all that is required is storing handles to these variables for later use.

Adaptor definitions are processed by constructing the instruction sequences to invoke the adaptor functions. The set of all adaptor invocations is guarded such that it will only execute once for every time the loop containing the original function is entered.

Composed variable definitions are processed by parsing the textual specification into a list of components, and storing the name of each field and the means of setting it for future use.

5.1.3 Injected code: structure

The driver injects three blocks of code around each optimisation site. Algorithm 5.4 shows how the original code is changed by these modifications. The three blocks each perform a specific function and are named for the point at which they are inserted: they are the *pre-loop block*, the *pre-call block* and the *post-loop block*. Each block is described below.

There is one modification that does not fall within these three blocks, which requires retargeting the optimisation site to invoke a stub function rather than the original function. This stub function serves two purposes.

The first purpose is to stop the original function being invoked - as the tunable functions may have different parameter lists, they cannot just be written over the original function call, because the instructions preceding that call have set up that call's parameters in a way that will not work for a function with a different signature. The tunable functions must therefore be invoked elsewhere, but the original function must not be invoked, because that would defeat the point of tuning. Therefore, the instruction that calls the original function is replaced by one that calls a stub function.

The second purpose of the stub function is to implement the binders whose specification was discussed in Section 5.1.1.3. Algorithm 5.5 shows an example stub function, which is designed to replace a function that multiplies a vector by a matrix stored in coordinate format. The replacement function monitors the variable `bind_req`, which is set by the pre-loop block. If this variable is non-zero, then the stub function records its arguments in the global variables named in the configuration file's binder specifications. The stub function also zeroes the guard variable, rendering it inert until reactivated by the pre-loop block. The final task of the stub function is to execute the original function. This is required so that,

if the tuned loop is set to execute for a fixed number k of iterations, the data will have been processed exactly k times - either by the original function or an alternative.

Algorithm 5.4: The main loop of the target application, before and after modification by the driver. Each line that is numbered in the modified version corresponds to the line with the same number in the original version. Lines in blue text and without numbers are injected by the driver.

Original Function

```
1 while not converged do
2   some_code
3   original_function( arguments )
4   some_code
end
```

Modified Function

Initialisation (see Algorithm 5.6.)

```
1 while not converged do
2   some_code
   pre-call block (see Algorithm 5.7.)
3   stub_function( arguments )
   tunable_function( possibly different arguments )
4   some_code
end
post-loop block(see Algorithm 5.8.)
```

Algorithm 5.5: An example stub function.

Input: row_array, column_array, value_array, element_count, input_vector**Output:** output_vector

```

1 if bind_req = 1 then
2   to_bind_row ← row_array
3   to_bind_col ← column_array
4   to_bind_nz ← value_array
5   to_bind_nnz ← element_count
6   to_bind_vec ← input_vector
7   to_bind_res ← output_vector
8   bind_req ← 0
9   original_function( row_array, column_array, value_array, element_count,
                      input_vector, output_vector )
10 end

```

5.1.3.1 Pre-loop block

The pre-loop block has only one purpose: to set a flag that indicates initialisation is required. It does not itself perform this initialisation - that is done by the pre-call block. This arrangement is required to handle situations in which the target application is modified while executing the loop of interest. Rather than duplicate the initialisation code over two blocks, the decision was made to confine everything to the pre-call block and merely flag that it is required within the pre-loop block.

Algorithm 5.6: Pre-loop block.

```

1 initialisation_required ← 1

```

5.1.3.2 Pre-call block

The pre-call block (Algorithm 5.7) has several purposes: it initialises the instrumentation on its first invocation, and on later invocations it performs the following functions:

- reporting elapsed time to the learning agent,

- incrementing the loop tracking counter used to estimate loop iteration counts [†],
- transforming program state into the set of attributes expected by the learning agent,
- querying the learning agent whenever a new action is required and
- triggering conversion between data formats.

The instrumentation is initialised by setting various global variables. Some of these variables are read directly by instrumentation and some are used by shared libraries. Some are allocated within the address space of the target application by the driver, using DynInst as an intermediary. The rest are exported by shared libraries. The *current action* variable records the index of the currently selected action - it is initialised to the special value *sentinel* so that no matter which action is first selected by the agent, data format conversion will always be triggered. The variable *counter* counts the number of times the loop iterates. While in some cases there is a loop induction variable in the target application that provides this information, this is not always true - for example, when the loop is terminated based on the result of a convergence test. The number of iterations before a new action is required is stored in the variable *current iterations*. The variable *site index* is used to distinguish between different invocations of the tuned function. It is set to a constant, which is determined at the time DynInst writes the instruction into the code. The flag *run adaptors* records whether adaptors need to be executed. Adaptors are executed only once for every transition from without to within the innermost loop.

The initialisation code is executed conditionally based on the value of the *initialisation required* flag. This flag is set by the pre-loop block and also by DynInst when it places the instrumentation. This second method is a fail-safe to handle the case when DynInst instruments the target application while that application is already executing the target loop. In those circumstances, the pre-loop block won't be executed until the next time the loop is reached. By setting *initialisation required* itself at the time of instrumentation, the driver prevents uninitialised use of the pre-call and post-loop blocks. All later executions of the loop will pass through the pre-loop block and get the flag set in that manner.

The pre-call block is also responsible for reporting timing data to the agent. The timing data is the wall-clock time elapsed during the execution of a set of

[†]See Section 3.2.6 for details on how the agent uses this counter to estimate iteration counts.

function invocations, whose size was specified by the agent. Timing data are only reported after at least two iterations of the loop have been observed [‡]. This is because the starting time for the first measured action is set during the second observed iteration, a delay which is due to initialisation only happening on the first observed iteration. Note that initialisation involves not only setting variables, but also binding binders, which happens during the first invocation of the `stub_tuned_function` after the instrumentation has been emplaced.

Additionally, the pre-call block must transform the program state (here considered synonymous with the arguments of the original function) into a set of attributes. This is done by invoking a function, whose name is specified in the configuration file - in Algorithm 5.7, this function is named `analyse_function`.

5.1.3.3 Post-loop block

The post-loop block (Algorithm 5.8) cleans up after the loop has terminated. It reports to the agent the number of iterations measured, which is used by the agent to estimate future iteration counts. If the learner's estimate of the iteration count was too high then the post-loop block also reports the time elapsed for the last set of iterations. Finally, the post-loop block also removes the tuning action from the code. This is done so that it is not called during the next loop invocation before the analysis phase of the pre-call block has been executed. While it would be possible to encapsulate the tuning function within the check for initialisation which guards the pre-call and post-loop blocks, this approach was used to avoid adding another branch instruction to the tuned loop.

5.1.4 Injected code: behaviour

This section describes how the driver carries out the tuning process, making use of the logic that has been injected into the target application as described in Sections 5.1.3 and 5.2

The process starts when a new action needs to be selected. This need arises when the number of iterations specified for the previous action has elapsed, or when there was no previous action. The pre-call block calls into the remote agent library [§], which obtains from the learning agent a pair (A, C) where A is the next

[‡]This wording is due to the previously mentioned issue with inserting instrumentation while the loop is iterating. If the instrumentation has been inserted on the tenth iteration, then timing data will not be reported before the twelfth iteration, which is the second observed iteration.

[§]This library is described in Section 5.2.2

Algorithm 5.7: Pre-call block.

```

1 if initialisation_required = 1 then
2   current_action  $\leftarrow$  sentinel
3   counter  $\leftarrow$  0
4   current_iterations  $\leftarrow$  0
5   run_adaptors  $\leftarrow$  1; site_index  $\leftarrow$  n
6 else
7   if counter = 0 then
8     analysis  $\leftarrow$  analyse_function( bound parameters )
9   end
10  counter  $\leftarrow$  counter + 1
11  if current_iterations = 0 then
12    if counter > 1 then
13      report_time_to_agent( site_index, counter, analysis,  $\delta_{\text{time}}$  )
14    end
15    new_action, new_iterations  $\leftarrow$  query_agent( site_index, counter,
16      analysis )
17    if new_action  $\neq$  current_action then
18      trigger_conversion( site_index, new_action )
19      await_conversion_completion()
20      current_action  $\leftarrow$  new_action
21    end
22    reset_timer()
23    current_iterations  $\leftarrow$  new_iterations
24  end
25  current_iterations  $\leftarrow$  current_iterations - 1
26 end

```

Algorithm 5.8: Post-loop cleanup.

```

1 new_action  $\leftarrow$  -1
2 trigger_conversion( site_index, new_action )
3 await_conversion_completion()
4 if initialisation_required = 0 then
5   if current_iterations  $\neq$  0 then
6     now  $\leftarrow$  get_current_time()
7      $\delta_{\text{time}}$   $\leftarrow$  now - then
8     report_time_to_agent( site_index, counter, analysis,  $\delta_{\text{time}}$  )
9   end
10  report_counter_to_agent( site_index, counter )
11 end

```

action to undertake and C is the maximum [¶] number of iterations before a new action must be selected.

Once a new action has been obtained, the pre-call block compares it to the action currently in use. If they are different, the new action needs to be emplaced over the old one. This process is carried out by the driver. The pre-call block in the target sends a message to the driver over a message queue. It then waits for a response from the driver. When this response is received, the driver has finished working and it is safe for the target application to continue executing.

The message to the driver contains two elements: the index of the optimisation site to be tuned and the index of the new action. Upon receiving this message, the driver removes any tuning actions already emplaced at the indicated site. As part of this process, cleanup functions associated with the removed action are triggered. The driver then builds up the instruction sequence to invoke the tuned function whose index was part of the message. This sequence is inserted into the target application immediately after the stub function which has replaced the call to the original function. A handle to the newly inserted function call is saved to enable its future excision. Once this has been done, the driver assembles the instruction sequence to invoke the appropriate conversion function. This sequence is then executed through DynInst's *one-time code* facility. This results in the conversion function being executed by the target application. Once the conversion function returns, the driver's work is done: the loop contains the instruction to

[¶] C is an upper bound because the loop may finish iterating because C is reached - it is an estimate, derived by the process described in Section 3.2.6

invoke the correct tunable function and the data is in the appropriate format for that function. The driver sends a message back to the target application and then waits for the next message which will restart the cycle.

There is one special case that modifies the above logic: a special action index of -1 is sent by the target application as part of the post-loop block. When the driver receives this index, it removes any existing tuning actions but does not emplace any new ones. This allows the loop to be cleaned up once it has finished a set of iterations.

5.2 Auxiliary Libraries

This section describes the various libraries used by the system. Some of these libraries are always used, while some depend on the type of tuning that is taking place. All are injected into the target application's address space when DynInst attaches to the target.

5.2.1 Sparse Matrix Format Libraries

One problem to which Swift has been applied is the problem of tuning sparse matrix-vector multiplication by selecting beneficial storage formats for the matrix. The set of multiplication algorithms and data conversion functions associated with each format are collected into the library *libspmv.so*

The multiplication functions are as described in Section 2.4.2, apart from one detail. During conversion to $r \times c$ -BCSR format, a matrix's row count will be padded, if necessary, to ensure it is an integer multiple of r . The matrix's column count is likewise padded to an integer multiple of c . This padding is expected by the multiplication routines, but it raises a problem: the input and output vectors haven't been padded. Because the vectors are not assumed to be unused outside the tuning function, they cannot be reallocated without changing every possible instance of their use within the loop, which would be complex. Instead, if required, larger temporary vectors are allocated within the multiplication functions themselves. Figures 5.9 and 5.10 shows the relevant portions of the 3×2 -BCSR multiplication routine.

The bulk of the extra code is placed within guards so that it is not executed when not necessary. These routines are expected to be applied to the same matrix several times in succession, in which case the size checks and any necessary allocation are only done infrequently - only when the matrix changes. Some data

Algorithm 5.9: Extra BCSR logic located before the multiplication

```

1 aligned_in_vec : static  $\leftarrow$  NULL
2 aligned_in_vec_size : static  $\leftarrow$  0
3 cached_in_vec_size : static  $\leftarrow$  -1
4 in_copy_needed : static  $\leftarrow$  False
5 aligned_res_vec : static  $\leftarrow$  NULL
6 aligned_res_vec_size : static  $\leftarrow$  0
7 cached_res_vec_size : static  $\leftarrow$  -1
8 out_copy_needed : static  $\leftarrow$  False
9 if cached_in_vec_size  $\neq$  max_col then
10     cached_in_vec_size  $\leftarrow$  max_col
11     aligned_in_vec_size  $\leftarrow$  ncol
12     if max_col + 2  $\geq$  ncol then
13         aligned_in_vec_size  $\leftarrow$  max_col + 1 + 2
14         aligned_in_vec  $\leftarrow$  reallocate( aligned_in_vec_size  $\times$  sizeof(double) )
15         aligned_in_vec[ncol:aligned_in_vec_size - 1]  $\leftarrow$  0.0
16         in_copy_needed  $\leftarrow$  True
17     end
18     else
19         aligned_in_vec  $\leftarrow$  in_vec; in_copy_needed  $\leftarrow$  False;
20     end
21 end
22 if in_copy_needed = True then
23     aligned_in_vec[0:ncol-1]  $\leftarrow$  in_vec
24 end
25 if cached_res_vec_size  $\neq$  nrow then
26     cached_res_vec_size  $\leftarrow$  nrow
27     aligned_res_vec_size  $\leftarrow$  nrow
28     if max_col mod 2  $\neq$  0 then
29         aligned_res_vec_size  $\leftarrow$  aligned_res_vec_size + 3 - ( nrow mod 3 )
30         aligned_res_vec  $\leftarrow$  reallocate( aligned_res_vec_size  $\times$  sizeof(double) )
31         aligned_res_vec[nrow:aligned_res_vec_size - 1]  $\leftarrow$  0.0
32         out_copy_needed  $\leftarrow$  True
33     end
34     else
35         aligned_res_vec  $\leftarrow$  res_vec; out_copy_needed  $\leftarrow$  False;
36     end
37 end
38 if out_copy_needed = True then
39     aligned_res_vec[0:nrow - 1]  $\leftarrow$  res_vec
40 end

```

Algorithm 5.10: Extra BCSR logic located after the multiplication

```
1 if out_copy_needed = True then
2   res_vec ← aligned_res_vec[0:nrow-1]
3 end
```

copy is unavoidable on every invocation, however, because one scenario that must be accounted for is a caller accumulating results in a single result vector. Thus, the raw result vector must be copied to the aligned version before the multiplication logic and the aligned vector copied back afterwards. It is also necessary to copy the initial input vector. All of these copies are themselves guarded and are not executed unless reallocation was required.

It may seem wasteful that the code initialises those portions of the reallocated vectors that fall outside the dimensions of the original vectors. This is done to ensure that these regions of memory do not contain values that, when interpreted as floating point values, appear special - for example, as $\pm\infty$. Passing such values through the multiplication logic could result in floating point traps being delivered to the target application, which would slow it down and may cause it to behave incorrectly

Before multiplication can take place, the matrix must be converted into a suitable representation. This is the job of the conversion functions. All $r \times c$ -BCSR conversion functions follow the same algorithm, albeit with different values for r and c . The algorithm processes the matrix in block-rows - the first block row contains the first r rows, the second block row contains the second r rows and so on. The block row is converted into an undirected weighted graph, with vertices corresponding to elements. Between every pair of vertices u, v there is an edge whose weight is $|column(u) - column(v)|$, where $column(x)$ is a function returning the column index of element x . Once constructed, the graph is processed to delete all edges whose weight exceeds $c - 1$. This breaks the single graph into a set of graphs, with the property that two elements u and v cannot belong to the same $r \times c$ block unless u and v are found in the same graph. Figure 5.1 shows an example of this conversion from a matrix into a set of graphs.

Once the set of graphs has been computed, the elements are grouped into blocks. Blocks are defined based on the graphs within the set: for each graph G , the functions $min(G)$ and $max(G)$ return the smallest and largest column index in the constituent elements of G . The graph G is converted into a set of blocks

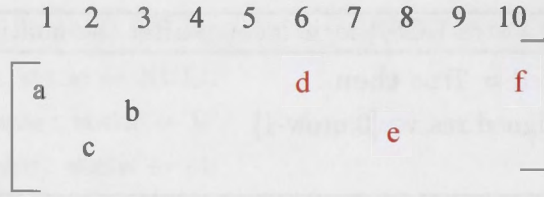
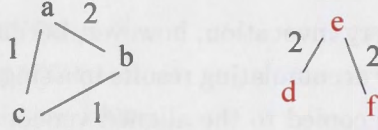
(a) Example 3×10 matrix.(b) The example matrix as a set of graphs for 3×3 -BCSR.

Figure 5.1: The conversion of matrices to graph sets for region detection.

covering all columns between $\min(G)$ and $\text{rounded}(\max(G), \min(G), c)$, where

$$\text{rounded}(\max, \min, c) = \begin{cases} \max & \delta = 0, \\ \max + (c - \delta) & \delta \neq 0 \end{cases}$$

Where $\delta = ((\max - \min) + 1) \bmod c$

The result of the conversion process is the set of all sets of blocks so generated, along with lists of column indices and row indices describing the layout of the blocks.

The purpose of using this arrangement of graphs and sets of blocks is to strike a balance between speed and the number of superfluous zero entries introduced by the conversion process. The simplest algorithm would have blocks whose column indices were integer multiples of c . However, this can introduce superfluous zeroes. Consider the following case: a 1×20 matrix with non-zeroes at (zero-based) columns 2,3,4,5,14,15,16 and 17. Using the simple algorithm when converting to 1×4 -BCSR format, blocks are formed as follows: (2,3), (4,5), (14,15), (16,17). There are explicitly stored zeroes at indices 0,1,6,7,12,13,18,19 - eight in total. However, using the above method with sets of graphs results in blocks with indices (2,3,4,5), (14,15,16,17), eliminating all explicitly stored zeroes. This is because the graph-based approach does not have to align blocks on multiples of c .

However, the graph-based approach also has its faults. Consider a different matrix, with non-zeroes at columns 2,3,4,5,9,10. Using the graph-based approach

for conversion to 1×4 -BCSR, all of these elements belong to the sample graph, resulting in blocks (2,3,4,5),(6,7,8,9),(10,11,12,13), with six explicit zero elements. It would be possible to eliminate four of these by starting the second block at index 9. However, for the agent to perceive this would require a more expensive analysis of possible block starting points.

5.2.2 Remote agent library

The remote agent library provides the target application with the logic to send and receive messages to the learning agent over sockets. It provides a set of routines that are invoked by the DynInst-inserted instrumentation. The first two routines are `query_agent` and `report_counter`. The `query_agent` routine contains the logic to send a message to the learning agent asking for a new format and iteration count and to receive reply containing this information. The `report_counter` routine reports the iteration count at the termination of every invocation of every tunable loop.

The remote agent library maintains a timer, which is used to measure the time required to execute tunable actions. The `reset_timer` function resets this timer, and the `report_time` function informs the learning agent of the length of time which has passed since the last timer reset.

5.2.3 Synchronisation library

The synchronisation library provides a means for the instrumented application to exchange messages with the driver. Messages are transferred over message queues. As well as being injected into the target application, the library is linked into the driver. The driver invokes the library's initialisation function to allocate two private message queue identifiers. One identifier is used for communications from the target to the driver and the other for the reverse. Once obtained by the driver, these identifiers are passed to the target by using DynInst to write them into global variables exported by the library. The library provides four functions to its users: `trigger_conversion`, `await_trigger`, `signal_completion` and `await_completion`. Each of these places a specific message on one of the two queues, or takes the first message (in a blocking fashion) from a specific queue and interprets it. The sequence of messages is as follows:

1. The driver calls `await_trigger` at the start of its main processing loop. This blocks.

2. Later, the logic injected into the target application calls `trigger_conversion`.
3. The tuning logic then calls `await_completion` and blocks.
4. The driver's invocation of `await_trigger` unblocks and returns. The driver adapts the target application and then calls `signal_completion`. It then starts another iteration of the main loop by calling `await_trigger` again.
5. Meanwhile, the tuning logic's call to `await_completion` returns and the target application is released to continue processing.

For `signal_completion` and `await_completion` the payload is ignored - the important event is the presence of the message. The other two functions transmit and receive both the site counter for the invoking optimisation site and the index of the action which is to be emplaced. This emplacement is the job of the main processing loop.

5.2.4 Matrix Analysis Library

The matrix analysis library analyses a matrix stored in coordinate format and produces a set of five matrix attributes: row count, column count, density, standard deviation in per-row non-zero counts and the mean neighbour count for each nonzero value. As explained earlier, these five attributes form the learning agent's perception of the matrix. The first three quantities are simple to calculate from the metadata stored with the coordinate matrix. Calculating the final two quantities involves a four stage process: the data are preconditioned, auxiliary data structures are created and populated, the non-zero deviation is calculated and then the mean neighbour count is determined.

The data is preconditioned by sorting it by row number and within rows by column index. The sort is performed by the `qsort` routine available in the C standard library. Use of this routine requires the three arrays of coordinate format (row indices, column indices and values) to be copied into a new representation: an array of structures, each having three elements: row, column and value. After the routine has returned, the reordered structure array is split back into the three arrays. Before these steps are executed, the input data is scanned. This scan assesses whether or not the matrix is already sorted. If it is, then the data copy steps as well as the sorting step are skipped.

The second stage of the analysis is the construction of auxiliary data structures. These data structures describe the indices at which the value of the row

array changes - that is, the offset within the three arrays at which each row begins. The procedure followed is shown in Algorithm 5.11. Given an input $R \times C$ matrix with N nonzero values, the amount of work done by Algorithm 5.11 will scale as $O(R + N)$.

Algorithm 5.11: Analysis data structure setup.

Input: a matrix with R rows, C columns and N non-zero elements

```

1
2 for  $r \in [1, R]$  do
3    $\text{first\_column\_in\_row}[r] \leftarrow C$ 
4    $\text{row\_start\_index}[r] \leftarrow N + 1$ 
5 end
6  $c \leftarrow$  smallest column index associated with a non-zero element
7  $\text{first\_column\_in\_row}[0] \leftarrow c$ 
8  $\text{row\_start\_index}[0] \leftarrow 0$ 
9  $\text{row\_start\_index}[R] \leftarrow N$ 
10  $i \leftarrow 0$ 
11 for  $e \in \text{matrix elements}$  do
12    $r \leftarrow \text{row}(e)$ 
13    $c \leftarrow \text{column}(e)$ 
14   if  $c < \text{first\_column\_in\_row}[r]$  then
15      $\text{first\_column\_in\_row}[r] \leftarrow c$ 
16      $\text{row\_start\_index}[r] \leftarrow i$ 
17   end
18    $i \leftarrow i + 1$ 
19 end

```

Given the array of row starting offsets constructed in stage two, the third analysis step can use the difference between successive pairs of offsets to determine the number of non-zeros per row. These quantities are used to calculate the standard deviation of the number of non-zeros over all rows.

The final analysis stage is the computation of mean neighbour counts. This computation assumes a matrix with at least three rows. Further, the algorithm used computes only an approximate neighbour count - it ignores elements in the leftmost column, the rightmost column and the top and bottom rows. This is done for the sake of simplicity, but it means that the results may be inaccurate for small matrices with specific arrangements of non-zero elements. The mean

neighbour analysis algorithm (Algorithm 5.12) processes triplets of rows: the i^{th} row is processed using data from rows $i - 1$ and $i + 1$. For every element e in the i^{th} row, all three rows are examined for the presence of elements at columns identical to or adjacent to the column containing e . The number of such elements is added to the total neighbouring element count. The final result is obtained by dividing this count by the number of elements processed.

Algorithm 5.12: Mean neighbour count analysis.

```

1 count  $\leftarrow$  0
2 total  $\leftarrow$  0
3 foreach row possessing a predecessor and a successor do
4    $K \leftarrow$  the set of elements in the current row
5    $P \leftarrow$  the set of column indices of elements in the previous row
6    $C \leftarrow$  the set of column indices of elements in  $K$ 
7    $N \leftarrow$  the set of column indices of elements in the next row
8   foreach  $curr \in K$  do
9      $c \leftarrow$  column(  $curr$  )
10     $n \leftarrow$  the set  $\{c - 1, c, c + 1\}$ 
11     $s \leftarrow$  the size of  $P \cap n$ 
12    total  $\leftarrow$  total +  $s$ 
13     $s \leftarrow$  the size of  $C \cap n$ 
14    total  $\leftarrow$  total +  $s$ 
15     $s \leftarrow$  the size of  $N \cap n$ 
16    total  $\leftarrow$  total +  $s$ 
17    count  $\leftarrow$  count + 1
18  end
19 end
20 return  $\frac{total}{count}$ 

```

5.3 Experimental Evaluation

Swift was evaluated in two experiments. The first experiment applied Swift to a benchmark scenario where the cost of switching between actions was low (Section 5.3.2). The second experiment uses Swift to perform sparse matrix format selection (Section 5.3.3).

5.3.1 Limitations

The experimental evaluation was constrained by the following problems with DynInst:

- DynInst does not work correctly with vectorised floating point operations (implemented using Intel's SSE/SSE2/SSE3 instructions). All BCSR formats have been implemented using only scalar instructions.
- DynInst contains bugs that prevent it working on the available 64-bit platforms. Thus, evaluation is restricted to the 32-bit Core2 system.

5.3.2 A synthetic benchmark

This section discusses the application of Swift to a synthetic benchmark problem. This problem has the advantage of not requiring costly analysis or data conversion operations. It is designed to minimise the costs specific to a particular tuning problem in order to focus on the overhead of Swift itself.

The benchmark is based on the target application shown in Algorithm 5.13. There are two tunable functions available, each built around the same two summations:

$$\begin{aligned}
 Big &= \sum_{i=0}^{999} \sum_{j=0}^{999} i \times (3 - j) \\
 Small &= \sum_{i=0}^{499} \sum_{j=0}^{499} i \times (3 - j)
 \end{aligned}$$

The two tunable functions f and g depend on the iteration count of the outermost loop, and are defined in such a way that g is faster on the first iteration of the outermost loop of the application and f is faster on the second iteration.

$$f(k) = \begin{cases} Big & \text{if } k \text{ is even or zero} \\ Small & \text{otherwise} \end{cases} \quad g(k) = \begin{cases} Small & \text{if } k \text{ is even or zero} \\ Big & \text{otherwise} \end{cases}$$

The agent will replace the original function with either f or g . The relative performance of these options changes over time. If the agent selects the function f at every opportunity, the system will perform poorly on the first 100 iterations and well on the last 200. If the agent uses g exclusively, the system will perform

Algorithm 5.13: The target application of the synthetic benchmark

```

1 function original_function
2    $\sum_{i=0}^{999} \sum_{j=0}^{999} i \times (3 - j)$ 
3 end
4 function main
5    $steps_0 \leftarrow 100$ 
6    $steps_1 \leftarrow 200$ 
7    $repeats_0 \leftarrow 50$ 
8    $repeats_0 \leftarrow 500$ 
9   for  $k \in [0, 1]$  do
10    for  $step \in [1, steps_k]$  do
11       $then \leftarrow$  current system time
12      for  $repeat \in [1, repeats_k]$  do
13        original_function(k)
14      end
15       $now \leftarrow$  current system time
16      record elapsed time  $now - then$ 
17    end
18  end
19 end

```

well to begin with and then suffer on the last 200 iterations. The agent's goal is to switch between the use of f and the use of g in such a way as to reduce the total execution time of the benchmark.

Two stages of preparation are required to apply Swift to this target. The first stage requires compiling the functions f and g into a shared library. The library must also contain an analysis function, conversion functions, functions to clean up after each action and a stub function. For this benchmark the conversion and clean up functions are empty and the stub function records its input (i) in a location accessible by Swift. The only complication is the analysis function: Swift currently lacks a general method of specifying the attribute vectors by which the learning agent identifies different states of the target application. The current method is explicitly coded to pass the five attributes designed to measure matrix structure. For this benchmark, the analysis function sets these attributes to the same arbitrary values every time it is invoked. This is sufficient for the learner - it attaches no meaning to any of the attributes, viewing them as a set of five numbers. Incidentally, the fact that the analysis function uses the same arbitrary values for every invocation means that the learning agent has to deal with perceptual aliasing.

The second preparatory stage is the construction of a configuration file. This configuration file is sufficient to teach Swift how to utilise the shared library.

The benchmark was executed with $w_{min} = 10^{-5}$, $w_{initial} = 0.001$, a random seed of 9,812 and the window method of counter estimation. Results were gathered on the Core2 platform described in Section 3.4.2 and Appendix C. The benchmark code was compiled without optimisations, which is acceptable because the goal of this experiment is to evaluate relative performance, not absolute performance. That being the case, disabling compiler optimisations made developing the benchmark easier, since there was no need to outwit the optimisation logic.

The results from the benchmark are shown in Figures 5.2 through 5.5. Figure 5.2 shows the manner in which Swift tracks the change in optimal function. The plot is a summary of seven experimental runs. The mean elapsed times for each iteration are plotted with lines. The error bars, plotted at every tenth point, represent the standard error in the mean.

It can be seen that Swift successfully tracks the changing optimal function. There is a small delay before it reacts to the change, which is the expected behaviour - Swift cannot react to a change until an exploration action is undertaken.

Figure 5.3(a) compares the cumulative execution time resulting from static use of the two functions f and g to the execution time required when the target

application is guided by Swift. The plotted points are the mean value of seven experimental runs. The latter begins slower than either action but quickly outpaces $f(i)$. After 100 iterations, Swift has consumed 7 seconds while $f(i)$ has consumed 18.7 - a reduction of 63%. At that point, however, $g(i)$ is still the best choice - it has consumed only 4 seconds, 43% less than Swift. After 106 iterations, Swift becomes the optimal choice. Swift remains optimal until iteration 264, at which point it has accumulated enough overhead to grow slower than $f(i)$. These transitions are shown in Figure 5.3(b). Figure 5.4 plots the ratio of Swift's cumulative execution time to the smallest cumulative execution time incurred by f or g . The figure has been cropped to show only ratios no greater than 1 - in other words, only those ratios where Swift provides a performance gain. The performance gain peaks at a ratio of 0.74 at the 111th iteration - at this point, Swift has required 26% less time to compute than either of the static functions.

Figure 5.5 compares timing data gathered from two different points within Swift. The data labelled *kernel timings* are the same data plotted for Swift in Figure 5.2. The data labelled *internal timings* are gathered by snooping on the timing data being sent to the learning agent. These data exclude some of the overhead imposed by Swift - they do not include the time required to communicate with the agent, to trigger conversion functions or to direct optimisation sites to new tunable functions. The data are the means for seven experimental runs, with standard errors plotted at every ten iterations. It can be seen that the overhead is rather small, especially when the total cost is high. It should be noted that this is a lower bound on overhead, for two reasons:

1. The design of the injected instrumentation means that some overhead will be included in the internal dataset, and
2. The data conversion functions are empty stubs. If real data conversion were required, the overhead could be considerably higher.

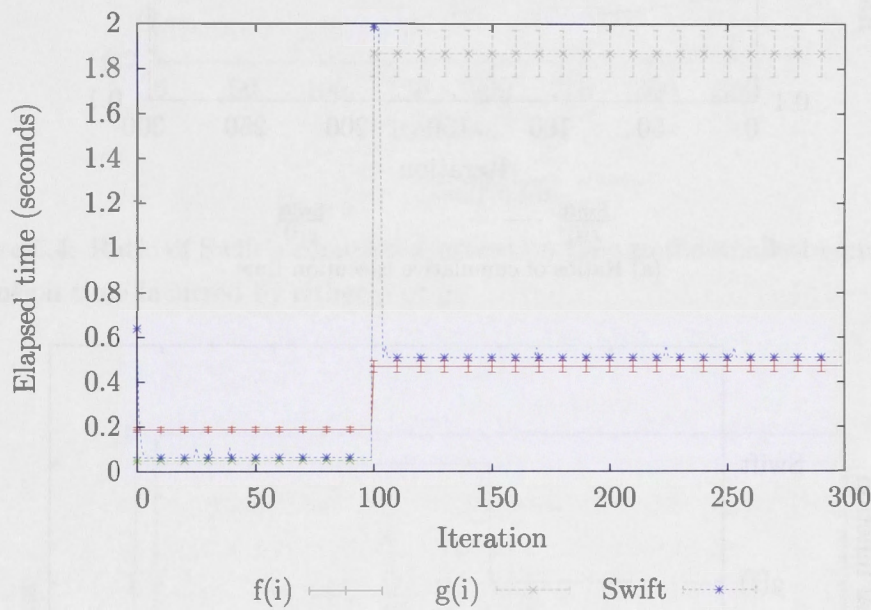
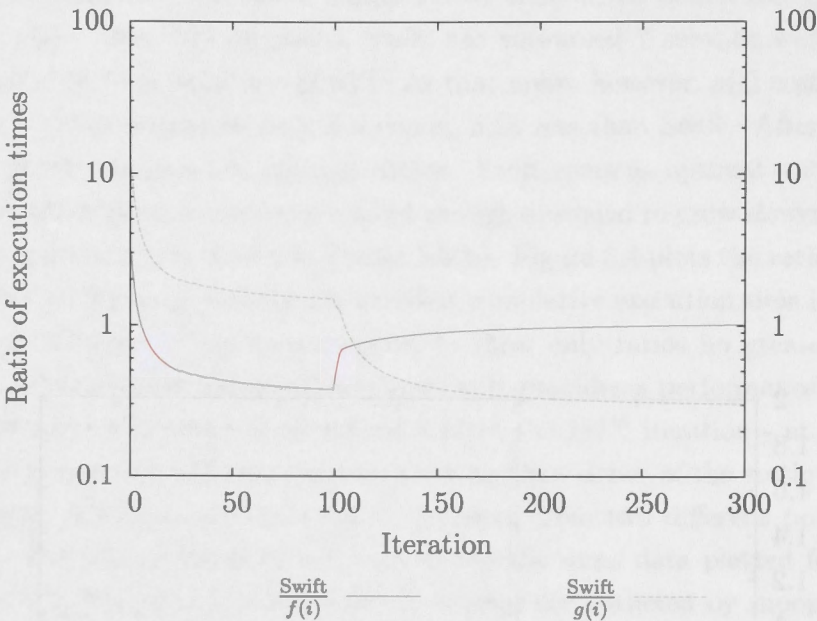
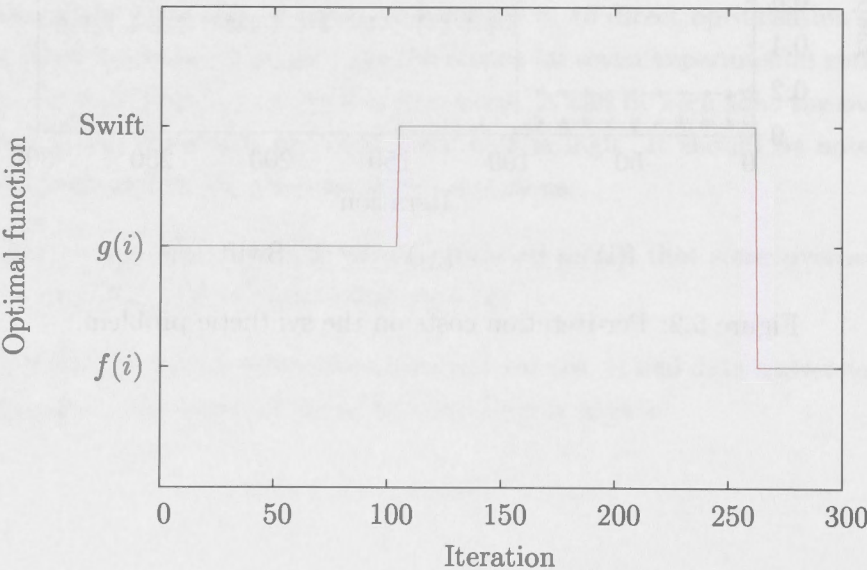


Figure 5.2: Per-iteration costs on the synthetic problem.



(a) Ratios of cumulative execution time.



(b) The change in optimal function as computation progresses.

Figure 5.3: Comparison of cumulative execution time on the synthetic problem.

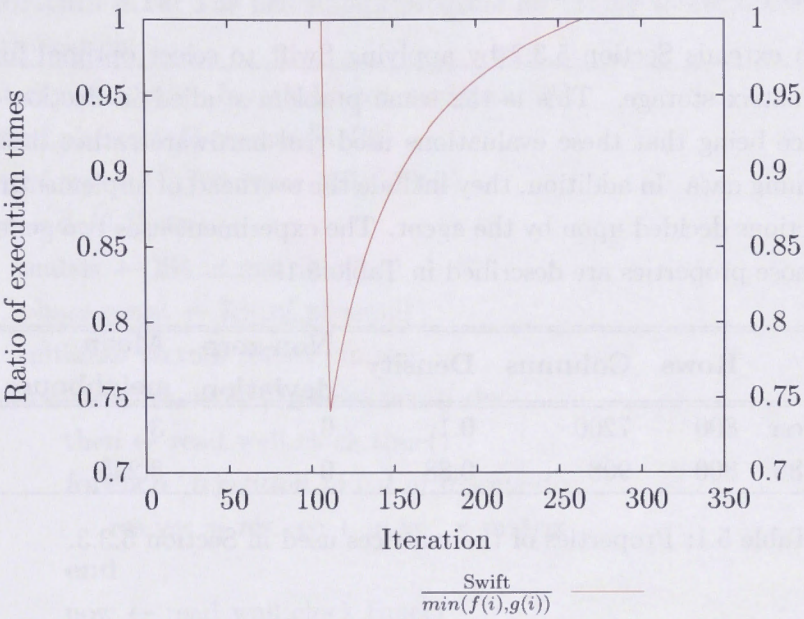


Figure 5.4: Ratio of Swift’s cumulative execution time to the smallest cumulative execution time incurred by either f or g .

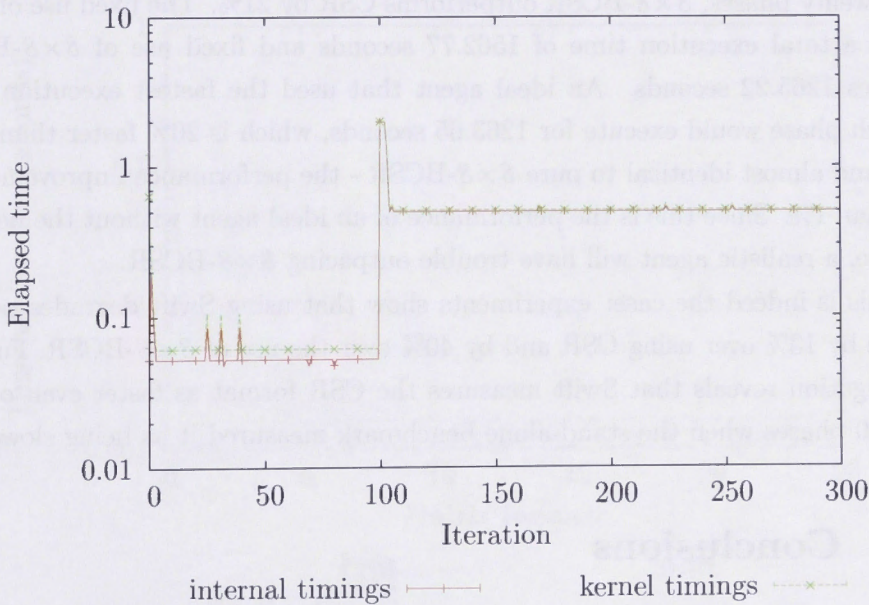


Figure 5.5: Overhead imposed by Swift while tuning the synthetic problem.

5.3.3 Sparse Matrix Format Selection

This section extends Section 5.3.2 by applying Swift to select optimal formats for sparse matrix storage. This is the same problem studied in Section 3.4.3, the difference being that these evaluations used real hardware rather than pre-gathered timing data. In addition, they include the overhead of implementing the format selections decided upon by the agent. The experiment uses two generated matrices whose properties are described in Table 5.1.

Matrix	Rows	Columns	Density	Non-zero deviation	Mean neighbours
synthetic.csr	800	7200	0.1	0	3
synthetic.88	800	908	0.88	0	8.245

Table 5.1: Properties of the matrices used in Section 5.3.3.

The matrices were processed by the program described in Algorithm 5.14. Figure 5.6 compares the performance of Algorithm 5.14 for two different realisations, each employing a different, fixed matrix storage format. For the first five phases, the CSR format is 73% quicker than the 8×8 -BCSR format. For the final twenty phases, 8×8 -BCSR outperforms CSR by 21%. The fixed use of CSR incurs a total execution time of 1562.77 seconds and fixed use of 8×8 -BCSR requires 1265.22 seconds. An ideal agent that used the fastest execution time on each phase would execute for 1263.65 seconds, which is 20% faster than pure CSR and almost identical to pure 8×8 -BCSR - the performance improvement is less than 1%. Since this is the performance of an ideal agent without the need to explore, a realistic agent will have trouble outpacing 8×8 -BCSR.

This is indeed the case: experiments show that using Swift degrades performance by 13% over using CSR and by 40% over the use of 8×8 -BCSR. Further investigation reveals that Swift measures the CSR format as faster even on the final 20 phases when the stand-alone benchmark measured it as being slower.

5.4 Conclusions

This chapter introduced the tuning driver, the third of three major components of Swift. The tuning driver was shown to be applicable to a range of target programs, with its behaviour driven by a configuration file. Elements of the

Algorithm 5.14: The benchmark program for tuning sparse matrix-vector multiplication.

```

1 list_of_matrices  $\leftarrow$  {synthetic.csr, synthetic.88}
2 list_of_phases  $\leftarrow$  {5, 20}
3 list_of_repeats  $\leftarrow$  {50, 50000}
4 for  $i \in [0, 1]$  do
5   matrix  $\leftarrow$  list_of_matrices[i]
6   phase_count  $\leftarrow$  list_of_phases[i]
7   initialise vectors res_vec, in_vec
8   foreach phase in [1, phase_count] do
9     then  $\leftarrow$  read_wall_clock_time()
10    foreach repetition in list_of_repeats do
11      res_vec = res_vec + in_vec  $\times$  matrix
12    end
13    now  $\leftarrow$  read_wall_clock_time()
14    report elapsed time (now - then)
15  end
16 end

```

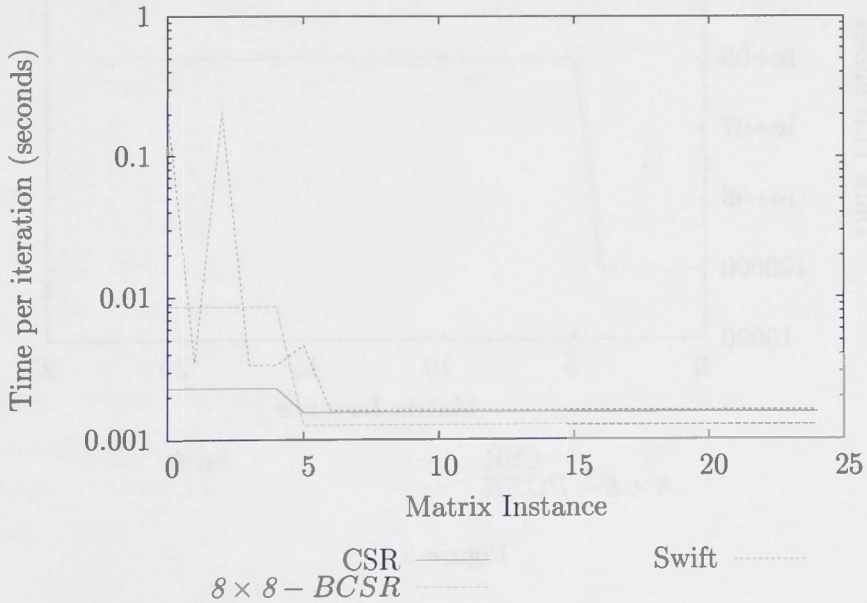


Figure 5.6: Cost per phase of Algorithm 5.14.

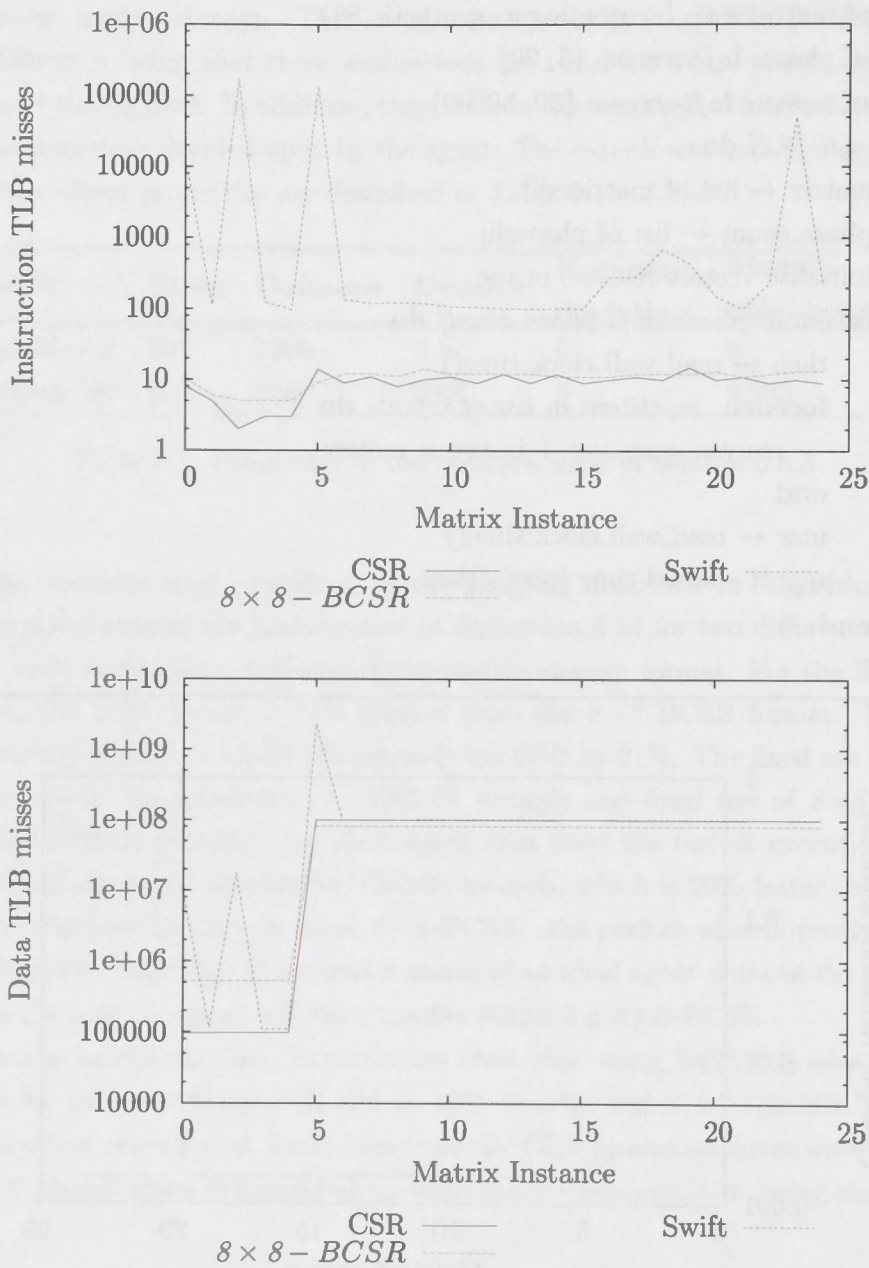


Figure 5.7

configuration file determined how the arguments of the tunable function could be derived from arguments of the original function. The derivation need not be direct - it can take place through a function provided by the user and called by the tuning driver when required. The discussion also covered the various supporting libraries required by Swift to facilitate the safe manipulation of the code and data of the target application.

The second part of this chapter presented an evaluation of Swift as applied to two different target applications. The first target examined a scenario where data analysis and action switching each took little time. Under these conditions, Swift achieved a speedup that varied over the lifetime of the benchmark.

The second target performed sparse matrix-vector multiplications, and Swift was unable to achieve a speedup when applied to this target. This result was disappointing, especially in the light of the positive results obtained in Section 3.4.3.1. One contributing factor was a change in hardware architecture - when gathering timing data for use in Section 3.4.3.1, preliminary analysis indicated that the largest scope for dynamic tuning was found on a 64-bit architecture - consequently, timings from such an architecture were used in the simulation of Section 3.4.3.1. Due to problems in DynInst, Swift is not able to execute on 64-bit architectures, and so the results in this chapter were gathered on a 32-bit platform.

Chapter 6

Conclusions and Future Work

This thesis has developed the Swift system for runtime tuning and used theory and experiment to evaluate its components. Swift was designed to use reinforcement-learning techniques to guide dynamic binary modification in order to speed up sparse matrix-vector multiplication kernels. It was shown in Chapter 3 that the learning algorithm was able to exploit performance differences to reduce execution time when compared to the best once-off format choice. The algorithm achieved performance gains of up to 20% over the best single matrix format. This figure varied with the nature of the tuning problem. On some problems, Swift was not able to outperform a single statically-selected format. The evaluation was conducted via simulation, which did not account for the overhead of Swift’s mechanisms for gathering profiling information and for implementing tuning actions.

Chapter 4 developed a suite of benchmarks to measure DynInst, the library used by Swift to carry out dynamic binary modification. Such benchmarks are useful for guiding the design of tools (like Swift) which rely on DynInst.

Chapter 5 provided detailed information about the tuning driver, the Swift component that links the decisions of the learning agent to the executive ability of DynInst. The chapter concluded with an evaluation of the complete Swift system in two non-simulated contexts. Swift was able to successfully tune one of these contexts, achieving a peak performance gain of 26%. However, Swift failed to correctly guide the application of the second target application where there was only a small potential for improved performance - Swift was at least 13% slower than the best statically-selected alternative format.

To summarise, combining a learning agent with the DynInst binary modification library creates a useful tool which has the potential to improve the per-

formance of a running application. The automated nature of the tuning process reduces the requirement for lengthy manual optimisation.

The remainder of this chapter discusses some of the issues that arose out of the development of Swift. Section 6.1 discusses some implications of isolating the learning agent in a dedicated process. Section 6.2 discusses some limitations of the learning algorithm employed.

6.1 The architecture of Swift

Swift was designed with the learning agent isolated in a separate process, communicating with the instrumentation over stream-oriented sockets. This separation was made because it provides potential for future expansion: a single learning agent can control and learn from many target applications on many different physical systems.

Swift has not yet been used to tune multiple systems at once, but it has suffered the costs of the split-process design. These costs occur in the communications between the learner and the instrumentation injected into the target application. Without the two-process design, this communication could occur via a procedure call or through a shared region of memory. Even with two processes, shared memory communication remains possible. However, the ultimate in flexibility is to operate across distinct physical systems. This is not possible using such a design, unless the physical systems support a system of distributed shared memory. This was considered overly restrictive and so socket-based communication was employed.

Appendix D contains details of a process by which these costs were quantified. It shows that the cost of communication is a major component of the time required to request actions of the agent.

6.2 The learning agent

The learning agent was shown to be successful in several evaluations, correctly determining the optimal action and correctly reassessing when a different action became optimal. However, there are several issues surrounding its design.

The first issue concerns the use of Gaussian distributions to estimate iteration counts. Section 3.2.6.1 explains how the number of iterations of tuned loops within the tuned application is modelled as a Gaussian distribution. This model

was designed to handle cases where the number of iterations is not constant - for example, when the loop in question is terminated by a convergence test, or is used to process arrays of differing sizes. While this model succeeds in capturing the range of iteration counts in those cases, any specific estimate drawn from the model is likely to be wrong.

The estimate is likely wrong because drawing a sample from the model's distribution and sampling the distribution of observed iteration counts are two independent events. The probability of the two samples returning the same value is thus the square of the probability of selecting that value. For Gaussian distributions this value is the mean, which occurs with probability $\frac{1}{\sigma\sqrt{2\pi}}$, where σ is the standard deviation of the distribution. Assuming $\sigma = 1$ gives a probability of 0.4. The chance of drawing this same value in two independent samplings is thus 0.16. In other words, 84 out of every 100 iteration count estimations will be either too high or too low. The effect of mispredicting the number of loop iterations depends on the direction of the error. A prediction that is less than the true iteration count will cause excessive communication between the target application and the learning agent as the agent is asked to select a format for the last few loop iterations. Over and above the cost of the communication comes the cost of data format conversion, should a change of format be directed.

The second issue is that the learning algorithm filters out the cost of converting between data formats. This filtering is required to obtain a set of measurements that all measure the same occurrence. The downside of this approach is that the learner bases its decisions on the cost of executing each action, and not on the costs of shaping the environment into a state where the action can be executed. This becomes problematic when tuning a target application that performs only a few executions with each action. When execution time is no longer dominant, the learner is likely to be misled into choosing quick actions that require extensive data conversion, resulting in an overall performance degradation.

6.3 Future Work

There are several directions in which the research program reported in this thesis could progress.

The framework can be applied to domains other than sparse matrix-vector multiplication. As discussed in Section 2.5.1 the notion of automatic performance tuning has been applied in a variety of guises to a range of different problems.

However, as noted in Section 2.5.1 the problems all bear a strong similarity to one another - they are mostly what can be termed *scientific* or *mathematical* problems. It may be profitable to branch out into more general applications. Of considerable interest are those applications where switching between tuning options is relatively cheap - recall from Figure 3.11 that data conversion costs required a beneficial tuning action to be applied for a long time to recoup data conversion costs. This situation is exacerbated in non-simulated environments when the cost of the tuning mechanism and the costs of data analysis must also be amortised.

This thesis has not investigated the manner in which architectural parameters influence the efficiency of different matrix storage formats. Such studies already exist in the literature, for example in the work by Nishtala *et al.* [59]. One promising avenue for future research is an analysis of the manner in which Swift's learning algorithm and tuning logic interferes with the execution of the tuned program when the two entities contend for resources such as a shared cache.

Another avenue for future research would compare the performance of Swift with some of the other automatic performance tuning tools discussed in Section 2.5.1. Such a comparison would very likely reveal that different classes of target program are best tuned by different manners of tuning algorithm. This opens up the possibility of using Swift for indirect tuning - a process wherein Swift tunes an scaffolding program that can use one of a set of tuning algorithms (including an instantiation of Swift) or applications to tune a target program.

This thesis has applied Swift to sparse matrix format selection and to a carefully designed synthetic benchmark. The application of Swift to other problem domains would be an interesting extension of this work. Such work could investigate those domains in which other automatic tuning systems have been successful *. Swift requires reasonably large optimisation opportunities in order to overcome its exploration costs, so the fact that another tuning system has proved effective for a problem does not assure that Swift will prove likewise. In contrast, Swift's ability to adapt applications to a dynamic environment may see fruitful application on the desktop. Desktop systems on modern multicore architectures host a wide range of applications that know nothing of each other but must run simultaneously, often contending for shared resources such as cache memory.

One of the failure modes of the learning algorithm occurs in situations where more than one action lies in the optimal set. Each action in the optimal set is

*See Section 2.5.1

associated with the maximum selection probability. This results in a high rate of exploration, as each element in the optimal set is repeatedly checked against every other element. This may appear unimportant, since by definition the performance of members of the optimal set is statistically indistinguishable. However, the cost of preparatory conversion actions required to switch between formats is not measured when determining the optimal set, although it is a component of the total running time. Preventing the algorithm from persistently alternating between indistinguishable formats would likely improve performance.

There are several paths along which the measurement of DynInst's performance can be extended. One such path is measurement on different architectures, although this would require DynInst to be repaired before it becomes feasible. Another extension is measurement of additional facilities provided by DynInst. Facilities of interest include DynInst's ability to execute *one-time codes*. One-time codes are chunks of user-supplied logic that DynInst causes to be executed by the target application. From the point of view of the DynInst-invoking code, execution is synchronous. One-time codes are used by Swift to implement conversion actions, and to ensure proper initialisation when first attaching to a target process. Also of interest is the manner in which trampoline insertion and removal at a single point scales with the number of insertions or removals. Section 4.3.2 reported measurements of the *first* trampoline insertion and removal at a particular instrumentation point. As pointed out in Section 4.2.2, multiple snippet insertions or removals at a single point require DynInst to parse and understand its earlier modifications to that point. The performance of Swift exhibits scaling behaviour that can be traced to this effect, and future work could more precisely quantify this scaling.

Appendix A

Performance Comparison of Binary Modification Tools

Runtime binary modification is a complex task, for which Swift employs a specialised third-party tool. Many such tools exist, most notably Pin [22], DynamoRIO [23], Valgrind [24], DynInst [25], Etch [26], Vulcan [27] and ATOM [28]. Swift is being developed for operation in a Linux environment on x86 and x86_64 architectures. These constraints rule out Etch and Vulcan (which require Microsoft Windows) and ATOM (which requires Alpha binaries).

The four remaining binary modification tools were benchmarked to assess the overhead they impose on the tuned application. Three benchmark applications were used: *csr*, *bcsr* and *rge*n. The first two applications repeatedly compute the product Ax of a vector x and a sparse matrix A . They possess the same basic structure (shown in Algorithm A.1), but use different in-memory arrangements of the matrix, and consequently employ different multiplication algorithms. Section 2.4 provides further details on different methods of storing and processing sparse matrices. The *rge*n benchmark depicted in Algorithm A.2 generates random numbers.

Each benchmark was executed several times while minimally modified by each binary modification tool. For Pin, each benchmark was executed using the command `pin -- benchmark <parameters>`. For DynamoRIO, benchmarks were invoked with the command `drrun benchmark <parameters>`. For Valgrind, the invocation used the command `valgrind --tool=none benchmark <parameters>`. For DynInst, the benchmarks were executed using a minimal wrapper program that performed two tasks:

- Locate the central function of each benchmark (the random number genera-

tion routine and the matrix-vector multiplication procedure), and make the changes required to place instrumentation at the beginning of that function.

- Launch and execute the benchmark

The goal of these invocation methods was to provide a minimal imposition of the tools onto the benchmarks. The overhead to perform useful work would be greater, but this assessment gives a useful point of comparison.

The tests were run on the Core2 machine described in Appendix C. The tools tested were version 2.10-41150-gcc.3.4.6-ia32_intel64-linux of Pin, DynamoRIO version 2.2.0-2, Valgrind versions 3.6.1 (on the Core2 machine) and DynInst versions 7.0.1 (on Core2). Table A.1 reports selected results from the *csr* and *bcsr* benchmarks which show that Pin performs the worst on every input matrix. The distinction between DynamoRIO and DynInst is less clear cut: for the *csr* benchmark, DynamoRIO is significantly faster on five out of ten matrices (*Hamrle1*, *can_24*, *cat_ears_2_1*, *dolphins* and *flower_4_1*), while DynInst is significantly faster on four out of ten (*GD01_b*, *cage4*, *ex5* and *impcol_b*) and neither significantly better for one matrix (*IG5-6*). When applied to the *bcsr* benchmark, DynamoRIO is significantly faster on just one matrix (*cage4*), while DynInst is significantly faster for the other nine. No results are shown for Valgrind, because it fails to handle some of the instructions within the benchmark.

The timing results from the *csr* and *bcsr* benchmarks measure only the core computational loop. A different picture emerges when considering the *rgen* benchmark, which was timed using the stand-alone utility `/usr/bin/time`. Consequently, the results include the start up phase of each binary modification tool. The results are presented in Figure A.1 for a variety of different input sizes - input size n corresponds to the generation of $n \times 10^5$ random numbers. The graphs show that DynInst suffers from larger start up costs, but for large enough inputs these are dominated by the main body of the benchmark, and under these circumstances, DynInst is the best of the evaluated tools.

Taken together, these results suggest that DynInst is a reasonable choice for use by Swift. These results do not consider factors such as ease of programming, the quality of documentation, the availability of source code or the extent and nature of each tool's community of users. This project is only concerned with speed of execution, and thus those additional factors have been ignored. DynInst, as the fastest evaluated tool, has been chosen as the technology on top of which

Swift will be constructed.

Algorithm A.1: The matrix-vector multiplication benchmarks *csr* and *bcsr*
 - the two variants have different implementations of $y \leftarrow y + Ax$.

Input: A: a matrix

```

1 A ← convert_matrix_to_desired_format(A)
2 y ← a vector full of zeros
3 x ← a vector full of ones
4 y ← y + Ax
5 start ← read_time()
6 for i ∈ [0,100] do
7   y ← y + Ax
8 end
9 end ← read_time()
10 runs_per_second ←  $\frac{100}{\text{end} - \text{start}}$ 
11 start ← read_time()
12 for i ∈ [0,runs_per_second] do
13   y ← y + Ax
14 end
15 end ← read_time()
16 return  $\frac{\text{end} - \text{start}}{\text{runs\_per\_second}}$ 
```

Algorithm A.2: The rgen benchmark. The operator $|$ denotes a bitwise-or operation.

Input: lim

```

1 x ← 0
2 for i ∈ [0,lim] do
3   for j ∈ [0,100000] do
4     y ← rand()
5     x ← x | y
6   end
7 end
8 return x
```

Matrix	Baseline	Pin	DynamoRIO	DynInst
<i>csr benchmark</i>				
GD01.b	$1.913 \times 10^{-7} \pm 8 \times 10^{-10}$	$2.48 \times 10^{-7} \pm 2 \times 10^{-9}$	$1.857 \times 10^{-7} \pm 3 \times 10^{-10}$	$1.84 \times 10^{-7} \pm 2 \times 10^{-9}$
Hamrle1	$4.28 \times 10^{-7} \pm 2.2 \times 10^{-9}$	$5.47 \times 10^{-7} \pm 3 \times 10^{-9}$	$4.18 \times 10^{-7} \pm 3 \times 10^{-9}$	$4.27 \times 10^{-7} \pm 1 \times 10^{-9}$
IG5-6	$9.06 \times 10^{-7} \pm 1 \times 10^{-9}$	$1.150 \times 10^{-6} \pm 2 \times 10^{-9}$	$9.041 \times 10^{-7} \pm 4 \times 10^{-10}$	$9.047 \times 10^{-7} \pm 5 \times 10^{-10}$
cage4	$1.701 \times 10^{-7} \pm 9 \times 10^{-10}$	$2.539 \times 10^{-7} \pm 1 \times 10^{-10}$	$1.734 \times 10^{-7} \pm 9 \times 10^{-10}$	$1.681 \times 10^{-7} \pm 5 \times 10^{-10}$
can_24	$5.51 \times 10^{-7} \pm 3 \times 10^{-9}$	$7.18 \times 10^{-7} \pm 5 \times 10^{-9}$	$5.50 \times 10^{-7} \pm 1 \times 10^{-9}$	$5.75 \times 10^{-7} \pm 3 \times 10^{-9}$
cat_ears_2_1	$1.451 \times 10^{-6} \pm 4 \times 10^{-9}$	$1.615 \times 10^{-6} \pm 9 \times 10^{-9}$	$1.243 \times 10^{-6} \pm 2 \times 10^{-9}$	$1.442 \times 10^{-6} \pm 2 \times 10^{-9}$
dolphins	$1.533 \times 10^{-6} \pm 3 \times 10^{-9}$	$1.894 \times 10^{-6} \pm 4 \times 10^{-9}$	$1.448 \times 10^{-6} \pm 3 \times 10^{-9}$	$1.538 \times 10^{-6} \pm 2 \times 10^{-9}$
ex5	$9.904 \times 10^{-7} \pm 5 \times 10^{-10}$	$1.2948 \times 10^{-6} \pm 6 \times 10^{-10}$	$1.0160 \times 10^{-6} \pm 8 \times 10^{-10}$	$9.905 \times 10^{-7} \pm 9 \times 10^{-10}$
flower_4_1	$1.942 \times 10^{-6} \pm 2 \times 10^{-9}$	$2.491 \times 10^{-6} \pm 9 \times 10^{-9}$	$1.888 \times 10^{-6} \pm 4 \times 10^{-9}$	$1.950 \times 10^{-6} \pm 4 \times 10^{-9}$
impcol.b	$9.62 \times 10^{-7} \pm 1 \times 10^{-9}$	$1.329 \times 10^{-6} \pm 3 \times 10^{-9}$	$9.70 \times 10^{-7} \pm 2 \times 10^{-9}$	$9.63 \times 10^{-7} \pm 2 \times 10^{-9}$
<i>bcsr benchmark</i>				
GD01.b	$3 \times 10^{-7} \pm 2 \times 10^{-7}$	$3.95 \times 10^{-7} \pm 2 \times 10^{-9}$	$3.22 \times 10^{-7} \pm 2 \times 10^{-9}$	$3.142 \times 10^{-7} \pm 8 \times 10^{-10}$
Hamrle1	$5 \times 10^{-7} \pm 5 \times 10^{-7}$	$6.93 \times 10^{-7} \pm 4 \times 10^{-9}$	$5.74 \times 10^{-7} \pm 4 \times 10^{-9}$	$5.36 \times 10^{-7} \pm 1 \times 10^{-9}$
IG5-6	$9 \times 10^{-7} \pm 1 \times 10^{-6}$	$1.080 \times 10^{-6} \pm 3 \times 10^{-9}$	$9.21 \times 10^{-7} \pm 3 \times 10^{-9}$	$8.99 \times 10^{-7} \pm 2 \times 10^{-9}$
cage4	$2 \times 10^{-7} \pm 3 \times 10^{-7}$	$2.29 \times 10^{-7} \pm 2 \times 10^{-9}$	$1.518 \times 10^{-7} \pm 3 \times 10^{-10}$	$1.64 \times 10^{-7} \pm 1 \times 10^{-9}$
can_24	$5 \times 10^{-7} \pm 7 \times 10^{-7}$	$5.92 \times 10^{-7} \pm 3 \times 10^{-9}$	$5.03 \times 10^{-7} \pm 3 \times 10^{-9}$	$4.90 \times 10^{-7} \pm 2 \times 10^{-9}$
cat_ears_2_1	$2 \times 10^{-6} \pm 2 \times 10^{-6}$	$2.5 \times 10^{-6} \pm 1 \times 10^{-7}$	$2.099 \times 10^{-6} \pm 5 \times 10^{-9}$	$2.059 \times 10^{-6} \pm 4 \times 10^{-9}$
dolphins	$2 \times 10^{-6} \pm 2 \times 10^{-6}$	$2.119 \times 10^{-6} \pm 5 \times 10^{-9}$	$1.908 \times 10^{-6} \pm 2 \times 10^{-9}$	$1.869 \times 10^{-6} \pm 5 \times 10^{-9}$
ex5	$6 \times 10^{-7} \pm 1 \times 10^{-6}$	$7.54 \times 10^{-7} \pm 3 \times 10^{-9}$	$6.29 \times 10^{-7} \pm 2 \times 10^{-9}$	$6.166 \times 10^{-7} \pm 8 \times 10^{-10}$
flower_4_1	$4 \times 10^{-6} \pm 2 \times 10^{-6}$	$4.54 \times 10^{-6} \pm 1 \times 10^{-8}$	$4.285 \times 10^{-6} \pm 6 \times 10^{-9}$	$4.235 \times 10^{-6} \pm 7 \times 10^{-9}$
impcol.b	$1 \times 10^{-6} \pm 1 \times 10^{-6}$	$1.28 \times 10^{-6} \pm 5 \times 10^{-8}$	$1.073 \times 10^{-6} \pm 4 \times 10^{-9}$	$1.057 \times 10^{-6} \pm 3 \times 10^{-9}$

Table A.1: Execution time (measured in seconds) for the Core2 platform.

Appendix B

Source code availability of
some programs

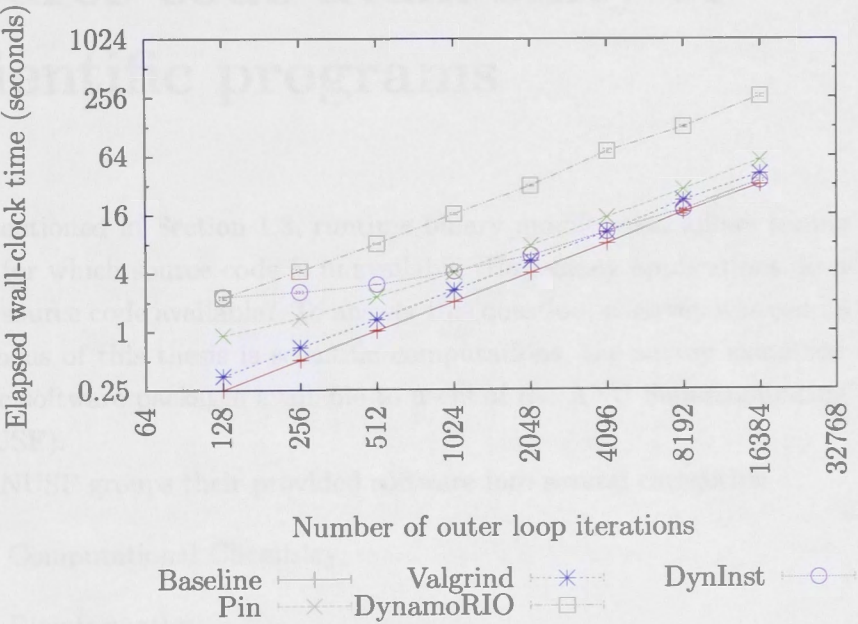


Figure A.1: Execution times for various binary modification tools on the *rgen* benchmark. Each point is the mean $\pm$ the standard error from five repetitions, but the error bars are too small to be easily perceived.



Appendix B

Source code availability of scientific programs

As mentioned in Section 1.3, runtime binary modification allows tuning of software for which source code is unavailable. How many applications do not make their source code available? To answer this question, a survey was conducted. As the focus of this thesis is scientific computations, the survey examined the scientific software packages available to users of the ANU Supercomputing Facility (ANUSF).

ANUSF groups their provided software into several categories:

1. Computational Chemistry,
2. Bioinformatics,
3. Mathematics libraries,
4. Statistics and mathematics environment,
5. CFD and Engineering,
6. Climate Modelling & Earth Systems,
7. Geophysics,
8. Geospatial tools,
9. Astronomy and Astrophysics,
10. Photonics,

11. Data mining,
12. Parallel Programming Libraries/Tools,
13. Compilers,
14. Debuggers & Profilers and Simulators,
15. Code Development Utilities,
16. Scripting Languages,
17. File Formats and Data management packages,
18. Graphics,
19. Network access and grid services

The last eight categories were excluded from the survey, as they do not contain scientific software. For every software package in the other categories, an attempt was made to determine if its source was available for download. The source was deemed available if any one of the following conditions was satisfied:

1. the software has an open-source license,
2. the software has a website which mentions downloading, building or installing from source code,
3. the availability of source code is known from personal experience

Note that neither cost nor restrictive license terms are sufficient to mark a package as having unavailable source code. Expensive or restricted source code will not serve all purposes, but framing the survey in this way minimises the number of packages without available source code. This survey is being used to support a claim that is stronger the more such packages there are, so biasing the survey in this manner increases the impact of its results.

[†]The website for these programs does not provide information, only marketing.

[‡]Unclear - commercial product

Computational Chemistry		
ABINIT	ACES III	ADF
Amber	CFOUR	CPMD
DALTON	DFTB-PLUS	ESPResSo
Gamess-PLUS	Gamess-US	Gaussian
GaussView *	GROMACS	LAMMPS
Molden	Molpro	MRCC
NAMD	Open Babel	Quantum ESPRESSO
SIESTA	VMD	WIEN2k †
Bioinformatics		
AbySS	BEAST	BLAST
bowtie	BWA	Cufflinks
HMMER	MAQ	mpiBLAST
MrBayes	PhyML	Velvet
wgsCABOG		
Mathematics Libraries		
ARPACK and PARPACK	BLACS	BLAS †
Boost	CPLEX	FFTW
GSL	LAPACK	LIBGOTO
Scalapack	Tao	
Statistics and Mathematics Environments		
Octave	PETSc	R
Climate Modelling & Earth Systems		
CCSM3	CCSM4	CDAT
CMOR2	ferret	GrADS
NCARG	NCL	wrf [§]
Geophysics		
Underworld		
Geospatial tools		
GDAL	GRASS	PROJ.4
Astronomy and Astrophysics		
cfitsio	DS9	IRAF
Data Mining		
lucene		

Table B.1: Software provided by ANUSF for which source code is know to be available

Computational Chemistry			
ORCA	Q-Chem	TURBOMOLE	
CASTEP¶	DMol3¶	GULP¶	NW-Chem
VASP¶			
Mathematics Libraries			
NAG			MKL
Statistics and Mathematics Environments			
Matlab	Maple	Stata	Mathematica
Photonics			
FDTD	FullWAVE		
CFD and Engineering			
ABAQUS¶	COMSOL¶	Fluent¶	LS-Dyna¶
Climate Modelling & Earth Systems			
Unified Model¶			

Table B.2: Software provided by ANUSF for which source code is not know to be available

Appendix C

Experimental Platforms

The experiments for this thesis were executed on several hardware platforms. This appendix provides the details of each system.

xeon-old Machine count 4

Processor 8×2.00 GHz Intel Xeon E5405 cores.

Address width 64 bits

Memory 8GB of RAM, 32 KB L1 instruction and data caches, 6MB unified L2 cache.

Operating System Three machines running Fedora 9, one machine running Fedora 14. All machines running a vanilla 2.6.37 kernel.

Compiler gcc 4.3.0 (three machines), gcc 4.5.1 (one machine)

xeon-new Machine count 2

Processor 8×2.40 GHz Intel Xeon E5620 cores.

Address width 64 bits

Memory 23GB of RAM, 32 KB L1 instruction and data caches, 256KB unified L2 cache, 12 MB unified L3 cache.

Operating System Fedora 14, running vanilla kernel 2.6.37

Compiler gcc 4.5.1

phenom Machine count 1

Address width 64 bits

Processor $4 \times$ AMD Phenom(tm) II X4 945 cores

Memory 3.8 GB of RAM, 64 KB L1 instruction and data caches, 512 KB unified L2 cache, 6MB unified L3 cache.

Operating System Ubuntu 8.04

Compiler gcc 4.2.4

core2 Machine count 1

Processor $2 \times 2.26\text{GHz}$ Intel Core2 Duo cores.

Address width 32 bits

Memory 3 GB of physical memory, 32 KB L1 data cache, 32 KB L1 instruction cache, 3 MB L2 Unified cache

Operating System Gentoo linux, running vanilla kernel 2.6.37

Compiler gcc 4.4.3

Pentium4 Machine count 1

Processor $1 \times 3\text{ GHz}$ Intel Pentium4 (hyperthreading disabled).

Address width 32 bits

Memory 1 GB of physical memory, 16 KB L1 data cache, 12K μop trace cache, 2 MB L2 Unified cache

Operating System Gentoo linux, running vanilla kernel 2.6.23

Compiler gcc 3.3.5

Opteron Machine count 1

Processor $1 \times 2.2\text{ GHz}$ AMD Opteron 848 core.

Address width 64 bits

Memory 2 GB of physical memory, 64 KB L1 data cache, 64KB L1 instruction cache, 1 MB L2 Unified cache

Operating System SUSE linux, running kernel 2.6.11.4-20a-default

Compiler gcc 3.3.5

Corei7 Machine count 1

Processor $8 \times 3.4\text{ GHz}$ Intel Core i7 cores

Address width 64 bits

Memory 8 GB of physical memory, 32 KB L1 data cache, 32 KB L1 instruction cache, 256 KB Unified L2 cache, 8 MB Unified L3 cache.

Operating System Gentoo Linux, running vanilla kernel 3.2.2
Compiler gcc 4.6.0

Appendix D

Communication costs between
the agent and the tuned
application

Both the used hardware and computer are 64-bit machines. The used agent - for this study, with exception to a few cases, written with Java - provides communication with the application. The data is transferred between different layers through some channels, which is not related with application itself. In this article, we have reported a general overview of the architecture of the agent.

The architecture of the agent generally is divided into three layers. The agent is divided into three layers: the agent core, the agent interface, and the agent application. The agent core is the core of the agent, which is responsible for the communication between the agent and the application. The agent interface is the interface between the agent and the application. The agent application is the application that is being tuned. The agent core is responsible for the communication between the agent and the application. The agent interface is responsible for the communication between the agent and the application. The agent application is the application that is being tuned. The agent core is responsible for the communication between the agent and the application. The agent interface is responsible for the communication between the agent and the application. The agent application is the application that is being tuned.

The agent core is responsible for the communication between the agent and the application. The agent interface is responsible for the communication between the agent and the application. The agent application is the application that is being tuned. The agent core is responsible for the communication between the agent and the application. The agent interface is responsible for the communication between the agent and the application. The agent application is the application that is being tuned.

Appendix D

Communication costs between the agent and the tuned application

Swift is based around two main components: the tuning driver and the learning agent. For this thesis, each component ran in a separate process, with inter-process communication occurring over sockets. This design means data transfer between components requires transmitting messages, which is more costly than using a procedure call. This appendix examines the impact of this communication cost on the performance of Swift.

The performance costs were gathered by modifying the tuning driver and the agent code to report timings around each communication event. The tuning driver was executed with the configuration from Table 3.2, altered by reducing μ_{iter} to 5,000 and the number of cycles to 1,000. Timing data for each type of inter-process message are summarised in Table D.1. The data have been processed by removing as outliers all values more than three standard deviations away from the mean and computing the mean and standard error. Each entry in Table D.1 is augmented by two numbers, tagged n and d . The former reports the number of data points remaining after outliers were discarded, and d reports the number of outliers discarded *. All standard errors in Table D.1 are at least two orders

*The reader may observe that the total $n + d$ is different for different combinations of socket type and measurement. This arises due to the process by which timers are calibrated - slightly different statistical distributions when setting the minimum acceptable elapsed time measurement can result in requiring more repeats per exploratory measurement, which in turn results in a different number of messages.

of magnitude smaller than the corresponding mean values. This allows the rest of this discussion to be simplified by examining only the mean values.

Dividing each *agent* time by the corresponding *driver* time shows the proportion of the driver time that is taken up by the computations within the agent process. For request messages the values are $\frac{9.261 \times 10^{-6}}{2.5874 \times 10^{-5}} \approx 0.36$ (UNIX) and $\frac{3.6996 \times 10^{-5}}{6.8670 \times 10^{-5}} \approx 0.54$ (TCP); for reports of execution time, the ratios are: $\frac{6.727 \times 10^{-6}}{6.247 \times 10^{-6}} \approx 1.07$ (UNIX) $\frac{4.1315 \times 10^{-6}}{2.1063 \times 10^{-5}} \approx 0.20$ (TCP); when reporting counter messages, the ratios are: $\frac{7.230 \times 10^{-6}}{3.769 \times 10^{-6}} \approx 1.12$ (UNIX) $\frac{2.1179 \times 10^{-5}}{1.9397 \times 10^{-5}} \approx 1.09$ (TCP). For the report messages, several of the ratios are greater than one. This is expected since report messages are unidirectional - there is no acknowledgement required from the agent. In these cases, the costs of using sockets to pass messages comes down to the cost of using the `write` system call. While this will be larger than using a procedure call to transfer the data, it allows the data to be processed by the agent in parallel with continued execution of the tuned application.

For request messages, however, the picture differs. These messages require a response from the driver, so not only must they suffer the cost of two `write` and two `read` calls, but they cannot execute in parallel with the learner, since they are blocked awaiting its reply. These processes only spend 36% (UNIX) and 54% (TCP) of their measured time waiting for the learning agent to calculate its reply. The rest of the time is consumed by the passing of messages.

Request messages

	Unix socket	TCP socket
Driver	$2.5874 \times 10^{-5} \pm 6.5 \times 10^{-8} n = 6,278 \ d = 2$	$6.8670 \times 10^{-5} \pm 5.9 \times 10^{-8} n = 13,430 \ d = 3$
Agent	$9.261 \times 10^{-6} \pm 7 \times 10^{-9} n = 12,524 \ d = 132$	$3.6996 \times 10^{-5} \pm 6.0 \times 10^{-8} n = 11,285 \ d = 3$

Execution time reports

	Unix socket	TCP socket
Driver	$6.247 \times 10^{-6} \pm 6 \times 10^{-9} n = 6,240 \ d = 40$	$2.1063 \times 10^{-5} \pm 3 \times 10^{-9} n = 13,249 \ d = 184$
Agent	$6.727 \times 10^{-6} \pm 1.1 \times 10^{-8} n = 12,592 \ d = 64$	$4.1315 \times 10^{-6} \pm 5 \times 10^{-9} n = 11,201 \ d = 87$

Report counter messages

	Unix socket	TCP socket
Driver	$3.769 \times 10^{-6} \pm 1.1 \times 10^{-8} n = 1,950 \ d = 50$	$1.9397 \times 10^{-5} \pm 1.5 \times 10^{-8} n = 1,964 \ d = 36$
Agent	$7.230 \times 10^{-6} \pm 2.2 \times 10^{-8} n = 1,993 \ d = 1$	$2.1179 \times 10^{-5} \pm 5.3 \times 10^{-8} n = 1,956 \ d = 43$

Table D.1: Mean and standard error of elapsed times - threshold is 3 standard deviations.

Table 1.1. Parameters of the tuned application and the agent

Tuned application	
WData	$1.55 \times 10^{-4} \pm 5.5 \times 10^{-6} = 1.55 \times 10^{-4}$
WData	$3.50 \times 10^{-4} \pm 1.1 \times 10^{-5} = 3.50 \times 10^{-4}$
Agent	
WData	$8.55 \times 10^{-4} \pm 1.1 \times 10^{-5} = 8.55 \times 10^{-4}$
WData	$3.1 \times 10^{-4} \pm 4.0 \times 10^{-6} = 3.1 \times 10^{-4}$
Tuned application	
WData	$6.76 \times 10^{-4} \pm 1.1 \times 10^{-5} = 6.76 \times 10^{-4}$
WData	$5.2 \times 10^{-4} \pm 1.1 \times 10^{-5} = 5.2 \times 10^{-4}$

Appendix E

Experimental Parameters

This appendix reports the detailed parameters used to obtain the results plotted in Figure 3.11. The parameters from Table 3.2 serve as a starting point. From those parameters, μ_{iter} varies between runs - the purpose of the experiment was to investigate the effect of different values of μ_{iter} . The other variance was in the random seeds fed to the simulation programs and the learning agent. These varying parameters are reported in Table E.1.

μ_{iter} /Repeat	Timing Seed	Sequence seed	Iteration seed	Agent seed
100/1	24901	14255	25328	12300
200/1	26325	27085	6234	30482
300/1	17074	29766	20879	12893
400/1	1919	18911	23686	460
500/1	3560	5323	18921	5441
600/1	640	19261	16207	1966
700/1	13248	11860	9271	14014
800/1	7548	18520	12158	11708
900/1	6300	19126	30502	2722
1000/1	13599	7986	3031	24777
1100/1	24819	10296	3409	25565
1200/1	20095	31821	11037	32530
1300/1	1109	3590	21591	11275
1400/1	12195	1319	22967	8268
1500/1	26589	29807	20460	8444
1600/1	1513	7521	28818	6129

Continued overleaf

μ_{iter}/Repeat	Timing Seed	Sequence seed	Iteration seed	Agent seed
<i>Continued</i>				
1700/1	32720	15272	16339	18694
1800/1	18810	1657	32693	21828
1900/1	5843	30477	4667	29213
100/2	748	25433	5821	24560
200/2	17915	602	29839	26869
300/2	18285	32339	12577	5092
400/2	222	11129	10299	21876
500/2	17401	6557	18124	3548
600/2	28768	21441	12503	5707
700/2	7390	26237	18431	25844
800/2	3379	14101	30932	17022
900/2	6849	6825	1264	21133
1000/2	10692	11181	32749	18552
1100/2	30429	13496	20800	25075
1200/2	7350	32248	20206	8649
1300/2	13092	6482	24186	11251
1400/2	6318	32330	24358	19209
1500/2	22895	12306	5537	13638
1600/2	13559	20395	474	19142
1700/2	12276	20455	30470	12033
1800/2	8904	6382	20622	10186
1900/2	27680	23897	13193	2029
100/3	28742	14022	4927	9560
200/3	22138	27028	30041	10754
300/3	10122	2962	21092	22235
400/3	30427	12867	28286	18053
500/3	19450	9998	7166	25713
600/3	28969	15229	15936	600
700/3	5142	28263	17076	14348
800/3	14144	19815	21538	11346
900/3	24567	22165	21773	30242
1000/3	13803	27807	18872	1815
1100/3	12027	27739	22341	31459
1200/3	2642	8627	7910	10693

Continued overleaf

μ_{iter}/Repeat	Timing Seed	Sequence seed	Iteration seed	Agent seed
----------------------------	-------------	---------------	----------------	------------

Continued

1300/3	30100	26383	2687	15275
1400/3	23479	22671	11162	14162
1500/3	1264	25591	32087	25694
1600/3	5679	6286	10325	7716
1700/3	623	21428	30330	25698
1800/3	4943	25085	13143	18837
1900/3	1516	19800	593	9316
100/4	23881	5321	18204	4768
200/4	28738	5945	17814	11143
300/4	24305	10387	28571	21023
400/4	32132	27855	12938	7938
500/4	21401	3268	9327	30251
600/4	14748	19199	17410	5122
700/4	15997	9524	8093	5333
800/4	21807	3875	28993	27382
900/4	17455	4893	29788	28019
1000/4	13907	5541	14760	32072
1100/4	2260	8981	16713	22422
1200/4	18186	26372	14668	16994
1300/4	26927	12176	15911	29388
1400/4	19149	4465	18092	26382
1500/4	19356	30095	2661	411
1600/4	26600	27242	23551	22631
1700/4	25845	18619	2077	25895
1800/4	26331	29560	28113	21787
1900/4	5922	16883	15288	20001
100/5	28970	1452	28166	20259
200/5	3938	32022	20937	31426
300/5	25637	24101	26949	13553
400/5	19639	1461	22019	1534
500/5	5119	25687	16068	28718
600/5	1684	24219	11367	13176
700/5	18875	8784	16044	3708
800/5	7421	18995	23323	19377

Continued overleaf

$\mu_{iter}/Repeat$	Timing Seed	Sequence seed	Iteration seed	Agent seed
---------------------	-------------	---------------	----------------	------------

Continued

900/5	30636	29751	29797	17515
1000/5	1696	12835	16128	9991
1100/5	24153	22069	27330	3899
1200/5	27843	4864	5611	5825
1300/5	26142	18492	32717	5044
1400/5	13081	26365	4178	13979
1500/5	5779	11599	14265	26603
1600/5	8856	15310	24627	14749
1700/5	31322	19190	4296	26124
1800/5	8634	29020	1830	31404
1900/5	26803	22523	24439	12564
100/6	17798	6615	3063	13498
200/6	24340	11767	26326	28214
300/6	19399	10830	32054	534
400/6	7357	21520	1282	20232
500/6	18364	18076	17232	16383
600/6	8187	20252	23976	25267
700/6	834	5697	15649	26970
800/6	17252	5976	15248	6961
900/6	25002	3035	29257	17913
1000/6	29532	15394	7455	24756
1100/6	28082	19251	3309	7886
1200/6	27860	26886	17634	30941
1300/6	9405	1020	8440	8855
1400/6	6580	30873	4382	19270
1500/6	4042	10567	9975	11495
1600/6	9089	7874	25581	27716
1700/6	31623	27592	14498	16030
1800/6	30623	1043	5779	9809
1900/6	9882	22502	29240	24638
100/7	13645	2578	15733	3170
200/7	31939	31219	24157	24495
300/7	24331	26089	16732	2119
400/7	409	4659	32255	4908

Continued overleaf

$\mu_{iter}/Repeat$	Timing Seed	Sequence seed	Iteration seed	Agent seed
<i>Continued</i>				
500/7	13761	11632	9701	3743
600/7	27542	30156	10377	31267
700/7	15656	3590	21749	16479
800/7	19026	897	13706	11473
900/7	27458	25204	24585	31562
1000/7	21855	23246	19172	16975
1100/7	32255	9538	19588	10495
1200/7	4636	9575	19220	13268
1300/7	18903	308	7451	215
1400/7	25073	8001	27828	12587
1500/7	7034	9652	27660	3460
1600/7	23611	16628	23653	28202
1700/7	15626	24693	24160	29528
1800/7	13517	677	17697	31308
1900/7	19548	16994	21090	14418
100/8	5007	18389	8824	13311
200/8	25219	5416	4102	12236
300/8	6240	18890	6866	30656
400/8	28785	6047	24825	11462
500/8	4580	12913	13281	8795
600/8	5262	32763	18878	2926
700/8	8312	24932	10575	8034
800/8	1544	6647	16601	30692
900/8	16026	4996	28753	27672
1000/8	21768	6606	22282	27303
1100/8	8384	20474	16589	1871
1200/8	32401	5419	28939	17135
1300/8	442	910	30119	25131
1400/8	2857	14946	32623	91
1500/8	24901	7176	22030	29178
1600/8	25117	25787	26933	8637
1700/8	15092	30511	14411	26824
1800/8	18665	25517	11788	6737
1900/8	24078	2743	2261	32015
<i>Continued overleaf</i>				

$\mu_{iter}/Repeat$	Timing Seed	Sequence seed	Iteration seed	Agent seed
<i>Continued</i>				
100/9	27511	29081	4387	11789
200/9	23097	30308	13567	24146
300/9	26129	29588	153	31155
400/9	23441	18638	22051	17781
500/9	16685	8390	12485	5542
600/9	19118	26403	26325	24242
700/9	7386	26242	27624	23473
800/9	22727	31160	17106	8380
900/9	19228	9077	29412	24332
1000/9	12457	19572	26955	18613
1100/9	5502	9205	17201	31044
1200/9	3086	6900	8920	21344
1300/9	29245	1450	2472	11390
1400/9	4076	31271	20657	14636
1500/9	4980	25003	22168	17453
1600/9	32536	13489	20810	3785
1700/9	20093	30045	27998	24047
1800/9	5057	869	31172	21938
1900/9	22581	6910	7906	14417
100/10	760	1060	30604	7954
200/10	2577	25856	29269	23487
300/10	28621	1969	2182	19449
400/10	19220	11133	12886	27508
500/10	16593	6113	23297	18045
600/10	20369	18424	31510	8397
700/10	6009	15994	28612	12644
800/10	14353	28537	12509	14250
900/10	6843	29914	11236	6591
1000/10	23783	25229	12699	26835
1100/10	3746	27176	1308	7535
1200/10	6830	7294	19008	17941
1300/10	8095	809	10378	6528
1400/10	13099	2942	5399	10290
1500/10	10385	26415	31363	13219

Continued overleaf

$\mu_{iter}/Repeat$	Timing Seed	Sequence seed	Iteration seed	Agent seed
<i>Continued</i>				
1600/10	10937	22453	19635	5384
1700/10	32160	12431	12430	17061
1800/10	32236	16432	5272	4463
1900/10	8127	26051	27245	16567

Bibliography

- [1] R. Clint Whaley, Antoine Petitet, and J.J. Dongarra. Automated empirical optimizations of software and the ATLAS project 1. *Parallel Computing*, 27(1-2):3–35, January 2001.
- [2] R.C. Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, 2005.
- [3] M. Frigo and S.G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.
- [4] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, and Others. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.
- [5] Eun-Jin Im, Katherine A. Yelick, and Richard Vuduc. Sparsity: Optimization Framework for Sparse Matrix Kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.
- [6] Richard Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. Phd, University of California, Berkeley, Berkeley, 2003.
- [7] Richard Vuduc, J.W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.
- [8] A. Buttari, J. Dongarra, J. Langou, P. Luszczek, and J. Kurzak. Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems. *International Journal of High Performance Computing Applications*, 21(4):457–466, November 2007.

-
- [9] D.E. Shaw, R.O. Dror, J.K. Salmon, JP Grossman, K.M. Mackenzie, J.A. Bank, Cliff Young, M.M. Deneroff, Brannon Batson, K.J. Bowers, and Others. Millisecond-scale molecular dynamics simulations on Anton. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, Portland, Oregon, 2009. ACM.
 - [10] Ruhong Zhou. Free energy landscape of protein folding in water: explicit vs. implicit solvent. *Proteins: Structure, Function, and Bioinformatics*, 53(2):148–161, 2003.
 - [11] Tsutomu Ikegami, Toyokazu Ishida, Dmitri G. Fedorov, Kazuo Kitaura, Yuichi Inadomi, Hiroaki Umeda, Mitsuo Yokokawa, and Satoshi Sekiguchi. Full Electron Calculation Beyond 20,000 Atoms: Ground Electronic State of Photosynthetic Proteins. In *ACM/IEEE Conference on Supercomputing*, page 10. IEEE Computer Society, 2005.
 - [12] CM Cantalupo, JD Borrill, AH Jaffe, TS Kisner, and R. Stompor. MADmap: A Massively Parallel Maximum Likelihood Cosmic Microwave Background Map-maker. *The Astrophysical Journal Supplement Series*, 187(1):212, 2010.
 - [13] M.B. Araújo and Mark New. Ensemble forecasting of species distributions. *Trends in Ecology & Evolution*, 22(1):42–47, January 2007.
 - [14] N.G. Bronson, Christos Kozyrakis, and Kunle Olukotun. Feedback-directed barrier optimization in a strongly isolated STM. In *Principles of Programming Languages (PoPL)*, volume 44, pages 213–225, Savannah, GA, USA, 2009. ACM.
 - [15] John Cuthbertson, Sandhya Viswanathan, Konstantin Bobrovsky, Alexander Astapchuk, E.K. Srinivasan, and Others. A practical approach to hardware performance monitoring based dynamic optimizations in a production JVM. In *Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 190–199. IEEE Computer Society, 2009.
 - [16] F.T. Schneider, Vijay Menon, Tatiana Shpeisman, and A.R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *Object-oriented programming, systems, languages, and applications (OOPSLA)*, volume 43, pages 181–194. ACM, 2008.

- [17] Christian Wimmer and Hanspeter Mössenböck. Automatic feedback-directed object inlining in the java hotspot virtual machine. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 12–21, San Diego, California, USA, 2007. ACM.
- [18] Xipeng Shen, M.L. Scott, Chengliang Zhang, Sandhya Dwarkadas, Chen Ding, and Mitsunori Ogihara. Analysis of input-dependent program behavior using active profiling. In *Proceedings of the 2007 workshop on Experimental computer science*, page 5, San Diego, California, 2007. ACM.
- [19] C.B. Cho and Tao Li. Complexity-based program phase analysis and classification. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 105–113, Seattle, Washington, USA, 2006. ACM.
- [20] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, November 2003.
- [21] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.
- [22] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Programming language design and implementation (PLDI)*, volume 40, page 190, June 2005.
- [23] Derek L./Supervisor-Amarasinghe Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, MIT, January 2004.
- [24] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI '07*, volume 42, page 89, New York, New York, USA, June 2007. ACM Press.
- [25] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

-
- [26] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of Win32/intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, page 8, August 1997.
 - [27] Andrew Edwards, Amitabh Srivastava, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. Technical report, Microsoft Research, 2001.
 - [28] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. In *Programming language design and implementation (PLDI)*, volume 29, pages 196–205, June 1994.
 - [29] D. Dubois, Andrew Dubois, Thomas Boorman, Carolyn Connor, and Steve Poole. Sparse Matrix-Vector Multiplication on a Reconfigurable Supercomputer with Application. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3(1):1–31, 2010.
 - [30] G.H. Golub and D.P. O’Leary. Some history of the conjugate gradient and Lanczos algorithms: 1948-1976. *SIAM review*, 31(1):50–102, 1989.
 - [31] M.R. Field. Optimizing a parallel conjugate gradient solver. *SIAM Journal on Scientific Computing*, 19(1):27, 1998.
 - [32] K.B. Theobald, Gagan Agrawal, Rishi Kumar, Gerd Heber, G.R. Gao, Paul Stodghill, and Keshav Pingali. Landing CG on EARTH: A case study of fine-grained multithreading on an evolutionary path. In *ACM/IEEE 2000 Conference Supercomputing*, pages 4–4, Dallas, Texas, United States, 2000. IEEE Computer Society.
 - [33] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.
 - [34] Warren Armstrong, Peter Christen, Eric McCreath, and A.P. Rendell. Dynamic algorithm selection using reinforcement learning. In *Integrating AI and Data Mining, 2006. AIDM’06. International Workshop on*, pages 18–25. IEEE, 2006.

- [35] Warren Armstrong and Alistair P. Rendell. Reinforcement learning for automated performance tuning: Initial evaluation for sparse matrix format selection. In *2008 IEEE International Conference on Cluster Computing*, pages 411–420. IEEE, September 2008.
- [36] Warren Armstrong and A.P. Rendell. Runtime sparse matrix format selection. In *International Conference on Computational Science*, volume 1, pages 135–144. Elsevier, May 2010.
- [37] Reiji Suda. A Bayesian Method for Online Code Selection: Toward Efficient and Robust Methods of Automatic Tuning. In *Proc. 2nd Intl Workshop on Automatic Performance Tuning (iWAPT2007)*, pages 23–32, Tokyo, 2007.
- [38] T. Kajiyama, Akira Nukada, Reiji Suda, Hidehiko Hasegawa, and Akira Nishida. Toward Automatic Performance Tuning for Numerical Simulations in the SILC Matrix Computation Framework. In *International Workshop on Automatic Performance Tuning (IWAPT)*. Springer-Verlag, 2007.
- [39] Lakshminarasimhan Seshagiri, Masha Sosonkina, and Zhao Zhang. Electronic Structure Calculations and Adaptation Scheme in Multi-core Computing Environments. In *Computational Science ICCS 2009*, pages 3–12. Springer, 2009.
- [40] Seyong Lee and Rudolf Eigenmann. Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 195–204, New York, New York, USA, 2008. ACM.
- [41] John R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 15:65–118, 1976.
- [42] Kate A. Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys*, 41(1):1–25, 2008.
- [43] Michail G. Lagoudakis and Michael L. Littman. Algorithm Selection using Reinforcement Learning. In *Proceedings of the Seventeenth International Conference on Machine Learning, ICML '00*, pages 511–518, San Francisco, 2000. Morgan Kaufmann Publishers Inc.

-
- [44] D Koulouriotis and a Xanthopoulos. Reinforcement learning and evolutionary algorithms for non-stationary multi-armed bandit problems. *Applied Mathematics and Computation*, 196(2):913–922, March 2008.
 - [45] Shie Mannor and John N. Tsitsiklis. The sample complexity of exploration in the multi-armed bandit problem. *The Journal of Machine Learning Research*, 5:623–648, 2004.
 - [46] R.S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in neural information processing systems*, pages 1038–1044, 1996.
 - [47] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. *2008 International Symposium on Computer Architecture*, (2):39–50, June 2008.
 - [48] Shivaram Kalyanakrishnan and Peter Stone. Characterizing reinforcement learning methods through parameterized learning problems. *Machine Learning*, 84(1):205–247, 2011.
 - [49] Steven D. Whitehead and Dana H. Ballard. Learning to perceive and act by trial and error. *Machine Learning*, 7(1):45–83, July 1991.
 - [50] Dyninst developers: Personal communication, September 2009.
 - [51] J.M. Millam and G.E. Scuseria. Linear scaling conjugate gradient density matrix search as an alternative to diagonalization for first principles electronic structure calculations. *The Journal of Chemical Physics*, 106(13):5569, 1997.
 - [52] Malik Silva. Sparse matrix storage revisited. In *Proceedings of the 2nd conference on Computing frontiers*, pages 230–235, Ischia, Italy, 2005. ACM.
 - [53] Ayaz Ali, Lennart Johnsson, and Jaspal Subhlok. Adaptive computation of self sorting in-place ffts on hierarchical memory architectures. In *High Performance Computing and Communications*, pages 372–383. Springer, 2007.
 - [54] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng

- Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 63–76, San Diego, California, USA, 2003. ACM.
- [55] Keith D. Cooper and Todd Waterman. Investigating Adaptive Compilation Using the Mipspro Compiler. *International Journal of High Performance Computing Applications*, 19(4):423–431, 2005.
- [56] Mark Stephenson, Saman Amarasinghe, Martin Martin, and U.M. O’Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Programming Language Design and Implementation (PLDI)*, volume 38, pages 77–90. ACM, 2003.
- [57] Eliot Moss, Paul Utgoff, John Cavazos, Doina Precup, Darko Stefanovic, Carla Brodley, and David Scheeff. Learning to schedule straight-line code. In *NIPS*, 1997.
- [58] Jeff Bilmes, Krste Asanovic, C.W. Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, pages 340–347, Vienna, Austria, 1997. ACM.
- [59] Rajesh Nishtala, Richard Vuduc, J.W. Demmel, and Katherine A. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):297–311, 2007.
- [60] S. Williams, K. Datta, J. Carter, L. Oliker, J. Shalf, Katherine A. Yelick, and D. Bailey. PERI-Auto-tuning memory-intensive kernels for multicore. In *Journal of Physics: Conference Series*, volume 125, page 012038. IOP Publishing, 2008.
- [61] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.
- [62] Masha Sosonkina. Adapting distributed scientific applications to run-time network conditions. In *Applied Parallel Computing*, pages 747–755. Springer, 2006.

-
- [63] N. Ustemirov, M. Sosonkina, M.S. Gordon, and M.W. Schmidt. Dynamic algorithm selection in parallel GAMESS calculations. In *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, pages 6–496. IEEE, 2006.
- [64] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPOPP '05*, pages 277 – 288, New York, New York, USA, 2005. ACM Press.
- [65] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the international symposium on Code generation and optimization*, pages 123–134. IEEE Computer Society, 2005.
- [66] Richard Vuduc, Shoaib Kamil, Jen Hsu, Rajesh Nishtala, J.W. Demmel, and Katherine A. Yelick. Automatic performance tuning and analysis of sparse triangular solve. In *ICS 2002: Workshop on Performance Optimization via High-Level Languages and Libraries*, New York, NY, USA, 2002. Citeseer.
- [67] Takahiro Katagiri, Kenji Kise, Hiroaki Honda, and Toshitsugu Yuba. FIBER: A generalized framework for auto-tuning software. In *High Performance Computing*, pages 146–159. Springer, 2003.
- [68] Serban Georgescu and Hiroshi Okuda. GPGPU-Enhanced Conjugate Gradient Solver for Finite Element Matrices. In *International Workshop on Automatic Performance Tuning (IWAPT)*, pages 71–80, 2007.
- [69] Lakshminarasimhan Seshagiri, M.S. Wu, M. Sosonkina, and Z. Zhang. Exploring Tuning Strategies for Quantum Chemistry Computations. In *International Workshop on Automatic Performance Tuning (IWAPT)*, 2009.
- [70] Haruki Imai and Yasushi Negishi. Automatic Parameter Search for Scientific Programs Using Performance Prediction. In *International Workshop on Automatic Performance Tuning (IWAPT)*, pages 13–22, University of Tokyo, 2007.
- [71] Y. Sawa and Reiji Suda. Auto Tuning Method for Deciding Block Size Parameters in Dynamically Load-balanced BLAS. *International Workshop on Automatic Performance Tuning (IWAPT)*, 2009.

- [72] Jaewook Shin, M.W. Hall, Jacqueline Chame, Chuan Chen, and P.D. Hovland. Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology. In *The Fourth International Workshop on Automatic Performance Tuning*. Citeseer, 2009.
- [73] Kaushik Datta, Samuel Williams, Vassily Volkov, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine A. Yelick. Auto-tuning the 27-point stencil for multicore. In *Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning*. Citeseer, 2009.
- [74] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael OBoyle. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming*, pages 1–32–32, January 2011.
- [75] Xianglong Huang, S.M. Blackburn, K.S. McKinley, J.E.B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: improving program locality. In *Object-oriented programming, systems, languages, and applications (OOPSLA)*, volume 39, pages 69–80, New York, New York, USA, 2004. ACM.
- [76] Markus Nordén, Henrik Löf, Jarmo Rantakokko, and Sverker Holmgren. Dynamic Data Migration for Structured AMR Solvers. *International Journal of Parallel Programming*, 35(5):477–491, September 2007.
- [77] Jing Xu. Rule-based automatic software performance diagnosis and improvement. *Performance Evaluation*, 67(8):585–611, August 2009.
- [78] S. Storie and M. Sosonkina. Packet probing as network load detection for scientific applications at run-time. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 62. IEEE, 2004.
- [79] Masha Sosonkina and Gan Chen. Design of a tool for providing dynamic network information to an application. *Parallel Computing Technologies*, 2127:350–357, 2001.
- [80] B. R. Buck. A New Hardware Monitor Design to Measure Data Structure-Specific Cache Eviction Information. *International Journal of High Performance Computing Applications*, 20(3):353–363, August 2006.

-
- [81] J. Odom, Jeffrey K. Hollingsworth, L. DeRose, K. Ekanadham, and S. Sbaraglia. Using Dynamic Tracing Sampling to Measure Long Running Programs. In *ACM/IEEE SC 2005 Conference (SC'05)*, SC '05, pages 59–59, Washington, 2005. IEEE.
 - [82] B.R. Buck and Jeffrey K. Hollingsworth. Data Centric Cache Measurement on the Intel Itanium 2 Processor. *Proceedings of the ACM/IEEE SC2004 Conference*, 00(c):58–58, 2004.
 - [83] Jaydeep Marathe, Frank Mueller, Tushar Mohan, Bronis R. de Supinski, Sally A. McKee, and Andy Yoo. METRIC: tracking down inefficiencies in the memory hierarchy via binary rewriting. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 289–300, March 2003.
 - [84] Nick Rutar and Jeffrey K. Hollingsworth. Software Analysis Techniques to Approximate Data Centric Direct Measurements. In *First International Workshop on High-performance Infrastructure for Scalable Tools*, 2011.
 - [85] Jaydeep Marathe, Frank Mueller, Tushar Mohan, Sally A. McKee, Bronis R. De Supinski, and Andy Yoo. Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Transactions on Programming Languages and Systems*, 29(2):12–es, April 2007.
 - [86] Mustafa M. Tikir and Jeffrey K. Hollingsworth. NUMA-Aware Java Heaps for Server Applications. *19th IEEE International Parallel and Distributed Processing Symposium*, pages 108b–108b, 2005.
 - [87] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Using Hardware Counters to Automatically Improve Memory Performance. *Proceedings of the ACM/IEEE SC2004 Conference*, 00(c):46–46, 2004.
 - [88] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Hardware monitors for dynamic page migration. *Journal of Parallel and Distributed Computing*, 68(9):1186–1200, September 2008.
 - [89] Tushar Mohan, Bronis R De Supinski, Sally A McKee, Frank Mueller, Andy Yoo, and Martin Schulz. Identifying and Exploiting Spatial Regularity in Data Memory References. *Computer Engineering*, 2003.

- [90] P. Unnikrishnan, G. Chen, M. Kandemir, and D.R. Mudgett. Dynamic compilation for energy adaptation. *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002.*, pages 158–163, 2002.
- [91] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient online computation of statement coverage. *Journal of Systems and Software*, 78(2):146–165, November 2005.
- [92] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. *ACM SIGSOFT Software Engineering Notes*, 27(4):86, July 2002.
- [93] Martin Schulz, John May, and John Gyllenhaal. DynTG : A Tool for Interactive , Dynamic Instrumentation. *Processing*, pages 140–148, 2005.
- [94] L. DeRose, T. Hoover, and Jeffrey K. Hollingsworth. *The dynamic probe class library-an infrastructure for developing instrumentation for performance tools*. IEEE Comput. Soc, 2001.
- [95] Tugrul Ince and J. Hollingsworth. Profile-driven selective program loading. In *Euro-Par 2010-Parallel Processing*, pages 62–73. Springer, 2010.
- [96] Charles Zhang and Hans-arno Jacobsen. TinyC 2 : Towards building a dynamic weaving aspect language for C. *Language*, 2003.
- [97] David S. Moore, George P. McCabe, and Bruce A. Craig. *Introduction to the Practice of Statistics*. Sixth edition, 2009.
- [98] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical algorithms*. Third edition, 1998.
- [99] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. In *ACM Transactions on Mathematical Software*, 2011.
- [100] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, October 2000.
- [101] S. Browne. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, August 2000.