



University of Warwick institutional repository: <http://go.warwick.ac.uk/wrap>

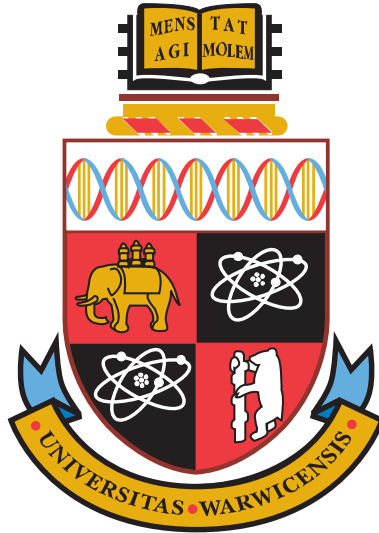
A Thesis Submitted for the Degree of PhD at the University of Warwick

<http://go.warwick.ac.uk/wrap/3713>

This thesis is made available online and is protected by original copyright.

Please scroll down to view the document itself.

Please refer to the repository record for this item for information to help you to cite it. Our policy information is available from the repository home page.



**Addressing Concerns in Performance Prediction:
The Impact of Data Dependencies and Denormal
Arithmetic in Scientific Codes**

by

Brian Patrick Foley

A thesis submitted to the University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

September 2009

THE UNIVERSITY OF
WARWICK

Abstract

To meet the increasing computational requirements of the scientific community, the use of parallel programming has become commonplace, and in recent years distributed applications running on clusters of computers have become the norm.

Both parallel and distributed applications face the problem of predictive uncertainty and variations in runtime. Modern scientific applications have varying I/O, cache, and memory profiles that have significant and difficult to predict effects on their runtimes. Data-dependent sensitivities such as the costs of denormal floating point calculations introduce more variations in runtime, further hindering predictability.

Applications with unpredictable performance or which have highly variable runtimes can cause several problems. If the runtime of an application is unknown or varies widely, workflow schedulers cannot efficiently allocate them to compute nodes, leading to the under-utilisation of expensive resources. Similarly, a lack of accurate knowledge of the performance of an application on new hardware can lead to misguided procurement decisions. In heavily parallel applications, minor variations in runtime on individual nodes can have disproportionate effects on the overall application runtime. Even on a smaller scale, a lack of certainty about an application's runtime can preclude its use in real-time or time-critical applications such as clinical diagnosis.

This thesis investigates two sources of data-dependent performance variability. The first source is *algorithmic* and is seen in a state-of-the-art C++ biomedical imaging application. It identifies the cause of the variability in the application and develops a means of characterising the variability. This 'probe task' based model is adapted for use with a workflow scheduler, and the scheduling improvements it brings are examined.

The second source of variability is more subtle as it is *micro-architectural* in nature. Depending on the input data, two runs of an application executing exactly the same sequence of instructions and with exactly the same memory access patterns can have large differences in runtime due to deficiencies in common hardware implementations of denormal arithmetic¹. An exception-based profiler is written to detect occurrences of denormal arithmetic and it is shown how this is insufficient to isolate the sources of denormal arithmetic in an application. A novel tool based on the Valgrind binary instrumentation framework is developed which can trace the origins of denormal values and the frequency of their occurrence in an application's data structures. This second tool is used to isolate and remove the cause of denormal arithmetic both from a simple numerical code, and then from a face recognition application.

¹Denormal or *subnormal* numbers are described in [Sec. 1.3](#) and [Sec. 4.3](#)

Acknowledgements

Firstly I'd like to thank my supervisor Stephen Jarvis for overseeing my years at Warwick, and providing encouragement and guidance through academia and the world of High Performance Computing.

Thanks to Dan Spooner for many stimulating and varied discussions over the course of three years. A dozen papers could be written on the subjects discussed, and a small fortune spent on the shiny toys from Silicon Valley we examined.

Thanks to Paul Isitt for being an accommodating lab-mate, paper co-author and general collaborator as we explored the work of HPSG and the performance prediction community.

Thanks to Russell Boyatt for providing much useful technical advice, many movie recommendations, and for being an obliging sounding board.

I'm extremely grateful to my friends and family for their moral support. Thanks mum and Niamh for your patience, continued support, and encouragement. I wish I could similarly thank dad for setting me on this path in the first place, but sadly he is no longer with us. Thanks to Gordon, Darina, and Patrick for providing levity, suggestions, and pep talks as the occasion required. I'm sure you'll be glad never to hear the word 'thesis' from me again!

This thesis is dedicated in loving memory of my father, 1952–2009

NON HABERI, SED ESSE



Declarations

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work described in this thesis has been undertaken by myself except where otherwise stated.

The following publications relate to the thesis text:

- Brian P. Foley, Paul J. Isitt, Daniel P. Spooner, Stephen A. Jarvis, and Graham R. Nudd. Implementing Performance Services in Globus Toolkit v3. In *Proceedings of the UK Performance Engineering Workshop*, July 2–8 2004, Bradford, UK. [[Appendix B](#)]
- Brian P. Foley, Daniel P. Spooner, Paul J. Isitt, Stephen A. Jarvis, and Graham R. Nudd. Performance Prediction for a Code with Data-dependent Runtimes. Awarded best paper at *UK eScience All Hands Meeting*, September 19–22 2005, Nottingham, UK. [[Chapter 3](#)]
- Stephen A. Jarvis, Daniel P. Spooner, Brian P. Foley, Paul J. Isitt, Graham R. Nudd. Predictive Workflow Scheduling for Multi-Clusters and Grids. *CORS/INFORMS Joint International Meeting*, May 16–19 2004, Banff, Alberta, Canada. [[Chapter 3](#)]
- Paul J. Isitt, Stephen A. Jarvis, Brian P. Foley, Daniel P. Spooner, and Graham R. Nudd. Towards Performance-aware Real-time Management for On-demand Computing Environments. In the *7th International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS)*, September 23–24 2005, Torino, Italy. [[Chapter 3](#)]
- Stephen A. Jarvis, Brian P. Foley, Paul J. Isitt, Daniel Rückert, and Graham R. Nudd. Performance Prediction for a Code with Data-dependent Runtimes. In *Concurrency and Computation: Practice and Experience* 19:1–12, 2007. [[Chapter 3](#)]
- Brian P. Foley, and Stephen A. Jarvis. Tracing Denormal Arithmetic with Valgrind. Submitted to the *37th International Symposium Computer Architecture (ISCA 2010)*, June 19–23, Saint Malo, France. [[Sec. 5](#)], [[Chapter 6](#)]

The thesis builds on the research on PACE, [[Pap95](#), [KHCN96](#), [CKPN99](#)] TITAN, [[SCT⁺02](#), [SJC⁺03](#), [SKD⁺03](#)] and performance modelling [[Har99](#), [TLHKN02](#), [MJSN06](#)] performed at the High Performance Systems Group.

Contents

Abstract	ii
Acknowledgements	iii
Declarations	v
Contents	vi
List of Tables	x
List of Figures	xii
Abbreviations	xiv
Chapter 1 Introduction	1
1.1 CPU performance prediction	1
1.2 Context and Previous Research	7
1.3 Thesis Contributions	8
1.4 Thesis Overview	10
Chapter 2 Performance prediction and its application	13
2.1 Performance modelling tools	14
2.1.1 SimpleScalar	14
2.1.2 PACE	15
2.1.3 WARPP	17
2.1.4 Prophesy	19
2.1.5 Performance Evaluation Process Algebra (PEPA)	21
2.2 Performance monitoring tools	25
2.2.1 Network Weather System	26
2.2.2 MDS	27
2.3 An analytical performance model of Sweep3D	31
2.4 Scheduling as an application	35
2.5 Summary	39
Chapter 3 Performance modelling and scheduling of a data-dependent code	41
3.1 nreg Image Registration	42

3.2	The nreg algorithm	43
3.3	nreg’s computational costs	45
3.4	Parallelising nreg	47
3.5	Predictive model	49
3.5.1	Pre-model work parameter	51
3.6	IXI workflows	54
3.7	Incremental prediction	54
3.8	User interaction	56
3.9	Speculative scheduling	57
3.10	Case study	58
3.11	Summary	62
Chapter 4	Floating point and denormal handling	65
4.1	Fixed point arithmetic	66
4.2	Floating point arithmetic	68
4.2.1	Scientific notation	68
4.2.2	Floating point calculations	69
4.3	IEEE-754	70
4.3.1	Gradual underflow	72
4.3.2	Mathematical properties	73
4.3.3	IEEE-754 implementations	74
4.4	DIP: A denormal profiler for Linux x86	76
4.4.1	Floating point exceptions on the 80x86	76
4.4.2	Using exception handlers	80
4.4.3	Exception handling in Linux	81
4.4.4	Library interposition to profile binaries	82
4.4.5	Example of DIP in use	84
4.4.6	Limitations of DIP and exception-based profilers	85
4.5	Summary	87
Chapter 5	Implementing a denormal tracing tool using Valgrind	89
5.1	Introduction to Valgrind	90
5.2	Shadow Memory	92
5.3	Taint analysis	94
5.3.1	Taint analysis using Valgrind	95
5.3.2	Taint analysis and denormal tracing	97

5.3.3	Denormal vs taintedness lifecycle	97
5.3.4	Reporting denormal events	98
5.4	DART: A denormal tracing tool for Linux	99
5.5	Memory management	100
5.5.1	Compile-time allocation	102
5.5.2	Runtime allocation	104
5.6	Metadata and tracing	106
5.6.1	Tag storage	106
5.6.2	Tag semantics	108
5.6.3	Tag usage	110
5.7	Summary	114
Chapter 6	Using the new profiling and tracing tools	117
6.1	Programs	117
6.1.1	jacobi	117
6.1.2	187.facerec	119
6.2	Analysis	121
6.2.1	Instruction profiles	121
6.2.2	Array heatmaps	122
6.2.3	Array origin maps	124
6.2.4	Optimising origin tracking	125
6.2.5	Garbage collecting fvals	126
6.3	Experiments	129
6.4	Profiling and tracing jacobi	129
6.4.1	Using DIP	129
6.4.2	Using DART	132
6.4.3	Comparing results from DIP and DART	132
6.4.4	Denormal heatmaps	134
6.5	Profiling and tracing 187.facerec	136
6.5.1	Using DIP	136
6.5.2	Using DART	138
6.5.3	Denormal heatmaps	140
6.5.4	Heatmap interpretation	141
6.5.5	Origin maps	143
6.5.6	Origin statistics	144
6.5.7	Missing arrays	146

6.5.8 Removing denormals	147
6.6 Limitations	149
6.7 Summary	152
Chapter 7 Conclusions	153
7.1 Summary	153
7.2 Contributions	153
7.3 Future work	154
7.3.1 Generalising DART	154
7.3.2 Observation and analysis	156
Appendix A Floating point representations	157
Appendix B Performance modelling in PACE	159
Appendix C A Linux/x86 LD_PRELOAD denormal profiler	163
Appendix D Integer Optimisations for FP on x86	167
D.1 Storage classes	167
D.2 Assignment	169
D.3 DWARF debugging information	170
Appendix E Valgrind Intermediate Representation and instrumentation	175
Appendix F 187.facerec denormal profile	177
Appendix G nreg case study data	180
Appendix H 187.facerec runtimes	182
Bibliography	183
Colophon	191

List of Tables

3.1	EvaluateGradient iterations	47
3.2	EvaluateDerivative costs	49
3.3	Parallel speedup of nreg vs sequential code	49
3.4	Comparing scheduling techniques with varying workloads	60
3.5	The effect of probe task speed on makespan	61
4.1	Single-precision storage in IEEE-754	72
4.2	Flush to zero behaviour	73
4.3	Gradual underflow with denormals	73
4.4	Divide by zero without denormals	74
4.5	Denormals at cfd-sor startup	84
4.6	Denormal instruction profile in cfd-sor	86
6.1	fval_info table for a simple expression	127
6.2	Denormal exception profile for jacobi	131
6.3	DART denormal profile for jacobi	132
6.4	Denormal loads/stores for jacobi arrays	134
6.5	Denormal exception profile for 187.facerec	137
6.6	DART denormal profile summary for 187.facerec	138
6.7	Denormal array accesses in 187.facerec	141
6.8	Denormal origins in 187.facerec	144
6.9	Genuine denormal origins in 187.facerec	147
6.10	Genuine denormal array accesses in 187.facerec	147
6.11	Denormal exception profile for patched 187.facerec	148
6.12	Denormal array accesses in patched 187.facerec	148
6.13	DART denormal profile for patched 187.facerec	150
A.1	5 bit unsigned minifloat values	158
F.1	DART denormal profile for 187.facerec, part I	177
F.2	DART denormal profile for 187.facerec, part II	178
F.3	DART denormal profile for 187.facerec, part III	179
G.1	Comparing scheduling techniques with varying workloads	180
G.2	Comparing different prediction models with varying workloads	181

H.1	Runtimes of 187.facerec on different architectures	182
-----	--	-----

List of Figures

1.1	Data hazard in a Central Processing Unit (CPU) with a 3 stage pipeline	3
1.2	Register renaming used to avoid anti-dependency of I_3 on I_1	6
2.1	NWS clique hierarchy	28
2.2	LDAP object class	29
2.3	LDAP distinguished name	29
2.4	Sweep3D wavefront behaviour	31
2.5	Run-time schedule of three example workflows	38
3.1	Slices from brain scans in need of registration	43
3.2	Transformation of 2-D image by nreg	44
3.3	Runtime variation for different images	50
3.4	Runtime scaling with subsampled images	52
3.5	Predicted runtimes	53
3.6	Prediction error	53
3.7	Operation of the performance-aware resource management system .	54
3.8	Workflow builder with performance model evaluation	56
3.9	Workflow scheduling a mix of speculative and real tasks	58
4.1	A 5 bit ‘minifloat’ format	69
4.2	IEEE-754 float and double formats	71
4.3	jacobi inner loop	78
4.4	Compiled code	78
4.5	Linux FPE handling	83
4.6	Kármán vortex sheet in cfd-sor	84
5.1	Valgrind instrumentation process	90
5.2	Heap overflow	93
5.3	Use after free error	93
5.4	Invalid data propagation	94
5.5	Linux 2.6.x x86 memory map	101
5.6	Dynamic and compile time allocation	104
5.7	Two level tag table	107
5.8	Tag word layout	108

5.9	Trace log from an example <code>jacobi</code> kernel	113
5.10	Graph of operations from an example <code>jacobi</code> kernel	114
5.11	Multi-copy code	114
5.12	Trace log of multi-copy code	114
5.13	Graph of multi-copy code operations	115
6.1	Laplace steady state approximation after N iterations	119
6.2	<code>187.facerec</code> local frequency measurements[PKvdM96]	120
6.3	Distinct tables for tag and read/write counts	124
6.4	Tags for a simple expression	127
6.5	<code>jacobi</code> kernel instructions	130
6.6	Heatmap showing sweep of denormal averages produced by <code>jacobi</code>	135
6.7	Map of number of denormals in <code>jacobi</code> array a derived from b	136
6.8	Array elements with denormal values in the five <code>187.facerec</code> kernels.	142
6.9	<code>187.facerec</code> temporary array denormal heatmaps	143
6.10	<code>187.facerec</code> origin maps	145
A.1	5 bit minifloat compared to IEEE-754 float	157
A.2	A 5 bit ‘minifloat’ format	158
B.1	Excerpts from the <code>SunUltra_10.hmc1</code> hardware model	159
B.2	The <code>async.la</code> parallel template	160
B.3	<code>blend.c</code>	160
B.4	<code>blend.la</code> – the blend subtask	161
B.5	<code>blend_app.la</code> – the application model	161
B.6	<code>Makefile</code>	162

Abbreviations

ALU	Arithmetic and Logic Unit
CHIP³S	Characterisation Instrumentation for Performance Prediction of Parallel Systems
CPU	Central Processing Unit
CTMC	Continuous Time Markov Chain
DAG	Directed Acyclic Graph
DIE	Debugging Information Entry
DIT	Directory Information Tree
DN	Distinguished Name
FIFO	First-In First-Out
FPU	Floating Point Unit
GIIS	Grid Index Information Service
GLUE	Grid Laboratory Uniform Environment
GRAM	Grid Resource Allocation Manager
GRIP	Grid Information Protocol
GRIS	Grid Resource Information Service
GRRP	Grid Registration Protocol
IDT	Interrupt Descriptor Table
IPC	Instructions Per Clock
IR	Intermediate Representation
JIT	Just In Time Compilation

LDAP	Lightweight Directory Access Protocol
LU	Least Upper matrix decomposition
MDS	Monitoring and Discovery Service
MMU	Memory Management Unit
MPI	Message Passing Interface
NWS	Network Weather System
ODE	Ordinary Differential Equation
OGSI	Open Grid Services Infrastructure
PACE	Performance Analysis and Characterisation Environment
PEPA	Performance Evaluation Process Algebra
QoS	Quality of Service
RTL	Register Transfer Language
SMT	Simultaneous Multithreading
SOAP	Simple Object Access Protocol
TLB	Translation Lookaside Buffer
WSRF	WS-Resource Framework
iFFT	Inverse Fast Fourier Transform
ulp	unit of the least place

Introduction

From the invention of the first electronic computers to the sophisticated systems of today, the increase in available processing power has been enormous. Cambridge University's EDSAC, a typical early computer, had approximately 500 words of memory and could execute 600 instructions per second [CK92]. In 2009, an inexpensive laptop is likely to have more than a million times as much memory and several million times as much processing power; and a high end cluster can provide more than 10,000 times as much usable compute power again.

The availability of large amounts of inexpensive processing power has enabled the mathematical and scientific communities to perform research that would otherwise have been impossible. Despite this continual growth, the demand for more processing power than is readily available has remained constant. von Hoerner's stellar dynamics simulations in 1960 modelled systems with about 16 particles; the current equivalents are cosmological models with 10^9 particles [vH60, TCPP98]. Protein folding [Pan02], medical image processing and number theory [Wol96] are examples of the many scientific domains that absorb as much CPU time and storage as can be provided.

Until recently, Moore's law¹ has provided continual increases in inexpensive sequential compute performance, but there are warning signs that this exponential growth is becoming difficult to sustain.

1.1 CPU performance prediction

For decades, single-threaded microprocessor performance has benefited from Moore's Law. Due to the relatively large transistor size, and limited die area on a silicon chip, early microprocessor designs had very limited transistor budgets, and were forced to have few registers, to omit any non-essential functionality, and to implement critical components such as Arithmetic and Logic Units (ALUs) and control units using as few transistors as possible. One way of doing this is to operate on the data serially² or a few bits at a time. The downside of this approach is that to perform a

¹An empirical observation that the number of components that can be fitted on a given area of silicon at a given price point seems to double approximately every 18 months

²i.e., one bit per clock cycle

single operation, pieces of a word of data have to pass through the [ALU](#) in several stages, increasing the number of clock cycles taken to perform the operation. As the transistor budget increased, widening the data paths and expanding the [ALU](#) was one obvious way to improve performance. When data paths become fully parallel, further performance improvements were achieved by reducing the depth of the logic trees required to implement operations such as add and multiply. This was done at the cost of extra transistors using techniques such as carry look-ahead blocks for adders and Wallace trees [[Wal64](#)] for multipliers.

As the transistor size decreased and transistor budget increased, performance rapidly increased along with it. Some of these improvements included:

- Both the amount of time and the power needed to switch a transistor on and off decreased, leading to an increased clock rate.
- [ALUs](#) became fully parallel, allowing additions, effective address calculations, and program counter updates to be performed in one clock cycle.
- Hardware multipliers could be implemented, first in serial-parallel form, and later fully parallel. This was accompanied by hardware dividers, although with less parallelism. Early multipliers and dividers typically generated one or two bits of output per clock cycle.
- When the transistor budget reached a certain threshold, floating point co-processors became practical, and later benefited from increasingly parallel implementations.
- Eventually Floating Point Unit ([FPU](#)) co-processors were integrated on the same chip as the [CPU](#). This reduced the communications overhead between the [CPU](#) and [FPU](#), further improving performance.

Although these advancements dramatically improved [CPU](#) speed, the behaviour of a [CPU](#) still remained relatively predictable and straightforward, and none of these advances causes substantial difficulties for performance estimation techniques based on simple instruction counting.

During the early to mid 1990s however, processor manufacturers started to include features that yielded substantial performance improvements, but would prove much more challenging to model. Techniques included:

- Functional units on [CPUs](#) started to become heavily pipelined. So, for example, an integer divide might take 7 clock cycles to complete, each cycle completing

Cycle	1	2	3	4	5	6	7	8
$I_1 : R_{10} = R_1 \div R_2$	Fetch	Exec	Exec	Exec	Exec	Store		
$I_2 : R_{11} = R_3 + R_4$		Fetch	Exec	Store				
$I_3 : R_5 = R_{10} \times R_{11}$			Fetch	<i>Stall</i>	<i>Stall</i>	<i>Stall</i>	Exec	Store

Fig. 1.1: Data hazard in a CPU with a 3 stage pipeline where divides have a 4 cycle latency. The third instruction cannot execute until after the two previous instructions have stored their results.

one stage of a 7 stage pipeline. However, once the first stage had completed, it became available for use by another instruction. One divide would take 7 clock cycles to complete, but 100 independent divides one after the other would only take 107 cycles to complete. Pipelining can be very effective, but fails to deliver a performance benefit when subsequent instructions depend on the results of an earlier instruction that is still going through a pipeline. When this happens the pipeline ‘stalls’ — the CPU must wait for the first instruction to complete before the second instruction can be issued. To model this, performance estimation software has to know the depth of each of the pipelines in the CPU, and track the dependencies between instructions. Fortunately this can be accurately modelled solely by static analysis of the instruction stream itself. A simple example can be seen in Fig. 1.1.

- Memory Management Units (MMUs) and memory caches began to be integrated on chip, and moved from direct mapped to more efficient and complex set-associative schemes reducing the need to access slow main memory. This had the benefit of decreasing the average amount of time a CPU had to wait to read/write memory, improving performance. However introducing extra layers into the memory hierarchy meant that memory access times were no longer constant. Two pieces of code with exactly the same mix of instructions could run at dramatically different speeds depending on whether their memory access patterns were cache-friendly or not. Unlike pipelining, cache behaviour is not readily amenable to static code analysis, and this causes considerable problems for performance estimation, although it is possible to apply heuristics to estimate the effects of some common memory access patterns [Har99].

The design of MMUs introduces a different, although related problem. To minimise the cost of translating virtual addresses into physical addresses, all MMUs have a small cache called a Translation Lookaside Buffer (TLB) that holds the mappings for commonly used pages. If a program uses a page that is not in the TLB, the MMU or OS are forced to fetch the mapping from main memory

at a considerable cost. If the program's memory access patterns touches many pages of memory, the *TLB thrashing* this causes can be a significant slowdown. As with memory cache performance analysis, this cannot be analysed statically, but similar heuristics can be applied; **MMU** pages in a **TLB** will have a similar eviction policy to cache lines in a memory cache.

- To reduce the impact of the pipeline stalling problems mentioned above, **CPUs** started to use more sophisticated control units to allow 'Out-of-order Execution'. Rather than issuing instructions strictly in order, the **CPU** queues up a number of instructions for execution. When the operands for that instruction become available and a functional unit is not busy, the instruction is issued for execution on that unit. As instructions complete, their results are *retired*, or queued for writing to the appropriate registers. Thus, a single instruction with a dependency will no longer stall the pipelines of all the functional units until the dependency is met. Instead the instruction itself will be delayed and other instructions are free to be issued ahead of it as long as they don't depend on any unavailable results.

To make this more effective, Out-of-Order **CPUs** tend to have large numbers of *rename registers* as well as multiple copies of functional units which are not directly visible to the programmer. The net effect of Out-of-Order execution is to allow a serial instruction stream with possible pipeline-stalling dependencies to be translated into a parallel stream of instructions, one stream per functional unit. When this works, a **CPU** can issue and retire several instructions per clock cycle, although in practice significant amounts of parallelism are difficult to achieve. Also, this makes life especially difficult for performance estimation software, as it now has to deal with an instruction stream which may be translated into several different instruction streams in a different order. Worse still, the streams can have interdependencies, and the same piece of code in memory may be translated into a differently ordered stream and issued to different units every time it is encountered depending on the internal state of each of functional units in the **CPU**. To determine the cost of an instruction, its overall *context* and the internal state of the **CPU** are far more important than the actual instruction itself. A simple example of register renaming can be seen in [Fig. 1.2](#).

- A final set of techniques that modern **CPUs** use to try to alleviate pipeline stalls and to keep functional units busy are those of *speculative execution* and *branch prediction*. Since it is impossible to know ahead of time whether a conditional branch will be taken or not, the most conservative technique is to stop issuing instruc-

tions until a branch's arguments are known, and to only continue after that. This causes serious pipeline stalls however, and since branch instructions are common, it causes a major performance penalty. To deal with this, Out-of-Order CPUs can guess which direction a branch is going to proceed, speculatively execute beyond the branch instruction, and store the results of speculative instructions to temporary registers. When the branch dependencies are finally resolved, if the guess was incorrect, the unneeded computations are discarded. If the guess is correct the results in the temporary registers are retired. If enough functional units and rename registers are available, the CPU can speculatively execute *both* paths of a branch, and discard the unnecessary one when the branch dependency becomes available.

Of course, if the branch prediction is unsuccessful, it causes unnecessary instructions to be issued, but as long as it is correct some of the time, it is still a performance benefit, as it allows some use to be made of otherwise idle execution units.

To improve the impact of speculative execution, modern CPUs maintain large tables of statistics describing which direction a particular branch instruction is likely to go. As each branch is taken or not, the statistics are updated, and thus the CPU builds up a model of the branch behaviour of the program. As with Out-of-order Execution, the internal state of the CPU, or more specifically of the branch prediction tables determines the cost of the instruction. Other than in simple cases, the probability of following a branch is related to the input data, and so, unlike Out-of-order techniques, the effectiveness of branch prediction is dependent on the program data itself. This is something that it is impossible for performance modelling software to analyse statically, and poses another problem for performance prediction systems like PACE [NKP⁺00]. Other systems, such as Prophesy and WARPP sidestep this problem by relying on coarser grained benchmarks for the basic blocks in an application.

The above techniques have all been inspired by the continually increasing transistor budget and a desire to extract as much parallelism as possible from a sequential instruction stream, but they come at a cost. In particular, aggressive speculative execution yields a linear increase in Instructions Per Clock (IPC) rates for an exponential cost in transistors, design complexity, and power consumption.

Recent trends seem to indicate that processor design is starting to push up against physical limits such the electrical properties of semiconductor materials, propagation delays, and particularly the ability to dissipate heat from CPU dies [AHKB00,

Without register renaming.

Cycle	1	2	3	4	5	6	7	8	10	11	12	13	14
$I_1 : R_1 = R_{10} \div R_{11}$	IF	EX	EX	EX	ST								
$I_2 : MEM(1000) = R_1$		IF	-	-	-	EX	ST						
$I_3 : R_1 = R_{12} \div R_{13}$			IF	-	-	-	-	EX	EX	EX	ST		
$I_4 : MEM(1001) = R_1$				IF	-	-	-	-	-	-	-	EX	ST

With register renaming.

Cycle	1	2	3	4	5	6	7	8	10	11	12	13	14
$I_1 : R_{1a} = R_{10} \div R_{11}$	IF	EX	EX	EX	ST								
$I_2 : MEM(1000) = R_{1a}$		IF	-	-	-	EX	ST						
$I_3 : R_{1b} = R_{12} \div R_{13}$			IF	EX	EX	EX	ST						
$I_4 : MEM(1001) = R_{1b}$				IF	-	-	-	EX	ST				

Fig. 1.2: Register renaming used to avoid anti-dependency of I_3 on I_1 .

[Bor03](#)]. As a result of this, the industry has moved away from scaling clock speeds and using aggressive speculative execution to speed up sequential codes. Instead the emphasis is now on making more efficient use of the growing transistor budget constrained by a fixed power budget by providing several less aggressive [CPU](#) cores on the one die, each running independent instruction streams. Commercial examples of this include Intel's Core family of [CPUs](#), IBM's POWER series, and Sun's recent UltraSPARCs. The same trends are driving the performance of graphics processing units capable of restricted forms of computation, and digital signal processors. Graphics processing units can take advantage of a more constrained computational model than general purpose [CPUs](#) and so have a much higher proportion of their transistors allocated to [FPU](#)s instead of control logic. This makes them significantly faster than general purpose [CPUs](#) for applications that fit their computational restrictions.

In addition to this, another technique called Simultaneous Multithreading ([SMT](#)) uses the same technology as speculative execution, but uses it to run as many independent threads of execution as possible on idle or stalled functional units. Every time a thread stalls because of an unmet dependency, another waiting thread is 'swapped in' to make use of an otherwise idle functional unit. This results in a lower [IPC](#) rate for each individual thread, but the overall instruction throughput is higher. The technique has the effect of masking memory latency to some degree and can be particularly useful for codes with unavoidably long chains of unpredictable branches or unfortunate memory access patterns, neither of which benefit from speculative execution. Many commercial workloads, such as web servers or database servers with large numbers of connections fall into this category.

1.2 Context and Previous Research

The High Performance Systems Group at the University of Warwick have demonstrated that performance prediction, i.e., the rapid estimation of the resource usage of an application on a given computer, is essential for the efficient utilisation of distributed systems. [JSK⁺06]

Under previous research, techniques and tools have been developed at Warwick such as the Performance Analysis and Characterisation Environment (PACE) [NKP⁺00] toolkit. PACE uses a combination of static code analysis, micro-benchmarking and event simulation to rapidly provide runtime estimations of performance and for both sequential and distributed applications [CKPN99], and has been used in application steering and job scheduling systems [KPN98, SJC⁺03].

Another researcher, also at Warwick, developed a scheduler called TITAN [SJC⁺03] which has shown that significant performance improvements can be achieved by using performance models as part of performance-responsive middleware services that address the implications of executing a particular workload on a given set of resources.

Since PACE was developed, both scientific applications and the hardware that they run on have increased in complexity. Binary compatibility means that old applications will generally run on new hardware; but due to architectural changes, code sequences which may have been optimal on an older processor might perform poorly on a more recent one. Performance prediction becomes much more difficult for the reasons mentioned above, and in these new environments, the speed at which a fragment of code executes now depends largely on its *context*, i.e., the state of the CPU and the mix of instructions and memory accesses used and not the individual instructions. Context, in turn is determined by the flow of control within an application, and for all but deliberately regular applications, the flow of control depends on the input data.

The applications themselves have also increased in complexity. Numerical applications now routinely take advantage of high-level languages such as C++; software engineering techniques such as object oriented programming and garbage collection; and more advanced numerical methods such as multigrids and adaptive mesh refinement. All of these make it more difficult to build performance models for applications. They can obscure the lower-level logic and flow of control within a program, as well as hiding aperiodic expenses associated with the maintenance of data structures, such as tree rebalancing, sweeps of a garbage collector, the deferred

allocation of copy-on-write objects as they are modified and more.

Concurrently with the work in this thesis, another PACE-like tool called WARPP is under development by a different set of research students at Warwick. WARPP, which is discussed in more detail in [Sec. 2.1.3](#) avoids some of the issues of application complexity and code context by basing its CPU simulations on benchmarks of the basic blocks of an application on the target hardware. Research is underway to automate the instrumentation and benchmarking process as much as possible. By their nature, these benchmarks only measure the average case and will not, for example, capture variability due to unpredictable memory access patterns or other data-dependent effects.

1.3 Thesis Contributions

These factors when taken together make it increasingly difficult to build performance models based solely on static code analysis and benchmarks — whether coarse or fine grained. This thesis explores this notion of data dependency. The term data dependency has multiple meanings — from the dependencies between in flight instructions in a CPU; to the abstract graph of computations that a compiler manipulates; to the relationship between groups of communicating processes. The meaning here is distinct from, but related to the first two: it refers to the notion that when the information content of an application’s input data changes, the performance of the application can change unexpectedly too. This can be simply because the different data makes an algorithm perform more work, by iterating more, or by using different code paths; or it can be because the different data triggers certain hardware behaviours.

To do this, I examine two types of application. The first is a medical imaging application with highly variable runtimes. These variations are caused by the application’s algorithms themselves, in part because of their convergence criteria, but also because of how the application samples its datasets. I build a performance model for this heretofore unpredictable application based on the application’s input data. This model requires some pre-execution CPU time for every task it models, and I adapt TITAN to accommodate this and to use the sequence of increasingly refined predictions it emits.

Later in this thesis I explore the performance variabilities introduced by some hardware implementations of denormal arithmetic. Denormal arithmetic, or *subnormal arithmetic* as it is termed in the 2008 version of the IEEE-754 standard, is the processing of floating point numbers smaller than the smallest *normal numbers*, but still

larger than zero. These numbers have a different representation to normal numbers, and require special handling by the floating point implementation. I describe a simple means of detecting denormal arithmetic and show its limitations. Based on this, I build a more sophisticated tool to track denormal data in an application back to its origins. This origin tracking aids the developer in eliminating denormal arithmetic in an application producing a faster, but more importantly, more predictable application.

The primary contributions of this work are as follows:

- **Developing a performance model for a state-of-the-art C++ medical imaging application.** Initial attempts to build a PACE-based performance model for this application, `nreg`, failed for two reasons. Firstly, `nreg` makes pervasive use of inheritance and C++ templates and needs compiler optimisations to run efficiently, neither of which PACE could handle. Furthermore, the application itself exhibits data-dependent runtime variability, i.e., two input datasets of exactly the same size can easily have more than an order of magnitude difference in runtime. PACE could only provide predictions for the average case, which are of little use in this situation. I develop an alternate data-driven model which makes it possible to estimate the runtime of `nreg`. This makes it possible to use `nreg` in scenarios where quality of service criteria are useful, such as in clinical diagnosis, or to allow efficient task scheduling.
- **Analysing how computationally intensive performance prediction processes may be used as a part of a scheduler.** As part of its scheduling algorithm, TITAN makes the assumption that performance data for its applications are available at negligible cost, and makes many queries to the prediction engine for every iteration of its genetic algorithm. The performance model for `nreg` takes some time to evaluate, and the more CPU resources given to the performance model, the more accurate the prediction. These performance prediction ‘probe tasks’ need to be run on the same compute resources as the actual tasks, and thus need to be scheduled along with them. I modify TITAN to do this, and analyse the effects the asynchronous stream of continually improving predictions have on scheduling quality, and how it may be used for ‘interactive scheduling’.
- **Profiling application runtime variability caused by denormal arithmetic.** Many processors now implement fully IEEE-754 compliant denormal arithmetic in hardware, but despite this, on some of these processors denormal arithmetic still executes very slowly relative to normal arithmetic. This slowdown is present

in recent Intel x86 processors. To measure the occurrence of denormal arithmetic, I write and use a floating point exception based profiling tool called DIP. I show why profiling is insufficient to identify the true sources of performance degradation.

- **Finding the origins of denormal arithmetic and removing it from an application using a novel tool.** The dynamic binary instrumentation framework Valgrind [NS07b] is described, and using its facilities, I write DART, a denormal arithmetic tracing tool. I implement three different methods for analysing the information gathered by DART. I then use DIP and DART to examine the flow of denormal data in the two numerical codes. The sources of the denormal data are found and eliminated, producing more predictable applications as a result.

1.4 Thesis Overview

The thesis is divided into 7 chapters. The remaining chapters are organised in the following fashion:

Chapter 2 Existing performance prediction techniques are reviewed, an existing analytical performance model is examined, and the iterative heuristic scheduler TITAN is described.

Chapter 3 The medical imaging application nreg is analysed, and a novel data-driven performance model is developed for it. The chapter describes how a continually improving series of performance predictions over time can be integrated into a workflow scheduler. It shows how this can provide performance information to the user and using this as a part of an ‘interactive scheduling’ process.

Chapter 4 The IEEE-754 Floating point arithmetic standard is described, with an emphasis on the performance aspects of denormal arithmetic and common hardware implementations. DIP, a profiling tool based on floating point exceptions is written, and its limitations are examined.

Chapter 5 Dynamic binary instrumentation and taint analysis is introduced. The idea of denormal tracking is developed by analogy with taint tracking. From this, DART, a novel denormal arithmetic reporting and tracing tool is designed and implemented.

Chapter 6 DIP and DART are used to analyse the behaviour of two applications. The sources of denormals are identified and removed from both applications, thus removing a source of performance variability. Limitations of DART are discussed.

Chapter 7 The final chapter summarises the problems presented in this thesis as well as the work undertaken and the results achieved. Possibilities for future opportunities in the area are outlined.

Performance prediction and its application

Performance prediction for high performance computing involves two separate phases. The first gathers performance data for the compute resources in question, and the second uses the data as part of a performance model.

Performance data is typically gathered using some sort of benchmarking or performance measuring technique. Since hardware failures are common in systems with large numbers of components, and failure is often preceded by performance degradation, it is common to integrate this into a cluster monitoring system rather than simply benchmarking the system once. On task completion, basic statistics are often recorded too, such as the overall runtime, number of CPUs used, and the working set of the tasks. The data gathered is then stored in some sort of repository where it can be queried and analysed.

Performance modelling of the applications themselves is the second stage of prediction. These models can be as simple as an averaging of the runtimes recorded by the performance monitoring system, or may include complex analytical models such as those to be described in [Sec. 2.3](#). Irrespective of the complexity, these performance models use the performance characteristics of the hardware as parameters, and so cannot generate performance predictions in isolation.

A major use for these performance prediction models is in resource allocation systems, such as job schedulers; in middleware for instantiating abstract workflows; in parallel applications which try to balance their workload across multiple CPUs; and in performance models used to guide hardware procurement by estimating application scalability on hypothetical systems. Performance prediction can also be useful when trying to determine why an application does not perform as well as expected on a new system [[PKP03](#)].

In this chapter some existing performance modelling techniques and two systems that gather and store performance data are examined. Details of an analytical performance model by Mudalige et al. that uses this kind of data are given, and finally a workflow scheduler by Spooner et al. which uses these performance models is explored.

2.1 Performance modelling tools

Depending on the problem domain, a whole spectrum of performance measurement and modelling techniques can be used. These vary in sophistication: Simple techniques use black box timings of a set of candidate workloads as the measurement, and curve fitting as the model; more sophisticated approaches gather sets of hardware microbenchmarks and feed them to application models of what computations are performed and how they are distributed across a cluster.

The more sophisticated models usually have the benefit of having more explanatory power, and providing more reliable predictions; however they are also more difficult to create, rely on a deeper understanding of the hardware and software, and may be slow to evaluate.

Another approach is to model software systems in a more abstract and mathematical manner by treating them as a set of interacting components specified using a formal language. When specified in this way, numerical and analytical tools can be used to find solutions to the steady state performance of the system.

2.1.1 SimpleScalar

Due to the complexities of modern microprocessors mentioned in [Sec. 1.1](#) there are many challenges to predicting the performance of a piece of code on a given processor. The most direct way to solve this problem is to exactly simulate each component of the processor's micro-architecture and to run the code directly on the simulation—or feed an instruction trace into it—and observe the results of the simulation. Tools such as SimpleScalar [\[BA97\]](#) do exactly this, and because of the precision of their results they are mostly used during the design of microprocessors. Because of the fact that they have to fully model the behaviour of the cache hierarchy, the functional units, the logic associated with Out-of-order Execution and maintain profiling information for all of these, simulators of this sort run much slower than native hardware. For example, the most detailed SimpleScalar 2.0 simulator running on a machine capable of more than 200 MIPS executes approximately 150,000 simulated instructions per second, a factor of more than 1,000 slowdown. SimpleScalar 4 has a similar slowdown: a full simulation on a 1.6 GHz Pentium 4 system runs at 350,000 IPS.

Simulators are not very useful as a part of a resource allocation system because of this slowdown — it would be simpler and much faster just to run the application directly on the candidate processor. They can, however, be used to accurately

benchmark application kernels on hypothetical systems, and these benchmarks can be used together with other less detailed performance modelling tools.

Because of their accuracy, these tools can also be useful both for low-level tuning of application kernels on a specific micro-architecture, and for designing new processors that work better with common workloads. One instance of this is the tool WATTCH [BTM00] built on top of SimpleScalar which predicts the power consumption of a processor under particular workloads. It does this by modelling the power draw of different primitive units within the processor (such as clocks, busses, logic, and memory arrays) and describes each part of the processor (such as TLBs or branch prediction units) in terms of these. When an instruction stream runs on the processor, the approximate number of transistor switchings in each unit is measured, and from this the overall power draw can be calculated.

2.1.2 PACE

PACE, the Performance Analysis and Characterisation Environment [NKP⁺00] was developed by the High Performance Systems Group at the University of Warwick in the late 1990s and early 2000s to predict the runtime and resource usage of scientific applications using pre-execution modelling and analysis. PACE provides rapid and accurate estimations for both sequential and distributed applications [CKPN99], and has been used in application steering and job scheduling systems [KPN98, SJC⁺03].

Rather than directly simulating the execution of a code-base as micro-architectural simulators do, PACE uses a language called Characterisation Instrumentation for Performance Prediction of Parallel Systems (CHIP³S) to build a static performance model of an application program. This language provides constructs to describe the flow control, overall instruction usage and communications patterns of an application in a parameterised fashion. PACE includes facilities to benchmark computation and network performance on a candidate hardware platform, and allows the application models to be evaluated against these hardware models.

CHIP³S performance scripts, written using a C-like syntax, are compiled and linked into an executable that can be run to provide performance analyses and predictive traces of application execution. Since PACE performance models are compiled, and since they calculate the runtimes of large blocks of code interleaved with computation instead of simulating individual instructions, they generate their predictions extremely rapidly. Depending on the degree of detail of the model, predictions typically take between 1 millisecond and 1 second to evaluate.

PACE's application models decompose into a set of subtasks, which in turn can contain other subtasks. Each subtask in [CHIP³S](#) consists of a sequence of flow control elements (bounded loops, and conditional statements) and each element (or block) contains a number of primitive operations. These primitive operations are characterised by a fixed delay and include such events as floating point multiplies, memory accesses and array indexing. Interprocess communications via Message Passing Interface ([MPI](#)) are characterised using a network model based on bandwidth and latency. The characterisation models the fact that small [MPI](#) messages have different performance characteristics to large [MPI](#) messages in most implementations.

The hardware models contain a list of costs (or characterisations) of each primitive operation for a given hardware architecture, and the file format PACE uses for hardware models is modular and extensible. Early PACE hardware models only had characterisations for instruction sequences used in C programs. Over time, various projects extended this to include characterisations for processor caches, and the memory hierarchy [[Har99](#)]; inter-node communications via [MPI](#), [MI](#) and [PVM](#); performance of SUIF primitives [[WFW93](#)]; and the cost of interpreting Java bytecodes [[Tur03](#)].

The final component in a PACE application model is the parallel template. This provides a means of expressing the costs and constraints associated with subdividing a task to run on multiple processors in a cluster. Applications within [CHIP³S](#) are eventually decomposed to blocks of primitive compute operations interleaved with communications. PACE's parallel templates have a `step` declaration which refers to a block of computation within a subtask, and statements indicating communication between two nodes.

The blocks from each of the subtasks are taken and compiled into a control flow of blocks. When run, each of these subtasks and each synchronous block is evaluated according to the model scripts and a predictive trace is made of operation usage. This is then translated into resource usage using the data from a hardware model. When communications occur they introduce dependencies between [CPUs](#), and a simple discrete event simulation is used to order the communications and computations on a time line.

When compared to direct simulation, PACE provides much less fined-grained detail, but is capable of modelling massively larger applications. PACE is less accurate at predicting the runtime of sequential blocks of code on modern processors than simulation, due to the effects mentioned in [Sec. 1.1](#), however PACE can model the

costs of network communication well, and in practice these contribute to a large portion of the runtime of typical parallel scientific applications.

A simple performance model is described in more detail in [Appendix B](#).

2.1.3 WARPP

Although successful in modelling some classes of applications, PACE's reliance on models derived from the source code of the application rather than from the optimised binaries produced by a compiler has become more and more of a problem. As discussed previously, modern processors are extremely complex internally and rely on sophisticated compiler techniques to schedule instructions, allocate registers, unroll loops etc. so that the best use is made of the processor's functional units and cache. PACE's assumption that a C construct such as a single iteration of loop or an array assignment can be characterised by a single timing is simply no longer true.

To deal with this problem and others, a new performance modelling framework called WARPP has been developed at Warwick [[HMS⁺09](#)] concurrently with the research in this thesis, but by different researchers. At a high level it is similar to PACE: parametric performance models for an application are written in a C-like scripting language. When executed, the models generates a trace of computation and communication events on all of the simulated CPUs which are fed into a discrete event simulator. A hardware model for both the network and CPUs orders assigns a timing to each event, and the simulator uses these timings to order the communications and computations on a timeline.

The differences lie in the details:

- PACE's CPU models are based on extremely fine grained microbenchmarks, but they do not accurately capture the behaviour either of modern compilers or modern CPUs. To avoid the problem of *context* mentioned in [Chapter 1](#), WARPP abandons fine grained microbenchmarks in favour of benchmarking each basic block of the application via compile time instrumentation.
- Like PACE, WARPP includes a model for network communications and for multiple different protocols. Both PACE and WARPP's models account for the fact that small message communications are usually handled differently to large messages, and thus have different scalability properties. In PACE this is handled by fitting two line segments and a crossover point to the scalability curve. In WARPP it is handled by explicitly subdividing protocols into regions, treating

small and large messages as different regions. WARPP's method is more flexible, allowing for multiple timings depending on the message size.

- WARPP allows a complex heterogeneous network layout to be specified on a CPU by CPU basis, for example extremely fast low latency IPC between CPUs on multicore chips, slower communications between CPUs on SMP compute nodes, and a system-wide interconnect between nodes. PACE simply assumes a flat, homogeneous network topology.
- WARPP allows multiple CPU types to be specified in a system, each with its own set of benchmark measurements. PACE assumes a cluster of identical CPUs.
- WARPP allows disc I/O to be characterised, whereas PACE only models network I/O and CPU.
- WARPP and PACE use a similar discrete event system to schedule network events and thus calculate the delays associated with communications, however WARPP's implementation is much more efficient, and scales effectively to very large numbers of CPUs. Initial attempts at a PACE performance model for Sweep3D as described in [MVJ08] had unacceptable runtimes and impractical memory usage for relatively small numbers of CPUs.

Furthermore, experimental work has been done with WARPP to model performance variability in the form of 'system noise', or random occurrences of low-level slowdowns that frequently occur on cluster systems [HMS⁺09]. These can be caused by interrupt handlers, by network contention, or by system daemons running in the background that periodically wake up and perform a small amount of work. Regardless of the cause, system noise can have an effect on application runtimes hugely out of proportion with the individual slowdowns caused on each node [PKP03]. Initial experiments that inject compute noise into a running model exhibit similar slowdowns to those seen in a 960 CPU commodity cluster in production use.

Additional WARPP work focuses on the automatic instrumentation of application basic blocks and MPI communications [SHM⁺09], and the automatic generation of parameterised performance models from the traces produced by the instrumentation [HSMJ09]. The trace analysis compares the traces produced by different CPUs and when differences exist, these are used to specialise the call graph used in the model for each MPI rank, and to later parameterise the model based on MPI rank. Although the automatically produced models do not have the predictive accuracy

of a hand-generated model, they are much faster to create, and can later be tuned by hand for further accuracy.

2.1.4 Prophecy

The Prophecy system[[TWL⁺01](#), [TWS03](#)] takes another high-level approach to modelling applications. Similar to PACE, it does not simulate individual instructions, but relies on higher-level abstractions. Instead of PACE's modelling language, it relies on the notion that many applications can be decomposed into a set of kernels which consume most of the application's runtime. If the performance characteristics of these kernels can be captured, and the kernels' interactions when run together can be described, then it is possible to build up a performance model of entire applications without needing to analyse individual instructions.

Prophecy provides three connected components.

The first component is PAIDE, the Prophecy Automatic Instrumentation and Data Entry system.[[WTS01](#)] This is the 'data gathering' part of Prophecy. It automatically instruments source code on one of several levels: on the level of entire functions, entire loops, or even individual basic blocks. The instrumentation design minimises overhead, and critical kernels or optimisation sensitive sections code can be instrumented less aggressively than the rest of the application. At runtime, the timings gathered by the instrumentation are sent to another Prophecy component along with a call graph of the application's execution.

This information is placed into a performance database, the second component of Prophecy. The database has a hierarchical structure reflecting that of the application: an application is assumed to consist of a group of modules, each in turn subdivided into functions composed of basic blocks. The performance information about an individual execution of an application (and its sub-components), the system it ran on, and the set of inputs used are stored in the database. Systems information includes the processor architecture, memory subsystem, operating system and network connections.

The final part of Prophecy is the modelling component. Here, individual kernels are modelled, and then composed to model the entire application.

The most basic form of kernel modelling is curve fitting. The user selects the performance data gathered for a subset of the execution of an application, along with a set of parameters—typically input parameters and the system it ran on—and Prophecy attempts to generate a performance model from the data points gathered using the

method of least squares. Since this is a ‘black box’ approach, it is of limited utility: it cannot predict how a kernel will run on different hardware configurations, because it cannot know how a kernel depends on different performance characteristics of the hardware (e.g., one function might be sensitive to memory bandwidth, another to memory latency, and a third to the speed of floating point divides). Also, if the kernel or application has many input parameters, a large number of data points may be required to produce a model. However, in its favour, the technique is simple and may be useful for examining the scalability of some applications.

The second form of modelling requires the developer to use performance measurements along with manual analysis to create an equation describing the runtime of an application kernel in terms of the input and the hardware system. This analysis is difficult and time consuming, but need only be done once. For example, by inspecting a function, the programmer might see that a function makes \sqrt{n} linear sweeps over an array of size n^2 and characterise the runtime in terms of memory bandwidth and array size.

The most novel feature that Prophecy introduces is that of *kernel coupling*.[\[TWGS02\]](#) This describes the effect that running one kernel will have on runtime of another kernel that executes directly after the first. It is expressed as a ratio of the runtime of kernels executed in isolation vs the kernels executed together, or if P_i is the runtime of kernel running alone, and P_{ij} is the runtime of two kernels running consecutively, C_{ij} , the coupling factor is

$$C_{ij} = \frac{P_{ij}}{P_i + P_j}$$

For example, kernels K_i and K_j might use similar datasets, and because K_i has ‘warmed the caches’ for K_j the runtime of K_i followed by K_j might be less than the sum of their runtimes in isolation, i.e., $C_{ij} < 1$. Conversely K_i and K_j might compete for resources and K_i might force data used by K_j out of main memory, requiring K_j to page it back in when it runs. In this case $C_{ij} > 1$. These two scenarios are called ‘constructive coupling’, and ‘destructive coupling’.

Kernels often occur in chains and loops, and Prophecy can use the information gathered for the coupling between individual pairs of kernels to predict the overall performance of an application. It was found in [\[TWGS02\]](#) that predictors using kernel coupling give much more accurate performance estimates than those based simply on summing the runtimes of each kernel.

Furthermore, by analysing NASA’s parallel benchmarks[\[BHS+95\]](#) on different systems the Prophecy researchers found that the coupling factors for a given set of

kernels often remain similar across different architectures. The structure of the memory hierarchy affects coupling, but the raw CPU speed of a system does not. This effect, called *Isocoupling* means that coupling values can be reused across similar classes of machines.

2.1.5 PEPA

The PEPA workbench, designed in the 1990s by Hilston et al., uses a more abstract performance modelling technique. PEPA is a process algebra with precise semantics that describes the interactions between a set of processes as they perform various actions and transform into other processes. It was inspired by earlier process algebras such as Milner's CCS [Mil80], and Hoare's CSP [HH78]. All of these algebras have in common the notion of a component with no internal state that can be transformed into another component by an activity¹. They all allow chains of activities to be performed sequentially on a component; they all allow some sort of choice for what a component turns into after an activity; and they all have some way of expressing multiple components and how they interact. PEPA differs from CCS and CSP in how it expresses parallelism, and in the fact that it includes timing information: each activity has an *action type* and a *rate*, and when an activity occurs it delays for an interval sampled from an exponential distribution.

The PEPA language is very parsimonious. It has only four combinators (or operators). These are

- **Prefix.** This is written as $(\alpha, r).P$ and represents a component that can have an activity with the type α performed on it. When this occurs, it will delay for a random time sampled from the negative exponential distribution with parameter r and then becomes the component P .

Sequences of activities that occur one after the other can be linked together either with explicitly named components, or by chaining activities together with the prefix combinator. In the later case, implicit, unnamed components exist in the model. For example, both the following models represent exactly the same model of a batch server with three states: 'idle', 'processing' and 'done' that occur strictly one after the other. The first uses explicitly named components:

¹*Component* and *activity* are PEPA's terms. Other algebras tend to call them processes and actions, however the term action has a special meaning in PEPA.

$$\begin{aligned}
Server_{idle} &\stackrel{def}{=} (submit, r_r).Server_{processing} \\
Server_{processing} &\stackrel{def}{=} (process, r_p).Server_{done} \\
Server_{done} &\stackrel{def}{=} (complete, r_o).Server_{idle}
\end{aligned}$$

Whereas this uses implicit components for a terser definition:

$$Server_{idle} \stackrel{def}{=} (submit, r_s).(process, r_p).(complete, r_c).Server_{idle}$$

- **Choice.** The choice combinator, written as $P + Q$ describes a component that can accept more than one possible activity. Both are sampled using their respective rates, and the component becomes the first one to complete. This can be looked on as two activities competing for the same resource. The first to complete wins, and the other is discarded. This can be used to model both components that change their behaviour depending on external events, and components that change in non-deterministic manner.

As an example, consider a simple porridge tasting model. In this Goldilocks will periodically taste the porridge. If it is too hot, she blows on it to cool it for a while; if it is too cold, she heats it on the stove. If it is just right, neither activity will occur.

Note that the rate at which *too_hot* and *too_cold* activities occur is unspecified for the $Goldilocks_{taste}$ component, because they depend on the temperature of the porridge and not Goldilocks herself. In PEPA terminology, Goldilocks is *passive* with respect to these two activities, i.e., PEPA requires some other component to define a rate for these activities, and the use of the cooperation combinator to allow these rates to be inferred for the complete system. Since there is no porridge component, and the rates of *too_hot* and *too_cold* are unknown, the model below is *incomplete*.

$$\begin{aligned}
Goldilocks_{taste} &\stackrel{def}{=} (too_hot, \top).Goldilocks_{cool} + (too_cold, \top).Goldilocks_{warm} \\
Goldilocks_{cool} &\stackrel{def}{=} (cool, r).Goldilocks_{taste} \\
Goldilocks_{warm} &\stackrel{def}{=} (warm, r).Goldilocks_{taste}
\end{aligned}$$

To see where non-deterministic choice can be used, consider an extended version of the batch server that occasionally crashes and needs to be rebooted. The $Server_{processing}$ component has a choice between two activities with the same action type, however PEPA distinguishes between all activities, no matter what their

type. When executing the model directly, the rates for both possible activities will be sampled, and occasionally (1 time in 100 in this case), the second activity will complete first, putting the server into the rebooting state.

$$\begin{aligned}
Server_{idle} &\stackrel{def}{=} (submit, r_s).Server_{processing} \\
Server_{processing} &\stackrel{def}{=} (process, \frac{99r_p}{100}).Server_{done} + (process, \frac{r_p}{100}).Server_{rebooting} \\
Server_{done} &\stackrel{def}{=} (complete, r_c).Server_{idle} \\
Server_{rebooting} &\stackrel{def}{=} (reboot, r_b).Server_{idle}
\end{aligned}$$

- **Hiding.** Written as P/L , hiding allows the activities that are internal to a component to be hidden from view by other components. The activities occur as normal, but to other components, they appear to have an action type of τ , and cannot occur in the cooperation set of the cooperation combinator which is described shortly.

Looking at the batch server model above, it could be argued that whether the server crashes or not is irrelevant to other components — they should just see the rate at which jobs complete and should not depend on implementation details of how the server transitions through internal states. To allow other components to interact with the server only by submitting jobs and receiving the results, a server where processing is hidden is defined using

$$Server \stackrel{def}{=} Server_{idle} / \{process, reboot\}$$

- **Cooperation.** PEPA's sole concurrency mechanism is the cooperation combinator written as $P \bowtie_L Q$. It defines two separate components P and Q running in parallel and that synchronise (or cooperate) on the list of action types in L . When cooperating, the overall (or *apparent*) rate of any shared activities is defined as the rate of the slowest component. The intuitive explanation here is that in a cooperative activity, whichever process is slowest becomes the bottleneck for that activity, such as a producer and consumer scenario, or in an assembly line.

If $L = \emptyset$ the components do not synchronise at all, and run completely independently of each other. For convenience, this $P \bowtie_{\emptyset} Q$ is written as $P \parallel Q$, and multiple instances of the same component can be written as $P[n]$ instead of $P \parallel \dots \parallel P$.

As an example, a job can be defined as

$$Job \stackrel{def}{=} (submit, \top).(complete, \top).Stop$$

Note that, as in the Goldilocks example, the rates of the *submit* and *complete* activities are undefined, making Job *passive* with respect to these activities. This is because the rate at which the job runs depends on the server's resources, not the job itself.² If $L = \{\text{submit}, \text{complete}\}$, then a single job submitted to a server can be represented using

$$\text{Job} \bowtie_L \text{Server}_{\text{idle}}$$

Here, the rates for *submit* and *complete* are defined in one of the components, so the overall rate will be $\min(\tau, r_s) = r_s$ and $\min(\tau, r_c) = r_c$ respectively.

Similarly, n independent jobs submitted to a pool of m independent servers can be described by

$$\text{Job}[n] \bowtie_L \text{Server}_{\text{idle}}[m]$$

Or, a heterogeneous set of machines and different types of jobs, with appropriately defined components, could be expressed as

$$(\text{BLAST}[n_1] \parallel \text{CHARMM}[n_2] \parallel \text{AMBER}[n_3]) \bowtie_L (\text{Opteron}_{\text{idle}}[m_1] \parallel \text{Itanium}_{\text{idle}}[m_2])$$

PEPA models, once built, can be analysed using multiple techniques. Since PEPA is a process algebra and has a formally defined semantics, PEPA models can be checked by machine for various logical properties, such as whether a model is incomplete, or free from deadlock. Two models can also be compared for equivalence using *bisimulation*.

PEPA models with few states can be transformed into a Continuous Time Markov Chain (CTMC). Using the CTMC, PEPA models can be analysed for steady-state or equilibrium behaviour, and by solving the CTMC, transient analysis can be performed to observe the evolution of the system over time. However, for anything other than small models, generating the CTMC is slow and solving it even slower due to the 'state space explosion' caused by having to generate, and then solve a system of linear equations involving every component in every possible state.

Fortunately, it is possible to translate PEPA models directly into a compact system of Ordinary Differential Equations (ODEs) with a size proportional to the number of distinct component types in the model. These ODEs can be solved efficiently using standard numerical techniques, allowing the evolution of the system over time to be examined.

²A more sophisticated model might contain a heterogeneous set of jobs each requiring different types of activity to occur in differing proportions, each activity having different rates defined on the server. This could model different resources such as network bandwidth, disk I/O, CPU etc.

As can be seen from this description, PEPA is far more abstract than either PACE or Prophecy. None of the language features are tailored to the peculiarities of processors and code execution, and it can be used to model things as diverse as software systems [GHLR04], biochemical pathways [CGH04] and peer to peer networking [Dug06].

However, this does not preclude the use of PEPA as a performance prediction tool for grid applications. In particular, if an application is written using algorithmic skeletons, and is structured using the Pipeline and Deal skeletons, it is possible to automatically translate this structure into a PEPA performance model [BCGH05]. This model can then be used along with performance data for a cluster of machines to find an optimal mapping of tasks to compute resources.

2.2 Performance monitoring tools

A typical computational grid may be composed of many compute nodes, each with a continually varying workload and availability. At any time, nodes can fail and will need to be replaced by new nodes, potentially with different capabilities. The amount of traffic flowing across the network infrastructure connecting these nodes will also vary over time due to network outages, changes in topology, and the communications patterns of the workloads themselves. This suggests that for scheduling and fault analysis purposes benchmarking a system once is insufficient. It is important to have a continuous performance monitoring facility as part of the infrastructure of a grid system. This performance monitoring consists of periodically executed probes which usually include measurements of the bandwidth and latency of MPI communications between various nodes, the I/O speed for local discs, and benchmarks of CPU and memory speed.

This continually changing performance can also be used as parameters to application performance models and scheduling systems: a workflow scheduler might decide not to run a particular parallel task on a node that has recently slowed down, as the application is known to perform poorly on heterogenous nodes, or the risk that the node will fail is higher.

One instance of a well known and frequently used end-to-end network benchmarking tool is NetPerf[Jon09]. For the reasons mentioned in the previous paragraphs, a benchmarking tool on its own is insufficient for grid or cluster management. Many grid and cluster monitoring tools exist to aggregate performance information and to facilitate the monitoring of cluster resources. These include popular tools such as Ganglia[Gan], Nagios[Nag], the Network Weather System and the Globus Moni-

toring and Discovery Service. We will examine these last two tools in the following sections:

2.2.1 Network Weather System

Network Weather System ([NWS](#)) is a distributed system for gathering performance data for large sets of computing resources and providing short-term forecasts based on the statistical analysis of the data gathered [[WSH99](#)].

It consists of four core components: a *Name Server* that acts as a directory which maps resource names to IP addresses; a set of *Persistent State Servers* that act as a long term data store for the performance data gathered; a set of *Sensors* that probe for performance data from particular resources and store them on the Persistent State servers; and a *Forecaster* component that uses a mix of statistical techniques to estimate the performance of a resource in the near future.

The Sensors gather two types of performance data. The first relates to compute resources: the Sensor combines the information available from UNIX's `vmstat` and `uptime` along with a periodic [CPU](#)-intensive probe program to calculate how much [CPU](#) is available on a node. To minimise the intrusiveness of this probe, NWS reduces the frequency of the probes when recent availability measurements are approximately static, and increases them as availability changes.

The second set of sensor data is measurement of the end-to-end network performance between pairs of compute nodes using two active probes: one which measures the round-trip time of a small TCP message, and the other using a large message. From this, the bandwidth and latency of a link, along with the time to open a TCP socket can be inferred.

Since there are $O(N^2)$ pairs of connections between N nodes, it would be prohibitive to directly measure the performance characteristics of all possible connections. Even if the number of measurements were limited somehow, with increasing numbers of nodes, the odds of multiple probes occurring simultaneously (and thus distorting the measurements) increases quadratically with the node count. To avoid this, the sensors arrange themselves into a distributed hierarchical system where each sensor belongs to a set of cliques [[WSH99](#)]. At the 'bottom' level, each sensor in a clique measures the inter-node performance with every other sensor in the clique. A token passing system makes sure only one sensor in the clique measures at once.

The cliques are arranged in such a way that one sensor in the 'bottom' level is chosen as representative of that clique, and belongs to a 'higher level' clique too.

Each of these representative sensors measure inter-node performance with each other in turn, and this continues up the hierarchy until the root clique.

The Forecaster component uses the time-stamped data from the Persistent State servers to generate forecasts of the performance data on demand. These are computed using a set of forecasting models to predict the recent and current measurements based on historical data and comparing them with the actual measurements (a technique dubbed *postcasting*). The model with smallest mean squared error is chosen as the best available, and is chosen to predict future performance data [Wol03].

Since the network and compute availability are continually changing, the performance data produced may be a non-stationary series, and limiting the amount of history available to a model may improve the accuracy of the resource predictions. NWS uses models with varying window sizes as part of the resource prediction process, and again uses the model with the smallest mean squared error.

Note that the predictions produced by NWS differ from those of an application model in that they are predictions of future *availability* of compute and network resources, rather than the resource *consumption* characteristics of applications. Both pieces of information are required: an application model cannot produce accurate estimates of an application's runtime if it does not know the resources available for that application to use.

2.2.2 MDS

The Monitoring and Discovery Service (MDS) is a built-in component of Globus that provides a scalable resource information system for grid services. It has evolved along with Globus through four major versions, and is designed to address the needs of grid computing infrastructure including tools such as resource brokers, meta-schedulers, and fault detection systems. These needs fall into the two categories of *monitoring* and *discovery*.

The requirements for MDS, or the Metacomputing Directory Service as it was originally called, were set out in [FFK⁺97], along with an initial early implementation that ran on Globus 1.1.2 and earlier. The paper examined a number of distributed directory systems, such as DNS and X.500 and showed how these were not suitable for grid computing. It determined that an MDS must have high performance and scalability; must have a uniform API and extensible data model; must deal with continually updating data; must be decentralised to allow for multiple information sources and no single points of failure; and must support secure access.

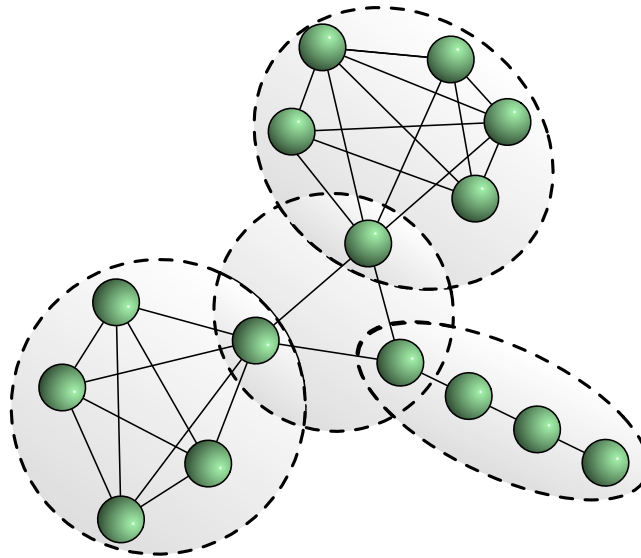


Fig. 2.1: NWS clique hierarchy

A data model and naming system based on that of the Lightweight Directory Access Protocol ([LDAP](#)) was chosen by the Globus Alliance for [MDS](#) version 1. The data model consists of set of elements, each of which has a type called an object class which is defined in a class hierarchy. The object class defines a set of mandatory and optional attributes for each element and what kinds of values each attribute may contain. In the case of [MDS](#), each element represents a grid resource, such as a compute node, or some network infrastructure. An example can be seen in [Fig. 2.2](#).

The naming system is also based on [LDAP](#). Each element has a unique identifier called a Distinguished Name ([DN](#)) which can be thought of as similar to the absolute path to a file. Entries are organised in a Directory Information Tree ([DIT](#)) where each entry is a child to some other entry, and each component in a DN 'path' represents a single specific DIT entry. [Fig. 2.3](#) shows an example distinguished name.

The [MDS](#) could be queried and updated using [LDAP](#)-style queries. For example, the query `(&(objectClass=GlobusHost)(o=University of Warwick)(c=UK)(totalMemory>256000000))` would find all compute nodes at Warwick with more than 256MB of memory.

The first implementation of [MDS](#) consisted of a centralised [LDAP](#) server for each organisation. A utility called `globus-gram-reporter` played a role similar to that of Sensors in NWS and periodically pushed resource information updates to the

```

GlobusHost OBJECT CLASS
SUBCLASS OF GlobusResource
MUST CONTAIN {
  hostName :: cis,
  CPU :: cis,
  OS :: cis,
}
MAY CONTAIN {
  totalMemory :: cis,
  totalSwap :: cis,
  dataCache :: cis,
  instructionCache :: cis
}

```

Fig. 2.2: LDAP object class

```

<
  hn = groupthink.dcs.warwick.ac.uk
  ou = HPSG,
  o = Department of Computer Science,
  o = University of Warwick,
  c = UK
>

```

Fig. 2.3: LDAP distinguished name

server, and clients queried the server via the standard [LDAP](#) mechanisms. This system could not scale and any failure of the server led to a complete loss of information services.

[MDS](#) v.2 expanded on version 1 by defining two classes of agent in [MDS](#), the Grid Resource Information Service ([GRIS](#)) and the Grid Index Information Service ([GIIS](#)). The GRISes are a more sophisticated version of the server in [MDS](#) version 1. GRISes use a set of Information Providers agents to collect data. This can be static information, periodically updated information, or information that is generated on demand by queries. GRISes are queried using the Grid Information Protocol ([GRIP](#)).

To reduce the load on a GRIS, an organisation may have multiple GIISes. These serve the role of both caches and aggregators of the information contained in a GRIS, but originate no information of their own. To connect a GRIS to a GIIS, the GRIS notifies the GIIS of what resource information it serves using the Grid Registration Protocol ([GRRP](#)).

Inspired by registration in NWS, GRRP is a ‘soft-state’ protocol, meaning that the GRIS must re-register with the GIIS on a regular basis: if it does not, the GIIS assumes the GRIS has failed or no longer exists and can remove cached information belonging it. Like GRISes, GIISes can be queried for resource information using GRIP, and if a GIIS lacks the information required, it may pass the query on to the original GRIS. GIISes can in turn register some or all of its resources with one or more other GIISes, and this can be used for load balancing or redundancy.

In a reversal of roles, GRRP can be used in the opposite direction too. A GIIS may use GRRP to invite a GRIS to join a ‘virtual organisation’, where a set of agents from many organisations can access a *view*, i.e., a subset of resources from many GRIS/GIISes. To enforce the restrictions on the view, GRIP can require encryption

and authentication of the connection using GSI.

Globus Toolkit 3 and 4 represented a major change in the protocols and implementations underlying all of Globus. Prior to version 3, the Globus Toolkit consisted of a collection of disparate programs and libraries that communicated using a number of ad hoc protocols. Globus Toolkit 3 replaced this with a set of web services running in a Java container that communicate using Simple Object Access Protocol (SOAP). Plain web services are missing important functionality required by grid service applications, such as the ability for each service to hold stateful data. To overcome this, Globus defined a set of standardised extensions that all grid services may use. The extended model is known as the Open Grid Services Infrastructure (OGSI) [TCF⁺03]. In OGSI every service running in the container has its own set of serviceData elements describing its internal state in XML and these can either be queried directly via SOAP messages, or remote agents can subscribe to receive periodic notification events from the container. Thus the push and pull information services of a GRIS are built directly into the Globus container in Globus Toolkit 3.

To complete the mapping of MDS GRIS functionality to grid services, Globus Toolkit 3 has a Provider mechanism that allows external scripts to become serviceData providers. This is similar to the Information Providers functionality in MDS v.2. Globus Toolkit 3's version of Grid Resource Allocation Manager (GRAM) uses this Provider mechanism to provide information about cluster hosts represented using the GLUE XML schema. Using OGSI's queryByXPath, the example MDS v.2 query above might become `/Cluster[@Name="warwick.ac.uk"]/SubCluster/Host[MainMemory/@RAMAvailable>256]`.

To provide the equivalent of a GIIS, Globus Toolkit 3 relies on an OGSI feature called a 'service group'. This allows a single grid service to represent and provide information for a group of other services. Globus Toolkit 3 ships with a service called the Index Service which uses service groups to provide the aggregation and caching functionality of MDS v.2.

After Globus Toolkit 3 was released, the web services community defined a collection of web services specifications collectively known as WS-Resource Framework (WSRF). These address similar concerns to those of OGSI, but with different terminology and as a modular set of Web Services specifications. For example, the WS-ResourceProperties [WS-03] specification contains the statefulness extensions of OGSI, WS-Notification [WS-04a] specifies push-based remote notification, WS-ServiceGroup [WS-04b] defines service groups extensions.

Globus Toolkit 4 is a refactoring of Globus Toolkit 3 to use WSRF [CFF⁺04]. Since

WSRF and OGSi are architecturally similar, the basic resource information and indexing services in MDS v.4 are essentially the same as MDS v.3.

MDS v.4 introduces one major new feature over MDS v.3, namely the Trigger service. This allows aggregators using the Index service to monitor incoming service data, and if a piece of data matches a rule, then an external script is executed. This could be used, for instance, to email a sysadmin if a host fails; or to reconfigure systems in response to varying load.

2.3 An analytical performance model of Sweep3D

The US Department of Energy benchmark Sweep3D and the analytical model built for it by Mudalige et al. serves as an example of the benefits of and difficulties in construction an analytical performance model. [MVJ08]

Sweep3D is a discrete ordinates neutron transport code designed to be representative of the majority of the application workload at sites such as the Los Alamos National Laboratory. In common with a number of other codes, it uses a parallel execution strategy based on a parallel wavefront decomposition. A typical run of Sweep3D will use a grid of between roughly 50^3 cells and 250^3 cells.

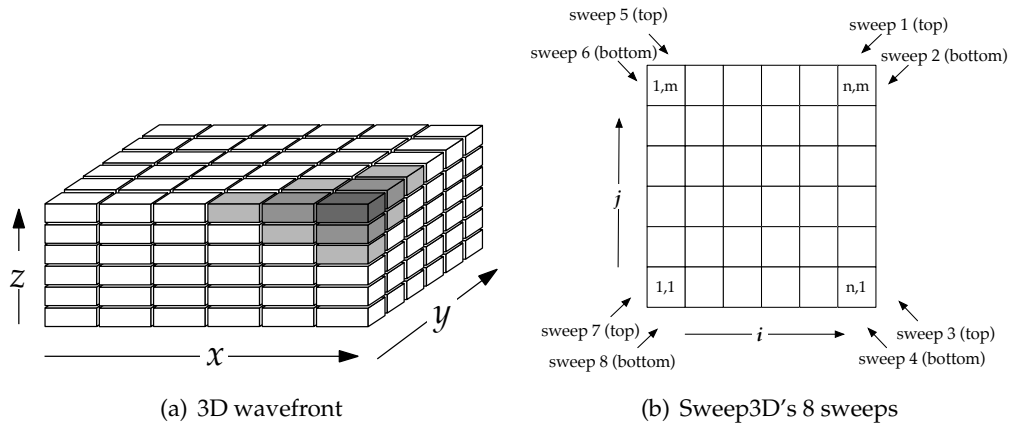


Fig. 2.4: Sweep3D wavefront behaviour

This wavefront method is based on the hyperplanes technique developed by Lamport. [Lam74] Sweep3D operates on an array of data of size $N_x \times N_y \times N_z$ and the data is split between processors by assigning the processors to a 2-D grid of size $n \times m$ and allocating a vertical 'stack' of cells of size $N_x/n \times N_y/m \times N_z$ to each processor. The algorithm operates by beginning computation at one corner of the data and sweeping through the array until it reaches the diagonally opposite corner. This is

implemented by processor (n,m) taking its topmost $N_x \times N_y \times 1$ tile and performing calculations on this block of data. When the results are ready, it passes the southern and eastern boundary cells of the updated block to the processors to the south and east of it, that is to say processors $(n-1,m)$ and $(n,m-1)$. Then processor (n,m) continues and performs calculations on the second tile in the vertical stack while its neighbours calculate the topmost tile of their stack of cells. When processor (n,m) completes N_z steps, it has finished its processing and another $m+n-1$ steps must pass before processor $(1,1)$ completes its final tile.

This leads to a pipelined effect illustrated in Fig. 2.4(a) where a diagonal wavefront of computation passes through the 2-D grid of processors. A processor cannot begin computation until the wavefront reaches it—the so called ‘pipeline fill’ delay—and the ‘width’ of the wavefront is the same as N_z , the height of the stack of tiles.

In a single iteration of Sweep3D, 8 sweeps occur, one for each corner of the 3-D data set. The order of these sweeps is shown in Fig. 2.4(b). The particular choice of sweep ordering has consequences for the pipelining behaviour. For example, immediately after processor (n,m) has completed sweep 1, it can immediately start sweep 2 because it already has all the data it needs. The wavefront for sweep 2 propagates across the 2-D grid of processors in exactly the same order as sweep 1, except the processors perform computations on their stack of tiles from bottom to top, rather than top to bottom.

However, for sweep 3 to begin, processor $(n,1)$ must first finish processing sweep 2, introducing a delay of m iterations for the pipeline fill between these two sweeps. Similar delays occur between sweeps 4 and 5, and between 6 and 7, and finally for sweep 8 to complete.

Other codes such as LU matrix decomposition and the Atomic Weapons Establishment’s Chimaera use the wavefront algorithm, but have different numbers of sweeps (2 in the case of LU), or order the sweeps differently (as in Chimaera).

As with most analytical models, the Sweep3D model in [MVJ08] has separate parts for the computation and communications. In Sweep3D, the pattern of computation is relatively simple: a processor simply performs a fixed set of calculations on a 2-D tile from the 3-D array after receiving boundary data from its northern and western neighbours. The runtime for this set of calculations is simply measured by benchmarking it for a representative tile size on the target hardware yielding a model parameter called W_g which is the average time to compute one cell. This benchmarking only needs to be performed on a single processor, as it only affected by CPU speed and the memory subsystem. Other applications such as LU per-

form some per-tile calculations before receiving data from its neighbours. This is measured in the same way and yields the parameter $W_{g,pre}$.

The other component of the model accounts for the delays due to communications. To allow this to be done, two problems need to be solved: the performance of the network on the target hardware needs to be modelled, and, independently, the communications patterns of the application need to be characterised.

The network and communications model is based on the *LogGP* methodology. This characterises communications with the following five terms:

L is the maximum latency which can be expected for an individual message.

o is the communications overhead, i.e., the amount of CPU time taken to process a message.

g is the gap between consecutive messages, which is a function of bandwidth and how messages are buffered.

G is the gap per byte for large messages, which is the inverse of bandwidth.

P is the number of processors.

The *LogGP* model ignores the effects of contention — the slowdown that occurs when multiple processors try to use the same links at once. This is not a significant problem for the Sweep3D model, as by its nature Sweep3D processors only ever communicate with their neighbours, there are no overlapping communications that use the same links, and the 2-D grid of the decomposition maps directly onto the 3-D links of the hardware under consideration.

The network model is for that of MPI on a Cray XT4, which provides a 3-D torus interconnect between processing elements, where each processing element is a dual core Opteron. MPI running on the interconnect uses a direct send protocol for small messages (up to 1 kB) and an acknowledgement based protocol for large messages (> 1 kB). MPI communication between processors on the same chip use memcpy between memory buffers for messages up to 1 kB and DMA for larger messages. The DMA takes longer to set up, but is much more efficient per-byte. For the Cray, g is found to be zero due to the hardware design, and measurements are made of L , o , and G for all four of the large message, small message, off-chip and on-chip permutations. These are combined into a communications model with a number of equations, two of which are

$$Total_Comm_{\leq 1\text{ kB off chip}} = o + Message_size \times G + L + o$$

$$Send_{>1\text{ kB on chip}} = o = o_{copy} + o_{DMA}$$

The communications model uses the network model just described as well as a number of model parameters. Two of them, W_g and $W_{g,pre}$ have already been mentioned. Another, n_{sweeps} specifies the number of sweeps in an iteration. Two more, n_{full} and n_{diag} capture the effect of some sweeps needing to wait for the full previous sweep needing to complete, or part of the previous sweep to complete before they begin.

The initial model derived assumes only 1 processor per compute node, thus neglecting the effects of on-chip communications and contention for the network interface. It consists of a set of recurrence relations for the start time for a sweep on a given node. Some of the equations are:

$$W = W_g \times H_{tile} \times N_x/n \times N_y/m$$

$$StartP_{1,1} = W_{pre}$$

$$StartP_{i,j} = \max(StartP_{i-1,j} + W_{i-1,j} + Total_comm_E + Receive_N, \\ StartP_{i,j-1} + W_{i,j-1} + Send_E + Total_Comm_S)$$

$$T_{diagfill} = StartP_{1,m}$$

$$T_{fullfill} = StartP_{n,m}$$

$$T_{stack} = (Receive_W + Receive_N + W + Send_E + Send + W_{pre})N_z/H_{tile} - W_{pre}$$

$$Time\ per\ iteration = n_{diag}T_{diagfill} + n_{full}T_{fullfill} + n_{sweeps}T_{stack} + T_{nonwavefront}$$

The equations are quick to solve by computer for any practical $n \times m$ grid size, and have a number of easily changed parameters. The models have been tested against a number of real world scenarios, and have a high degree of accuracy with a 10% error even on a system unfavourable to the model. More importantly, the equations can be intuitively understood and can be modified to examine what the behaviour of Sweep3D and other wavefront codes would be with various platform changes or with modifications to the software design and configuration. One change examined later in [MVJ08] shows how by changing the expression for $StartP_{i,j}$ a simplification in the initial model can be resolved and a model for systems (such as the Cray XT4) with 2 processor cores per network interface can be derived. Other experiments investigate Sweep3D's scalability as the processor count increases, and the effect of

application parameters such as H_{tile} on performance.

A later work investigates the effects that a mixed OpenMP/MPI implementation would have on Sweep3D, as well as the potential for speedup by running different sweeps on different CPUs in parallel on chip multiprocessor systems. The model allows various changes to be speculated upon and tested for their potential benefits without requiring a lengthy and error prone rewrite of the code itself. [MHSJ09]

As can be seen in the above, an accurate analytical performance model allows many speculative scenarios to be investigated. However building the model requires a deep understanding of the application's behaviour, which becomes increasingly difficult as applications grow in size and complexity. For Sweep3D, model creation has been in part possible because it has a deliberately uniform structure, and sets out to avoid performance reducing effects such as network contention which also happen to be difficult to model.

2.4 Scheduling as an application

For grid applications to execute efficiently they rely on middleware services to manage resources and allocate them to tasks among variable and unpredictable application workloads. Many tools exist to model application performance based on performance data and other tools gather and store performance data. Used together, these tools can be part of a scheduling system that attempts to allocate applications to resources subject to various constraints as efficiently as possible.

Previous performance work at Warwick has focused on performance prediction with the PACE framework described in Sec. 2.1.2 and Appendix B and an adaptable scheduling system called TITAN [SJC⁺03]. The PACE tools assist the user in generating analytical performance models, and the evaluation of these models allows the performance and scalability of parallel applications to be estimated on different architectures. A recent example of this can be found in [MJSN06].

TITAN is a multi-cluster scheduling system that uses a rapid genetic algorithm and application performance models to create schedules in real-time. [SJC⁺03] In order to scale well, a complete TITAN system may consist of a number of distribution brokers in a loosely hierarchical structure which pass tasks between each other and determine whether any workload manager within the hierarchy has the available resources and meets any Quality of Service (QoS) criteria. The individual workload managers are responsible for a local pool of resources and interact with local batch schedulers for job submission and monitoring services. The workload managers

also query local performance prediction engines such as PACE for performance data. This section will ignore most of these components and focus on the workflow manager itself.

In contrast to planning-based scheduling systems such as OpenCCS[Ope] which plan the start times for each task, and produces a complete schedule for the future, TITAN's workflow manager uses a queuing system, and simply tries to use available resources as efficiently as possible. The workflow manager uses a genetic algorithm to explore the task mappings required to minimise the overall runtime of all tasks, resource idle time, and average delay of a schedule. The genetic algorithm achieves this by continually generating scheduling solution sets, and replacing the current best schedule as it discovers improved schedules.

The scheduling problem is one where a set of tasks $T = \{T_1, T_2, \dots, T_n\}$ is mapped onto a set of hosts $H = \{H_1, H_2, \dots, H_m\}$. The tasks are considered to be independent of each other and are submitted to run in some order $\ell \in P(T)$ where $P(T)$ is the set of all possible permutations of T . For each task ℓ_j within this ordering, there is a mapping of that task onto one or more hosts β_j , where $\beta_j \subseteq H$ and $\beta_j \neq \emptyset$. A schedule consists of a 2-D matrix where the rows are the available hosts, and the columns the ordered tasks. S_k as defined below is the k^{th} schedule in a set of schedules $S = S_1, \dots, S_p$ manipulated by the genetic algorithm. The columns of the matrix are simply another way of representing β_j as a bitmap where an element in the column is 1 if the task is allocated to a particular host, and 0 if it isn't.

$$M_{i,j} = \begin{cases} 1, & \text{if } H_i \in \beta_j \\ 0, & \text{if } H_i \notin \beta_j \end{cases}$$

$$S_k = \left(\begin{array}{c|cccc} & \ell_1 & \ell_2 & \dots & \ell_n \\ \hline H_0 & M_{0,0} & M_{0,1} & \dots & M_{0,n} \\ H_1 & M_{1,0} & M_{1,1} & \dots & M_{1,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ H_m & M_{m,0} & M_{m,1} & \dots & M_{m,n} \end{array} \right)$$

One of the unintuitive aspects of this definition of a schedule is that it makes no reference to time. Also it appears at first glance to be impossible to specify two or more tasks to run concurrently on different hosts. However this appearance is misleading. The reason for this is that the schedule isn't a schedule in the sense of being a timetable, rather it simply defines the order in which tasks are submitted

to a batch queue, and a host mapping for each task. If, for example, if a schedule has two tasks, T_1 and T_2 , and ℓ_1 and ℓ_2 specify a non-overlapping set of hosts, then when T_1 and T_2 are submitted to an empty batch queue, they will both start executing concurrently. If ℓ_1 and ℓ_2 *do* share a subset of hosts, then T_2 will not start executing until T_1 completes and frees up the hosts they have in common.

This representation has a major benefit in the context of a genetic algorithm. When the genetic algorithm randomly permutes or changes the schedule, it is impossible for it to invalidate the schedule by producing overlapping host requirements: tasks are simply submitted to run on a specific set of hosts, and they begin to execute as soon as they are able.

Some means of knowing how long a task will take to complete, and how well it scales to different numbers of hosts is needed to convert the schedule to one where time is involved. This is provided by the performance prediction engine, which provides runtimes for every query $p_{ext}(\ell_j, \beta_j)$ where ℓ_j is a task and β_j the host allocation.

The end time for a task is te_j , and the start time ts_j . tr_{ji} is the earliest *release time* possible for task ℓ_j on an individual host H_i independent of all the other hosts. These are defined as

$$te_j = ts_j + p_{ext}(\ell_j, \beta_j)$$

$$ts_j = \max_{i \text{ st. } H_i \in \beta_j} tr_{ji}$$

tr_{ji} can be defined as a recurrence relation on the release time for all the earlier tasks $\ell_1 \dots \ell_{j-1}$ running on the same host. The release time for ℓ_1 by definition is the current time t . The maximum in the definition of ts_j occurs because the job can only be released when all the hosts for the task become available, and this occurs at the latest release time for an individual host.

The genetic algorithm operates in an iterative fashion by measuring candidate schedules using fitness functions. The best schedule becomes the one used by the scheduler, and the poorest schedules are removed and replaced by new ones created by crossover and mutation. Crossover takes a random pair of schedules and mixes them together producing a new schedule, whereas mutation produces a new schedule by randomising the contents of an existing schedule.

Crossover of task orders takes task orders ℓ^A and ℓ^B from two random schedules

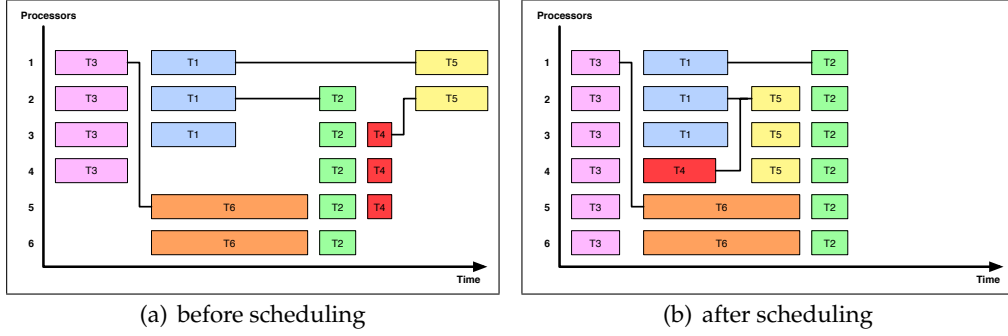


Fig. 2.5: Run-time schedule of three example workflows

and splices them together at some random point i so that $\ell'_1 \dots \ell'_i$ are from ℓ^A and $\ell'_{i+1} \dots \ell'_n$ from ℓ^B . This can produce invalid task orders with duplicated tasks (which is not permitted for a permutation of T), and this can be resolved by repeatedly replacing the first occurrence of each duplicated task by the original task from ℓ^A until no duplications remain. Mutation of a task order simply swaps random tasks within ℓ , which is always guaranteed to produce a valid task order.

Crossover of hostmaps β^A and β^B performs a similar splice operation to crossover of task orders. Mutation of a hostmap simply flips random bits within a hostmap β . After crossover and mutation, hostmaps are then checked for ‘topology’ constraints such as a task being allocated to no hosts, or a task running on a subset of hosts that is not permitted for some reason.

The fitness functions include minimising the makespan, minimising idle time, and minimising the over deadline time of any te_j .

Because it is based on a continually iterating genetic algorithm, TITAN’s workflow manager copes well with change. Due to the schedule representation, addition or removal of tasks does not invalidate a schedule, and the genetic algorithm quickly adapts the schedule to remove any idle gaps that emerge in such an event. Similarly, the genetic algorithm can adapt to imprecise performance estimations, such as if a task completes earlier or later than expected, or if, for some reason the performance estimate *changes*.

By the same token, a schedule can adapt to news from a performance monitoring tool that hosts have been added or removed from a cluster. Added hosts simply appear as a new empty host in each of the schedules S_k . Failed hosts are likewise removed from the schedule, and each β_j has the failed hosts removed from their hostmaps. Any tasks running on the hosts while they failed are simply resubmitted

for scheduling.

Subsequent work on TITAN added the ability to schedule workflows by allowing dependencies to be considered across a process flow. [SCJ⁺04] This can be accommodated by adding an extra restriction to ts_j that it cannot start before any of the tasks it depends on complete.

Scheduling entire workflows differs from scheduling individual tasks in that the tasks within a workflow will usually have dependencies that introduce a certain amount of serialisation between tasks in the workflow. When multiple workflows occur in a schedule, and their descriptions request the scheduler to allocate as many CPUs as possible to each task, this can lead to large amounts of idle time in the schedule. A scheduler that is aware of the scalability properties of an application has substantial opportunities to optimise these kinds of inefficient schedules.

Fig. 2.5(a) and Fig. 2.5(b) show a simple example of how the scheduler can optimise a schedule by allocating individual subtasks to fewer or more CPUs as appropriate, and by interleaving the subtasks from different workflows. Furthermore, interleaving tasks is an optimisation that mutating a task order is likely to find. Fig. 2.5(a) shows the time composition of three workflows as submitted to the system prior to scheduling. Fig. 2.5(b) shows the same three workflows after scheduling by the resource management system. It can be seen that the makespan of all the tasks has been reduced and the utilisation of resources has increased significantly, i.e., the idle white-space in the schedule diagram is reduced.

2.5 Summary

In this chapter a number of performance monitoring tools which gather performance data on the hardware resources in a system, and performance modelling tools which aid in modelling software performance on this hardware have been examined. Both these sets of tools are used in concert by schedulers such as TITAN to attempt to find the most efficient mapping of software tasks to the available resources.

To allow performance models of software to be built many assumptions are made. On the micro-architectural level, PACE makes assumptions that are thwarted by branch prediction, caches, and optimising compilers amongst other factors. Systems such as WARPP and Prophecy avoid this by benchmarking basic blocks. However on a higher level all these systems assume that it is enough to model the average runtime of an application, ignoring the effect of input data on the applica-

tion. WARPP makes some allowance for certain kinds of variability by modelling the slowdowns introduced by random system noise, but again does not model data-dependency.

Even though none of the systems attempt to model the performance variations introduced by changing an application's input data, it can cause major changes in runtime. One obvious source of this variation is due to changes in the amount of work an application does — it might require more iterations to reach convergence, or it might operate differently on different input data. A less obvious source are micro-architectural variations: an application might execute the same code sequences, but trigger a slow case in the hardware implementation of some operation.

In the following chapters both of these sources of data-dependent runtime variability will be examined. I find a data-dependent algorithmic slowdown in a medical imaging application, *nreg*, and micro-architectural slowdowns caused by denormal arithmetic in other applications. I will show how these effects can be detected and either modelled or removed.

Performance modelling and scheduling of a data-dependent code

This chapter presents a parallel implementation, a predictive performance model, and an efficient scheduler for a biomedical imaging application in the UK e-Science IXI (Information eXtraction from Images) project [HHL⁺03]. The IXI project [HHL⁺03] demonstrates how grid-computing technologies can be used to enable large scale image processing and medical image analysis. Connectors in IXI's workflow manager can submit jobs to dedicated clusters, Condor-managed workstations, or to the National Grid Service.

The application considered is `nreg`, a 3D non-rigid registration tool. This application exhibits highly variable runtimes depending on the specific input data provided. Since the runtime can vary by a factor of 50, without a performance model it was challenging to apply meaningful quality of service criteria to workflows that use this code. The model developed here is used in the context of an interactive scheduling system which provides rapid feedback to users, allowing them to tailor their workloads to available resources, or to allocate extra resources to scheduled workloads.

Sec. 2.1.2 described a performance prediction toolkit, PACE, and discussed how it models application performance. The `nreg` codebase has several properties that makes it unsuitable to model with PACE. As a modern C++ application, `nreg` makes heavy use of inheritance and templates. This alone presents difficulties as PACE's `capp` cannot parse C++ source code. Furthermore, because of the C++ features used, `nreg` relies heavily on aggressive compiler optimisations to achieve acceptable performance. Since PACE works at the level of source code, it cannot model the performance improvements that result from template specialisation, inlining, dead code elimination and the like.

WARPP discussed in Sec. 2.1.3 could be used sidestep some these issues by instrumenting the important methods in the application. However this would need to be done manually and with some care, as some of the inlined image processing methods are called a huge number of times during a typical execution, and instrumentation in the wrong place could remove some of the compiler optimisations

thus distorting the benchmarks. Furthermore, the flow control of the application is made less obvious by the use of inheritance and template specialisation.

Most importantly, quite apart from language-level details, the algorithm used by `nreg` is sensitive to the information content of the input data, and exhibits highly variable and data-dependent runtimes as a result. Neither PACE nor WARPP are capable of modelling this effect.

`nreg`'s algorithms are discussed, and I produce a parallelisation strategy that scales reasonably up to about 16 nodes. I analyse the cost of each part of `nreg` and identify the sources of variability. Although tools like PACE and WARPP cannot be used to characterise this variability, I produce a pre-execution performance model for `nreg`.

This model is not analytical, and takes a substantial amount of time to execute. It also can be tuned to trade predictive accuracy for execution speed and can emit a sequence of increasingly accurate predictions over time, a feature called *incremental prediction*. I adapt TITAN to work with performance models of this kind.

This performance model along with a feature of TITAN's called *speculative scheduling* can be used either to directly schedule tasks for execution, or to evaluate the implications of adding extra tasks to an existing workflow.

Speculative scheduling allows the end user to dynamically construct workflows and uses the scheduling and prediction systems to provide estimates on how long the workflow would take to complete if it were scheduled to run along with the scheduler's current mix of tasks. The user can then examine this schedule and decide whether it meets their requirements. If it does not, they can edit the workflow as they see fit, either in the hope of doing extra useful work, or of getting usable results back more quickly. As the workflow changes, the predicted schedule updates with it and this *closing the loop* between the user, application and scheduler allows the both the user and scheduler to provide valuable feedback to each other that would not otherwise be available.

3.1 `nreg` Image Registration

`nreg` is a medical imaging tool [RSH⁺99] used to perform non-rigid registration on pairs of 3D MRI scans. Registration is the process of aligning two images so that the corresponding features in both are in the same location. It is used in medical imaging because pairs of images are often compared or displayed together using image processing techniques, and these produce meaningless results if the images aren't aligned properly. An example of two such images is Fig. 3.1 from a set of

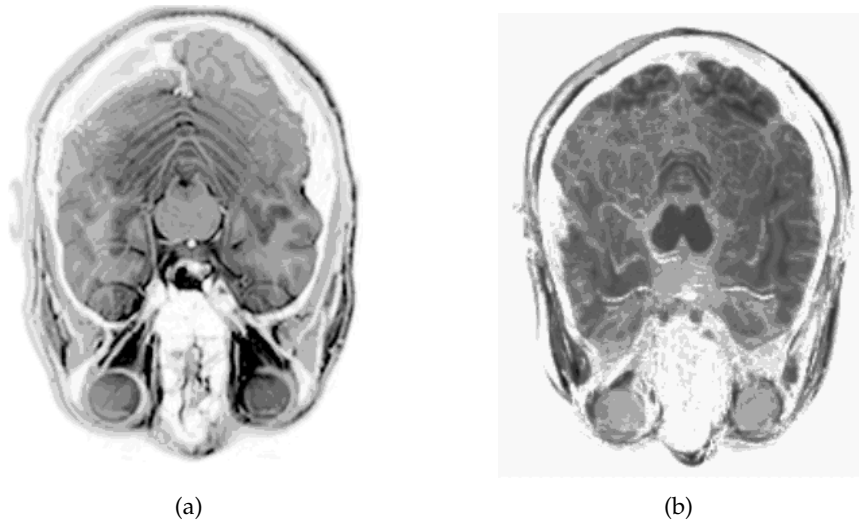


Fig. 3.1: Slices from brain scans in need of registration

brain scans made using different MRI imaging techniques. `nreg` differs from other registration algorithms in that it uses a mesh of B-splines to capture both local deformation and global motion between the two images and its similarity measure is based on normalised mutual information which allows it to align images from different MRI modalities such as CT, MR and PET. It has been shown to be highly effective at compensating for misregistration in breast MR images and for isolating tumour growth for visualisation purposes.

The use of B-splines has desirable numerical attributes, including smoothness, continuity, and the property that moving a control point only affects the transformation in the local neighbourhood of the control point, making it computationally tractable to use large numbers of points. An example of a 2-D version of the transformation can be seen in [Fig. 3.2](#).

3.2 The `nreg` algorithm

The core of the algorithm is an optimiser based on gradient descent for a system with about 1,000 degrees of freedom (three for each control point). The algorithm fits a uniform mesh of control points over the 3D image. A function called `EvaluateDerivative` takes one of the x , y and z axes of a control point and experimentally moves it by a given step-size, and measures the effect of this individual motion on the transformation using a similarity function. The effect is measured with both positive and negative movements, and the difference between them is a discrete approximation of the partial derivative for this degree of freedom. A func-

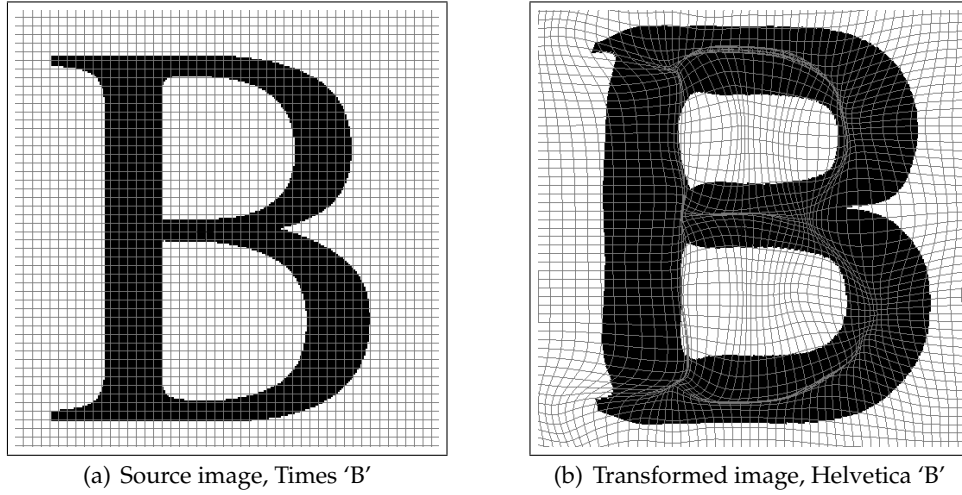


Fig. 3.2: Transformation of 2-D image by nreg

tion named `EvaluateGradient` calls `EvaluateDerivative` for each control point and axis using a specific step size, and stores the results in a gradient vector. The magnitude of this vector is calculated, and the vector itself normalised.

Another routine, the gradient descent optimiser, calls `EvaluateGradient` with a given step size. If the magnitude of the gradient is sufficiently large, the optimiser assumes it can improve the similarity of the two images by moving along this gradient. It repeatedly does this and uses `Evaluate` to measure the similarity of the newly transformed image each time, until the improvement in similarity drops below a threshold.

This gradient descent optimisation process loops, causing a new gradient to be found and followed each time until a maximum iteration count is reached, or the similarity improvement is sufficiently small. The gradient descent loop is then repeated for a fixed number of steps, halving the motion step size each time, and this completes the registration process.

To account for global motion as well as local deformations, this entire registration process is first performed on a coarse mesh with a low resolution image, and then refining the mesh and increasing the image resolution. These different resolutions are called levels in the application. After each registration, the image resolution is doubled in each axis, and the number of points in the mesh is doubled in each dimension using a B-spline subdivision algorithm to generate the new points.

The similarity measure consists of two components: the first imposes some penalties on the transformations to make sure they are well formed. For example, one penalty

encourages smoother transformations by calculating a 3D analogue of an equation that describes the bending energy of a thin sheet. Transformations that require more bending energy (and so have more local irregularities) are penalised. Calculating this penalty involves a small fixed number of floating point calculations for each control point and its immediate neighbours.

The second part of the measure quantifies the similarity between two images using the normalised mutual information of the two images. It is implemented as a 2D histogram. When a new transform is generated, the grey values of corresponding pixels in the source and transformed target images are read and the count for that pair of values is incremented in the histogram. When divided by the total number of samples in the histogram, each entry in the histogram gives the probability of a particular pair of values occurring in the source and transformed target images. When a control point is moved, the old sample values are removed from the histogram, and new values added. By summing over all the samples in the histogram, the conditional entropy, joint entropy, and thus the normalised mutual information $I(X; Y)$ are calculated as follows:

$$H(Y|X) = - \sum_{x \in X} \left(\sum_{y \in Y} p(x, y) \log \sum_{y \in Y} p(x, y) \right) \quad (3.1)$$

$$H(X|Y) = - \sum_{y \in Y} \left(\sum_{x \in X} p(x, y) \log \sum_{x \in X} p(x, y) \right) \quad (3.2)$$

$$H(X, Y) = - \sum_{x \in X} \sum_{y \in Y} p(x, y) \log p(x, y) \quad (3.3)$$

$$I(X; Y) = \frac{H(Y|X) + H(X|Y)}{H(X, Y)} \quad (3.4)$$

3.3 nreg's computational costs

The runtime of `nreg` is limited by the speed of the [CPU](#) and main memory. Upon initialisation `nreg` reads two image files into main memory and constructs the set of subsampled images that are used later in the registration process. For images that take a long time to register this setup phase takes a negligible proportion of the total runtime. Other than these reads and the output of the final transformation parameters no disk I/O occurs.

Several factors account for the bulk of the runtime of `nreg`:

1. As the number of voxels in the image increases, the runtime increases also. Calculating the image similarity, which is done 6 times for the B-spline patches around each control point in `EvaluateDerivative`, and once for the image overall using `Evaluate` in the gradient descent, involves transforming the image and updating the histograms, which has a per voxel cost.
2. As the number of control points increases, the number of voxels moved by a control point decreases in proportion to the number of control points, so the amount of work to transform each voxel stays approximately constant.
3. As the number of control points increases, the overall proportion of the runtime spent calculating the normalised mutual information increases. The 2D histogram has a fixed size, and the cost of calculating the NMI is independent of the number of samples added to it — calculating the NMI for a newly initialised histogram takes exactly the same time as one with samples from all the voxels from both images. Since a temporary histogram is created and its NMI is calculated 6 times per control point in `EvaluateGradient`, this cost increases in proportion to the number of control points.
4. All else being equal, the runtime is proportional to the number of times the gradient descent optimisation is called. However this must be measured carefully: the algorithm performs registrations at different image and mesh resolutions, and an iteration using a finer mesh and resolution takes longer than one using a coarser mesh and lower resolution.
5. For each call of `EvaluateDerivative`, there is a highly variable cost associated with gathering the sample data, i.e., interpolating and transforming local patches of the target image. This occurs because patch sampling loops over every voxel in the bounding box of the patches moved by a given control point. When a patch is not approximately cuboid in shape or is not aligned with the x , y and z axes, this leads many more voxels in the bounding box than those that properly belong to the patch. As a result the bounding boxes for different patches overlap, sometimes substantially, leading to the same voxels being read several times during an iteration of `EvaluateGradient`.

The first two factors are predictable and can be reduced to simple analytical expressions.

Factor 3 exhibits some variance depending on the input images, but it is several times less than the large differences in runtime in [Table 3.1](#).

Target	Source	Iterations	Runtime	Samples
b7_s2	b7_s2	5/5/5	3,863 s	4.7×10^9
b7_s2	b7_s1	19/13/21	14,210 s	18.9×10^9
b9_s2	b9_s1	26/29/31	26,940 s	30.9×10^9
b9_s4_e2	b9_s3_e2	49/45/47	4,284 s	3.9×10^9
b9_s3_e2	b8_s3_e2	47/38/35	3,515 s	4.1×10^9

Table 3.1: EvaluateGradient iterations

Intuitively, factor 4 seems very difficult to predict, as it is related to the overall difficulty in matching the two images. However, as can be seen in Table 3.1, although it has a strong correlation to the runtime, variation in the number of iterations can often be masked by other effects, and the bulk of the variation comes from factor 5.

Table 3.2 was generated by running a number of sample registrations under the profiling simulator Callgrind [JWT04]. Callgrind is a tool in the Valgrind binary instrumentation framework which is used again in Sec. 5. Callgrind uses Valgrind’s x86 CPU emulation to execute user processes, and each memory access is instrumented and fed into a model of the level 1 and level 2 CPU caches. Callgrind uses this model to estimate the per-function cost of running a code on a particular memory hierarchy. The table shows that while the cost of calculating the statistics is quite stable, the cost of calculating the transformed voxel values resulting from moving a control point is highly variable: in some images it dominates the runtime, for other images it is a much smaller factor. The table shows only average costs across the entire execution of the program, but when parallelising nreg it was found that the cost of EvaluateDerivative varies widely from one control point to another.

3.4 Parallelising nreg

Since the runtime of nreg can be extensive (tens of hours), it may be desirable to have a parallel version of nreg. Despite the inefficiencies introduced by parallelisation overheads such as duplicated computations, or delays waiting on inter-node communications, a parallel implementation can improve turnaround and resource utilisation in cases where a user has many free machines and wishes to run small registration workflows. Depending on the requirements of a workflow, a mix of sequential and parallel tasks may provide the best overall use of the system. For example, when a workflow consists of a group of registration tasks, each could be allocated CPUs in proportion to their expected runtime to balance the workload more

evenly and to avoid scenarios where one slow process delays the overall makespan unnecessarily. Similarly, a high priority individual task could be allocated a number of CPUs to ensure it meets a deadline.

The processing performed by `nreg` as described in the previous section is almost entirely CPU and memory bound, and this lack of I/O removes one potential barrier to a scalable parallelisation. On the other hand, the internal workings of `nreg`'s algorithm appear less amenable to parallelisation: they involve an unknown number of iterations of mesh transformation, each depending on the last. Furthermore this occurs at several different step-sizes and resolutions.

However, the amount of work done inside each iteration is substantial. It involves the experimental movement of thousands of control points and, as observed before, the use of B-splines means that the movement of control points only affects voxels in the vicinity of those control points. The effect of each of these movements can be calculated independently and thus in parallel. The simplest parallel decomposition would involve dividing the work into N equal pieces and handing one to each CPU. This decomposition is simple and needs no communications to arrange, but is inefficient on heterogeneous systems or systems with varying load: the slowest CPU limits the overall runtime. This analysis also shows that the variable cost of gathering the image statistics means different amounts of work occur per candidate move, making it impossible to statically divide up the workload into equal chunks. This leads to a first parallelisation technique: a simple master/slave decomposition of the workload. Approximately 95% of the application's runtime is spent in a very small part of the code — `EvaluateDerivative` called repeatedly from `EvaluateGradient`. The remainder of the code runs identically and in lockstep on each CPU. When the partial derivative loop is reached in `EvaluateGradient`, one CPU (the master), instructs each of the slave CPUs to perform a small batch of `EvaluateDerivative` calls independently. When this work completes, the results are passed back to the master and more work is received. When all the partial derivatives for an iteration have been completed, the master distributes all the results to all the CPUs. This involves a broadcast of the gradient vector, a small data structure. Then every CPU returns to 'lockstep mode', executing the same code as every other until they enter the next iteration.

The code changes needed to implement this were small and non-invasive: about 200 lines of new code were introduced. Table 3.3 shows the runtime of the sequential code on a single CPU along with the speedup of two different parallelisation strategies relative to the sequential code timed using the same datasets. The test

Target	Source	Sampling	NMI eval/reset
b7_s2	b7_s2	721 kc/iter	244 kc/iter
b7_s2	b7_s1	816 kc/iter	254 kc/iter
b9_s2	b9_s1	945 kc/iter	282 kc/iter
b9_s4_e2	b9_s3_e2	99 kc/iter	246 kc/iter
b9_s3_e2	b8_s3_e2	93 kc/iter	286 kc/iter

Table 3.2: EvaluateDerivative costs

CPUs	Runtime	Speedup 1	Speedup 2
1	58,320 s	1.00×	1.00×
2	31,895 s	1.80×	1.9×
4	19,340 s	3.09×	3.7×
8	14,045 s	4.64×	6.2×
16	9,625 s	6.06×	10.8×
24	—	—	13.9×

Table 3.3: Parallel speedup of nreg vs sequential code

environment was a cluster of 16 dual core Opteron 246 running 64 bit SUSE Linux Enterprise Server 9 with version 2.6.5 of the kernel, and connected using a gigabit ethernet switch. The first parallelisation strategy simply uses the master/slave division of EvaluateDerivative described above. As can be seen from the first speedup column in the table, it is fairly effective for such a simple parallelisation. It scales adequately on smaller experimental clusters, but the scaling is insufficient for a larger production system, especially for a code that is in principle quite amenable to parallelisation.

To improve the speedup further, some of the remaining sequential code is parallelised. The Evaluate steps in the gradient descent optimisations calculates all the voxels in the transformed image, and measures the similarity with the source. The voxel calculation can be divided up into stripes which are executed in parallel on all the CPUs, and the results returned to the master CPU and broadcast back to all the slaves. After the broadcast, all CPUs can continue in lockstep to calculate the similarity. Unlike the earlier parallelisation, this is relatively bandwidth intensive and will perform poorly with a slow interconnect, but it can provide some extra scalability on larger systems.

3.5 Predictive model

The IXI-based workflows that TITAN supports include nreg and two other codes — BET [Smi02] and FAST [ZBS01]. They pre-process input images for use with

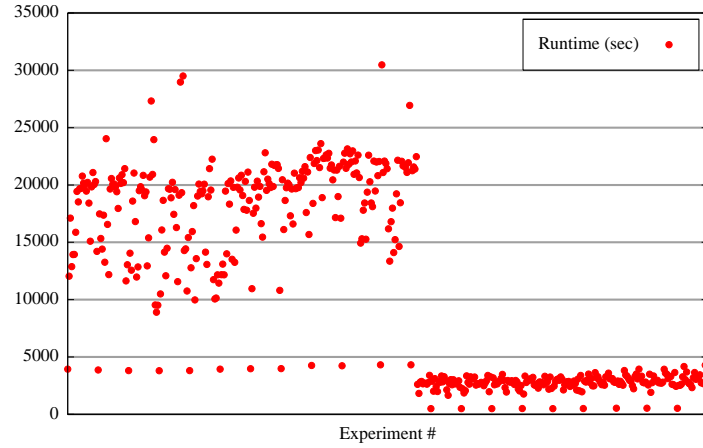


Fig. 3.3: Runtime variation for different images

`nreg` and perform a fixed amount of per-voxel work that is directly related to the input parameters and size of the input image. Both codes can be analysed statically and analytical expressions for each are readily formed. Performance tests reveal that these models yield a high level of predictive accuracy, with the relative error of predicted vs. measured execution time typically less than 10%.

The `nreg` model is less straight-forward. Observation shows that the program exhibits highly unpredictable execution times. This variation is caused partly by the variation in the number of mesh fitting iterations that occur and partly because of the variable amount of work required to calculate the improvement generated by moving each point. In some execution scenarios, there are many rapidly executing iterations of `EvaluateGradient`. In other scenarios, the routine is called far less frequently, but the runtime is also much higher due to the large number of samples made. From source code inspection, study of the internal data structures and analysis of profiling traces, it is apparent that the execution time of `nreg` correlates most strongly with the choice of 'target' image. This is the second of the two input images and is the image that the first image (the source) is registered against. The sensitivity to the target image is dramatic and can affect the overall execution time by a factor of 50 \times .

[Fig. 3.3](#) illustrates the variability in the overall runtime and [Table 3.1](#) shows the number of iterations that occurs at each level for different images and how it may affect runtime. The difficulty in predicting the execution time for `nreg` reduces to estimating how many iterations will occur and their overall costs.

3.5.1 Pre-model work parameter

Despite the variations in runtime, patterns occur in the execution behaviour of `nreg`. Four bands of runtimes can be observed, and these are characterised by the iteration count at each resolution level and the number of image samples taken per iteration. The steps between these bands are significant — an average ‘fast’ band will run for 2,500s on a 2.8 GHz P4 computer, while an average ‘slow’ band will run for 18,000s on the same machine. Since TITAN works by continually refining its schedules to adapt to new information and any changes in the workflow, it is able to compensate when tasks complete earlier than expected or over-run (see below). Because of this TITAN benefits from even coarse grained predictions, and identifying which band a particular image pair belongs to will improve TITAN’s schedules.

One approach to estimating the runtime is to pre-process the destination image to distinguish whether it is likely to cause more (or less) work than other images. Earlier work on PACE addressed a related problem where a data-dependent application, a lossless video compressor, performed an inexpensive initial scan over the data to identify potential features that would affect the runtime [TLHKN02]. The result of this analysis yields a model parameter that can be used by PACE’s evaluation engine. Unfortunately, while some statistics such as intensity variations or various information theoretic properties can be found in the MRI scans, these do not directly reveal the effect the image has on the gradient descent solver.

Inspection of the code shows that while data sensitivity is significant, the program’s runtime is also related to the size of the input images and the number of control points used. If the images are of a lower resolution, all else being equal, there is less work to do. This feature can be exploited by using `nreg` itself to generate the pre-model parameter. Running the program with a significantly subsampled version of the images, it is possible to obtain an indicator of execution time and determine which band the image will fit into.

As seen in [Fig. 3.4](#), even when two pairs of images have the same global features (by virtue of one pair being subsampled copies of the other), the runtime does not simply scale linearly with the image size. There is a loose correlation, but it is insufficient to make adequate performance predictions. The runtime scaling factor depends partly on which ‘band’ the target images fall into.

The bounds of this problem are the best and worst case runtimes for both the actual and subsampled images. The lowest runtime is for a ‘self-registration,’ that is a registration of an image with itself. This causes one call of `EvaluateGradient` for

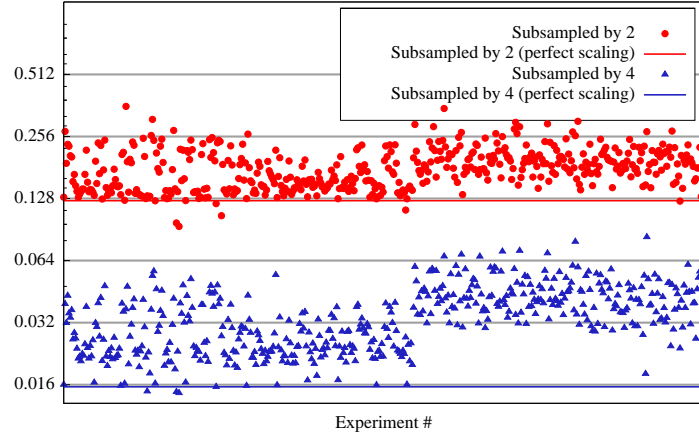


Fig. 3.4: Runtime scaling with subsampled images

every step size, and every image level. Each EvaluateGradient will show perfect similarity with no transformation, and any attempt at moving a control point will at best, preserve the similarity, but more likely reduce the similarity. Since the magnitude of the gradient vector is zero, the optimiser will attempt no movements of the control points, and the registration will immediately move on to the next step size. Thus self registration causes the least amount work for a registration in a particular band.

The self registration cases can be seen as the lowest points in Fig. 3.3. As can be seen, even for self-registration, there is a substantial difference in runtime between the ‘fast’ band of target images, and the ‘slow’ band of target images.

The worst possible case, for a given band, is an image where for all image levels, and for all step sizes, the optimiser is run for the maximum allowed number of iterations. This would require a very ‘jagged’ gradient descent path, and serves as a strict upper bound and is not a case that is likely to occur.

In practice, for a large cohort of scans a combination of brain images that produced the greatest runtime was empirically found. Registering these two images combines a relatively high iteration count with many samples per iteration.

Using these upper and lower bounds on the execution time it is possible, by subsampling an image, to hugely shorten the runtime but still keep the salient features of the image. When compared with the subsampled worst case registration and subsampled self-registration it is straightforward to identify a candidate workload parameter. Fig. 3.5 shows the effectiveness of this performance prediction technique.

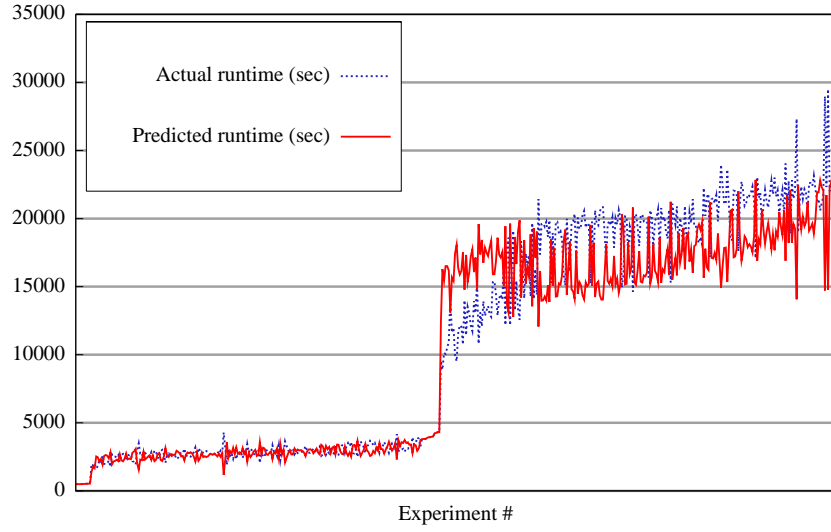


Fig. 3.5: Predicted runtimes

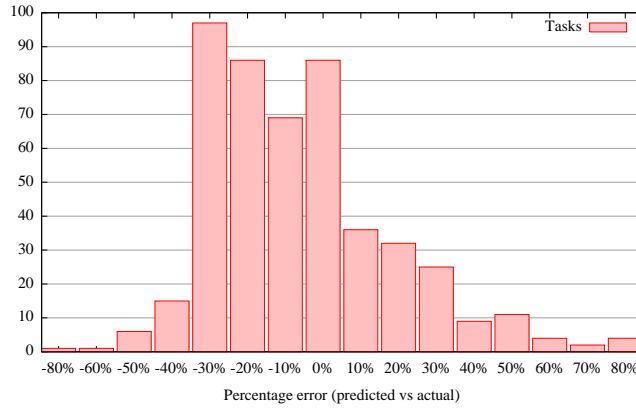


Fig. 3.6: Prediction error

Fig. 3.6 shows that although the worst case predictive error is quite large (on the order of $\pm 80\%$), the average error is under 20%. Even in the worst case, an 80% error is substantially smaller than the factor of $50\times$ uncertainty in runtime when no performance prediction is available.

With this technique predictions of the overall runtime can be made at the cost of performing one self-registration using a subsampled version of the target registered against itself and one registration with subsampled versions of the source and target images. These two subsampled registrations are called ‘probe tasks.’ It has been found that subsampling in each axis by a factor of 4 provides good results, although it may be possible to subsample further without sacrificing accuracy. Furthermore, it is a simple matter to cache the results of each of these probe tasks. One typical

use case for `nreg` is to register several images against the one reference image and, in this case, the reference self-registration only needs to be predicted once.

3.6 IXI workflows

IXI workflows are composed of standalone applications which can be joined to form an image processing pipeline [RBH⁺04]. The initial tasks in a typical workflow focus on image extraction. After this an image is then segmented to obtain an area of interest, which is used as input to a series of rigid registrations. These registrations scale, rotate and translate the images so that they are correctly aligned with each other. The output from this process is a set of transformations, which could then be passed to a more advanced registration algorithm, such as that provided by `nreg` or used directly.

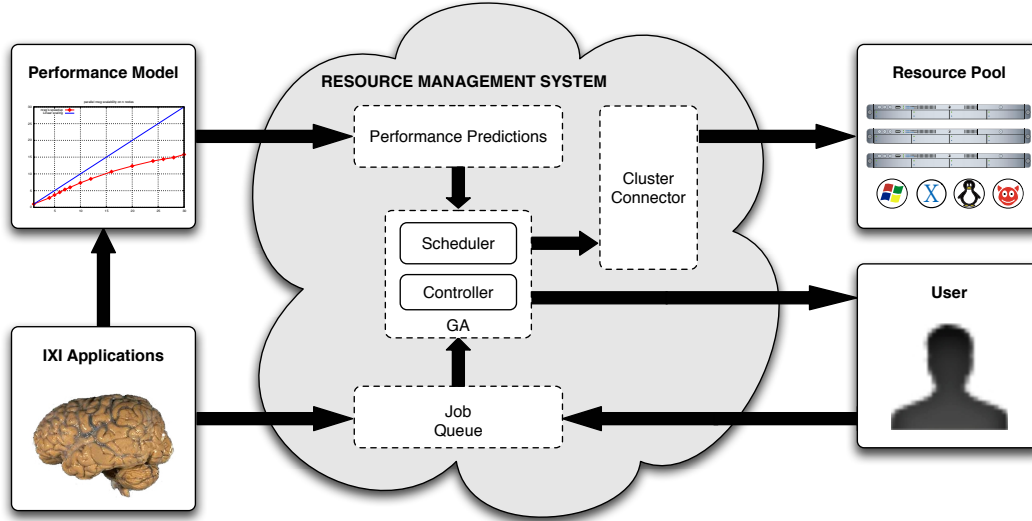


Fig. 3.7: Operation of the performance-aware resource management system

Managing how these workflows are scheduled in the context of a time critical, on-demand environment, such as in clinical diagnosis, is essential. In addition to minimising runtime, physicians need to know when results will be available. When provided with an estimated completion time, it may also be desirable to modify or remove subtasks in a workflow to reduce runtimes.

3.7 Incremental prediction

To date, TITAN, described in Sec. 2.4 has relied on analytical performance models. Two of the assumptions made by TITAN are that the analytical models are

inexpensive to execute (running in under a second), and can be evaluated on demand directly on the same machine that hosts TITAN itself without causing any substantial slowdown.

As described earlier, the probe tasks must be run on a machine with the same hardware configuration as the target machines for the full sized jobs to obtain meaningful results, require the same input data as the full sized jobs, and take a significant amount of time to execute.

In practice, a subsampling factor of $4\times$ in each dimension, yielding images with $\frac{1}{64}$ of the voxels and a runtime between $\frac{1}{20}$ and $\frac{1}{64}$ of the original gives reasonable estimates. Even though this gives a runtime for the probe tasks measured in minutes instead of hours, this is still much slower than the performance estimates provided by the analytical models, and is large enough that the probe tasks require scheduling in their own right. To accommodate this, when an nreg task is submitted to TITAN it adds a pair of probe tasks to the schedule and makes the nreg task depend on the probe tasks' completion. These probe tasks are scheduled and executed on the cluster hardware just as ordinary tasks are.

To schedule both the probe and real nreg tasks, TITAN requires some estimate of their runtime, but until the probe tasks complete, no estimate is available. To resolve this bootstrapping problem, as an initial estimate each probe and nreg task is given a runtime which is the average of all the probe or nreg tasks that have run so far.

When the probe tasks execute and complete, they report their performance estimates to TITAN. TITAN caches these results and updates the runtime estimates for the real nreg tasks. It would appear that the new runtime estimates will either lead to new gaps appearing in the schedule (where the initial estimate was longer than the updated estimate), or overlaps in the schedule (where the initial estimate was shorter than the estimate). However due to the representation of schedules within TITAN as discussed in [Sec. 2.4](#) these gaps and overlaps do not occur – each task simply consumes more or less time and subsequent tasks requiring the same nodes begin earlier or later than before. This motion may introduce other gaps elsewhere in the schedule that the ordinary operation of the scheduler will attempt to remove.

So that an initial performance estimate is produced quickly, the probe tasks can be modified to perform a number of registrations at varying levels of subsampling, such as $6\times$, $5\times$ and $4\times$ in each axis. As each registration completes, it reports its results to TITAN, which can then immediately make use of the less accurate, but more timely information to begin packing a more realistic schedule.

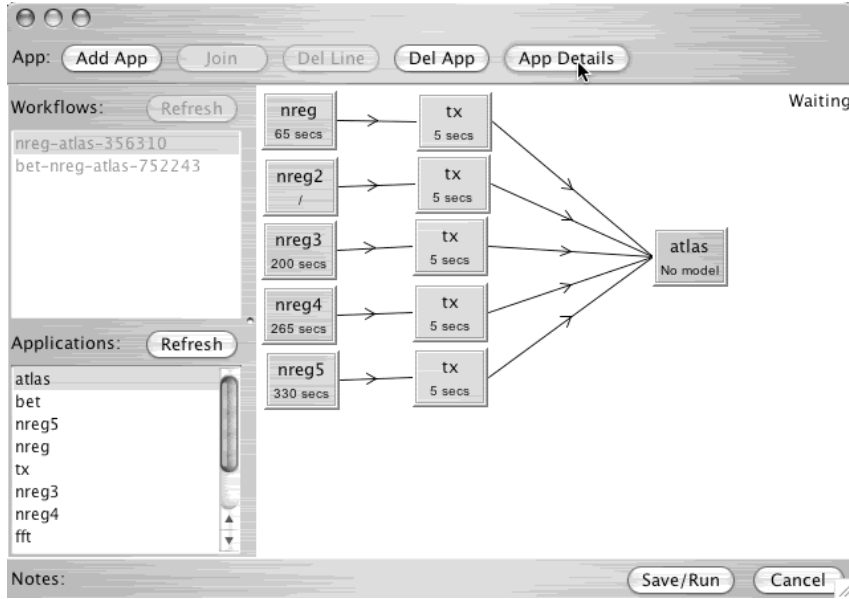


Fig. 3.8: Workflow builder with performance model evaluation

3.8 User interaction

To allow the user to have a degree of interaction with the scheduler, an improved front-end has been written that allows workflows to be constructed, submitted for execution, and monitored. This front-end resembles many other DAG construction editors, but with the difference that the components in the front-end represent the performance models and not the applications themselves. As a result of this, the user can interactively determine the resource requirements of the workflow and verify that it can be run within the desired timeframe. In practice, instead of requiring ‘yet another workflow editor’, it is more likely that the services that TITAN exposes would be connected to the end user’s preferred DAG tool such as the IXI Workbench [RBH⁺04].

When a workflow is constructed, performance model probe tasks for each task are scheduled to be evaluated as quickly as possible on a machine with the same hardware configuration as the target compute nodes. This can be done either by reserving a node on the cluster specifically for probe tasks, or by giving the probe tasks a higher priority. As before, when the probe tasks complete they report their results to TITAN which then caches them. This provides rapid feedback to the user of how long each task will take to run in isolation, and can be used as the basis for setting realistic QoS targets. This information is particularly useful for applications such as `nreg` with highly variable runtimes. The user can build up the workflow

incrementally, and each component will provide an indication of the CPU time required and the scalability of the component.

3.9 Speculative scheduling

When scheduling workflows, particularly ones with a number of internal dependencies, TITAN can often fill the idle time introduced by dependencies in one workflow using tasks from another workflow. This interleaving of the tasks from different workflows allows for more scheduling opportunities, and thus more efficient schedules. The interleaving could be regarded as similar to SMT in some CPU cores, and leads to similar throughput improvements.

A consequence of this is that if a workflow tool tries to calculate how much a new workflow will increase the makespan of the entire schedule simply by considering the runtimes of the workflow's tasks in isolation, it will likely overestimate. In the most extreme case, a new workflow might simply 'fill in the gaps' in an existing schedule leading to no increase in the overall makespan.

Since how to allocate compute nodes to a specific task is best decided by examining both the workflow and other workflows in the schedule, predictions from the performance models need to be evaluated in the context of a complete schedule.

To accommodate this, after assembly a workflow can be flagged as a *speculative* workflow and then submitted to the scheduler. Speculative tasks are scheduled like other tasks, but have the lowest possible priority and are marked as tasks that should never be executed. To prevent speculative tasks from blocking the execution of any other tasks, non-speculative tasks will automatically 'jump over' speculative tasks if they reach the front of the queue for any compute node.

By *speculatively* submitting a workflow, the scheduler is able to take advantage of its schedule packing algorithms to produce candidate schedules that optimise for all the workflows in the system, not just the speculative ones. The schedules returned by the scheduler give a more realistic runtime prediction for the entire speculative workflow because they includes the effects of mixing the tasks of workflow with those of the rest of the workload.

With *speculative* execution, it is likely that users will add and then remove registrations from their speculative schedules many times. This will have the effect of submitting many probe tasks for execution. Each of these that completes will increase the hit-rate of the performance estimate cache.

The use of speculative scheduling and the interactive workflow monitor together

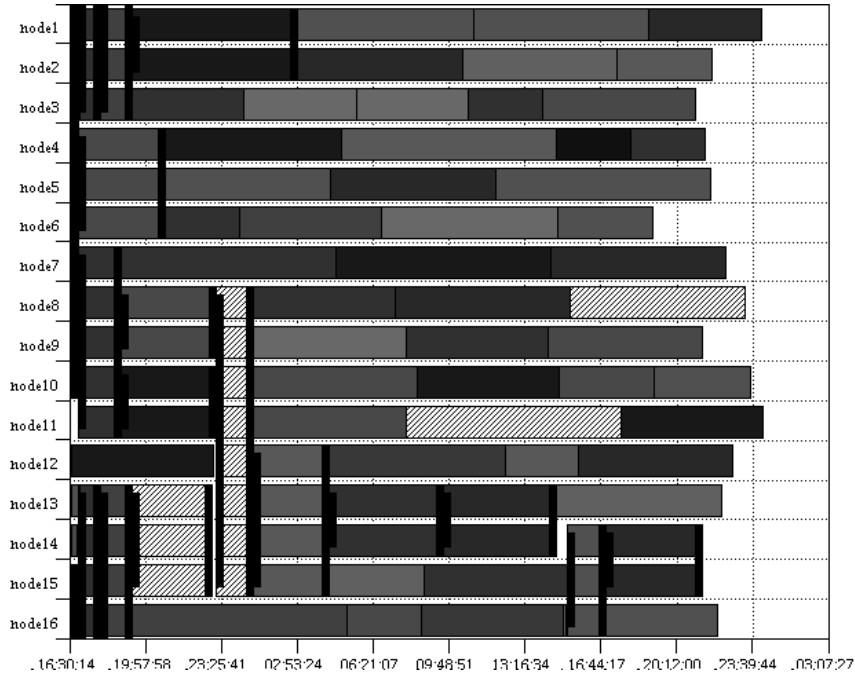


Fig. 3.9: Workflow scheduling a mix of speculative and real tasks

allows the user to experiment with different workflows and to view preliminary estimates on how long the workflows will take to execute. This feedback allows users to examine a proposed schedule and decide if it meets their requirements, and to modify the workflow, for example to do extra work or obtain usable results more quickly, if it does not. For applications such as `nreg` whose execution time can vary greatly for different input data, the ability to understand the implications of executing tasks in a given configuration before executing a workflow can provide significant benefits.

3.10 Case study

A typical IXI workflow is a three stage process that constructs a brain atlas from a number of registered brain scans. The input to this workflow is a set of brain images obtained from MRI scans. The first stage of the workflow extracts a usable brain image from raw scan data using BET [Smi02], a brain image extraction tool. The second stage of the workflow registers the output brain images from two BET tasks using `nreg`. The final stage of the workflow combines all the registered brain images produced by the `nreg` tasks to create a brain atlas.

Both the BET and ATLAS tasks have negligible runtimes. Typically they take tens of seconds to run, whereas `nreg` runtimes range from hours to tens of hours. The

main effect they have on a workflow is to act as serialisation points — an ATLAS task cannot run until all the nreg tasks it depends on have completed.

Two experiments are run. The first demonstrates that a scheduler using the performance information from the nreg probe tasks generates more efficiently packed schedules increasing the overall CPU utilisation and decreasing the makespan. This is true for light, medium and heavy workflows.

In the first experiment, the workloads consist of brain atlas workflows involving a number of brain registration tasks. Representative input brain scan images of varying sizes were selected from IXI’s database of sample brain scans. A light workload has 16 registration tasks, a medium workload 32 tasks, and a heavy workload 64 tasks. For each size of workload, five different random candidate workloads are generated, each workload scheduled in different ways, and the resulting schedule is measured.

The three scheduling techniques are:

1. The tasks use the sequential version of nreg scheduled in a First-In First-Out (FIFO) manner, where the first task submitted ran on the first available host.
2. The tasks use the parallel version of nreg and all tasks run on all CPUs.
3. TITAN schedules the tasks. The nreg performance model is evaluated for each task with TITAN matching tasks to suitable resources and scaling the tasks as appropriate to minimise the makespan.

Rather than submitting the nreg tasks directly for execution, the scheduler is allowed to ‘pack’ the schedule for a large number of iterations, and then the overall makespan and idle time of the final schedule is calculated using a table of measurements of the runtime of different nreg registrations gathered on real hardware. These runtime measurements are used in the second experiment to measure the behaviour of TITAN with a perfect performance model.

The results for the first experiment in Table 3.4 show the average effect of using TITAN and the performance model for five different workloads. More detailed figures are available in Table G.1. With a light workload TITAN gives a 22% reduction in makespan over the other scheduling techniques. With medium and large workloads the resource management system gives a 24% improvement in makespan. The tests assume that the probe tasks for the performance models were executed and cached at workflow construction time and do not contribute any overhead by requiring compute resources of their own in the schedule. The effect

Experiment	Makespan	Idle time
light workload		
sequential, FIFO scheduling	51,413 s	58.2%
parallel, FIFO scheduling	37,068 s	0.0%
TITAN scheduling	28,880 s	7.3%
medium workload		
sequential, FIFO scheduling	86,692 s	50.5%
parallel, FIFO scheduling	74,100 s	0.0%
TITAN scheduling	56,464 s	7.2%
heavy workload		
sequential, FIFO scheduling	159,730 s	45.1%
parallel, FIFO scheduling	151,558 s	0.0%
TITAN scheduling	115,847 s	8.8%

Table 3.4: Comparing scheduling techniques with varying workloads

of probe tasks on a schedule is measured in the next experiment, but for now it is evident that even requiring a $4\times$ subsampled probe task for each nreg task and assuming no caching of probe task results, the additional runtime overhead will be somewhere between 1.5% and 5%, which is less than the makespan improvement achieved.

The idle time is lowest, and thus the resource utilisation is highest when each of the nreg tasks are ‘greedily’ allowed to run in parallel on all nodes. However, because of the sub-linear scaling of nreg this is a very inefficient scheduling strategy. From the idle times in each of the schedules it is seen that both the makespan and percentage of unused CPU are significantly lower when managed by TITAN. If the total number of CPU seconds used is calculated, the TITAN-managed schedules are shown use more CPU time than that of a FIFO schedule consisting of just single CPU nreg tasks. The greater CPU load comes from nreg’s sub-linear scaling and the fact that TITAN decided to run some of the tasks on more than one CPU. However this tradeoff is worthwhile, as it led to a more efficiently packed schedule and thus reduced the makespan when compared to the most CPU efficient schedule.

The second experiment examines the costs and tradeoffs of running the probe tasks themselves. Several random mixes of tasks are submitted to TITAN, and the scheduler is configured so that time appears to pass $200\times$ faster than real time. TITAN submits tasks at the front of the schedule to a simulated batch queue where they appear to execute, also at $200\times$ real time. The simulated queue uses the measurements of the runtimes of the nreg tasks gathered on real hardware mentioned in the previous experiment. The scheduler submits tasks for execution

	FIFO	Fast	Medium	Slow	Perfect
40 tasks					
Makespan	65,827 s	43,680 s	43,813 s	43,680 s	39,986 s
Idle time	60.0%				16.8%
80 tasks					
Makespan	109,283 s	94,453 s	95,173 s	93,319 s	81,787 s
Idle time	51.0%				19.0%
160 tasks					
Makespan	184,204 s	208,853 s	212,480 s	208,080 s	177,196 s
Idle time	42.1%				23.1%

Table 3.5: The effect of probe task speed on makespan

as simulated nodes become available, repeatedly optimises the schedule, and the simulated queue notifies TITAN when the simulated tasks complete. The duration of the experiment is measured from the time when the first task is submitted to when the final task completes. This, multiplied by the scaling factor of 200, yields a makespan.

This simulation is more demanding on the scheduler than using a real batch queue, as the genetic algorithm runs for fewer iterations to find a good schedule before a task executes, and has less time to adapt when the schedule changes due to a task taking more or less time to execute than predicted.

The workloads consist of a random mix of nreg registrations, with 40, 80 and 160 registrations. On average 20% of the tasks submitted depend on other tasks in the workflow, thus introducing additional constraints to the scheduler. The simulated batch queue has 16 processors, and uses the same runtime timings as used in the first experiment.

Three types of tests are run: firstly with no performance predictions available and FIFO scheduling (the FIFO column); secondly with ‘perfect’ performance predictions available and zero cost for the predictions (using the timings gathered previously); and thirdly with three different classes of probe tasks — fast, medium and slow. The fast probe tasks subsample the input images 5×, and have the fastest runtime. The medium probe tasks subsample by 4×, and the slow probe tasks subsample by 3×.

The FIFO case is similar to the current execution strategy employed by the IXI demonstrator — it has no knowledge of the task runtime or scalability. The perfect case shows how the scheduler would behave if perfect (and instant) runtime predictions were available. The effectiveness of the fast, medium and slow cases are

compared with these two extremes.

From [Table 3.5](#) it is evident that for the small and medium workloads, the effectiveness of the probe task scheduling falls between the [FIFO](#) and perfect scheduling. The runtime estimate provided by the probe tasks allows the scheduler to more accurately anticipate idle time in the schedule and to reorder tasks, or run them on different numbers of [CPUs](#) to avoid unused resources. This can be seen in the decreased makespan of the fast, medium, and slow schedules. The medium probe tasks are of less benefit to the overall makespan than either the slow or fast probe tasks. This unexpected result shows how there is a tradeoff between the overhead of performing the probes themselves and the benefits they bring. The predictions from the medium probes are insufficiently more accurate to justify their extra overhead.

Also in this experiment, it can be seen that the medium workload benefits less from the genetic algorithm than the small workload, and the heavy workload benefits least, and with the three probe-based predictions is in fact slower than the [FIFO](#) case. This is an artefact of how the experiments were performed. The 200× scaling of time for the probe-based simulations meant that the genetic algorithm had far fewer iterations to react to changes in schedule. Since larger schedules are proportionately slower to pack than small schedules, and with probe-based predictions, the genetic algorithm has to schedule the probe tasks as well as the real tasks, what is seen here is the effect of the genetic algorithm being overloaded rather than a failure of the predictive model. More detailed figures can be found in [Table G.2](#).

3.11 Summary

This chapter examined the medical imaging application `nreg` and determined that the factor of 50× runtime variability it exhibits is for the most part algorithmic in origin. `nreg`’s performance variability stems from how many times the gradient descent optimiser runs, and how many samples are made in each call to `EvaluateDerivative`. In other words, depending on the input data, the algorithm decides to do more or less work. Although difficult to capture directly in an analytical performance model, I show how running ‘probe tasks’ based on highly subsampled input images provides information that can be used to classify and predict the runtime of the full sized application. The average error rate is under 20%, and even the least accurate estimations are on the order of $\pm 80\%$.

A parallel version of `nreg` which scales reasonably up to 16 [CPUs](#) was developed with identical performance characteristics to the sequential version. *Interactive scheduling* and *speculative scheduling* were introduced, and experiments run to show

how TITAN can use the performance characterisation to schedule parallel nreg more efficiently than a FIFO scheduler despite the overheads of the probe tasks.

The existence of both the parallel implementation and the predictive model remove one obstacle preventing a tool like nreg being used where quality of service criteria need to be applied. Once such environment is that of clinical diagnosis, where the ability to process data within a strict deadline is critical. The nreg performance model allows the runtime of an nreg registration to be estimated, and this estimated runtime can be used to allocate as many CPUs as necessary for parallel nreg to complete within a specified deadline.

Floating point and denormal handling

This chapter examines an issue that can cause performance variability even for applications with no ‘algorithmic’ variability: that is to say, applications which perform a fixed amount of work in a fixed order on some or all of their input data. These applications would appear at first sight to be perfect candidates for a simple performance model: 1) perform a few simple benchmarks on the target machine; 2) find an equation to determine how the runtime varies with different input arguments and input data sizes; 3) determine the upper bounds for the application’s working set to fit in cache and main memory, and how the performance changes when it exceeds these and the model is completed.

However if the application uses floating point instructions, operations involving denormal floating point values can be a source of substantial data-dependent slowdowns, even on systems which handle the operations entirely in hardware. On the Pentium 4 for example, the use of denormal arguments makes floating point loads and stores approximately 70× slower than they would otherwise be. This slowdown can cause problems in codes based on numerical stencils, amongst others, as it frequently manifests as small localised regions of denormals that move throughout the code’s data sets. These localised slowdowns can be problematic in applications that presume a workload can be partitioned evenly by giving $1/N$ of the data to each CPU. The effect is amplified when parallel applications use barrier operations after each iteration, as every CPU will be forced to wait for the slowest one.

Denormal value arithmetic can be disabled by switching on ‘flush-to-zero’ mode, however this is undesirable as it leads to a silent loss of precision, and in certain scenarios, various mathematical properties that compilers rely on no longer hold. What is desirable is to find a way to remove the unneeded denormal values from an application without disabling denormal arithmetic for those rare cases where it is really required.

In this chapter, some details of how IEEE-754 floating point arithmetic is implemented are discussed, and the trapping mode that it mandates is used to implement a small denormal arithmetic profiler for Linux on Intel x86 processors. This profiler can be used to detect the occurrence of denormal instruction arguments

in an application, but has significant limitations. These limitations are discussed and lead to the implementation of a more sophisticated tool in the next chapters to isolate and remove the sources of denormal values.

4.1 Fixed point arithmetic

Scientific applications often need to perform calculations involving extremely large or extremely small numbers, and frequently both at the same time. The standard integer arithmetic found in all microprocessors cannot represent fractional values at all so one workaround, used in cases where floating point hardware is not available, is to scale all values by a fixed amount, and perform integer calculations using these scaled values.

A representation that is sometimes used in graphics and sound processing is to treat a 32 bit integer as having a 16 bit integral component, and a 16 bit fractional component. This can be thought of as moving the ‘binary point’ 16 places to the left. The input data is converted into this form, intermediate calculations are performed using fixed point arithmetic to avoid accumulated rounding errors, and the final results are converted back to 16-bit integers.

Every number that enters the system must be multiplied by 2^{16} , i.e., 65536. Additions and subtractions of scaled numbers occur as normally, as there is no change in scale; however multiplications and divisions need to be carried out to a higher precision, and re-scaled to compensate for the doubling or cancellation of scaling that occurs. Square roots, logarithms and trigonometric functions also need to be treated with care.

The steps required for each operation can be derived using basic algebra. Taking the real numbers a , b , and c , and using a scaling factor of s ($s = 2^{16}$ for a 16.16 fixed point format), then the fixed point representations are $\|s.a\|$, $\|s.b\|$ and $\|s.c\|$, where $\|x\|$ is x rounded to the nearest integer. We wish to determine $\|s.a\|$ in terms of $\|s.b\|$ and $\|s.c\|$. When discussing precision, for simplicity’s sake a fixed point format with 16 signed integer bits and 16 fractional bits is assumed.

For addition and subtraction, if $a = b \pm c$, then

$$\begin{aligned}\|s.a\| &= \|s.(b \pm c)\| \\ &= \|s.b \pm s.c\| \\ &= \|s.b\| \pm \|s.c\|\end{aligned}$$

Since no further scaling is used for these operations, 32-bit precision is sufficient, and overflow occurs exactly as with the integers.

For multiplication, if $a = b \times c$, then

$$\begin{aligned}\|s.a\| &= \|s.(b \times c)\| \\ &= \left\| \frac{s.b \times s.c}{s} \right\| \\ &= \left\| \frac{\|s.b \times s.c\|}{s} \right\| \\ &= \left\| \frac{\|s.b\| \times \|s.c\|}{s} \right\|\end{aligned}$$

Since the result is produced by scaling down an intermediate value, the intermediate $\|s.b\| \times \|s.c\|$ calculation needs to be carried out to 48-bit precision if the result is to have 32 significant bits. In practice, multiplying two 32 bit values to yield a 64 bit result is commonly implemented in [CPU](#) hardware or has hardware assistance.

For division, if $a = b \div c$, then

$$\begin{aligned}\|s.a\| &= \left\| s. \frac{b}{c} \right\| \\ &= \left\| \frac{s.s.b}{s.c} \right\| \\ &= \frac{s. \|s.b\|}{\|s.c\|}\end{aligned}$$

However, due to the fact that integer arithmetic is used, it is important that the operations are performed in the correct order — so that precision not be lost, the multiplication in the numerator must be done before the division. This leads to a 48-bit numerator and a 32 bit divisor.

For square roots, if $a = \sqrt{b}$, then

$$\begin{aligned}s.a &= s. \sqrt{b} \\ &= \sqrt{s.s.b} \\ &= \sqrt{s. \|s.b\|}\end{aligned}$$

As with division, this involves an intermediate 48-bit value that is required so that precision is not lost during the integer square root operation.

However, apart from being unwieldy and error prone, fixed point arithmetic has

a major problem that precludes it from use in many applications, namely that the range of the numbers is too small. For example, in a 32-bit fixed point system, no matter what the scaling factor is, the difference between the biggest and smallest numbers is a factor of approximately 2 billion, or just over 10^9 . Using 64-bit values only doubles this to just under 10^{19} . This might seem adequate for most problems, but it is not. For example, a simple mechanics problem might involve calculating the cumulative effect that a 10 micronewton ion thruster will have on the trajectory a 1000 kg spacecraft. The thruster acceleration is $\frac{10^{-5} \text{ N}}{10^3 \text{ kg}} = 10^{-8} \text{ ms}^{-2}$. If the ship is moving at 10^4 ms^{-1} , the velocity is $10^{12} \times$ the acceleration, requiring 12 decimal places to represent both in a calculation such as $v = u + at$. For the final velocity to be accurate to 6 decimal places requires the use of 18 of the almost 19 decimal places leaving very little for rounding errors introduced by the intermediate calculations.

4.2 Floating point arithmetic

4.2.1 Scientific notation

The solution to this problem is to adopt a scheme that distinguishes between the scale of the number and precision in the significant digits. In the ion thruster example, the velocity of the ship is 10^{12} larger in magnitude than the acceleration of the thrusters, but that does not mean that the velocity of the ship is known to 18 decimal places compared to the 6 of the thrust. Scientific notation makes this distinction by splitting numbers into two components, the mantissa m , and the exponent e . By convention, the mantissa is a fractional number with one digit before the decimal point, and a limited number of digits after the decimal point. The mantissa is in normal form, that is to say, it has a non-zero digit before the decimal point, i.e., $1.0 \leq m < 10.0$ ¹. The exponent is an integer, and any number can be represented as $n = \pm m \times 10^e$. The precision is controlled by limiting the number of digits in the mantissa, and the range can be very large due to the fact that e controls the range on an exponential scale. The thruster calculation can use 6 digits in the mantissa and a 2 digit exponent instead of the 18 or more required for a fixed point scheme.

Scientific notation can be generalised to any base, by expressing numbers as $n = m \times b^e$ where $1.0 \leq m < b$. For floating point arithmetic on digital computers, it is convenient to use a power of 2 as the base, and 2, 8 and 16 have been chosen by popular implementations in the past.

¹Scientific notation also allows the exceptional case where $m = 0.0$ exactly

Fig. 4.1 shows an extremely compact 5-bit floating point format. It can represent a larger range of numbers than the 0–31 of an unsigned integer, but the absolute precision of the format varies over the range of possible values. The details for this format can be found in [Appendix A](#)

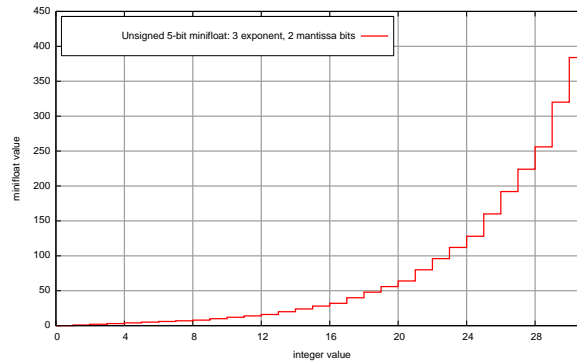


Fig. 4.1: A 5 bit 'minifloat' format

4.2.2 Floating point calculations

Calculation with floating point arithmetic is more complex than fixed point arithmetic. To perform addition and subtraction on two numbers the following steps must be taken:

- The values must be 'unpacked' by unbiasing the exponents, and inserting the implied leading 1s into the exponents. During unpacking, a check must be made whether either value is exactly zero, in which case no leading 1 is inserted.
- The number with the smaller exponent must be denormalised by 'shifting it right' so that the two numbers are aligned, i.e., have the same exponent. For rounding to occur correctly, this requires two more digits of precision than the mantissa usually stores.
- It must be determined whether a 'true addition' or 'true subtraction' is to be performed by comparing the two sign bits requested operation. For example $(+a) + (-b)$ is in fact a true subtraction, and $(-a) + (-b)$ is a true addition.
- The two mantissas are now added or subtracted as if they are integers, and the resulting sign bit updated to match the calculation. If the two exponents were the same, the result mantissa will overflow and must be shifted one digit to the right, increasing the exponent by 1.

- If the result is exactly zero, this can be returned, otherwise the result must be normalised by shifting the mantissa as appropriate and updating the exponent. This may lead to an exponent that is too large or small for the floating point format. These cases (leading to $\pm\infty$ or a zero value with the underflow status set) as well as the denormal cases must be handled here.

Multiplication and division are similar:

- The numbers must be unpacked as in the addition/subtraction case.
- The exponents are added, or subtracted depending on whether multiplication or division is performed.
- For division, the divisor must be at least as large as the dividend, otherwise the resulting mantissa will overflow. This can be arranged by ensuring both values are properly normalised and the exponents updated appropriately. The mantissas are multiplied or divided as if they are fixed point numbers, producing a result with double the precision of the source mantissas.
- The result sign bit can be calculated by examining the source signs. If they differ the result is negative, otherwise it is positive. If the result is exactly zero, this can be returned, otherwise the resulting mantissa is normalised, updating the exponent. As with addition and subtraction, the exponent may not fit the floating point format. These cases, as well as the denormal cases and divide by zero case must be handled here.

This extra complexity has a significant cost both in terms of transistors and speed for hardware implementations.

4.3 IEEE-754

There have been numerous different computer floating point arithmetic implementations each with differing formats, precision guarantees, rounding modes, handling of signedness or signalling of exceptional situations. For example, base-16 implementations can lose up to 3 units of the least place (ulps) in basic arithmetic calculations; some implementations distinguish between +0.0 and -0.0, and some do not; and some use guard bits to preserve extra precision during calculations. Due to the difficulties of writing numerical algorithms that worked within the limitations of all these implementations, one particular scheme, IEEE-754, was standardised in 1985, and revised in 2008 and all current implementations support it. IEEE-754

implementations are required to provide at least two floating point formats: the 32-bit single precision, and the 64-bit double precision. Numbers are stored as base-2 floating point numbers with a biased unsigned exponent, and normalised mantissa.

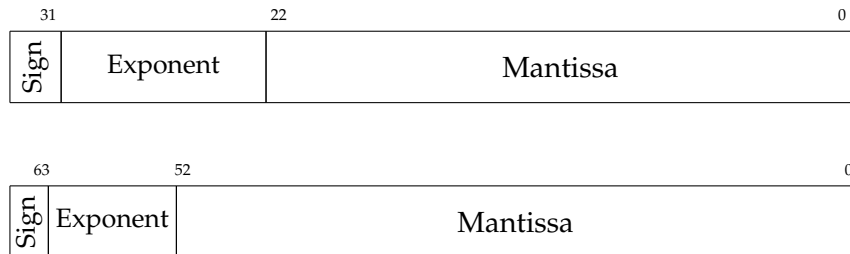


Fig. 4.2: IEEE-754 float and double formats

In both formats, the floating point number is divided into 3 fields as shown in Fig. 4.2. These fields are interpreted as follows:

- The sign bit s , which specifies whether the number is positive or negative: 1 indicates a negative number, and 0 a positive number.
- The mantissa m , which is 23 bits in single precision and 52 bits in double precision. When normalised numbers are used, the mantissa is $1.000 \leq m < 2$ and therefore the most significant digit must be 1. Since the most significant digit is always 1, only the fractional part of the mantissa needs to be stored in the floating point representation. This 1 is referred to as the hidden bit, and gives the mantissa one more effective bit of precision leading to 24 bits in single precision and 53 bits in double precision.
- The exponent e , which is 8 bits in single precision and 11 bits in double precision. It is stored as a biased unsigned integer. A biased or excess- n integer is stored as if it is n greater than its real value. In single precision the range of the exponent is $-127 \leq e \leq 128$ and the bias is 127. In double precision the range is $-1023 \leq e \leq 1024$ and the bias is 1023. In IEEE-754, the largest possible exponent and the smallest possible exponent are reserved for special purposes as discussed below.

Floating point numbers with the exponent is set to its maximum value represent two classes of exceptional numbers. The first class is the two infinities, $+\infty$ and $-\infty$. These occur when the result of a calculation is too great to be represented by

Exponent value	Mantissa	Interpretation	Value
-127	Zero	± 0	$-1^s \times 0.0$
-127	Non-zero	Denormal numbers	$-1^s \times 0.m \times 2^{-126}$
-126 – +127	Any value	Normal numbers	$-1^s \times 1.m \times 2^e$
+128	Zero	$\pm \infty$	$-1^s \times \infty$
+128	Non-zero	Not a number values	NaN

Table 4.1: Single-precision storage in IEEE-754

a normal number, e.g., $2^{127} + 2^{127}$, or -10×2^{127} for floats. The second class is NaN — Not a Number — used to indicate the result of calculations with no meaning, or operations with invalid values, e.g., division by zero, the log of a negative number, or $\sin^{-1}(x)$ where $x \notin [-1, 1]$. There are in fact two classes of NaNs, quiet NaNs and signalling NaNs. The distinction between them is not important here.

Because of the hidden bit described above, it is impossible to represent 0.0 as a normal number, since a hidden 1 bit is implied in the mantissa. To handle this, there is a special class of numbers in IEEE-754 called denormal numbers² which represent a set of positive and negative numbers smaller than any normal number, but with less precision than the normal numbers. In single precision, when the exponent is set to -127, the exponent is treated as if it is -126, the mantissa is treated as if the ‘hidden’ most significant digit is 0, and the 23 bits from the mantissa field are used directly as the fractional part of the mantissa. If the 23 bits are all zeros, then the number represented is exactly zero, since $n = -1^s \times 0.0000000000000000000000_2 \times 2^{-126} = 0.0$. Although this representation of zero has the same form as a denormal number, it behaves like a normal number in all implementations, and is not usually regarded as denormal.

If the 23 bits are non-zero, then the number will be less than the smallest normal number. For example if $m = 00000000100000000000000_2$ then $n = -1^s \times 0.00000000100000000000000_2 \times 2^{-126} = 2^{-135}$.

4.3.1 Gradual underflow

This is useful for two reasons. Firstly, it allows for *gradual underflow*. Consider a 32-bit floating point representation of π repeatedly divided by two.

In the first case, all 24 significant figures are preserved until the final divide by two, where suddenly the number is truncated and all 24 bits are lost.

In the second case, $\pi/2^{127}$ is accurate to 24 figures, $\pi/2^{128}$ to 23 figures, $\pi/2^{129}$ to 22

²Current versions of IEEE-754 use the term *subnormal numbers*, but Intel still refer to them as *denormal numbers*.

$$\begin{aligned}
\pi &\approx 1.10010010000111111011010_2 \times 2^1 \\
&\dots \\
\pi/2^{100} &\approx 1.10010010000111111011010_2 \times 2^{-99} \\
&\dots \\
\pi/2^{126} &\approx 1.10010010000111111011010_2 \times 2^{-125} \\
\pi/2^{127} &\approx 1.10010010000111111011010_2 \times 2^{-126} \\
\pi/2^{128} &\approx 0.000000000000000000000_2
\end{aligned}$$

Table 4.2: Flush to zero behaviour

$$\begin{aligned}
\pi/2^{100} &\approx 1.10010010000111111011010_2 \times 2^{-99} \\
&\dots \\
\pi/2^{126} &\approx 1.10010010000111111011010_2 \times 2^{-125} \\
\pi/2^{127} &\approx 1.10010010000111111011010_2 \times 2^{-126} \\
\pi/2^{128} &\approx 0.11001001000011111101101_2 \times 2^{-126} \\
\pi/2^{129} &\approx 0.01100100100001111110110_2 \times 2^{-126} \\
\pi/2^{130} &\approx 0.00110010010000111111011_2 \times 2^{-126} \\
&\dots \\
\pi/2^{140} &\approx 0.00000000000011001001000_2 \times 2^{-126} \\
&\dots \\
\pi/2^{150} &\approx 0.00000000000000000000001_2 \times 2^{-126} \\
\pi/2^{151} &\approx 0.0000000000000000000000_2 \times 2^{-126}
\end{aligned}$$

Table 4.3: Gradual underflow with denormals

figures, and so on. This gradual loss of precision lets some numerical algorithms fail more gracefully, and also increases the effective range of the floating point format.

4.3.2 Mathematical properties

Denormal arithmetic is important for a second reason as well, namely that without it, certain mathematical equivalences that apply to the reals do not hold true for floating point numbers.

One such property is when $a, b \in \mathbb{R}$, $a - b = 0 \Leftrightarrow a = b$. However, in floating point arithmetic using flush to zero, if a and b are very small numbers, and the difference between them is less than the smallest normal number, then $a \neq b$, but $a - b \approx 0$ because of the flush to zero behaviour.

As an example, in single precision, if $a = 1.1_2 \times 2^{-126}$ and $b = 1.0_2 \times 2^{-126}$, then $a - b = 0.1 \times 2^{-126}$. There is no normal number representation for this, so the result is rounded to zero. This rounding means that apparently safe calculations such as the following can lead to unexpected divide by zeros.

In contrast, it can be seen that if two small normal numbers differ, then the smallest

```

if (t1 != t2) {
    v = d / (t2 - t1);
}

```

Table 4.4: Divide by zero without denormals

possible difference between them must be in the unit of the least place ([ulp](#)) of their mantissa, i.e., the 23rd fractional digit in single precision. Because of how they are defined, a denormal number can exactly represent this difference as a non-zero number: it is the smallest denormal). Any other pair of small normal numbers that differ will differ by a larger amount, which will either be another larger denormal, or sufficiently large to be a normal number.

Another way of seeing this is that the small normals will have an exponent of 2^{-126} for floats, as do the denormals. When subtracting two normals, the two hidden 1 bits in the mantissa will cancel out, leaving a 0 as the most significant figure. The fractional parts of the mantissa will have to differ for them to be non-equal, and this fractional difference preceded by a 0 as the units digit is precisely the definition of a denormal number as seen in [Table 4.1](#).

For the same reason as for the divide by zero calculation above, without denormal arithmetic, simple compiler optimisations such as transforming $(a \times b) - (a \times c)$ into $a \times (b - c)$ become unsafe. When b and c are very small, and close in value, then some or all of the significant digits may be lost.

4.3.3 IEEE-754 implementations

The full IEEE-754 specification is relatively demanding to implement fully in hardware, in part due to the requirement that calculations be performed as if exactly accurate arithmetic were available and the results rounded appropriately. To provide as much accuracy as possible, this rounding requires 4 different rounding modes, and the use of 3 extra rounding bits — a guard bit, a sticky bit, and a rounding bit. Along with this, an IEEE-754 implementation must deal with denormal values and non-numerical values; and must have a mode where exceptions are signalled to an application immediately when the offending instruction executes. This last requirement, called precise floating-point exceptions, is particularly difficult to implement in a heavily pipelined or superscalar [CPU](#), and in some cases, such as on the DEC Alpha, the implementation requires the compiler to insert trap barrier instructions between every floating point instruction.

Since a the complete IEEE-754 specification is complex to implement in hardware,

some designs omit certain parts of the specification, and require software support to complete the implementation. One example of this is the DEC Alpha mentioned above which in its fast mode cannot handle any floating point values other than normal ones. Underflow is flush-to-zero, and any denormal, infinite, or exceptional values encountered immediately terminate the program.

For IEEE-754 behaviour, the Alpha compilers need to use floating point instructions which generate software traps, and also must interleave every floating point instruction with trap barrier instructions so that software traps can identify the instruction that caused the trap. The trap barrier instructions cause a significant slowdown by themselves, but the trap-generating floating point instructions run as quickly as the fast instructions, as long as non-normal operands are not encountered. When non-normal operands occur, a software handler is invoked, performs the required calculation using a software implementation and inserts the required value into the destination floating point register. The slowdown can be dramatic, being anything from 8× and 75× depending on the Alpha implementation. The cost of the penalty comes from the need to flush pipelines and discard any in-flight instructions, save user-mode state, perform the calculations and return to the user's code.

Other implementations, such as many MIPS and SPARC implementations have an intermediate strategy. They require no trap barrier instructions, and normal values and infinities are handled correctly in hardware. When NaNs occur, they trigger a software trap, incurring a slowdown of between 10× and 25×. Similarly, when underflow occurs a trap is generated, and the denormal calculations are performed in software. Again the slowdown is about 20×. Underflow can be handled either in flush-to-zero mode, or using full denormal arithmetic.

Curiously, on some RISC implementations, such as the MIPS R4400 and PA-RISC, the flush to zero mode is just as slow as gradual underflow, negating one of the major reasons to use this mode on these implementations.

Examples of RISC implementations with full performance hardware implementations for all types of floating point calculations are the TI SuperSPARC up to 50 MHz, the MIPS R5000 up to 180 MHz, and the IBM's PowerPC implementations up to 533 MHz. Recent implementations of the PowerPC architecture, such as the 1.8 GHz G5, step back from this ideal a little to achieve greater overall performance. They introduce two extra pipeline stages when non-normal arithmetic occurs. This incurs a 20% slowdown for gradual underflow, flush to zero, and overflow to infinity.

All popular Intel 80x86 compatible chips have a full hardware IEEE-754 implementation. Normal arithmetic is full speed, and on most versions of the [FPU](#) calculations that generate underflow have a slowdown of about 5× and those that generate NaNs or Infinities have a slowdown of about 3×. However despite this, the Pentium 4, various models of which have been popular in high performance computing, has an unusually high overhead when floating point instructions with denormal operands are encountered. The slowdown is about 70× even though the situation is handled in hardware. This occurs because of Intel's use of an 80-bit 'extended double' register file on the [FPU](#). Loads and stores are converted to and from this format, and when a load is denormal, or a store will become denormal, a microcode assist routine internal to the [FPU](#) performs the conversion.

4.4 DIP: A denormal profiler for Linux x86

The Intel 80x86 implements all of the IEEE-754 floating point standard in hardware. A part of this standard is a set of flags that cause a processor interrupt when one of a number of exceptional conditions occurs. These interrupts allow software to halt a running program when a problematic event occurs, perform some sort of fixup or reporting in the interrupt handler, and resume program execution by restarting the offending instruction.

Linux has operating system features which allow access to this functionality. I use these to write a tool called DIP (Denormal Instruction Profiler) that allows a profile to be generated of exactly where denormal arithmetic is used an application.

To do this, a knowledge of how floating point exception handling is implemented on the 80x86 is needed, along with how Linux wraps this functionality and makes it available to user programs.

4.4.1 Floating point exceptions on the 80x86

For the purposes of this thesis we shall only examine denormal arithmetic when 80x87 floating point instructions are used, rather than the newer SSE instructions supported by the Intel Pentium and later. The SSE hardware supports denormal arithmetic in a similar way to the x87 hardware, with similar performance penalties and like the x87, there are masks to enable and disable exception reporting and flush-to-zero behaviour. To simplify the presentation, we restrict our focus to the x87 only.

When running 32-bit code and using 80x87 instructions, a group of bits called *masks* in the [FPU](#) Control Word control floating point exceptions. These masks control

one of six possible types of floating point exceptions, and two of the exceptions signal denormal arithmetic.

The first type of exception is called a Numeric Underflow Exception. This occurs when the **FPU** has performed a floating point calculation and is attempting to normalise the result. Whether underflow is reported depends on whether the result is *tiny*, i.e., the resulting exponent is too small to be represented in the floating point format in normal form; and whether the result is *inexact*, i.e., if the result can be represented without truncation in the floating point format. If the underflow mask is set in the **FPU** Control Word, underflow is reported when the result is both tiny and inexact. If the underflow mask is cleared, underflow is reported for tiny results. This corresponds to the underflow signalling in the IEEE-754 specification, and enabling the underflow mask allows software to determine when denormal *output* is produced by a floating point instruction. When underflow is reported, if the value's destination was memory, the result is left on the floating point stack and the exception handler must perform the necessary denormalising and write. If the value was to be written to another floating point register, the value is scaled by 2^{24576} before writing to the register, and the exception handler must deal with any necessary re-scaling and denormalisation.

The second type of exception is called a Denormal Operand Exception. This occurs when one or more of the operands to a floating point instruction is found to contain denormal data and the denormal mask in the **FPU** Control Word is cleared. In other words, clearing the denormal mask allows a program to be informed when a floating point instruction receives denormal *input*. There is no corresponding signalling mode in the IEEE-754 specification.

When either the underflow or denormal exceptions occur, the DE or UE flag is set in the **FPU** status word, and the general purpose software exception handler is triggered when the next **FPU** instruction is encountered. The handler can read the status word, perform any appropriate actions, and on completion, performs an IRET instruction to resume program execution from where it left off.

However, there is one crucial difference between the denormal or underflow exceptions that complicates profiling for the denormal case. A underflow exception is a *post-operation* exception, i.e., it occurs after the instruction has finished calculating and has stored some intermediate result³ in the destination. In fact, due to how the original 8087 co-processor communicated with the 8086, it actually occurs at the beginning of the next floating point instruction. So when the IRET instruction exe-

³The software exception handler is free to modify this result.

cutes in the exception handler, the program resumes execution of the next floating point instruction after the instruction that caused the exception⁴. As a consequence, it is impossible to use this exception to ‘fix up’ the denormal output of an operation, but this is of no concern to a profiler which should not modify the behaviour of the program being profiled.

In contrast, denormal exceptions, are *pre-operation* exceptions. They occur when the instruction is fetching its operands, and before any calculation has been performed.

```
for (i=1; i<N-1; i++) {
    for (j=1; j<N-1; j++) {
        cur[i*N + j] = 0.25 * (
            prev[(i-1)*N + j] +
            prev[(i+1)*N + j] +
            prev[i *N + j-1] +
            prev[i *N + j+1]);
    }
}
```

Fig. 4.3: jacobi inner loop

```
0x08048730: flds    (%edi)
0x08048732: add     $0x1,%eax
0x08048735: add     $0x4,%edi
0x08048738: fadds   (%esi)
0x0804873a: add     $0x4,%esi
0x0804873d: fadds   (%ebx)
0x0804873f: add     $0x4,%ebx
0x08048742: fadds   (%ecx)
0x08048744: add     $0x4,%ecx
0x08048747: fmul    0x080488e0
0x0804874d: fstps   (%edx)
0x0804874f: add     $0x4,%edx
0x08048752: cmp     $0x1ff,%eax
0x08048757: jne     0x08048730
```

Fig. 4.4: Compiled code

To illustrate when floating point exception handling and the distinction between *pre*- and *post-operation* exceptions, we shall use the inner loop of the jacobi application from [Sec. 6.1.1](#). The inner loop, and the machine code instructions it compiles to are shown in [Fig. 4.3](#) and [Fig. 4.4](#).

We shall assume that the 32-bit value read from memory by the third `fadds` instruction at `0x08048742` is a denormal value, and that the 32-bit value written by the `fstps` at `0x0804874d` is also denormal. This leads to the following sequence of events:

The first 7 instructions execute normally:

```
0x08048730: flds    (%edi)
0x08048732: add     $0x1,%eax
0x08048735: add     $0x4,%edi
0x08048738: fadds   (%esi)
0x0804873a: add     $0x4,%esi
0x0804873d: fadds   (%ebx)
0x0804873f: add     $0x4,%ebx
```

⁴This may be many instructions after the instruction that caused the exception if there are integer instructions between the two floating point instructions.

The 8th instruction begins:

```
0x08048742: fadds  (%ecx)
```

The FPU loads the 32-bit value from memory, decodes it, sees it is a denormal, and sets the FPU's DE (Denormal Operand Exception) flag. It then abandons the instruction by performing no operation and leaving the instruction pointer unmodified.

The CPU begins execution of the next instruction. Since the instruction pointer hasn't been changed, this is still the instruction at 0x08048742. Before starting the instruction, the FPU sees the DE flag is set and triggers a Floating Point Exception.

The FPE handler runs, and on completion, the CPU returns to the instruction at 0x08048742 and executes it.

The 9th and 10th instructions execute normally, and produce a denormal on the stack:

```
0x08048744: add    $0x4,%ecx
0x08048747: fmul    0x080488e0
```

The 11th instruction begins:

```
0x0804874d: fstps  (%edx)
```

To perform the store, the FPU converts the 80-bit float to a 32-bit float, and produces a denormal value. It performs the store to memory and then sets the FPU's UE (Underflow Exception) flag and increments the instruction pointer.

The CPU continues normally until the next floating point instruction is encountered:

```
0x0804874f: add    $0x4,%edx
0x08048752: cmp    $0x1ff,%eax
0x08048757: jne    0x08048730
```

At the next iteration of the loop (4 instructions after the store), a floating point load occurs:

```
0x08048730: flds   (%edi)
```

At the beginning of the instruction, the FPU sees the UE flag set (from the previous store), and before starting the instruction, triggers a Floating Point Exception. The FPE handler runs, and on completion, the CPU returns to the instruction at 0x08048730.

4.4.2 Using exception handlers

Because of the fact that denormal exceptions are *pre-operation* exceptions, a floating point exception handler must be careful to remove the conditions that caused the denormal exception in the first place. Because of the *pre-operation* nature of the exception, when the IRET instruction returns from the exception handler the program restarts the offending instruction. If the instruction's operands are still denormal, and the denormal mask is still cleared, another denormal exception will immediately be triggered for exactly the same instruction, leading to an infinite loop.

However, it is not sufficient for the exception handler to simply set the denormal mask flag to resolve this infinite loop. If the exception handler does this, when the offending instruction is restarted it will not trigger a denormal exception, and will execute completely, which is as desired. However, now denormal trapping is disabled for the rest of the program, and the exception handler has been called for just one instruction.

At first sight, it would appear that the only workaround for this scenario would be either to implement a software [FPU](#) interpreter in the exception handler that simulates the behaviour of the offending instruction, sets the return address to the next instruction, and then returns to normal program execution; or to somehow rescale the instruction operands before resuming execution so that they are all normal, but still produce the same results. Both of these schemes are complex, slow and error prone.

Fortunately, the 80x86 has a feature that can be used to avoid this. The 80x86 has a register called EFLAGS which, among other things, holds a number of 'system flags' that control program execution. One of these flags is called the trap flag, and setting it enables a single-step mode that is usually used for debugging purposes. In single-step mode, the [CPU](#) generates a debug exception after every instruction completes execution. This is perfect for a profiler's needs. Denormal profiling can be enabled as follows:

- At the beginning of program execution set up an exception handler for denormal and debug exceptions and clear the denormal mask and the trap flag.
- When a denormal operand is encountered, the denormal exception handler will be entered. This logs the event, sets the denormal mask, sets the trap flag, and returns to the program.

- The offending instruction is restarted and runs to completion because denormal is set. When the instruction finishes, the debug exception handler is entered. This handler clears the denormal mask again, clears the trap flag, and returns to the program to start the instruction directly after the offending instruction.
- Now the denormal mask is clear, so when the next denormal operand occurs, the denormal exception handler will be entered again.

This use of two different trapping facilities means two exceptions are required to detect denormal inputs instead of the one required by denormal outputs.

4.4.3 Exception handling in Linux

DIP will need to be notified whenever floating point and debug exceptions occur. However, for security and stability reasons, modern operating systems cannot allow ordinary user processes to install their own exception handlers directly. On the x86, the set of defined interrupt and exception handlers is stored in an area of memory called the Interrupt Descriptor Table (IDT). Each entry in the IDT is called an *Interrupt Gate*, and holds the address of the handler along with flags defining what processor mode the CPU should enter before running the handler.

If ordinary user code was allowed to write directly to the IDT, it could, for example, install an invalid opcode exception handler that switches the CPU to supervisor mode, examines the kernel's process descriptor table, and changes the user ID of the current process to 0, the root user. Once the handler is installed, user code could then attempt to execute an illegal instruction, thus triggering the exception handler and gaining root access and full control over the entire machine.

Another reason for disallowing direct access to the IDT is that the correct functioning of the OS requires the kernel to have complete control over the IDT entries associated with device I/O interrupts, timer interrupts, and page fault handling.

Instead of allowing direct IDT access, the Linux kernel writes its own Interrupt Gates into the IDT at startup, pointing at a group of general-purpose kernel exception handlers. A user program can register a user function called a signal handler to be called using the UNIX signals API. Some of these signals—such as the process control signals SIGHUP, SIGINT, and SIGKILL—correspond to abstractions provided by UNIX itself, but others—such as SIGILL and SIGFPE—map to processor exceptions. When a processor exception occurs, the kernel gathers information about the state of the processor at that point in time, populates a *ucontext* (user context) data structure with this state, switches back to user mode and calls the

registered signal handler.

Once invoked, the signal handler may inspect the ucontext, which stores the values of all the CPU and FPU registers including the instruction pointer and may modify some of them before returning. When the signal handler returns, control returns to the kernel, which updates the processor state using the contents of ucontext, and finally exits from the kernel exception handler returning to user mode.

This process is illustrated in Fig. 4.5.

The requirements of DIP can be implemented on Linux on 32-bit x86 as follows:

- On x86, the instructions to read and write the FPU Control Word are unprivileged, so the appropriate FPU exception masks can be cleared directly from user code.
- Under Linux, the x86 general purpose floating point exception handler maps to the SIGFPE signal, and the debug single-step exception handler maps to the SIGTRAP signal.
- Since copies of the FPU Control Word and the EFLAGS register are stored in ucontext, and the kernel updates these registers after the signal handler returns, it is possible to set the denormal mask and enable single-step mode in the SIGFPE handler. Similarly it is possible to clear the denormal mask and disable single-step mode in the SIGTRAP handler.

4.4.4 Library interposition to profile binaries

Using the APIs above, DIP can now be implemented as a shared library. The profiler is notified when a denormal operand exception occurs and can perform any necessary logging subject to the limitations on the use of system calls within signal handlers. DIP can be used with any program by linking the program against it, and calling its initialisation function somewhere near the entry point to the program. Other than the slowdown caused by the invocation of the signal handlers, there should be no changes in behaviour visible to the program itself. Indeed, this is one of the benefits of the profiler: if there are no denormal exceptions in the profiled program, it will run at full speed.

However, sometimes it is undesirable or impossible to recompile and relink a program. This can occur if the program is supplied only as a binary, or if the toolchain and development libraries necessary to build the program are unavailable or difficult to install. A feature available in a number of mainstream UNIXes including Linux, the BSDs and Solaris called *library interposition* can be exploited

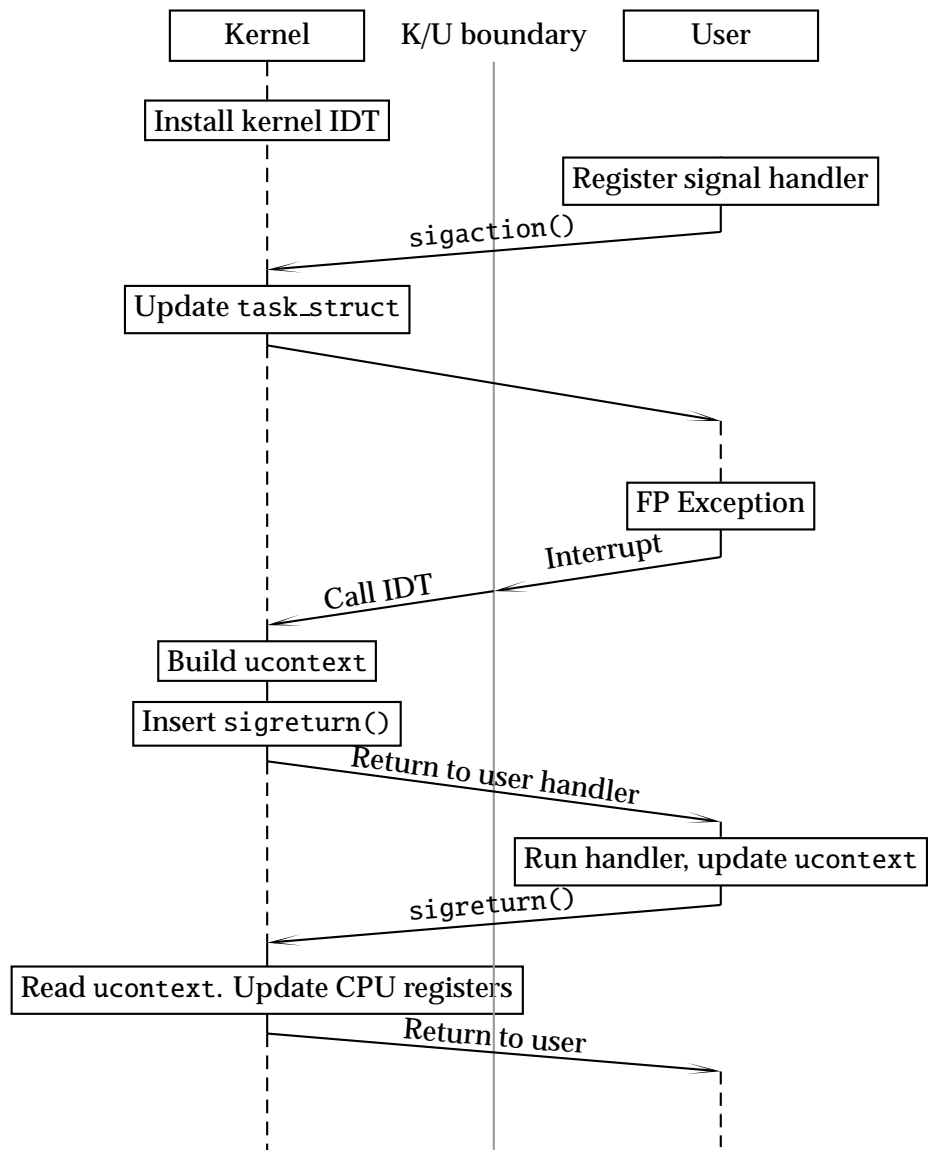


Fig. 4.5: Linux FPE handling

Iteration	Denormals
1	13969
2	30043
3	23805
4	15608
5	0
...	0

Table 4.5: Denormals at cfd-sor startup

to make DIP more convenient to use under these circumstance. This feature takes advantage of the fact that on program startup, the dynamic linker checks if the environment variable `LD_PRELOAD` is set, and if it is, reads a list of dynamic libraries to load into the process before loading the libraries specified in the executable file itself.

Library interposition is sometimes used to wrap library functions (such as `malloc`, `open` or `gettimeofday`) by providing a small wrapper function that performs whatever replacement activity the preloaded library requires, and then, optionally, passes control to the real function provided by the expected library (such as `libc`). The wrapper can perform logging, argument validation, or may modify either the arguments or return value of the real function.

DIP does not need this function wrapping capability, but the compiler can be instructed to mark a function in a shared library with the *constructor* attribute. When a function in a library is marked as a constructor, the function is called at library load time and before a program that depends on that library starts running. Using the constructor attribute in combination with `LD_PRELOAD`, DIP can initialise itself, install the appropriate signal handlers and clear the needed [FPU](#) exception masks before the target program enters `main()`. A simplified version of DIP showing how the main mechanisms are implemented on Linux can be seen in [Appendix C](#).

4.4.5 Example of DIP in use

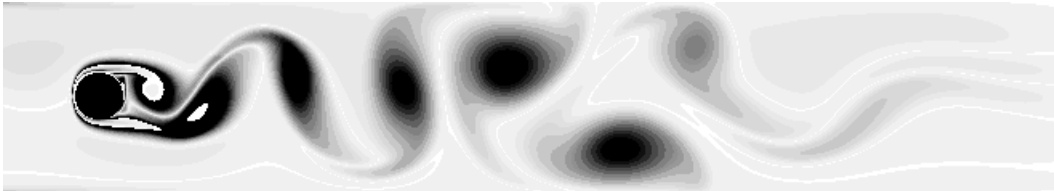


Fig. 4.6: Kármán vortex sheet in cfd-sor

A small application known to generate denormal values at startup can be used

to show the behaviour of DIP. The application is a simple 2D computational fluid dynamics code which operates on a regular grid with obstacle cells and a number of different boundary conditions. It generates a tentative velocity term for each cell based on assuming a uniform pressure across the grid; calculates the pressure value for each cell by solving the Poisson equation using red/black successive over-relaxation; and finally updates the velocity based on the new pressure values.

The algorithm is simplistic and inefficient, but is useful for pedagogic purposes. During testing it emerged that the first few iterations ran more slowly than expected, and later iterations ran faster. By running it under a version of DIP modified to emit denormal counts and denormal instruction profiles, it was found that the first four iterations of the program produced denormal values.

[Table 4.5](#) shows that over the first four iterations, some denormal values are generated, then they build up to a peak, and finally die away. This information can be used to determine if any of the application's slowdown is caused by large numbers of denormal values.

[Table 4.6](#) is generated by instructing DIP to record the addresses of the instructions that cause the exceptions. This shows the instructions that cause the exceptions. They all appear in the `compute_tentative_velocity` and `poisson` functions, and are all load, multiply, add and subtract instructions. However, DIP provides no information beyond that, for example where the values came from or where they will be stored.

This leads to a discussion of the limitations of DIP and other profiling tools based on floating point exceptions.

4.4.6 Limitations of DIP and exception-based profilers

DIP as described above is useful for identifying the exact instructions in a program where a denormal value is generated or used, but has a number of important limitations:

- DIP depends on the target application not setting the [FPU](#) exception masks. If the target code modifies the [FPU](#) Control Word, the profiler stops working. This occurs, for instance, with Sun's Java Virtual Machine on Linux, preventing this tool from profiling Java code.
- As it stands, DIP can only identify the instruction that triggered the exception, it does not report the value in question. This value is available to the exception handler, and with some extra coding could be reported if necessary.

Address	Denormals	Instruction	
compute_tentative_velocity			
804a02c	457	fadds	-0x78(%ebp)
804a055	429	flds	-0x14(%ebp)
804a092	429	flds	-0x14(%ebp)
804a1cc	498	fadds	-0x78(%ebp)
804a1f8	426	flds	-0x14(%ebp)
804a22f	179	flds	-0x14(%ebp)
poisson			
804a7f7	7473	flds	(%eax)
804a80e	7473	flds	(%eax,%ecx,1)
804a825	7462	flds	(%eax,%ecx,1)
804a838	6974	flds	0x8(%ebx,%eax,1)
804a851	7473	fmuls	-0x1c(%ebp)
804a86b	6971	fmuls	(%ebx,%edx,1)
804a886	200	fsubs	(%eax,%ecx,1)
804a951	7473	flds	(%eax,%edx,4)
804a958	7473	flds	(%esi,%edx,4)
804a976	7462	flds	(%eax,%ecx,1)
804a989	6974	flds	(%esi,%ecx,4)
804a9b0	6971	fsubs	(%esi,%edx,4)
804a9ca	200	fsubs	(%eax,%edx,1)

Table 4.6: Denormal instruction profile in cfd-sor

- DIP does not report the source address of denormal arguments, or the destination address of underflowing writes. If the operand is on the x87 stack, or is an immediate address, this location can be read directly from the instruction itself. However, usually an address is specified using one of Intel’s indirect addressing modes. For array accesses, addresses are typically specified using operands of the form *base register + (index register × scale)*. To perform the necessary effective address calculation, DIP would need to parse enough of Intel’s opcode layout to identify all the floating point instructions, how many operands they take, and needs to decode the addressing bytes of the arguments to the instruction. This information can then be used along with the saved copy of the [CPU](#) registers in the signal handler to calculate the required effective address.
- However, this leads to another limitation: DIP as it stands cannot perform any data-flow analysis. Frequently the sources or destination of an instruction will either be temporary values on the [FPU](#)’s internal stack, or a function’s arguments/temporary working space on the program’s call stack. These temporary values are of no direct use to the programmer, as they either reflect the details

of the compiler code generation or lower level details of the application's implementation rather than the higher level semantics of the programmer's code. For example, knowing that all the instructions with denormal arguments are in a small helper function is of little use if that function is called from dozens of places throughout the application. The programmer is interested in how the code takes values in the application's data structures and generates or propagates exceptional values from them. To do this, a tool needs to be able to track some the instructions between the denormal exceptions as well as the ones that cause the denormal exceptions.

- Furthermore, from a performance variability perspective, the programmer is interested in how the denormal values and arithmetic are distributed throughout the application's data sets. If the denormals are distributed completely uniformly, then there will be a slowdown, but no performance variability and denormal arithmetic becomes less of a concern.
- Finally, due to the design of the x86 instruction set, it is quicker to copy floating point values from one location to another using a single integer instruction instead of a floating point load/store pair. This occurs when copying data into temporary variables, or during function calls. Similarly, assignments are often performed using integer `mov` instructions, and arguments are generally passed to functions using the integer `pushl` instruction. Some instances of this integer optimisation can be seen in [Appendix D](#). Because these are integer instructions, they will not generate floating point exceptions under any situation, and thus completely bypass DIP or any other exception-based profiler. This adds an additional complication to data-flow analysis.

4.5 Summary

This chapter has examined why floating point arithmetic is necessary for many applications, the complexities of implementing it, and why a hardware implementation is desirable. It discussed salient details of the IEEE-754 standard, and showed some of the implementation choices made in some popular [CPUs](#).

The exception handing models of the Intel x86 [CPUs](#) and how they map to the signal handling API in Linux are described. This API is used to write DIP, a simple floating point exception based denormal profiling library, which is then used to profile a small application. The profiler allows a profile to be generated which lists each of the floating point instructions that read denormal values from memory

and how often they occur. Apart from when denormal exceptions occur, the rest of the application runs at full speed. This information allows the programmer to determine whether or not any of the performance variability in an application is due to denormal arithmetic.

Finally the limitations of this profiler are described. These limitations stem from the fact that profiles of individual instructions aren't very useful without further context — it is desirable to see how the values are distributed throughout the data sets and where they came from and where they go to, and this context cannot be provided by the exception causing instructions alone. Some of these profiler issues could be resolved by a more complex implementation, but others are unavoidable given the x86 instruction set.

The next chapter introduces a dynamic binary instrumentation tool called Valgrind, and shows how it and a variation of taint analysis can be used to implement a denormal tracer that overcomes the of limitations exception-based profilers.

Implementing a denormal tracing tool using Valgrind

The previous chapter described how denormal arithmetic can be the source of data-dependent slowdowns even in otherwise regular and predictable applications. In it, I wrote a profiling tool for Linux x86 to identify which floating point instructions read denormal values from memory as arguments. Although this tool can be useful to determine whether significant numbers of denormal operations occur, it is of limited use when trying to isolate the original causes of denormal arithmetic in an application. In order to find the origins of denormal data, a tool would need to perform data-flow analysis, and to do this, it would need to watch the behaviour of more than just the instructions that cause denormal exceptions.

No simple exception-based tool can provide the primitives needed for this, and so a more powerful set of tools must be employed — those that allow arbitrary sequences of instructions to be monitored and allow values to be tagged and traced throughout a program. This process bears some similarities to an area in software security called taint analysis, and the monitoring of running programs is called dynamic binary instrumentation. A number of tools exist to facilitate dynamic instrumentation, such as DynamoRIO[BGA03], Pin[LCM⁺05], and Valgrind[NS07b].

Both DynamoRIO and Pin work by preserving the existing instruction stream of a program where possible, and by explicitly adding instrumentation instructions or calls to analysis routines at particular points in the program. This approach is suitable for implementing profiling tools that need to interrupt or monitor program behaviour at specific, well defined points during a program's execution, but is less useful when all or most of a program's instructions need to be monitored. It will be shown later in the chapter that monitoring most of a program's instructions is required for denormal tracing.

Valgrind, by contrast, operates by decompiling an entire program binary into an intermediate representation, adding instrumentation code, and finally optimising and recompiling the result back into blocks of machine code. It also supports a scheme called 'shadow memory', and makes a clear distinction between memory used by the instrumentation for workspace or metadata and that used by the program under analysis. Both of these attributes significantly facilitate the tracking process, making Valgrind a more suitable tool for denormal tracing.

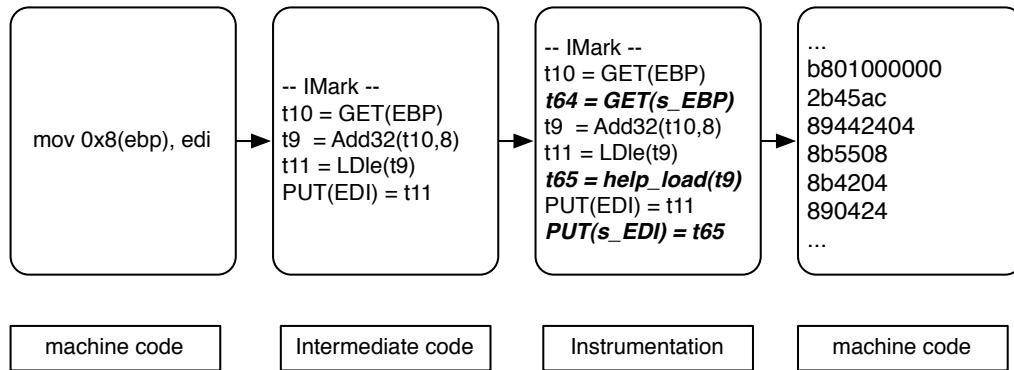


Fig. 5.1: Valgrind instrumentation process

This chapter describes both Valgrind and taint analysis, and examines how Valgrind has previously been used to implement a taint analysis system called TaintCheck. It shows how denormal tracing is a more open-ended problem than taint analysis, but by applying suitable constraints it is possible to design a denormal tracer which I implement using Valgrind. This tracer will help identify where the denormals in an application come from, and this can be used to isolate and remove the unnecessary denormals thus producing an application with less data-dependent performance variability.

5.1 Introduction to Valgrind

Valgrind is an open source binary profiling framework written to aid the construction of dynamic analysis tools. It functions by taking an executable¹, and on a [JIT](#) basis converting basic blocks of code in the executable into an Intermediate Representation ([IR](#)) on demand, inserting instrumentation instructions into the [IR](#), optimising and compiling the [IR](#) into native machine code, and finally saving the resulting code in a cache from where it can then be executed.

The intermediate representation consists of a sequence of μ Ops somewhat reminiscent of a cross between a RISC instruction set and the graph structures produced by the intermediate stages of a compiler. In Valgrind's [IR](#), access to memory is performed using explicit Load expressions and Store statements. Extending this, Valgrind maintains what it calls the 'guest state' which is essentially the register file of the guest [CPU](#). Guest registers are read from and written to using explicit Get expressions and Put statements. To store values and the results of operations,

¹The target executable is called the guest program in Valgrind terminology.

Valgrind's [IR](#) provides an effectively unlimited supply of temporaries which can have expressions assigned to them only once, but can be read from many times. The [IR](#) provides a variety of side-effect free operations which can be used to build up expression trees. These expressions can then be used as the arguments to statements. The [IR](#) also provides facilities to perform conditional jumps from one basic block to another, to perform system calls, and to 'call out' to tool functions in the host for cases where it would be difficult to implement functionality in the [IR](#) itself.

Once Valgrind has translated a basic block into [IR](#), it transforms it so that expressions and statements only take temporaries as arguments, and the results of expressions are immediately assigned to new temporaries. This has the effect of 'flattening' any expression tree, and simplifies matters for the instrumentation code. The instrumentation code is given the [IR](#) for a basic block, and may transform it, usually by adding extra instrumentation expressions or statements necessary to its monitoring role.

After instrumentation, the [IR](#) is optimised, removing any redundant or dead expressions, and compiled back into host machine code which is stored in a basic block cache. Valgrind has an execution engine which produces compiled basic blocks as needed, and executes them from the cache. When a block finishes running, it returns to the execution engine which selects the next block to be run, compiles it if necessary and then runs it.

Because of how the [IR](#) is constructed and modified, Valgrind controls the apparent execution of every guest instruction, and from the point of view of the guest program it appears that it is running directly on the host hardware. As is to be expected, the instrumented code runs slower than the original program, but unlike traditional instrumentation, the effects of this slowdown can be hidden from both the instrumentation tool and the guest program.

Consider, for example, `gprof`, the UNIX call graph execution profiler [[GKM82](#)]. It works by adding a call to a monitoring routine to the entry and exit points of each function in a program at compile time. When the program runs, and the monitoring routine is called, it records the run time of each function, along with where it was called from, and the number of times it has been called. One of the problems with this approach is that it adds a variable amount of overhead to the execution time of the program: when compared to the uninstrumented code, programs with many calls to small functions will be penalised more than programs with fewer calls to larger functions. Furthermore, the memory accesses caused by the instrumentation updating its internal data structures may, in some cases, distort

the cache utilisation patterns of some programs. All this occurs because there is no clear way to distinguish instructions in the instruction stream that are part of the instrumentation tool from those that are a part of the guest program.

Valgrind tools can avoid this problem entirely. A simple gprof-like tool could be written as follows:

- The tool maintains its own data structures, which includes a list of all the functions in the guest program, and the statistics associated with them. The tool also maintains a pointer to the currently executing function.
- When the [IR](#) is generated, the instrumentation tool can add [IR](#) code after every guest instruction to increase the current function's instruction count by one.
- When the [IR](#) is generated, if the tool detects a function entry or exit point in the guest code block, it adds an [IR](#) instruction that calls an external instrumentation routine to change the current function pointer.

The net result is that although many [IR](#) instructions or external instrumentation calls may be executed for each instruction of guest code, and a good deal of house-keeping will occur in the background, the tool will only count one instruction for every instruction of guest code that would have been actually executed without instrumentation. In other words, there is a clear separation between the instrumentation code and the guest program itself, thus the instrumentation process can be arbitrarily slow and expensive without affecting the statistics gathered.

5.2 Shadow Memory

As well as providing a framework for dynamic analysis tools, Valgrind has built-in support for a powerful feature called Shadow Memory [[NS07a](#)]. Shadow Memory is an instrumentation technique where every byte of memory and every byte of the user-accessible [CPU](#) registers has a piece of metadata associated with it. This metadata is controlled solely by the instrumentation code, and allows it to monitor the usage of memory and how the guest program reads, modifies and writes memory while it executes.

Perhaps the most widely used Valgrind tool that uses Shadow Memory is Memcheck. Memcheck associates 1 bit of addressability metadata with each byte of memory, and 8 bits of validity metadata with each byte of memory and each byte of the [CPU](#) registers.

The addressability metadata determines whether that byte of memory may be read or written by the guest program. On initialisation, all memory except for the stack, heap and mapped data from the executable is marked as non-addressable, and Memcheck updates the addressability bits by monitoring allocations and deallocations by the OS and C runtime from functions such as `mmap`, `malloc` and `free`. Attempts to access non-addressable bytes can be used to detect heap overflows, ‘use after free’ errors, and reads and writes to random memory addresses.

```
void f() {
    char *p;

    /* Allocate 100 bytes */
    p = malloc(100);

    /* Write to 101st byte */
    p[100] = 42;
}
```

Fig. 5.2: Heap overflow

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    char *p, *s = "Hello world\n";

    p = strdup(s); /* Copy string */
    free(p);       /* Deallocate it */
    printf("%s", p); /* Use it */
}
```

Fig. 5.3: Use after free error

The validity metadata describes whether each bit in a byte of memory or CPU register has been explicitly assigned a well-defined value. For example, if some memory is allocated using `malloc`, the entire memory range will be marked as addressable but invalid. Instructions that read from these addresses will copy the validity metadata along with the data, as will instructions that perform computations based on invalid operands. Memcheck silently propagates invalid data as far as it can, and signals an error when invalid data is used to calculate the destination of a jump, or as arguments to a system call, or as the address for a memory load/store.

The code in Fig. 5.4 shows 8 invalid bits being copied into an integer; that integer having various operations performed on it — some of which propagate invalidity, some which remove it — the results being stored and then printed.

Note that the correct way to compute validity bits for arithmetic and logic instructions is far from obvious. The scheme used by Memcheck is an approximation that avoids many false positives and is described by Seward and Nethercote in §2.5 and §2.6 of [SN05].

```

void f() {
    char buf[4];
    int a = 0, b, c, d, e, f;

    /* Assign only 3 of the 4 bytes */
    buf[0] = 0x11; buf[1] = 0x22; buf[2] = 0x33;

    /* Copy uninitialised data */
    memcpy(&b, buf, 4);

    /* Operate with uninitialised data */
    c = a * b;

    /* Operations destroying uninitialised data */
    d = b & 0x00ffffff; /* Clears top (invalid) byte */
    e = b | 0xff000000; /* Sets top (invalid) byte */
    f = b ^ b;          /* Always 0 */
    printf("%x %d %d %d\n", c, d, e, f);
}

```

Fig. 5.4: Invalid data propagation

5.3 Taint analysis

With the rise of the World Wide Web, web sites that delivered dynamic content via CGI scripts became common. Since these scripts were generally written by relatively inexperienced programmers, the inputs to the scripts were often used as arguments to potentially dangerous commands without first being sufficiently validated or quoted.

One example of this is ‘SQL injection’[BK04, Mon07]. If the user provides the username ‘_OR_’X=’X to the following script fragment, the query will return a list of all the users, instead of just the one required:

```

$name = $query->param("username");
$db->execute("SELECT * FROM users WHERE name='$name'");

```

This works because the WHERE clause becomes WHERE NAME=’ ’ OR ’X=’X’ which is always true.

A similar problem called ‘shell injection’ occurs when scripts construct shell commands from insufficiently validated input strings and then execute them.[HYH⁺04]

```

$file = $query->param('file');
system("/usr/bin/unzip -d /tmp $file");

```

Here, if the user provides a filename such as `foo.zip; rm -rf /`, the string passed to the shell becomes `/usr/bin/unzip -d /tmp foo.zip; rm -rf /`, which is two commands — one to unzip a file, and the other to erase all files on the system.

Because of the prevalence of these kinds of programming errors, a *taint mode* was introduced into the PERL scripting language to help detect these problems.

When PERL's taint mode is enabled, the PERL interpreter automatically associates a taint flag with every 'scalar'² value in a PERL script. Data internal to the script, e.g., literals and values derived entirely within the script are marked as untainted. However data that enters the script from external sources, such as environment variables, command line arguments, and the results of some system calls are marked as tainted. Operations on tainted data generate more tainted data; for example concatenating a tainted and an untainted string generates a new tainted string.

Unlike MemCheck's validity computations, PERL uses a pessimistic scheme to calculate the taintedness of the results of operations on data. It simply asserts that if any of the operands are tainted, then all the output is too. This may seem to have the scope to generate false positives, but since there is only a one bit taint flag per scalar, and the purpose of taint mode is to enforce string validation, it is a reasonable approximation for this problem domain.

If tainted data is used with a function that performs a potentially dangerous action, such as writing to a file, running an external program, or accessing a database, then the PERL interpreter generates an error and halts the script.

In PERL, the taint flag is removed from input data by first validating the data using a regular expression. The substrings returned from a regular expression match are flagged as untainted, and these untainted values can then be used safely with the actions mentioned above.

5.3.1 Taint analysis using Valgrind

By virtue of the fact that PERL is an interpreted language, and that all values in the language are strongly typed and have a data structure internal to the interpreter associated with them, it was relatively straightforward to add a taint mode to the PERL runtime. To perform dynamic taint analysis on arbitrary binaries is much more complex.

A 2005 paper by Newsome et al.[[NS05](#)] describes a Valgrind-based system called

²In PERL, scalars are the non-composite data values; that is to say individual strings, numbers, or references.

TaintCheck which provides tools to trace the propagation of user-generated data through a program and to detect if it is eventually used in dangerous ways.

As with PERL's taint mode, and Valgrind's Memcheck, TaintCheck has three major components.

- The first is called TaintSeed and marks data as tainted or trusted. Every byte of memory and the CPU registers in TaintCheck is shadowed by a pointer to a taint data structure. When a program receives data from a network socket, TaintCheck's default policy marks all the data received as tainted. The taint structure associated with each byte records the parameters passed to the last system call, the results from the call, and the user stack at the time of the system call. This later allows the developer to determine the initial source of the data.
- TaintCheck's second component is called TaintTracker and provides the taint propagation of TaintCheck. The taint propagation policy is done on a per-byte basis, and is similar to that of PERL's — loads and stores directly copy the taint information, and arithmetic and logic operations assume pessimistically that if any of the source operands are tainted, then the result should be too. TaintTracker includes a couple of special cases to handle some common x86 instruction patterns, such as XOR-ing a register with itself to zero a register (and thus making its value untainted). When propagating taint information, TaintTracker can either make the new value point at the same taint structure as the tainted source value, or create a new taint structure which records the current stack, and contains a pointer back to the earlier taint structure. This latter scheme consumes much more memory, but provides an exact chain of all the operations that propagated the tainted value from the point where it was initially injected into the program to the current moment in time.
- The final component in TaintCheck is TaintAssert, which detects whether tainted data is used dangerously. By default, TaintAssert defines as dangerous the use of tainted data as a jump address, or as the format argument to a printf-style function. These two scenarios cover a majority of internet-based security exploits. When dangerous uses are detected, they can be logged, along with a chain of all the taint structures for offline analysis.

As with PERL's taint mode, TaintCheck focusses on detecting dangerous uses of user data that can potentially be used as attack vectors by malware. Because of this, the pessimistic propagation of taint information is appropriate and simplifies the implementation substantially.

5.3.2 Taint analysis and denormal tracing

TaintCheck tags user inputs as tainted, traces tainted data throughout the program, and warns and logs dangerous uses of tainted data. At first glance, it would seem that a taint tracing mechanism like this could be adapted for tracing denormal arithmetic. Certainly an analogue of TaintTracker could be written that instruments every floating point operation and if any operands or the output of an operation is denormal, uses shadow memory to ‘tag’ the value with the appropriate metadata. Any further uses of these tagged values could be examined to allow propagation of denormal values through the code to be monitored.

However, in practice there are two substantial difficulties with implementing a scheme like this. Firstly the differences in the ‘lifecycle’ of a denormal value compared to a tainted value, and secondly how a TaintAssert analogue might be defined.

5.3.3 Denormal vs taintedness lifecycle

In TaintCheck, there is a clear point where tainted data enters a program, and taintedness is a well defined binary state: all the data read from a network socket is defined as tainted, everything else is not. Furthermore, at least for the common cases, since TaintCheck is using a ‘pessimistic’ approach, there is a reasonably natural definition of how taintedness should be propagated as the result of various CPU instructions that copy or operate on data previously marked as tainted. Because of these definitions, although taintedness can be copied or propagated, it is impossible for tainted data to be created *de novo* — taintedness always originates from a memory buffer written to by the read and recv family of system calls. Similarly, taintedness cannot be destroyed as such. Tainted values can be overwritten thus removing the value along with the taint, but taintedness on its own cannot be removed from a value.

The end point of the lifecycle of a tainted value is also well defined. Three possible things can happen to a tainted value: after some number of harmless operations are performed on it, it can be ignored for the rest of the program; or it can be overwritten by another value, thus destroying it; or it can be used dangerously, thus triggering TaintCheck to generate an error. In this third case, because the origin of all tainted data is explicitly recorded, and because each operation on tainted data may also be recorded, TaintCheck can build a complete history of everything that happened to that value, from its injection into the program, to its final dangerous use.

In comparison, a denormal tracer can not rely on similar properties. Denormals

may be injected into a program as user data, but it is equally possible for a sequence of operations to create denormal values ‘mid program’ from what had been normal data. One common way this can happen is by some iterative process causing input values to converge on zero. Another is by a routine that uses the difference between two almost identical data sets.

By the same token, it is possible for the denormalcy to be removed from a value while preserving some of the information contained in it, either by adding random normal noise to it, or by scaling it to a sufficiently large value. Denormals can also be removed by flushing to zero, but this is the equivalent of overwriting the denormal with ± 0 , thus destroying it.

These two facts combined mean that unlike for taintedness, short of recording a trace of every operation in a program, there is no clear way to identify the history of a given denormal once it is detected. Even if the original user data was denormal — something of which there is no guarantee — the data may have become normal at some point in the program, and then denormal again later on.

The problem of an incomplete history is compounded by the fact that because of the design of the x86 instruction set, and because of Intel’s calling conventions, often an optimising compiler will perform floating point copies, floating point assignments, and manipulate the call stack on function entry/exit using integer instructions.³ These optimisations mean that the first operations on some user data might be to copy them to the stack using integer instructions. Even if the floating point value is itself denormal, it must be determined that the memory area in question is definitely used for floating point values. If not, a tool will incorrectly flag integer instructions writing small integer values to memory⁴ as denormal. Values in this range commonly occur as flags and loop indices. This inability introduces further gaps into the history of an eventually denormal value.

5.3.4 Reporting denormal events

For TaintCheck, not only is there a clear and unbroken lifecycle for tainted data, there is also a small set of discrete operations that are regarded as dangerous and when they occur an error can be reported.

From the performance point of view, the very fact that denormal arithmetic occurs at all is a problem, so for a denormal tracer there is no clear distinction between the

³This issue was also a limitation with DIP in [Sec. 4.4](#). Some of these kinds of optimisations can be seen in [Appendix D](#).

⁴All 32-bit integers less than 8,388,608 except 0 have the same binary representation as 32-bit floating point denormals.

operations that cause a denormal to be propagated and a dangerous operation that should be reported.

Also, from a performance point of view, individual denormal operations are not of much interest — what needs to be determined is the how many of them occur, how are they distributed throughout the program's data sets, and whether they persist over time or not. A denormal tracer that simply records individual events will generate a huge trace log that may not offer much insight into a program's behaviour. This implies that unlike for a taint analyser an important aspect of a denormal tracer is how the raw trace data is reduced and reported.

5.4 DART: A denormal tracing tool for Linux

As can be seen from the above, a general purpose implementation of a denormal tracer faces real difficulties. However, if some assumptions are made about how a numerical code is likely to behave and the language used for implementation, it is possible to avoid or resolve these issues.

Using these assumptions, I write a Valgrind-based tool called DART, the Denormal Arithmetic Reporter and Tracer.

- The program is assumed to be written either in Fortran 95 or earlier, or C99 or earlier. These are the two main systems languages used for numerical programming, so this is not a major restriction.
- The program is assumed either to use fixed-sized data structures allocated at compile time, or to use the standard memory allocation functions to allocate variable-sized structures at runtime from the heap.
- The program is assumed to primarily manipulate data from these fixed or dynamic structures.
- The program is not assumed to be compiled with low levels of optimisation, but the compiler must produce debugging information so the program's data structures can be identified. Specifically, DWARF3 annotations, described shortly, are required. Debugging information is not needed for any dynamically linked libraries.

By restricting the language to C or Fortran, and requiring the system memory allocator, the debugging information and a knowledge of stack frame layouts can be used to locate the major floating point data structures at runtime. Once this is

determined, those memory locations can be marked and watched for loads, whether performed by integer or floating point instructions.

When denormal values are loaded from these locations, the values are tagged with their origin address, and traced as they propagate through the program. This solves some of the origin identification problem mentioned in [Sec. 5.3.3](#).

Since the program is assumed to manipulate primarily these marked data structures, the lifecycle of most floating point values is going to start with a load from a marked data structure, followed by a number of uninteresting ‘housekeeping’ operations. Then it will be operated on by one or more floating point instructions, possibly interleaved with more housekeeping instructions, and finally the value might be written back to another marked data structure.

Since all denormal values are tagged as soon as they are loaded from marked memory, they can be traced until they reach a floating point operation. Then the values used with the floating point operation can be recorded, and the resulting value traced until it is written back to one of the marked data structures. By analysing the patterns of these load/floating point operation/store sequences, the flow of denormals through the program can be traced, as well as what floating point operations are responsible for the transformations of the program data.

5.5 Memory management

To implement the denormal tracing scheme outlined above, a detailed understanding of how memory is divided up and used by Fortran and C programs under Linux is needed. To simplify the discussion, and to avoid irrelevant complications in the implementation, we assume the process is running on a default Linux 2.6.x kernel on x86 running in 32-bit mode. This assumption is not one that fundamentally restricts the design of the denormal tracer — with some expansions of the data structures, it can be modified to deal with 64-bit addressing.

Under the 32-bit arrangement, each process has access to 4 GB of virtual address space divided up into several areas.

- The very bottom of the memory map, specifically the addresses from 0x00000000 to 0x08047fff are unmapped by the kernel and no process can read or write to these memory locations.
- 0x8048000 is the default load address of an executable. On process startup, various sections of the ELF executable file are mapped into memory at this address.

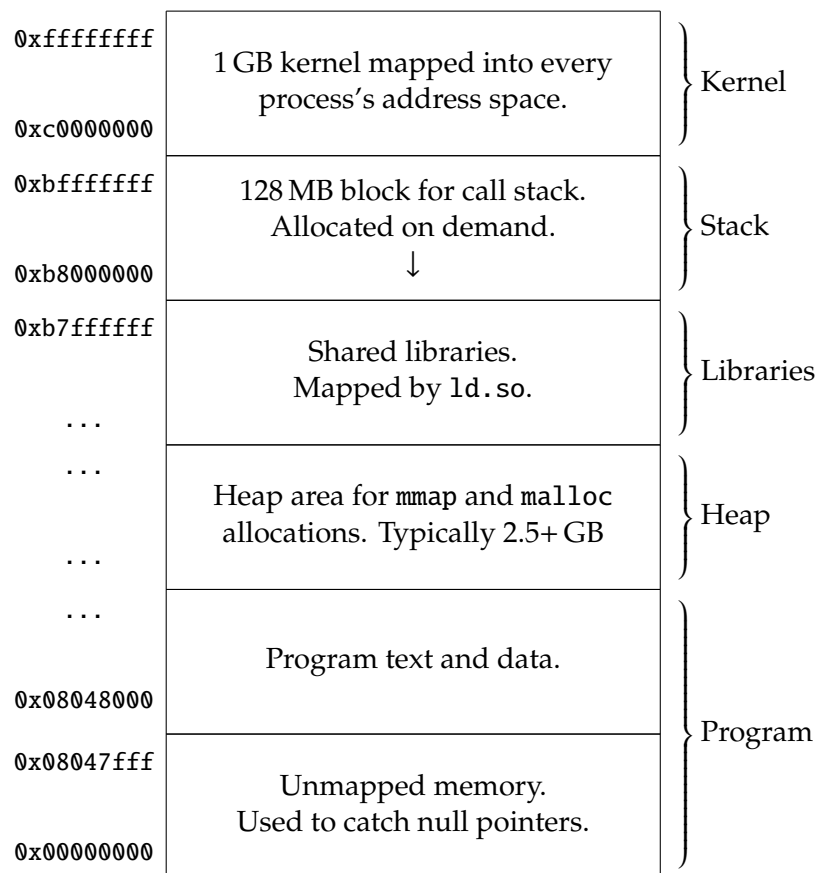


Fig. 5.5: Linux 2.6.x x86 memory map

These sections include `.text` — the program code itself; and the `.rodata`, `.data` and `.bss` sections — blocks of read only data, preinitialised data, and uninitialised data used by the program. Fixed-sized compile-time allocated data is stored in these sections. Collectively the `.rodata`, `.data` and `.bss` sections are often called the data segment of the program.

- The top 1 GB of the memory map, 0xc0000000 to 0xffffffff, is where the kernel is mapped into every process's address space. This space is not accessible to user programs, and is used by the kernel to minimise the overhead of user/kernel context switches.
- Directly below the kernel, from 0xbfffffff down, is the user stack. This holds all the stack frames, i.e., the local variables, function arguments, and return addresses used by the program's functions. It grows downwards in memory and is allocated as required. It can have a maximum size of up to 128 MB.

- Directly below the user stack, from 0xb7ffffff down, is where the dynamic linker maps the shared libraries used by a process. These typically include parts of the dynamic linker, libc and libm.
- Between the lowest area used by a shared library, and the end of the program text is an area called the heap. This is completely unmapped at program startup, and blocks of memory are allocated from this range, either directly using the `mmap` system call, or indirectly by the libc memory allocator.

DART reads the DWARF3 debugging information compiled into the executable at compile time. DWARF is a tree based debugging format that describes the compilation units (i.e., source files) used to build an executable, and all the global variables and subprograms (i.e., functions) in each compilation unit. Each subprogram in turn describes all its arguments and the variables local to the subroutine. DWARF is complicated by the need to describe all the features and formats supported by a large number of languages (such as enumerations, records, unions, COBOL's packed decimals, Ada's ranges, and Fortran's allocatable arrays.) An example of some DWARF annotations can be seen in [Sec. D.3](#). DART limits itself to the basic features required by numerical codes written in C and Fortran, namely floating point values, fixed sized arrays, pointers to arrays, and Fortran array descriptors.

The DWARF3 debugging information for each variable includes a sub-tree defining the exact type of the variable (e.g., a read-only 1024-element array of unsigned bytes, or a dynamically allocated 3D array of reals with unspecified bounds), as well as its location (e.g., starting at memory location 0x8049160, or at offset -0x10c relative to the function's stack frame). DART needs to parse enough of the debugging information to enumerate all the compile-time and run-time allocated floating point arrays used by the entire program, their sizes and types, if available, and the entry point of each subroutine with one of these arrays.

To understand how DART works, the two cases of compile-time allocation and run-time allocation will be examined separately.

5.5.1 Compile-time allocation

Both C and Fortran support the use of either *static* or *automatic* fixed sized data structures. A static variable is one which is allocated a fixed-size space at a fixed location in the data segment in an executable's memory map. Static variables may be declared globally⁵, in which case they can be accessed from any point within a

⁵Global variables are called common variables in Fortran.

program, or may have a local declaration, in which case they are only accessible from within the enclosing subroutine or function. The distinction between local and global variables, however, is only a lexical one which is enforced by the compiler. Static local variables, just like global variables are stored in the data segment, and values stored there are preserved for the entire lifetime of a program. From a machine code point of view, static variables are read and written by hardcoding their addresses into the instructions that access them.

From a DWARF perspective, a static variable is one whose location attribute has a fixed address, that is to say the bytecode describing the location consists of the `DW_OP_addr` opcode followed by the absolute address of the variable.

Automatic variables are ones which are assigned space when a subroutine or function is entered, and return the space when it is exited. This space is taken from the function's stack frame and is allocated for all a function's variables at once by decrementing the stack pointer at function entry, and increasing it again at function exit. As such, an automatic variable has no fixed absolute address, however it does have a fixed offset relative to the current stack frame. Automatic variables are accessed from code by referring to a fixed offset from the frame pointer, and thus depend on the value of the frame pointer when the function is entered.

In DWARF, the location of an automatic variable is defined relative to the stack frame base, i.e., the bytecode of the location is the `DW_OP_fbreg` opcode followed by an offset.

DART can automatically determine the base and size of a static data structure based on the location, its data type, and the array bounds if it is an array. It can do this before the guest program starts executing, and adds the memory range to an internal watch list to monitor for reads and writes.

For automatic variables, DART can determine the entry point for the subroutine it belongs to, and the offset and size of the variable relative to the stack. During JIT compilation, DART instruments the entry point to the subroutine, and when the subroutine executes, an instrumentation helper function is called that gets the current stack pointer, calculates an absolute address for any automatic variables in the subroutine, and dynamically adds the memory range to a watch list. When the function exits⁶ DART removes the memory range from the watch list.

On most operating systems, the stack has a relatively limited size (by default on Linux it can grow up to 8 MB). Because of this limit, using automatic variables for

⁶Determining function exit points in the general case is quite tricky, but for the codes considered in this thesis, instrumenting the return instructions is sufficient.

large data structures is regarded as a bad programming practice and is rarely done. Given this, and since one of the assumptions made about the guest programs is that they spend most of their time manipulating large and persistent data structures, it may seem odd to pay any attention to automatic variables. The next section shows why this is necessary.

5.5.2 Runtime allocation

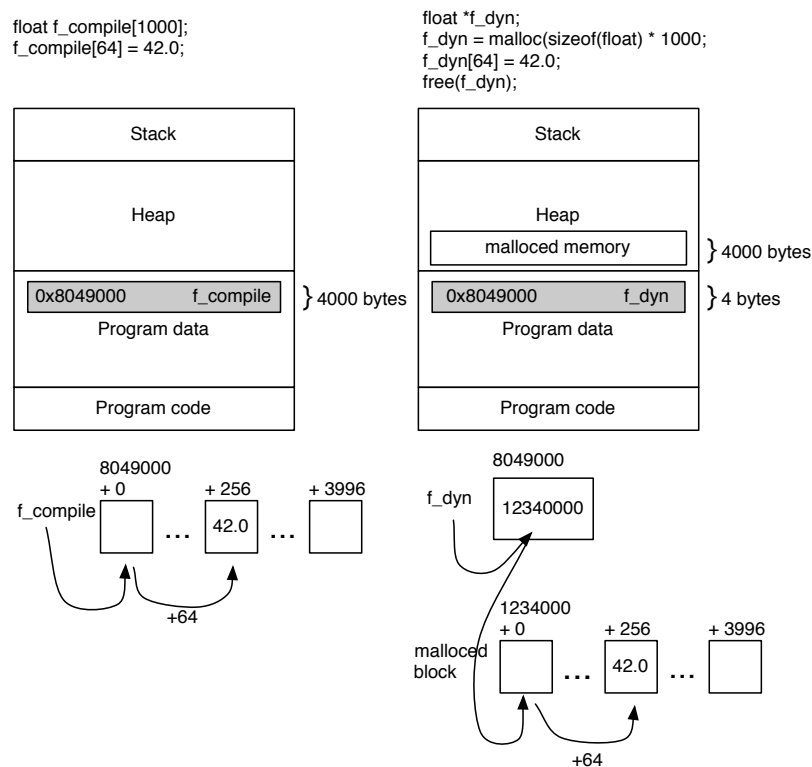


Fig. 5.6: Dynamic and compile time allocation

Dynamically allocatable arrays have always been available in C, via array pointers and the C memory allocator; and were added as a standardised feature to Fortran 90 via array descriptors and some extra keywords for the language. Arrays in C are simpler, and so will be explained first.

A one dimensional array in C is nothing more than a specially typed variable that points at the base address of the array. For statically allocated arrays, this storage is allocated at compile time, as described above. For dynamically allocated arrays, the programmer must use the `malloc` routine to allocate some storage, and then assign the pointer `malloc` returns to the array pointer. The distinction between

dynamically allocated and compile-time allocated memory is illustrated in [Fig. 5.6](#). The array pointer may be a global variable, or as is often the case, it may be an automatic variable allocated on the stack. Once the array has been allocated, a pointer to it is often passed as an argument to other functions which can then operate on it. When the array is no longer required, space can be returned to the system using `free`.

For multi-dimensional arrays, two approaches can be used. The one directly supported in the language is for an N-dimensional array to be a 1 dimensional array of pointers to an N-1 dimensional arrays. This requires many pointers to be set up at initialisation time, and means each array access involves N inherently sequential pointer indirections, so the more common approach is to use an appropriately sized 1 dimensional array and have a small macro to calculate the offset into the 1-D array based on the N array indices.

To monitor C array allocations, DART uses the debugging information to find a list of variables that store global or local pointers⁷ to floats. DART immediately adds the global variables to a ‘to be allocated’ watchlist. At function entry/exit points it adds/removes the local variables to the ‘to be allocated’ watchlist as described in [Sec. 5.5.1](#). After `malloc` returns, the guest program will assign the pointer to the memory allocated to one of these variables. When this assign occurs, DART will detect the write to the array pointer, determine the size of the memory allocated, and add the memory range to the watch list.

After the freeing allocated memory, C programs do not always update the corresponding pointers (by writing zero to them), so instead of watching for zero to be written to an array pointer, DART instead intercepts the `free` routine. If it is called with the address of one of the arrays on the watchlist, that memory range is removed from the watchlist.

For Fortran, the situation is similar to that with C, except that, instead of having array pointers, Fortran uses array descriptors. An array descriptor is a small opaque data structure maintained by the Fortran runtime that contains a base pointer, the data type of the array elements, the dimensionality of the array and the stride⁸ and upper and lower bounds of each dimension. This extra data means that Fortran code can pass an array as an argument to a subroutine without having to specify its dimensionality or bounds as additional arguments. It also allows subsets of the

⁷That is to say static or automatic variables.

⁸The stride of an array is the number of memory locations between adjacent elements of a particular dimension.

array to be presented by creating a new descriptor with the same base pointer and appropriately adjusted bounds and strides.

In Fortran, arrays are allocated and freed using the `allocate` and `deallocate` keywords. Internally, these keywords call routines in the Fortran runtime library that in turn call `malloc` to allocate enough memory, and assign the address returned to the array descriptor's base pointer. Similarly `deallocate` returns memory to the system by calling `free` and marks the array as deallocated by writing zero to the array's base pointer.

This means that DART can handle Fortran dynamic array allocation almost identically to C, except that instead of watching an array pointer directly, it must find the offset of the base pointer within the array descriptor, and watch that instead.

5.6 Metadata and tracing

Once the major data structures manipulated by the guest program are identified, according to the earlier assumptions the sources and sinks that are the start and endpoints of most of the chains of floating point calculations in the program are then known. What remains is to find a way of 'tagging' these values as they flow through the program.

For the purposes of simplifying DART's implementation, it is assumed that only 32-bit floating point values are used in the codes examined. As with the assumption of a 32-bit address space, this is not something fundamental to the design of the denormal tracer — with additional implementation work, and some data structure changes, the tracer could be adapted to deal with both 32-bit and 64-bit values, as well as 64-bit addresses.

As will emerge in the next chapter, depending on the code in question, the assumption of 32-bit values can lead to some rounding differences when comparing simulated execution to direct execution, but it is possible work around this issue.

5.6.1 Tag storage

To perform the tagging, the 32-bit floats the program uses are assumed to be always stored at addresses aligned to the nearest 4 bytes. This is an entirely reasonable assumption, as unaligned loads and stores elicit a substantial performance penalty, and all compilers and memory allocators will arrange data structures to avoid this penalty unless explicitly instructed to do otherwise. As a consequence of this, the bottom two bits of the address of a floating point value will always be zero, and

there are 2^{30} or approximately 1 billion distinct addresses where floats can be stored.

With this alignment assumption, all memory is divided into 4-byte words. Most of the words will contain uninteresting values, but some will contain floating point values to be traced by DART. To do this, Valgrind's shadow memory is used to associate a 4 bytes tag with every word of memory and with every integer and floating point register.

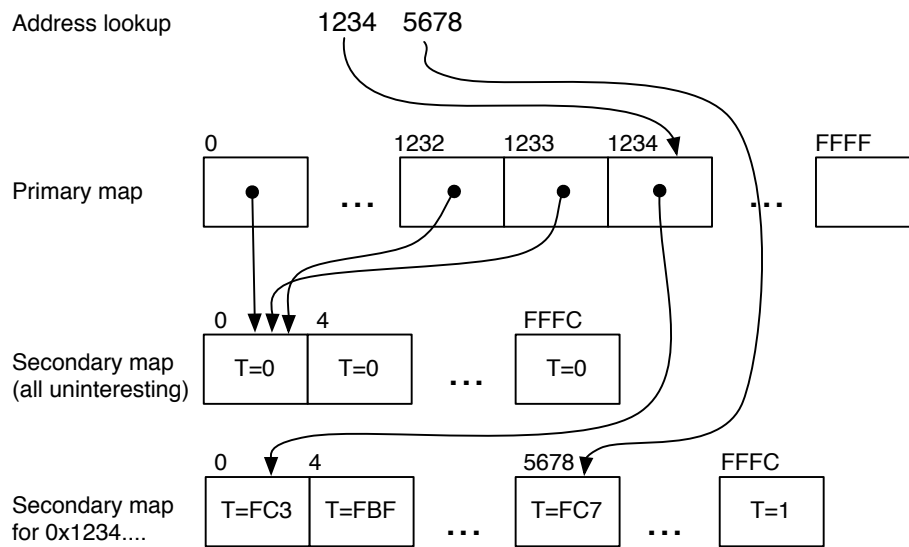


Fig. 5.7: Two level tag table

Because Valgrind tools share the same address space as the programs they are instrumenting, and because DART needs metadata for the entire address space, it cannot store the tags directly in an array — that would require another 4 GB of mostly wasted metadata storage. This, of course, is impossible — DART would need the entire address space just to store metadata for the address space itself. Instead, DART takes advantage of the fact that the address space is sparsely occupied, and that a lot of the space used will have predictable values. To do this, the idea of a two level page table is borrowed from virtual memory implementations[Tan01]. Rather than treating the 30 significant bits of an address as an index into a (huge) flat table, there are two levels of tables. The primary map is a 256 kB table of 2^{16} pointers to a set of secondary maps. These secondary maps are allocated on demand and contain the tags for 2^{14} 32-bit words, or 64 kB worth of words. To read the tag associated with a word, the top 16 bits of the word's address act as an index to the appropriate secondary map in the primary table, and the remaining 14 bits locate the specific tag in the secondary map. The savings come from using

a special read-only secondary map with every metadata value set to ‘uninteresting’ (interestingness will be defined shortly). When the primary map is initialised, every element in it points to this special ‘all uninteresting’ secondary. This way, an entire 4 GB of uninteresting address space can be covered with 320 kB of tables. When a tag is written, if the primary map points at this special table, DART creates a new copy of the all uninteresting secondary, updates the pointer in the primary map to point to this new secondary, and writes the tag to the new secondary. This two level arrangement is illustrated in [Fig. 5.7](#).

5.6.2 Tag semantics

To trace floating point behaviour, three components are needed: Reads from the memory ranges of major data structures must be detected and the values read from these ranges tagged with the source address. Floating point calculations must be recorded if they use one or more of these values, or values derived from these as operands, or produce a denormal result. Finally, the floating point results need to be identified and traced until they are written back to one of these major data structure memory ranges.

To do this, a property called *interestingness* and its associated tag values are defined. ‘Interestingness’ can be regarded somewhat like ‘taintedness’ in TaintCheck. *Marked* memory ranges are those occupied by the major data structures identified in [Sec. 5.5.1](#) and [Fig. 5.5.2](#). Values are interesting if they are in marked memory ranges, or are copied from these ranges, or are the results of floating point calculations on interesting values, or are the result of any floating point calculation if the result is denormal. To meet these requirements, the tag word T is divided into three fields. The top field is 30 bits in size, and it either stores the top 30 bits of an address, or an identifier.

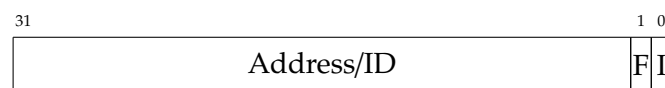


Fig. 5.8: Tag word layout

The least significant bit of a tag is a flag called I, and it flags when the word is from a marked memory range, or has a value copied from one of these ranges.

The second least significant bit is a flag called F. It flags whether the value is a new value produced by a floating point operation with either interesting inputs or a denormal output.

This leaves three possible cases:



If $I = 0$ and $F = 0$ the word is uninteresting and not the result of a calculation, in this case the other 30 bits are required to be zero also, and therefore $T = 0$ defines the word as uninteresting and not to be traced. Requiring the top 30-bits to be zero has two benefits — it simplifies the check for uninterestingness by removing the need for a mask operation, and it allows for an optimisation of the two level tag table scheme in [Sec. 5.6.1](#). This optimisation is a simple form of garbage collection: DART maintains a counter of how many new secondary maps have been allocated since the last sweep, and after it grows beyond a threshold, it scans all the existing secondary maps, and deallocates all secondary maps that contain nothing but $T = 0$ values (i.e., uninteresting tags) and replaces pointers to them in the primary map to the special ‘all uninteresting’ secondary.



If $I = 1$ it is required that $F = 0$. $I = 1$ signifies the word is either copied from a marked address, or it is *at* a marked address. If the latter is the case the top 30 bits are set to zero, thus $T = 1$ identifies a word at a marked address, not just one copied from a marked address. If a word is copied from a marked address, the I flag in the new tag is set to 1, but the top 30 bits of the new tag is set to be the address copied from. This allows a tag to record both the fact that the value is interesting, and its source address.



If $F = 1$ it is required that $I = 0$. $F = 1$ signifies an interesting new floating point value. These come from denormal outputs to floating point operations (which are

always interesting no matter how they are generated), or floating point operations with one or more interesting floating point arguments, irrespective of whether the arguments or result are normal or not. To trace floating point values, DART logs the floating point calculation, its arguments, address and results, and assigns a unique 30-bit identifier to the value, called an fval. This identifier is stored in the top 30 bits of T.

5.6.3 Tag usage

From [Sec. 5.1](#) we recall that the guest program machine code instructions are decoded into a simpler IR based on Loads and Stores from memory locations; Gets and Puts from guest registers; assigns to an unlimited number of temporary values that can be assigned to once but read from many times; and expressions composed of operations on these temporaries.

5.6.3.1 Shadow temporaries

To complete DART, a policy is defined based on loads, stores and floating point operations that propagates and updates tags, and show how that maps to Valgrind's μ Ops. Interesting values are actively propagated by adding relevant instrumentation statements, and anything that isn't explicitly interesting is regarded as uninteresting.

To do this, just as every memory location and every guest register has corresponding tag in shadow memory, every temporary in the IR must also have a 'shadow temporary' that holds the tag for that temporary. DART provides these shadow temporaries by creating a table every time a basic block is instrumented. This table holds a mapping between 'real' temporaries and shadow temporaries. The table is sparse and the default value for each of the shadow temporaries is $T_t = 0$. As DART instruments a basic block, it encounters statements that can propagate interestingness (i.e., by loading a value from memory, or by performing a floating point operation), and for each of these it creates a corresponding shadow temporary and an instrumentation statement that calculates the correct shadow values to be assigned to that shadow temporary.

5.6.3.2 Tag lifecycle

As described in [Sec. 5.6.2](#), 'interestingness' is similar to 'taintedness' in TaintCheck. To illustrate this, the full lifecycle of a value is described, and the propagation of tags through each stage is explained.

1. A lifecycle begins with a read of a value from memory. Therefore every load instruction in the IR must be accompanied by a ‘call out’ to a helper function that reads the tag for that address. The result of this ‘call out’ is assigned to a shadow temporary in the IR. The load helper has the additional function of logging the read if it is from a marked location (i.e., $T_{src} = 1$) and the value is denormal. The logging reports the address of the read instruction, the value itself, and the memory location it is read from. This allows DART to identify the source of denormal values which are read directly from data instead of being computed later.
2. This value may now be assigned to a guest register. In the IR this is done using the Put and PutI statements with a temporary as an argument. This is instrumented simply by inserting statements that write the associated shadow temporary to the appropriate shadow guest register. No logging is needed for this, as it is purely a ‘housekeeping exercise’ that copies a tag.
3. This value might then be read back from the register and written to memory, for example to a variable on the stack. The register read phase is performed with Get or GetI, and this is instrumented by adding a Get/GetI from the matching shadow guest register and storing the result in a shadow temporary. Again, no logging is required for this phase, as it simply copies a tag.

The write phase will be performed using the Store statement. All stores need to be instrumented with a call out to a store helper function which has two responsibilities. Firstly it needs to check the destination address in case the guest code is writing to an array pointer or array descriptor as described in Fig. 5.5.2 after allocating some memory. If this occurs, it needs to mark the memory range (i.e., it needs to set $T_r = 1$ for every address r in that range.)

Secondly, the store helper needs to check if the guest is writing to a marked memory range. It does this by reading the tag for the destination address. If it is not a marked range, i.e., $T_{dst} \neq 1$, it will simply overwrite the tag at this address with the tag of the value written. This has the effect of propagating the interestingness (if there is any) to an unmarked memory location such as the stack or a temporary variable. The $T_{dst} = 1$ case is discussed in a moment.

NB, it is entirely possible for a memory read and write to occur in the same instruction without the value being stored in a guest register in the interim. An instance of this is the instruction `movl (%eax), (%ebx)` which copies the value at EAX to the location EBX. In this case the propagation will occur with the load

and store as described in steps 2 and 3, except without the intervening Put and Get.

4. At some point, one or more values will be operated on. If the operator is an integer operator, i.e., one that performs integer arithmetic, logical operations, shifts and the like, DART knows that the result cannot possibly be an interesting value, as even if the input to the operator was an interesting float, the operation itself will produce an integer as output, which by definition is not interesting. This is because only in extremely unusual cases can integer operators perform meaningful calculations on floating point values⁹. To ignore these operations, DART simply adds no instrumentation for the operation in question, and does not add a shadow temporary in to the sparse table. As a result, any when trying to instrument any later statement in the basic block that uses a temporary assigned to by an integer operation, a read from the shadow table will just get a constant $T_t = 0$.

Floating point operators are instrumented with a ‘call out’ to one of two floating point helper functions — one for unary operators and one for binary operators. These are passed the tags for the operator arguments, as well as the result of the operator. The helpers check if either of the arguments are interesting, or if the result is denormal. If any of these conditions are the case, it will create a new fval identifier for the result, and return that as the tag. It will also log the tags of both the result and the arguments along with the address of the instruction and the result calculated.

5. The resulting fval may be copied from and to registers, or from and to memory in which cases the relevant parts of steps 2 and 3 will apply. The fval may also be used as an argument to another floating point operation, in which case step 4 will be repeated. The log will show the fval as one of the arguments, and the resulting tag will be another fval.
6. Finally, another store will occur, this time to a memory location in a marked memory range, i.e., a repeat of step 3, but with $T_{dst} = 1$. In this case interestingness—i.e., the tag—should not be propagated any further, as a write to a marked memory location is regarded as the end point of the lifecycle of a value. Furthermore, if the tag of the destination were overwritten, it would no longer be the case that $T_{dst} = 1$, so it would no longer be a marked memory location.

⁹Integer operations such as logical OR with 0, multiply by 1 or add 0 completely preserve the value, but these would be optimised away by a compiler, and a properly functioning compiler would never use integer arithmetic on floating point values anyway.

However DART does need to log the store so it can later retrieve the full history of the value. There are four possible situations DART needs to be concerned with, depending on whether the destination and the source value are normal or not. A normal write on top of an existing normal value is uninteresting — this is the usual state of affairs, and does not need to be reported. A denormal write on top of either an existing normal or denormal is interesting, this has the potential to propagate further denormals. Finally, a normal write on top of an existing denormal is also of interest — this indicates a possible process where denormals are destroyed. It also provides useful information for one of the analyses performed later. In the three cases where DART reports the write it logs the instruction that performs the store, the location the value is written to, the value written, and its tag.

[Appendix E](#) has a detailed example of the [IR](#) and instrumentation for a short sequence of instructions.

When executed under DART, the kernel of code in [Fig. 6.5](#) will produce a sequence of denormal loads, floating point operations and a store when it encounters denormal input. A single iteration of this kernel will produce a trace log similar to [Fig. 5.9](#). This trace log can also be visualised as a graph of operations that culminates in a store, as can be seen in [Fig. 5.10](#). The trace log and graph do not have entries or nodes for any of the integer instructions which update the array pointers. This is because the integer instructions do not at any point manipulate interesting data.

```
0x08048730 LD: 0x042b0190 -> @0x042b0190
0x08048738 FP: @0x042b0190 + @0x042b1190 -> f1
0x0804873d FP: f1 + @0x042b098c -> f2
0x08048742 FP: f2 + @0x042b0994 -> f3
0x08048747 FP: f3 * uninteresting -> f4
0x0804874d ST: f4 -> 0x041b0980
```

Fig. 5.9: Trace log from an example jacobi kernel

One of the features of DART that this log does not illustrate is that only loads from marked memory locations are logged. If values loaded from marked locations are subsequently stored in other unmarked memory locations, and then loaded again later, the second load is not logged, because the origin of the value can be traced without recording intermediate stores. The tag says where the value originally came from, and that is sufficient. Similarly, newly computed interesting values can be stored to unmarked memory locations, and this is also ignored. Only the final write to a marked location is logged. An example of this can be seen in the contrived code fragment in [Fig. 5.11](#). [Fig. 5.12](#) shows only three of the instructions are logged:

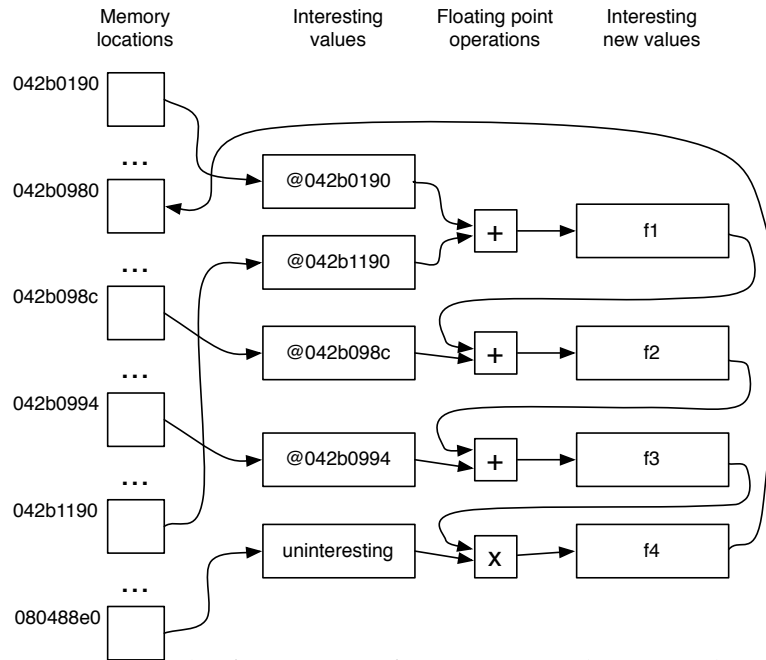


Fig. 5.10: Graph of operations from an example jacobi kernel

```

0x08048800: mov    $0x20000000, %edi
0x08048807: mov    0x10000004, (%edi)
0x0804880e: mov    (%edi), 0x4(%edi)
0x08048811: mov    0x4(%edi), 0x8(%edi)
0x08048814: flds   0x8(%edi)
0x08048817: fsqrt
0x08048819: fstps  0xc(%edi)
0x0804881c: mov    0xc(%edi), 0x10(%edi)
0x08048822: mov    0x10(%edi), 0x10000000

```

Fig. 5.11: Multi-copy code

the initial load, the floating point operation, and the store. The intermediate copies are ignored. It is also interesting to note that the initial load, the final store, and the intermediate copies are all performed by integer instructions. This is illustrated in [Fig. 5.13](#) where the missing nodes are shown with a dashed outline.

```

0x08048807 LD: 0x10000004 -> @0x10000004
0x08048717 FP: sqrt @0x10000004 -> f1
0x08048722 ST: f1 -> 0x10000000

```

Fig. 5.12: Trace log of multi-copy code

5.7 Summary

This chapter introduced the dynamic binary instrumentation framework Valgrind. It examined taint analysis in languages such as PERL and discussed how other

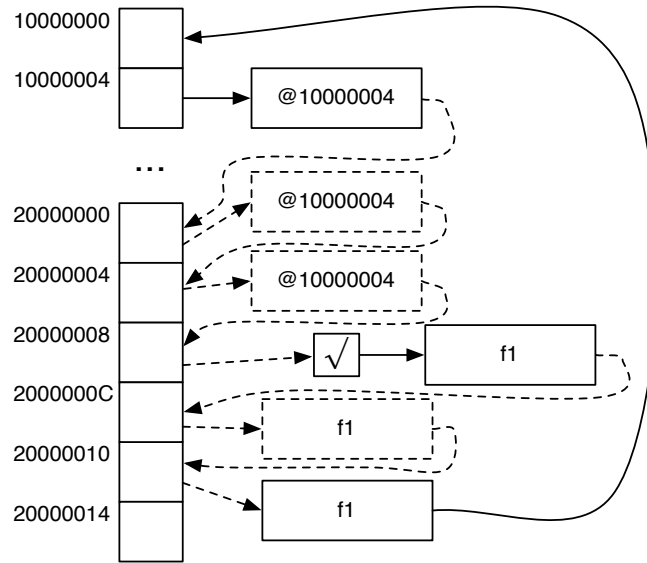


Fig. 5.13: Graph of multi-copy code operations

tools use Valgrind's shadow memory feature to perform taint analysis on arbitrary binaries. I compared and contrasted taint tracing with denormal value tracing. By constraining the problem domain, I specified a design for DART, a Valgrind-based tool which traces the use of denormal values in binaries with sufficient debugging information. This tracing is impossible for DIP or other exception-based profilers to perform. I implemented DART and show how it operates on some small fragments of code.

The next chapter shall show how the large amounts of tracing data produced by DART can be summarised. Two detailed case studies are performed, one with a simple C application, and another with a more complex Fortran benchmark.

Using the new profiling and tracing tools

This chapter describes two numerical codes, one written in C and the other in Fortran 90, that suffer from excessive denormal arithmetic and shows what DIP, the exception-based profiler from [Sec. 4.4](#), and DART, the tracing tool from [Sec. 5.4](#), can tell about the program behaviour.

I show how the raw information logged by DART can be reduced and presented in three different ways, each of which produces useful information..

Finally, I discuss the overheads and limitations of DART.

6.1 Programs

The first program, `jacobi`, is a small example written in C that solves the Laplace equation in 2 dimensions using Jacobi iteration. The second program, `187.facerec`, is a more complex Fortran 90 code taken from the SPEC CPU2000 benchmarks.

`jacobi` is sufficiently simple to contain few surprises and can be used to verify that all the parts of DART behave as they should. The same kernel has been used in other publications [[BA05](#)] to illustrate how performance variability can be introduced by denormal arithmetic.

`187.facerec` is a more substantial application based on computer vision research done at Ruhr University Bochum in the mid 1990s [[Lad93](#)]. As part of SPEC CPU2000 it represents a realistic numerical application, and its size, portability and scalability make it convenient to analyse with DIP and DART.

6.1.1 `jacobi`

`jacobi` is a short program written in ANSI C that solves the Laplace equation in 2 dimensions using Jacobi iteration.

The Laplace equation

$$\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} = 0$$

can be used to represent many physical phenomena. For example, when φ is the temperature of a conductor, the solution to the Laplace equation gives the steady-

state heat flow for the system; and when φ is electrical potential in a metal sheet, the solution gives the steady-state voltage at every point in the sheet.

The Laplace equation can be solved numerically by splitting the plane into a grid of discrete cells, $\varphi_{i,j}$, and defining an approximation of the first and second order partial derivatives based on the differences between neighbouring cells.

Doing this produces the approximation

$$\varphi_{i,j}^{(n+1)} = \frac{1}{4} \left(\varphi_{i-1,j}^{(n)} + \varphi_{i+1,j}^{(n)} + \varphi_{i,j-1}^{(n)} + \varphi_{i,j+1}^{(n)} \right)$$

This equation is called the *update equation*, and the 4 cells on the right hand side of the equation are called the *stencil*.

To produce a solution, the edge cells are held fixed, and an approximation generated by applying the update equation to each interior cell in the grid. A second approximation is iteratively produced by applying the equation to every interior cell in the first approximation, and this can be continued for sufficiently many iterations to converge on the solution.

To implement this, two arrays of 32-bit floats, a and b , are allocated. a is initialised with the desired initial and boundary conditions. Then N iterations of the update equation are calculated. On even iterations a will hold the existing approximation and b the new approximation, and on odd iterations the two arrays swap roles. For each iteration the edge cells are copied directly from one array to the other, and the interior cells updated using the update equation to calculate a new approximation based on cell values from the old array.

Due to its simplicity, this algorithm is often used to illustrate parallelisation techniques in high performance computing courses, however if the initial values are chosen poorly the algorithm can generate large numbers of denormal values.

Consider, for example, the 1 dimensional Laplace equation, and the corresponding update equation.

$$\frac{\partial^2 \varphi}{\partial x^2} = 0$$

$$\varphi_i^{(n+1)} = \frac{1}{2} \left(\varphi_{i-1}^{(n)} + \varphi_{i+1}^{(n)} \right)$$

Beginning with an initial approximation where every interior cell is 0, and the left boundary is non-zero, the approximation on [Fig. 6.1](#) evolves over a number of

iterations:

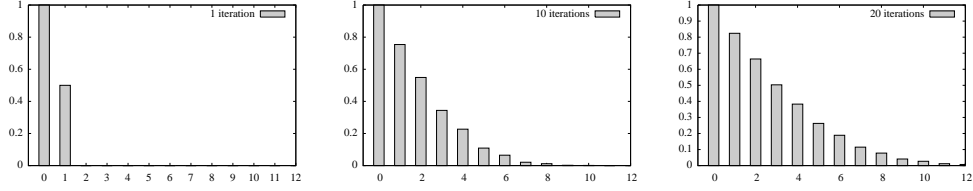


Fig. 6.1: Laplace steady state approximation after N iterations

As the iterations pass, the non-zero boundary value is ‘smeared’ out across the interior cells. Because of the repeated averaging, the values decay exponentially towards zero, and if there are enough cells in any direction denormal values will occur in a large band of interior cells.

This can be avoided by initialising all the interior cells to a negligibly small normal value such as 10^{-20} , and DIP and DART should detect and highlight the cause of this problem.

6.1.2 187.facerec

187.facerec is a simplified version of an image database search application applied to a set of face images.

Image features, for this application, are calculated by the Gabor Wavelet Transform. [Lad93] This takes an input image and performs a 2D FFT on it. Once in the frequency domain, it takes a group of 40 kernels described below, and for each of them multiplies every pixel in the image by the corresponding kernel value, and applies an Inverse Fast Fourier Transform (iFFT) to the result. The set of 40 transformed values for each input pixel is called a ‘jet’ and each jet is stored as a normalised absolute value. Finally, a graph is fitted on top of the image, optionally transformed in some way, and ‘graph extraction’ is performed by reading the jets at each graph node location into a features array.

The kernels are biologically motivated — they behave like an abstract version of the simple cells in V1 in the visual cortex which detect localised frequency components in an image. There are 5 ‘levels’ of kernels, each generated from the equation below by varying k , with smaller k s leading to a more spread out waveform:

$$K(x, y) = e^{-\sigma^2(x^2+y^2)/2k^2}$$

Each level of kernel is offset from the origin and rotated at 8 angles from $-3\pi/8$ to $+4\pi/8$ producing a total of 40 different kernels which are used for the multiplication in the transform above. The transform has the effect of picking out local sets of frequencies at various points in the input image.

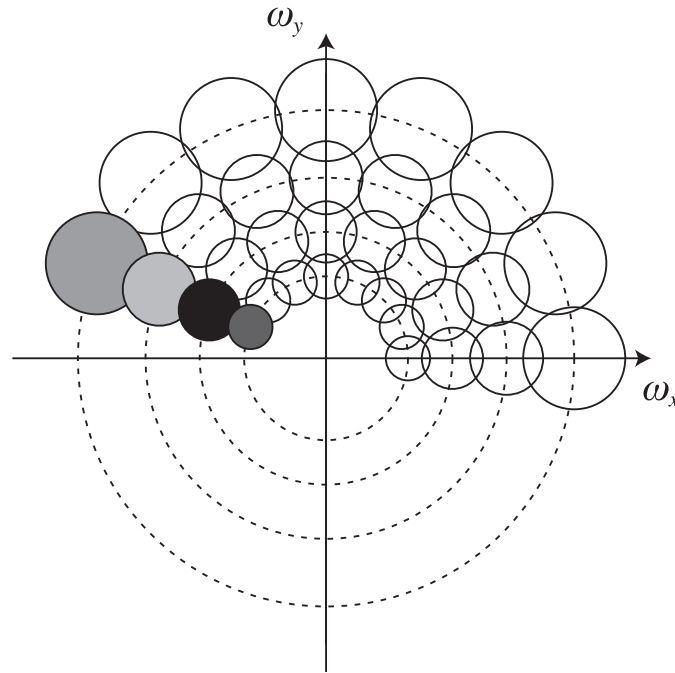


Fig. 6.2: 187 . facerec local frequency measurements[PKvdM96]

At startup, 187 . facerec reads a configuration file, and uses this to load a set of 256×256 pixel greyscale images of people's faces. One of these serves as a 'canonical' image used to calibrate the system, a group of images called the 'album gallery' made up the database to be searched, and a final set called the 'probe gallery' are the search query terms.

The canonical image is represented by applying the Gabor Wavelet Transform to it, superimposing a graph (in this case a rectangular 2D grid of nodes) on top of the transformed image, and then applying the graph extraction directly to this grid.

Each image in the album gallery also has the Gabor Wavelet Transform applied, and the graph superimposed, but then a graph fitting stage, which consists of a global move step takes place. The global move translates the entire graph in the X and Y axes over the image and searches for the offset that maximises the similarity between the canonical image and the album image. After this is found, the graph extraction describe above is applied, and the graph parameters stored for later use.

In the recognition phase, each image in the probe gallery is loaded, the Gabor Wavelet Transform is applied to it, and the global move step from graph fitting stage occurs. After this, the probe image is compared to every image in the album in turn. For each album image, a local move step occurs. This works by randomly visiting all the nodes in the probe graph in what is termed a sweep, and seeing if moving that node a small distance in the X and Y axes will improve the similarity between the probe and album images. If it does, the result is saved, and a hop is said to have occurred. The sweeps repeat until no hops occur for an entire sweep. For a given probe image, the album image with the highest similarity is chosen as the match to the probe image.

As can be discovered fairly easily with DIP, many denormal calculations occur in this application — primarily in the Gabor Wavelet Transform routine and to a lesser degree in the iFFT. However, it is far from obvious why these denormal values are occurring, and what can be done to avoid them.

6.2 Analysis

As described in [Sec. 4.4](#), a profiling library such as DIP can use exception trapping features of the 80x86 to log when denormal arguments use in floating point instructions. As noted in [Sec. 4.4.6](#) a simple tool like this has a number of limitations, however one of the limitations not mentioned is the huge slowdown inherent in such an approach and the large volumes of trace log produced. A program that runs for only a few seconds might execute on the order of 10 million denormal operations, and produce a gigabyte of trace log. Clearly this is unwieldy and will not scale well to programs with longer runtimes. Furthermore, the raw logs in themselves give little little insight into the behaviour of the program.

To avoid having to process these volumes of logging information, some form of data reduction must be applied to the logs, preferably as they are created. Three kinds of reduction have been implemented for DIP and DART.

6.2.1 Instruction profiles

In the case of DIP, there are only two pieces of information in each logging item: the time the event occurred and the address of the floating point instruction itself.

Unless the order in which operations occur is of interest, the timestamps and the ordering it provides are unnecessary. This leaves a single piece of data: the instruction address. A natural way to condense this data is simply to record the number of times a particular instruction is logged. If there is no need for 'phase

analysis' where the behaviour of a program is examined at different stages of its execution, the total counts for each address can simply be reported after the program completes. If phase analysis is required, the totals can be reported at specific times during the program's execution.

One data structure suitable for storing the address counts is a hash table. Given the sparse distribution of the addresses involved, the two level tables from [Sec. 5.6.1](#) could also be used.

To facilitate phase analysis, a means of telling DIP to dump the current totals could be provided by getting it to install a signal handler for one of the user signal handlers e.g., SIGUSR1 and getting either the program itself or an external program to send a signal to the process.

Instruction profiling can just as easily be added to DART too. If the two level table approach is used, instead of adding an extra word (or more) for each entry in the second level maps, it would be better to use a separate set of tables. The reasoning behind this is that the code and data occupy disjoint regions of memory, so larger secondary entries would needlessly double (or more) the size of the secondary maps used for the data areas only to store zeroes in the counts for those addresses. Likewise, the code areas will never contain data, and memory will be wasted storing always-uninteresting tags for those blocks of memory.

Rather than using the potentially brittle method of signal handling to request that DART dump the current totals, use can be made of the Valgrind 'client request' functionality. This provides a set of macros that compile to an unusual sequence of instructions in the guest program that perform no action. When the Valgrind x86 parser detects these instruction sequences, it adds a host call out to the [IR](#) that when executed calls the appropriate request handler in DART.

6.2.2 Array heatmaps

Examining instruction profiles can be helpful in some scenarios, but often they show little about the patterns of data flow throughout a program, or the evolution of the distribution of denormals in a data set over time. Sometimes this extra information is necessary to gain insight into the overall behaviour of the program.

DART reports all the denormal loads and all the denormal stores to marked arrays. The tag metadata can easily be extended with a pair of counters storing how often denormals are read or written to each marked address, and corresponding code added to the load and store helper routines to increment the counters every time a

denormal load/store occurs. As occurred with the instruction count metadata, the read/write counts only apply to specific areas of memory, so instead of bloating all the tag tables with extra unused data, it is better to store the read/write counts in a separate set of tables.

This is illustrated in [Fig. 6.3](#). In it are three memory locations:

- 10000000 is a location in a marked data structure. Denormal values have been read from it 100 times, and denormal values written to it 4 times. Its tag value is thus $T_{10000000} = 1$, and the read/write count values are $R_{10000000} = 100$, $W_{10000000} = 4$.
- 20000000 is not in a marked structure, therefore denormal read/write counts are not tracked for it. It currently stores a value copied from the marked memory location 10000004. Its tag value is $T_{20000000} = 10000005$ (the I flag is set) and $R_{20000000} = 0$, $W_{20000000} = 0$.
- 30000000 is neither a marked location, nor does it store an interesting value. Therefore $T_{30000000} = 0$, $R_{30000000} = 0$, $W_{30000000} = 0$.

As with the ordinary tag tables, the read/write information is stored as a two level table. 64 kB blocks whose read/write counts are all zero will share one immutable all-zeros secondary table, minimising the overhead of storing the extra information.

Thanks to the DWARF debugging information and knowledge of the runtime allocation methods, the name, location, dimensionality and bounds of the marked arrays are all known. This information can be used to identify specific memory ranges, and periodically create a ‘heat map’ of the denormal reads and writes to a particular array. These heat maps can be saved as images (perhaps coding reads as red and writes as blue) and later viewed as animations or subjected to further analysis. The animations can reveal a lot about the behaviour of an algorithm, particularly if related patterns of denormal accesses occur in different arrays.

A less detailed, but more quantitative approach can be taken simply by summing all the read and write counts for each array, and reporting the totals for each array at regular intervals. These totals can be interpreted in a number of ways.

- If an array has only a few denormal writes, but many denormal reads, it is likely to be a source of denormals in a program — the writes were the denormals created at initialisation time, and the reads are the repeated accesses since then.

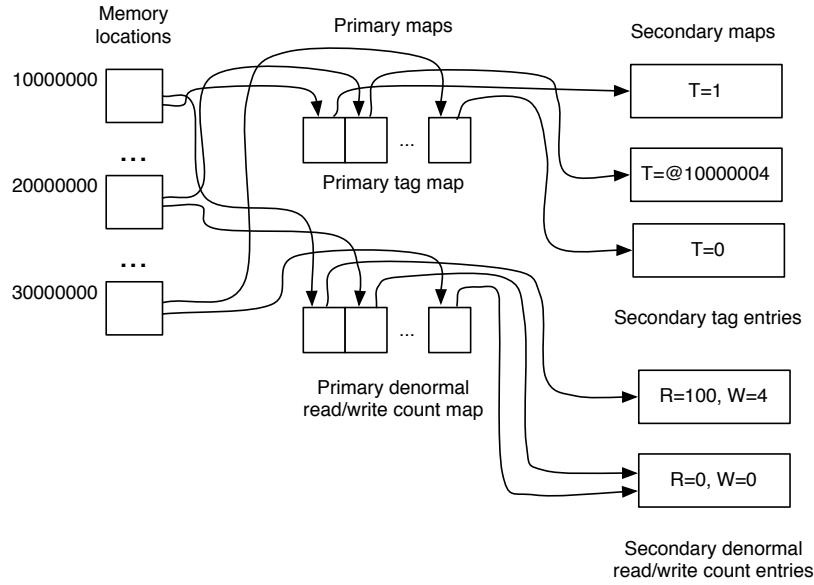


Fig. 6.3: Distinct tables for tag and read/write counts

- If the denormal read and write counts are similar, then either the denormals are propagated within the array, or the array is a data structure used to store intermediate calculations.
- If the denormal read counts are low, but the write counts are high, the array is a denormal sink. Either the values in this array are frequently overwritten without use, or some code is filtering the denormals from the array after they are first written.

6.2.3 Array origin maps

Generating read/write heatmaps for the marked arrays in a program can reveal a lot about its behaviour, but sometimes the programmer would like to ask ‘where did these values originally come from?’

As described in [Sec. 5.4](#) and detailed afterwards, DART uses the simplifying assumption that most of the computation in the guest programs will be on values in the marked data structures of the program. Thus guest values have a short lifecycle which starts at a read from a marked array, followed eventually by a chain of one or more floating point operations which generate temporary fval results, and finally a write to a marked memory location. In some sense, a group of related read/calculate/write events all with the same source and destination arrays will

define a small phase of a program's execution. For example, array initialisation, element by element operations on an array, FFTs, convolutions, and stencil kernels all fit this pattern. If, for a given phase, the source of the denormals in the destination array can be determined, then by backward chaining, the source of denormals can be tracked across multiple phases.

As with the heatmap generation, the origin metadata is stored in a new set of two level tables in DART that hold a fixed number of origin counters for each word in memory. These tables will only be updated when a value is written to a marked memory location. For this to be practical, there must be a small number of marked origin arrays, perhaps on the order of 10 or so, each identified by a small integer, making it feasible to add this many counters worth of overhead to every marked location.

When a store to a marked location occurs, DART will have to effectively 'trace backwards' through the tree of operations that occurred to create that value. In the case of fvals, DART finds the source arguments to the floating point operation that created the fval, and recursively finds the sources to those in turn until it ends up with a list of uninteresting tags or source addresses. The uninteresting tags are discarded, as are the source addresses if the values read were not denormal. Finally, for each remaining source address, the array it belongs to is found by bounds checking. Then the corresponding origin array counter in the destination metadata is incremented.

Once this metadata is available, DART can periodically produce heatmaps for a specific array, except instead of the heatmaps showing denormal reads and writes, they show writes to array *a* that derive their values from denormal reads from array *b*.

6.2.4 Optimising origin tracking

If origin tracking is performed naïvely as described, DART will need to store the entire trace of interesting loads, floating point operations and stores, and search backwards through them to find the sources. This is both extremely slow and consumes huge amounts of memory. Instead, the algorithm can be optimised as follows:

- The tag metadata tables are augmented with a 1 bit denormal flag for every word of memory, increasing the secondary map size from 64 kB to 66 kB. This flag is set to 1 whenever a word is loaded from a marked memory location and the

word's value is denormal. This word is necessary because when backtracking is performed, DART needs to know whether the word was denormal when it was first read — the memory location may later have been overwritten with a normal value.

The machinery that propagates the tag will need to be updated to handle two pieces of metadata instead of just one. Fortunately, there are two shadow guest state structures available in Valgrind, and Valgrind provides as many extra temporaries as are needed.

- When a floating point operation occurs, it takes one or two arguments and produces an fval. If the fval is denormal, its tag denormal flag is set to 1. A two level table is created called `fval.info` which is indexed by fval IDs. Entries in `fval.info` store the tag for the two arguments which were used to create the fval, the fval denormal flag, and optionally the address of the instruction that did so. If there is only one argument, the second argument is set to a special empty tag value.¹ This table is used to trace recursively from the root fval through the tree of fvals and operations that created it, until ultimately memory load leaf nodes for marked or uninteresting memory are reached.
- When a store to a marked location occurs, the origins tree can be build using the same algorithm as described above. The source arguments to an fval are found by retrieving the arguments from the `fval.info` table, applying this recursively until a list of addresses remains. These addresses are mapped to the arrays they occur in, and for each of these, the appropriate origin counter in the destination metadata is incremented.

Optionally, when creating this list all stores of not denormal values can be ignored, or even all intermediate fvals that are not denormal, however it may be useful to see all the sources that could potentially have ended up generating denormal results if some intermediate calculation had not normalised the value.

A simple mathematical expression $(a \times b + c)/d$ might produce the tags in [Fig. 6.4](#) and the corresponding `fval.info` in [Table 6.1](#).

6.2.5 Garbage collecting fvals

This optimisation functions well, but since large numbers of temporary fvals are created, a scheme is needed to clear and remove fvals from `fval.info` once they are

¹This is defined to be `T=ffffff`, as no valid tag can have both I and F set at the same time.

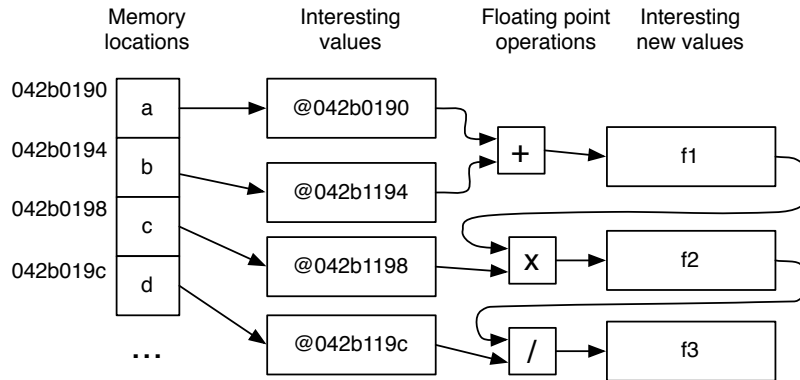


Fig. 6.4: Tags for a simple expression

fval	Instruction address	Argument 1	Argument 2
f1	0x08048556	From 0x042b0190	From 0x042b0194
f2	0x0804855a	f1	From 0x042b0198
f3	0x0804855e	f2	From 0x042b019c

Table 6.1: fval_info table for a simple expression

stored to memory and no longer used. To do this, the following properties of fvals are observed:

1. fvals are similar to [IR](#) temporaries in that unless they are explicitly stored somewhere, they will cease to exist once DART leaves a basic block. Also like temporaries, fvals can be written into either registers (in the guest state), or memory locations.
2. An fval can potentially be used for a store to a marked memory location if there is a copy of it anywhere in memory or in the register file. Information about it must be kept 'alive' as long as any copies of it exist, since if a store does occur, DART will need to discover which arguments it was derived from as a part of the recursive backtracking.
3. An fval may have no copies of it either in the register file or in memory, but DART may still need to keep a copy of the fval if a second fval derived from this fval is still alive. This occurs when an fval is used as one of the arguments to a floating point operation.

These constraints can be met by adding a reference count field to each `fval_info` entry. If the reference count is non-zero, the `fval` is ‘alive’. It is incremented for every reference to it that exists, and decremented when those references are removed, and once it is decremented to zero the `fval` is ‘dead’ and the entry in `fval_info` can be wiped. An entry is wiped by setting the reference count and flags to zero, and the argument fields to the empty value.

To honour property 2, every time the `fval` is written to unmarked memory or registers (i.e., a copy is made of it and its tag), the reference count is incremented. Each time the `fval` is overwritten by something else (i.e., a copy is destroyed), the reference count is decremented.

To honour property 3, every time an `fval` is created (as the result of a floating point operation), if either of the arguments is an `fval`, their reference counts are incremented — the `fval` must keep its ‘parents’ alive for as long as the `fval` itself lives. If an `fval`’s reference count is decremented to zero, it is first wiped and then the reference counts of its parents are decremented also — after wiping an `fval` ‘lets go of’ any references to its parents. This decrementing may in turn reduce the parents’ reference counts to zero in which case they too must wipe themselves and decrement the reference counts of their parents, and so on recursively.

Once this reference counting scheme is in place, DART can periodically compact the `fval_info` structure by scanning the entire `fval_info` table and deallocating the secondary maps that contain nothing but wiped entries in the same way as described in [Sec. 5.6.2](#).

This description might suggest that there is race condition at the point where a new `fval` is created. When an `fval` is created, its reference count is zero (but one or both of its arguments are non empty). Since there is an interval between the `fval`’s creation and the point where its reference count is incremented from zero when it is used as an argument to create another `fval` or stored to memory/a register, it would appear a compaction at this point would remove the value. However, the compaction is not where the garbage collection is really performed. The garbage collection occurs when the value is wiped. The compaction merely reduces the redundant storage by removing unneeded all-empty secondary maps. Since a newly created `fval` will have at least one non-empty field, the secondary containing it will not be removed.

This does bring up the possibility of an `fval` being created but never stored or used to create another `fval`. Such an `fval` would never be removed since there would be no references to it, and nothing to decrement its reference count. In practice though, Valgrind’s [IR](#) generator will not produce these μ Op sequences, and even if

it or DART's instrumentation did, they would be optimised out by an [IR](#) dead code elimination pass before the basic blocks are compiled.

6.3 Experiments

To evaluate the effectiveness of DIP, DART, and data reduction, each of the data reduction methods is applied to both `jacobi` and `187.facerec`.

All the tests are run on a Linux machine with a 2.0 GHz Core Duo T2500 [CPU](#) and 2 GB of 667 MHz DDR2 SDRAM.

A standard install of the Debian testing distribution 'squeeze' from July 2009 is used. The kernel version is 2.6.26-2-686, and `gfortran-4.3.3-10` and `gcc-4.3.3-10` are used as the compilers. The version of Valgrind used for DART is r10229 from the SVN repository. This was the most current development version at the beginning of June 2009 and is between the stable releases 3.4.1 and 3.5.0.

Both `jacobi` and `187.facerec` are compiled with DWARF debugging information enabled. `187.facerec` is compiled at optimisation level `-O1` to make the profiling information clearer, and `jacobi` is compiled at optimisation level `-O3`.

The precise setup of the experiments are not strictly important, the relevant details are that a 32-bit version of Linux for the x86 is used running on an Intel-based CPU² and the compiler generates x87 floating point instructions and DWARF3 debugging information. The application performance is independent of whether the OS is running directly on the hardware or in a virtual machine, as apart from when DIP is used, no floating-point exceptions are triggered by the experiments that might require intervention by the VM hypervisor.

6.4 Profiling and tracing `jacobi`

`jacobi` is run with a 512×512 grid of cells for 768 iterations. The edge cells are initialised to 1.0 and the interior cells to 0.0.

6.4.1 Using DIP

The program takes approximately 5.52 seconds to run without any profiling. With DIP described in [Sec. 4.4](#) and the logs redirected to `/dev/null` to remove the overhead of writes to disc, the runtime increases to 152.3 seconds, a slowdown of 27.6×. Some of this overhead is in the string formatting and log printing. Replacing the logging with the basic instruction profiling from [Sec. 6.2.1](#) reduces the runtime to 109.3

²We see in [Appendix H](#) that AMD CPUs have different performance characteristics.

```

0x08048730: flds    (%edi)
0x08048732: add     $0x1,%eax
0x08048735: add     $0x4,%edi
0x08048738: fadds   (%esi)
0x0804873a: add     $0x4,%esi
0x0804873d: fadds   (%ebx)
0x0804873f: add     $0x4,%ebx
0x08048742: fadds   (%ecx)
0x08048744: add     $0x4,%ecx
0x08048747: fmul    0x080488e0
0x0804874d: fstps   (%edx)
0x0804874f: add     $0x4,%edx
0x08048752: cmp     $0x1ff,%eax
0x08048757: jne     0x08048730

```

Fig. 6.5: jacobi kernel instructions

seconds, a slowdown of 19.8×.

If jacobi is configured to initialise the interior cells to a normal nonzero value (e.g., 0.1), this avoids the unnecessary denormal arithmetic, resulting in an executable that runs almost twice as fast (2.81 seconds) when the profiling is disabled. With profiling enabled the runtime increases by a tiny fixed amount (0.03 seconds) due to the time taken to load the profiling library. Since DIP only incurs an overhead when a floating point exception occurs, and there are no exceptions in this modified run, there is no other slowdown.

The computational core of jacobi is this single statement in a nested loop, where `cur` and `prev` are arrays of 32-bit floats.

```

for (i=1; i<N-1; i++) {
    for (j=1; j<N-1; j++) {
        cur[i*N + j] = 0.25 * (
            prev[(i-1)*N + j] +
            prev[(i+1)*N + j] +
            prev[i*N + j-1] +
            prev[i*N + j+1]);
    }
}

```

This compiles to the code in [Fig. 6.5](#). There is a floating point load, 3 adds, a multiply and a store.

DIP reports denormal reads for the following addresses:

Address	Count	Instruction
0x08048730	1,075,922	flds prev[...]
0x08048738	1,075,922	fadds prev[...]
0x0804873d	1,075,922	fadds prev[...]
0x08048742	1,075,922	fadds prev[...]
Total	4,303,688	

Table 6.2: Denormal exception profile for jacob1

As expected, there are an equal number of denormal argument traps for the load and the 3 adds. Perhaps surprisingly though, the multiply and store cause no denormal exceptions. The reason for this is straightforward but requires examination.

The `flds` generates a denormal exception because its argument—the 32-bit value read from memory—is denormal. This value is written to the `FPU` register at the top of the stack. On the x87, all `FPU` registers are 80 bit extended precision floats with 15 bits of exponent and 64 bits of mantissa with an explicit leading 1. When a value is loaded from memory, it is automatically converted to extended format, and any arithmetic on it will also occur in extended format. Although it is possible to put the `FPU` into modes where the mantissa is rounded to 53 or 24 bits after each operation rather than the full 64 bits of the 80 bit format, the exponent is not similarly reduced. Thus the exponent is always 15 bits — larger than the 11 bits of double precision and the 8 bits of single precision.³ This means that the value on the stack is now normal.

When the following `fadds` occur, they also generate denormal exceptions because their 32 bit argument is denormal. However they too are converted to 80 bit extended precision on reading, and the result of adding even the two smallest single precision denormal numbers is well within the range of the 15 bit exponent, and thus remains normal.

Similarly, for the `fmul`s, the memory-based argument—a constant 0.25—is normal and will not generate a denormal exception, and the result of the multiply will reduce the exponent by 2, keeping it well within the range of extended precision normals.

Finally, the `fstps` instruction’s argument is the top register on the `FPU` stack. From the above, this will definitely be an 80 bit normal, so will not generate a denormal argument exception. The value written to memory will be converted from 80 bits

³The smallest possible single precision denormal is $2^{-23} \times 2^{-126} = 2^{-149}$ which is hugely larger than the smallest normal extended precision number (2^{-16382}) due to its 15-bit exponent.

to 32 bits, and this value may well be denormal, however this does not generate a denormal exception, but rather an underflow exception, which DIP does not monitor due to the extra complications it involves.

There are further consequences to the x87's floating point rounding which will be examined shortly.

6.4.2 Using DART

Tracing `jacobi` using 1.0 for edge cells and 0 for interior cells with DART takes 72.5 seconds to run. This is $13.1\times$ slower than the `jacobi` running natively. Interestingly, this is faster than the runtime under DIP, despite the significant overhead for every instruction in the guest program. When there large numbers of denormal values, the cost of trapping is higher than the cost of simulation.

However tracing `jacobi` using 1.0 for edge cells and 0.1 for interior cells under DART takes 62.1 seconds to run. This is $21\times$ slower than `jacobi` running natively, showing that the fixed per-instruction overhead is approximately $20\times$.

6.4.3 Comparing results from DIP and DART

DART generates the following profile for `jacobi`:

Address	Loads	FP ops	Stores	Instruction
0x08048730	4,477,168			<code>flds prev[...]</code>
0x08048738	4,477,168	4,306,412		<code>fadds prev[...]</code>
0x0804873d	4,477,168	4,252,566		<code>fadds prev[...]</code>
0x08048742	4,477,168	4,216,460		<code>fadds prev[...]</code>
0x08048747		4,477,184		<code>fmls 0.25</code>
0x0804874d			4,477,144	<code>fstps cur[...]</code>
Total	17,908,672	17,252,622	4,477,144	

Table 6.3: DART denormal profile for `jacobi`

The table shows that DART reports a load for the `flds` instruction; a load and a floating point operation for each `fadds`; only a floating point operation for the `fmls` (since the load is normal, and the second argument is from the stack); and stores for the `fstps` instruction. Already this gives more information than provided by DIP.

When these results are compared to the results given by DIP, the overall pattern is seen to be the same — an equal number of denormal loads for each of the `flds` and `fadds` instructions. However, the total number of denormal loads reported by DART is more than $4.1\times$ that reported by DIP. To explain this the differences

between Valgrind's floating point implementation and that of the x87 need to be examined.

As part of its guest state, Valgrind mimics the x87 floating point stack, which stores 80 bit values. However, in Valgrind, all floating point arithmetic on the stack is ordinarily performed using 64 bit long floats and rounded to the nearest value. When the values are stored to memory, these results are converted to the appropriate format whether that be 32-bit, 64-bit or 80-bit floats, so that if the program examines memory the results are in the correct format. Furthermore, to simplify DART, all floats are assumed to be 32-bit floats, so all intermediate calculations are in fact performed at 32-bit precision. Since the stencil has a chain of 3 fadds each of which can lose half a unit of least precision, and the numbers involved at the edge of the denormal 'wavefront' are tiny and decreasing in size, this repeated rounding leads to the values in this area being smaller than if they had been calculated to a higher intermediate precision.

This behaviour can be simulated directly on the x87 by compiling jacobi with the -O0 compiler switch. This turns off all GCC's compiler optimisations, and instead of emitting arithmetic instructions that read one argument directly from memory and the other from the stack, GCC emits separate load and arithmetic instructions, presumably mirroring the compiler's internal Register Transfer Language (RTL) more closely.

Because of the separate instructions, the rounding that occurs between each operation is the same as with DART. If DIP is used on this unoptimised binary, exactly the same number of denormal exceptions are reported as denormal loads with DART.

```
...
0x0804876f: flds    (%eax)
...
0x08048783: flds    (%eax)
0x08048785: faddp   %st,%st(1)
...
0x0804879d: flds    (%eax)
0x0804879f: faddp   %st,%st(1)
...
0x080487b3: flds    (%eax)
0x080487b5: faddp   %st,%st(1)
0x080487b7: flds    0x8048950
0x080487bd: fmulp   %st,%st(1)
0x080487bf: fstps   (%edx)
```

```
...
0x080487cc: jle    804874c
```

However, the precise number of denormals counted is less important than how they're distributed and how they are propagated. DART can provide both of these details.

6.4.4 Denormal heatmaps

DART's heatmap mode described in [Sec. 6.2.2](#) can be used to come to an understanding of where the denormals are in `jacobi`'s data structures.

Firstly, the heatmap provides the with the unsurprising information that both denormal loads and stores are divided almost perfectly evenly between the two arrays in the program, *a* and *b*. This is unsurprising as `jacobi` alternates between using which array is the source and which is the destination for each iteration.

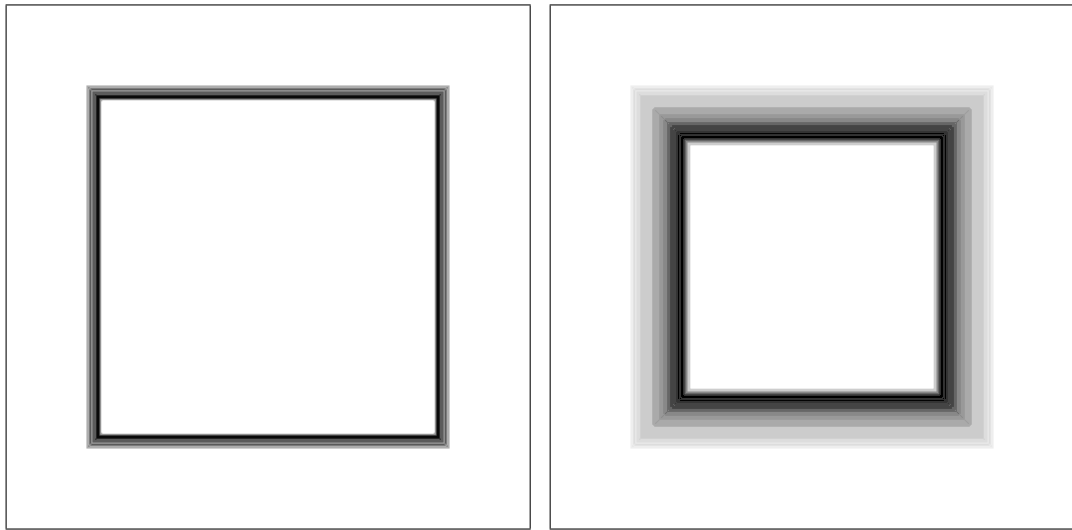
Array	Loads	Stores
<i>a</i>	8,914,864	2,228,716
<i>b</i>	8,993,808	2,248,468
Total	17,908,672	4,477,184

Table 6.4: Denormal loads/stores for `jacobi` arrays

Secondly, plotting the heatmap at different points in time, say after 100 and 200 iterations as in [Fig. 6.6\(a\)](#) and [Fig. 6.6\(b\)](#), gives the more interesting result that the denormal writes occur as a wavefront spreading from the edges towards the centre of the arrays. This wavefront moves inwards over time, and the number of denormal reads and writes that occur towards the centre of the array are higher than towards the edges.⁴

Closer inspection will show why this must be: towards the centre of the array, the values of the cells are very close to zero, and the neighbouring cells are only slightly higher, so the averaging process applied by the stencil is relatively slow at increasing a cell's value out of the denormal range. Nearer the edges, the relatively large values of the edge cells have a more direct influence on a cell's value, and its value increases more quickly as a result. Eventually, if iterated for long enough, every cell's value increases above the denormal range, and after continued iteration converges on the edge value.

⁴For printing purposes, these images have all been normalised to increase the contrast. In the unaltered images, the heatmap at 100 iterations is much fainter than it appears here.



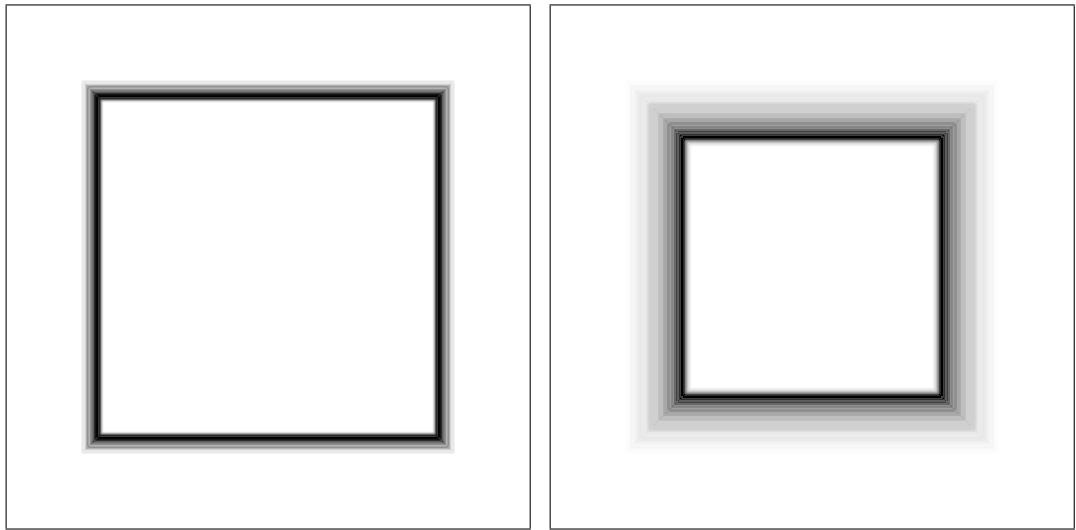
(a) After 100 iterations; denormal writes to array a (b) After 200 iterations; denormal writes to array a

Fig. 6.6: Heatmap showing sweep of denormal averages produced by jacobi

Examining the heatmap closely, it can also be seen that the cells along the diagonals seem to have lower denormal counts than their neighbours. This is because the stencils of the corner cells covers two edge cells, not just the one of the rest of the boundary cells, thus their values are higher and move out of the denormal range earlier.

Origin tracking can also be applied to this code, however the results (Fig. 6.7(a) and Fig. 6.7(b)) are not particularly interesting. Origin simply reveals that all the denormals in array a come from calculations on values that originate from b , and vice versa, something which was already known. An origin map of the denormal values in a that derive from denormals in b will look very similar to the heatmaps above. This is unsurprising, as the structure of the stencil already shows that a cell derives its value from its neighbours, causing values to spread like a wavefront. There are more denormals towards the centre of each array, because it takes the longest for the wavefront to reach there, and the edge values influence on these cells is very indirect.

In fact, in this case, what the origin map shows is a subset of what the heatmap contained: The heatmap shows all denormal loads and stores, including the denormal stores that come from averaging very small normal numbers with zeros. The origin map only shows the denormal stores that came from averaging one or more denormal loads.



(a) After 100 iterations; map of denorms in a from b (b) After 200 iterations; map of denorms in a from b

Fig. 6.7: Map of number of denormals in jacobi array a derived from b

6.5 Profiling and tracing 187.facerec

187.facerec is compiled as described and run using the test dataset. This uses an album and probe gallery each with two images. One of these image is the canonical image. Run directly on the host hardware, the runtime of 187.facerec is about 5.1 seconds. This is very short, and as the 187.facerec documentation notes, the runtime of each stage increases either with the sum of the number of images in the probe and album gallery, or the product of the number of images in the two galleries. Two larger datasets are available if a longer runtime is needed, but they are unnecessary for these tests — the denormal behaviour of the code can be determined with the smallest dataset.

6.5.1 Using DIP

Run directly on the hardware, the runtime of 187.facerec is 5.1 seconds, and under DIP, the runtime is approximately 18.1 seconds, a 3.5× slowdown.

DIP produces the profile in [Table 6.5](#) for 187.facerec

On inspection the debugging information shows that the `gaborRoutines.f90` instructions lie in the subroutine `GaborTrafo` which applies the Gabor Wavelet Transform described in [Sec. 6.1.2](#). This multiplies each pixel in an image by the corresponding pixel in a scaled and translated kernel, and generates almost 53% of the total

Address	Source	Count	Percentage
0x0805154a	gaborRoutines.f90:110	604,000	26.430%
0x08051542	gaborRoutines.f90:110	604,000	26.430%
0x0804a8aa	cfftb.f90:291	131,016	5.733%
0x0804a87d	cfftb.f90:287	130,505	5.711%
0x0804a8c7	cfftb.f90:293	111,952	4.899%
0x0804a89d	cfftb.f90:289	111,953	4.899%
0x0804a895	cfftb.f90:289	93,624	4.097%
0x0804a8bc	cfftb.f90:293	93,508	4.092%
0x0804a8d0	cfftb.f90:294	93,508	4.092%
0x0804a885	cfftb.f90:287	83,471	3.653%
0x0804a8af	cfftb.f90:291	83,033	3.633%
0x0804a8f8	cfftb.f90:301	64,776	2.835%
0x0804a8ff	cfftb.f90:302	64,776	2.835%
0xb7ee9660	libm expf()	15,100	0.661%
Total		2,285,250	

Table 6.5: Denormal exception profile for 187 . facerec

denormal exceptions.

The instructions in cfftb.f90 all lie in the PassB4 subroutine, which is the first stage of the butterfly transform used by the 1D iFFT subroutine in cfftb.f90. In total, this generates 46% of the denormal exceptions in the test program. The 1D iFFT is only called from the FFT2DB subroutine in fft2d.f90 which performs a 2D iFFT. FFT2DB, in turn is only called from two places — GaborTrafo, as part of the wavelet transform; and from ReadImage to upsample the image. The image has a 2D FFT performed on it, is padded with zeros in the frequency space, and then a 2D iFFT to convert back to an upsampled image. Without further information, such as a stack trace, it is difficult to say which of GaborTrafo or ReadImage is responsible for most of the denormal FFT values.

Finally, the 32-bit e^x function in the standard math library generates a very small number of denormal exceptions (less than 1% of the total). This function is only called from two places — once in the body of GaborTrafo, and in an inner loop of the ComputeKernel routine which initialises the five levels of kernels described in [Sec. 6.1.2](#).

Based on the above, it is difficult to see any significant pattern to the flow of denormals through the application. An examination of the 1D iFFT code shows that PassB4 is the first of a sequence of butterfly passes, and that denormal exceptions do not occur for any of the other passes. So it could be assumed that the inverse

FFT destroys the denormal values passed to it.

6.5.2 Using DART

Running 187. facerec under DART takes about 185 seconds on the test dataset. This is approximately 36.3× the runtime of the 187. facerec running normally. Running 187. facerec under the ‘none’ Valgrind tool, which performs all the translation and compilation steps, but adds no instrumentation produces a runtime of 23.7 seconds or 4.6× slower, leading to an overhead of 7.8× for the denormal tracing functionality.

Since DART can monitor all the denormal floating point operations, and loads and stores from and to marked addresses, it generates a considerably more comprehensive profile than the one produced by DIP. A complete profile can be found in [Appendix F](#), and a summary of that profile is discussed here.

Address	Source	Loads	FP ops	Stores
PassB4 in the 1D inverse FFT				
0x0804a64b-985	cfft.b.f90:267-308	3,918,747	11,454,638	2,942,303
Array copy in FFT2DB				
0x0805035e-77	fft2d.f90:172	976,444	0	976,444
Transpose in FFT2DB				
0x080504f5-fb	fft2d.f90:182	0	0	684,919
Fortran complex array transpose				
0x040a48a0-b5	_gfortran_transpose_c4	684,919	0	0
GaborTrafo				
0x08051542-616	gaborRoutines.f90:110	604,000	2,401,884	976,444
libc’s memcpy				
0x041c9b76	memcpy	604,000	0	0
CFFTB1 in the 1D inverse FFT				
0x0804b32a-51	cfft.b.f90:18-23	512,000	0	0
CFFTF in the 1D inverse FFT				
0x0804dc3b-62	cfft.f.f90:18-23	12,800	0	0
libm’s exponential function				
0x0410c094	_ieee_754_expf	0	15,100	0
exp(x) in ComputeKernel				
0x080508c1	gaborRoutines.f90:199	0	0	15,100
Total		7,312,910	13,871,622	5,595,210

Table 6.6: DART denormal profile summary for 187. facerec

One of the first features evident in this profile is that like with jacobi, the overall denormal read count reported by DART is higher than the denormal exception

count produced by DIP. Unlike `jacobi`, this is not because of repeated 32-bit rounding, but because DART instruments all loads, not just floating point loads. Two examples of this are the libc `memcpy` implementation, which uses only integer instructions to copy blocks of memory, and the Fortran runtime's implementation of array transposition, which also uses only integer operations. DIP cannot capture either of these, but DART can since the operations are on marked arrays and it watches all loads and stores from these arrays.

Another feature of this profile is that some of the load counts are balanced by corresponding stores. For example, line 172 of `fft2d.f90` consists of the array copy statement `FTemp = Freq`. For some reason, instead of using the highly optimised `memcpy`, `gfortran` has chosen to compile this as a tight loop that copies 4 bytes at a time using integer instructions. Since the copy is from a marked array to another marked array, DART counts all the loads and stores, and the profile shows that each loaded value has no floating point operations performed on it and is immediately stored to the destination array.

This balancing of loads and stores occurs more indirectly between line 182 of `fft2d.f90` and the Fortran runtime's implementation of array transposition. The Fortran runtime allocates a temporary array, transposes the source array into it using `_gfortran_transpose_c4` and then copies the temporary array to the destination and frees the temporary array. DART has not marked the temporary array, and so although it tracks the tags for these reads and writes, it does not report the writes to the temporary, or the reads from it. However it does report `_transpose_c4`'s reads from the source array, which is marked memory, and `FFT2DB`'s writes to the destination array, which is also marked. If nothing else, this highlights an inefficiency in `gfortran` — the runtime can determine the source and destination arrays are the same types and of the correct size, and a more efficient compiler would have code to check for this case and would write directly to the destination array.

A similar balancing occurs between the denormal values produced by the floating point operation in `_ieee_754_exp` and the store in `ComputeKernel`.

However, the use of intrinsic functions which operate on entire arrays and produce temporary arrays are common in this code. This includes functions such as `TRANSPOSE` and `CSHIFT` and expressions that extract slices from an array. If the use of these functions is nested, as it is in `GaborTrafo`, copies from one temporary to another will not be reported at all since neither the source nor destination are marked, however as elsewhere the tags are tracked whether they are reported or not.

It should be noted that not all of these denormal loads and stores directly indicate performance problems. For example, the integer loads and stores performed by `memcpy` and `_gfortran_transpose_c4` run at full speed, since the loads and stores do not interpret the values in any way. Valgrind's [IR](#) assigns a type to each load and store based on the instruction that uses it. DART could be extended to distinguish between integer and floating point loads and stores of denormal data and to account for them separately.

From this profile, similar conclusions can be drawn as from the profile generated by DIP: most of the denormal arithmetic occurs in PassB4 in the 1D iFFT and in the Gabor Wavelet Transform. DART's profile shows this, although a much higher proportion—almost 80%—of the denormal floating point operations occur in PassB4. Inspecting the binary shows that many of these are not reported by the exception based tool because they are stack-stack operations performed with the 80-bit registers and the values are always normal in the 80-bit representation. Confirming the earlier conclusion, this profile shows that PassB4 permutes and operates on a large number of denormals, but since the later stages of the iFFT do not, PassB4 must indirectly remove the denormals from the input data.

As with the DIP `187.facerec` profile, it is difficult to see any other pattern to the flow of denormals simply by instruction based profiles. The two other data reduction methods will reveal this.

6.5.3 Denormal heatmaps

Using the DWARF3 debugging information compiled into the `187.facerec` executable, DART detects 20 variables that refer to arrays of floating point or complex variables. All of these arrays are dynamically allocated, and some of the variables are pointers to existing arrays and not distinct arrays of their own right. For example, an array pointer `gaborimage` is declared in both `GenerateGraphs` and `FaceRec` itself, but is passed to `GaborTrafo` and allocated there. `gaborimage` holds the wavelet transformed images which are later passed to other subroutines.

Using the heatmap technique from [Sec. 6.2.2](#), DART can report the array sizes, when they are allocated, and how many denormal reads and writes occur in each array. Figures for these are presented in [Table 6.7](#).

This table shows a number of arrays never store denormal values (the load and store counts are both zero). These arrays can be ignored.

Of the denormal bearing arrays, `ftemp` and `wsaverow` in `FFT2DB` have the most

Type	Subroutine	Name	Dimensions	Loads	Stores
Stack pointer	ReadImage	cimage	256:256	0	0
Stack pointer	ReadImage	finput	256:256	0	0
Global allocatable	ReadImage	fimage	256:256	0	0
Stack pointer	GenerateGraphs	gaborimage	256:256:5:8	0	0
Global allocatable	LocalMove	jetsim	108	0	0
Global allocatable	GaborTrafo	cimage	256:256	0	0
Global pointer	GaborTrafo	ctemp	256:256	684,919	684,919
Global pointer	GaborTrafo	fcimage	256:256	0	0
Global allocatable	GaborTrafo	fctemp	256:256	976,444	976,444
Global pointer	GaborTrafo	kernel	256:256:5	1,208,000	15,100
Global allocatable	FFT2DF	stemp	256:256	0	0
Global allocatable	FFT2DF	wsavecol	1039	6,404	6
Global allocatable	FFT2DF	wsaverow	1039	6,404	6
Global allocatable	FFT2DB	ftemp	256:256	2,410,480	2,410,480
Global allocatable	FFT2DB	wsavecol	1039	256,004	6
Global allocatable	FFT2DB	wsaverow	1039	1,764,271	1,508,273
Stack pointer	Facerec	gaborimage	256:256:5:8	0	0
Stack pointer	Facerec	graphs	8:5:108	0	0
Stack pointer	Facerec	prototype	8:5:108:2	0	0
Stack allocatable	Facerec	probesim	2	0	0

Table 6.7: Denormal array accesses in 187 . facerec

denormal values written to and read from them. Along with fctemp and ctemp in GaborTrafo, they are written to almost as often as they are read from. This suggests that although these arrays manipulate denormal data, they are not the sources of it.

Of the remaining arrays, kernel stands out. It has a small number of denormal writes, but many times more reads. In fact there are exactly $80\times$ as many reads as writes, suggesting this may be a source of denormal values.

Plotting heatmaps for the writes to each of the five kernels, produces the images in Fig. 6.8. In these images each of the kernels has a very distinct band of denormal values. These kernels are initialised by the ComputeKernels subroutine, and it can be seen from there that the array is initialised using values from a negative exponential function. The occurrence of denormals is explained by the shape of the negative exponential curve $y = e^{-x}$ — as $x \rightarrow \infty$, $y \rightarrow 0$, and since it is a continuous function, it passes from $e^{-0} = 1$ through the range of denormal values while doing so. This will be returned to later.

6.5.4 Heatmap interpretation

We now focus on the reads/writes to the fctemp, ftemp, and ctemp arrays. The heatmaps in Fig. 6.9 show all the reads and writes to these arrays. As the figures clearly illustrate, each of these arrays has many denormal accesses, and the patterns

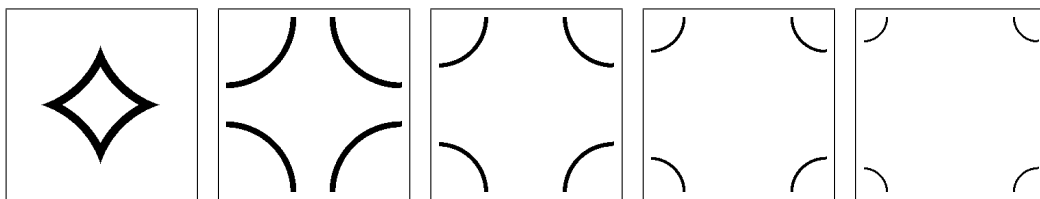


Fig. 6.8: Array elements with denormal values in the five 187 . facerec kernels.

of denormals are translated and distorted copies of the denormals in the kernels.

The contents of `fctemp` are generated by taking an image transformed into the frequency domain, and doing element by element multiplies with translated versions of the kernels. Small values multiplied by denormals also become denormals, which explains why `fctemp`'s heatmaps look so similar to copies of the kernels.

For ordinary images, most of the energy, i.e., the highest values, are concentrated in the low frequency components of the array at the bottom left. This can be seen by the fainter rings towards the bottom of `fctemp` — more values are large enough to be multiplied out of the denormal range by high-valued frequency components.

Once `fctemp` is calculated, a 2D iFFT is performed on it producing `ctemp`. The 2D iFFT is implemented as a copy into `ftemp` (a temporary array in `FFT2DB`); then 1D iFFTs on each row of `ftemp`; followed by a transpose into `ctemp` (the output array); and finally another 1D iFFT on each row of the output array. The initial copy writes from `fctemp` to `ftemp` explaining the similarity of the patterns in `ftemp` and `fctemp`.

The figures show that the horizontal bands in `ftemp` are strongest where the left-most elements have had the strongest denormal contributions from the kernels. An ordinary FFT transformed image will have most of its energy in the lower frequencies. If the lower frequencies are denormal, as they are where the arcs in `ftemp` meet the y axis, it is likely that the higher frequencies are smaller again, and therefore denormal or zero. The 1D iFFT is implemented using the butterfly computation which means that the lowest frequency element will, indirectly, be added to all the other elements in a row. If these values are mostly zero or denormal, they end up becoming or remaining denormal, producing the bands.

The bands are weakest towards the bottom of the image, this is because most of the energy of the image is concentrated here. Adding denormals to a normal will generally produce a normal — the only exceptions are when a narrow range of oppositely signed tiny normals are added to a denormal.

The contents of `ctemp` are produced by transposing `ftemp`, which will have the effect of copying the denormal bands in `ftemp`, and then applying 1D iFFTs to the results, which will produce normal values because the lowest frequency components are normal, and summing normals and non-normals mostly produces normals. The heatmap shows an approximately equal number of reads and writes — the writes from the transpose, and the reads from the iFFTs.

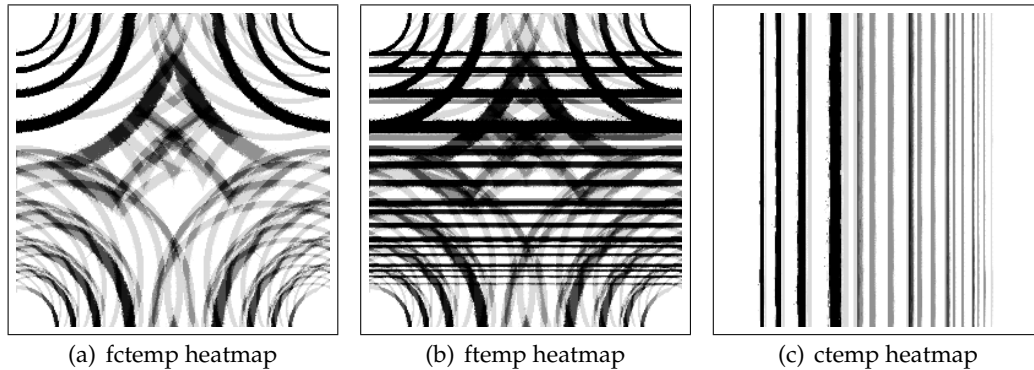


Fig. 6.9: 187.facerec temporary array denormal heatmaps

If the heat maps are plotted at different phases during the execution of the program, the pattern of denormal accesses caused by the wavelet transform and FFTs is shown to be almost identical for each image. This makes sense, as this process of preparing the image for the graph processing is fixed and deterministic — it is performed exactly the same way for each image.

In contrast, the graph fitting process uses random search, and thus the access patterns will vary from image to image. However the jets it uses have entirely normal values, as can be seen in [Table 6.7](#), so it does not contribute to the denormal activity of the application.

6.5.5 Origin maps

Corroborating evidence for the statements above can be provided by examining the origin maps produced by DART. A denormal value written to a marked array is the result of a Directed Acyclic Graph (DAG) of calculations. The root of the DAG is the value and the memory location it is written to; the interior nodes are the floating point operations that produce a result; and the leaves of the DAG are the uninteresting values or reads from marked sources. The graph is a DAG, not a tree because any value in the graph may be the argument to one or more subsequent

Destination	Source	Denormal Origins
fctemp	kernel	477,207
wsaverow	ftemp	5,349,536
ftemp	wsaverow	5,393,897
ftemp	fctemp	976,444
ctemp	ftemp	684,919

Table 6.8: Denormal origins in 187. *facerec*

operations, e.g., $c = a + b$ and $d = c + a$.

For each write to marked memory, the origin tracking finds all the leaves which are reads from marked memory, and finds which source arrays these belong to. For each of these, it then increments the appropriate origin-from-source- X counter at the destination address. Note that it only tracks one ‘lifetime’ for each value. Recall that in [Sec. 5.6.3.2](#) the final step in does not overwrite the tag of a marked memory location when it is written to, because to do so would mean the memory location is no longer marked. So if there are three marked arrays A , B , and C , and values are read from A and written to B , then read from B and written to C , then C ’s values will be regarded as originating from B , not A , as the tags will be followed as far as a marked memory range, and no further.

6.5.6 Origin statistics

As with the heatmaps, DART can produce overall statistics for each pair of arrays, or more detailed plots for specific pairs of arrays. The statistics are listed in [Table 6.8](#). This list is surprisingly short and appears to be missing some arrays such as *wsavecol* in *FFT2DB*.

The first feature of note is that the counts in this table appear to be significantly higher than the equivalent read and write counts in [Table 6.7](#). The fact that the origin graph is a DAG explains this apparent anomaly. A single read from a marked location may be, and sometimes is, reachable by more than path through the origin graph, and each of these paths is counted as a source. This is particularly true of the butterfly transposes in the FFTs.

Next, the only marked source of denormals in *ctemp* is from *ftemp*. This is as expected — the *GaborTrafo* subroutine gets *FFT2DB* to allocate *ctemp* and write the result of an inverse 2D FFT on *fctemp* into it. *FFT2DB* allocates *ftemp* as a temporary array and eventually transposes *ftemp* into *ctemp*.

This also explains why one of the marked sources of denormals in `ftemp` is from `fctemp` — the contents of `fctemp` are copied into `ftemp` at the start of `FFT2DB`.

The other, and more significant source of denormals in `ftemp` is `wsaverow`. This array is the working space used by the inverse 1D FFT which is performed on every row of `ftemp`. Intermediate values are stored alternately in the source row and the working space at different phases of the execution of the 1D FFT. This also accounts for the fact that `ftemp` is the source of all the marked denormals in `wsaverow`.

Finally, this leaves the `fctemp` array — as has already been determined, the only marked source of denormals for `fctemp` is from the kernel.

For every destination array, an origin map may be plotted showing which array elements with denormal values received values from a particular denormal source.

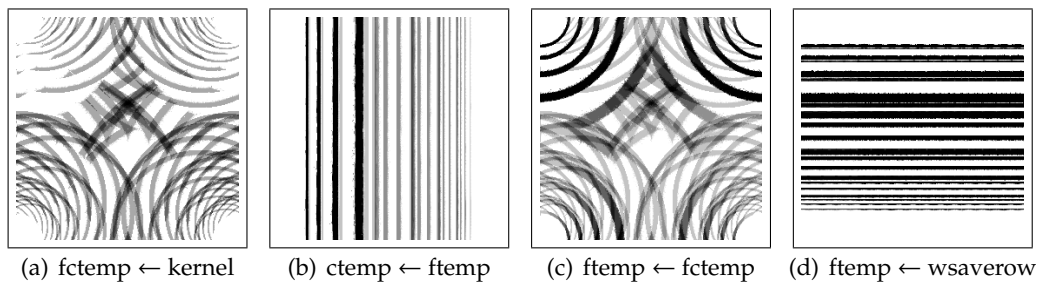


Fig. 6.10: 187 . facerec origin maps

It has already been noted from the statistics in [Table 6.8](#) that `fctemp` gets all its denormals from kernel, and `ctemp` gets all its denormals from `ftemp`. Because of these single sources, it is unsurprising that the origin maps [Fig. 6.10\(a\)](#) and [Fig. 6.10\(b\)](#) have similar structures to [Fig. 6.9\(a\)](#) and [Fig. 6.9\(c\)](#).

What is more interesting is to compare the heatmap for `ftemp` at [Fig. 6.9\(b\)](#) to the two origin maps [Fig. 6.10\(c\)](#) and [Fig. 6.10\(d\)](#). The origins table already showed `ftemp` receives denormal contributions from both `fctemp` and `wsaverow`, and this is confirmed by the denormal heatmap for `ftemp` showing features from both of the origin maps. The denormal arcs in the figures come from `fctemp` (and thus indirectly from the kernel), and the horizontal lines come from `wsaverow`. That the denormal bands come from the results of the iFFT provides further evidence for the explanation in [Sec. 6.5.4](#) for these bands.

6.5.7 Missing arrays

One of the arrays that appears to be missing from this table is kernel itself. However from the earlier profiling, the kernel's denormal values are known not derived from any marked source — they came from the working of libm's expf function.

The other two arrays missing from the table are the wsavecol array in FFT2DB and both wsaverow arrays. To understand 187.facerec's use of these arrays requires an examination the source code for the FFTs. Despite being declared as single arrays of floats, all four working space arrays are in fact treated as composite data structures. For an N element FFT, the workspace is a $4N + 15$ element array. The first $2N$ elements store temporary values for the FFT itself. This shows why the ftemp and wsaverow have such high origin counts for each other — phases of the FFT operate on one array and update the other, and the next phase swaps role of source and destination. The next $2N$ elements are initialised by the CFFTI routine and store multiplicative constants used by the FFTs. CFFTI also initialises the final 15 elements of the array, but despite the array as a whole being declared as floating point, these final 15 are in fact small integer factors derived from the input size N .

A modern Fortran implementation would have used Fortran 90's derived types⁵ to describe the layout of the workspace, but since the FFT routines in question are from the fftpack libraries in the netlib repository, and were written in June 1979, they can perhaps be forgiven this omission.

Since the compiler does not know this, the debugging information it produces does not provide the internal structure of the workspaces, and DART treats the loads and stores as floating point values. All 32-bit integers less than 8,388,608 (except 0) will appear as denormal values and contribute to the denormal load and store count. This appears as the 6 stores and 6,404 loads in FFT2DF's workspace arrays, and the 6 stores and 256,004 loads for FFT2DB's wsavecol array. FFT2DB's wsavecol has a much higher load count than the other two arrays because all these workspace arrays are initialised once and used many times. The 4 loads and 6 writes come from initialising the integer factors, and the remaining loads are reads from within the FFTs themselves. The inverse FFTs are performed 40× as often as the FFTs, leading to the elevated load counts. The FFT2DB wsaverow array will have the same number of spurious denormal loads and stores, but these are masked by the large number of genuine denormal loads and stores into the temporary working space area of the array.

⁵Fortran derived types are similar to structs or records in other languages.

Destination	Source	Denormal Origins
fctemp	kernel	477,207
wsaverow (tmps)	ftemp	5,349,536
ftemp	wsaverow (tmps)	5,393,897
ftemp	fctemp	976,444
ctemp	ftemp	684,919

Table 6.9: Genuine denormal origins in 187 . facerec

The above can be verified by instructing DART to disregard the debugging information and to treat the workspace arrays as 3 separate arrays, and re-profiling the application. This leads to Table 6.9 and Table 6.10 which can be compared to Table 6.8 and Table 6.7. The (spurious) denormal reads all occur in the integer area of each workspace and do not come from any marked source. FFT2DB’s wsaverow is the one exception to this — it has many denormal loads and stores in its temporary value area. Furthermore, the load and store counts are identical, suggesting it is not a source of denormal data.

Subroutine	Name	Dimensions	Loads	Stores
GaborTrafo	ctemp	256:256	684,919	684,919
GaborTrafo	fctemp	256:256	976,444	976,444
GaborTrafo	kernel	256:256:5	1,208,000	15,100
FFT2DF	wsavecol (temporaries)	512	0	0
FFT2DF	wsavecol (constants)	512	0	0
FFT2DF	wsavecol (integers)	15	6,404	6
FFT2DF	wsaverow (temporaries)	512	0	0
FFT2DF	wsaverow (constants)	512	0	0
FFT2DF	wsaverow (integers)	15	6,404	6
FFT2DB	ftemp	256:256	2,410,480	2,410,480
FFT2DB	wsavecol (temporaries)	512	0	0
FFT2DB	wsavecol (constants)	512	0	0
FFT2DB	wsavecol (integers)	15	256,004	6
FFT2DB	wsaverow (temporaries)	512	1,508,267	1,508,267
FFT2DB	wsaverow (constants)	512	0	0
FFT2DB	wsaverow (integers)	15	256,004	6

Table 6.10: Genuine denormal array accesses in 187 . facerec

6.5.8 Removing denormals

From the origin tracking and heatmaps, a path can be traced backwards through the program of where the denormals came from:

- The origin tracking shows that wsaverow and ftemp repeatedly pass denormals

back and forth between each other, and their equal load and store counts suggest neither of them are the source of the denormals.

- Origin tracking shows that ftemp receives some of its denormals from fctemp.
- Origin tracking also shows that fctemp gets all of its marked denormals from the kernel.
- Finally, the profiling showed that the kernel gets its denormals from its initialisation with the expf function in ComputeKernel.

If ComputeKernel is modified to store 0.0 in the kernels when expf produces a result smaller than 10^{-20} , and 187.facerec is rerun, the output of the program is identical to that of the unmodified 187.facerec. This shows that 187.facerec's algorithm is not sensitive to these tiny denormal values.

Re-running the patched 187.facerec using DIP produces [Table 6.11](#). The profile shows the 15,100 denormals calculated in ComputeKernel, but these values are immediately discarded, and no further denormal operations occur.

Address	Source	Count
0xb7ee9660	libm expf()	15,100
Total		15,100

Table 6.11: Denormal exception profile for patched 187.facerec

Running the patched 187.facerec under DART gives the denormal profile in [Table 6.12](#). The only loads and stores in this table are the spurious use of integers in the FFT working space data structures.

Subroutine	Name	Dimensions	Loads	Stores
FFT2DF	wsavecol (integers)	15	6,404	6
FFT2DF	wsaverow (integers)	15	6,404	6
FFT2DB	wsavecol (integers)	15	256,004	6
FFT2DB	wsaverow (integers)	15	256,004	6

Table 6.12: Denormal array accesses in patched 187.facerec

The instruction profile from DART is in [Table 6.13](#). The only denormals that occur are the spurious ones from the FFT workspace arrays, and the libm exp calls that calculate denormal values. The profile shows these 15,100 values are calculated and returned to ComputeKernel, but never stored to a marked array.

This means that the source of unnecessary denormal values in `187.facerec` has been identified correctly and removed without affecting the numerical output of the program. Since there are no longer any denormal values in any of the data structures in the program, there will be no denormal arithmetic except for the few initial calculations in `ComputeKernel`.

Although there was no large scale denormal-based performance variability to speak of in the original `187.facerec`—all the image processing that occurred had a fixed structure to it, so every image filtered would be penalised by denormal arithmetic in exactly the same way—variability could still be seen on a finer grained level. [Fig. 6.10\(d\)](#) shows when the `ftemp` array receives values from `wsaverow`. This happens when the 1-D iFFT routine is called once for each row of `ftemp`. As can be seen, some of the rows are almost entirely black (signifying a large number of denormal calculations for all or most of the elements in the iFFT), and others have no denormal values whatsoever. So in the original `187.facerec` some of these iFFTs will run substantially slower than others.

As shown in [Table 6.6](#) only 7.3 million denormal floating point operations occurred in `187.facerec` when run for approximately 5.5 seconds. This may seem relatively insignificant, however due to the overhead of the denormal operations this leads to a noticeable slowdown. [Appendix H](#) shows the results of benchmarking `187.facerec` with and without the patch on a number of CPUs released from late 2002 to mid 2007. As can be seen, on the AMD CPU, the speedup when denormal operations are removed is a relatively low 6%, but on all the Intel CPUs the speedup is between 20% and 25% due to their less efficient denormal implementation. This is a substantial performance improvement considering the patch simply prevented a small number of denormal operations that are irrelevant to the output of the program.

6.6 Limitations

The above analyses show that with the aid of a denormal tracing tool such as DART, it is possible to infer a great deal about the working of an application and the flow of denormal values within it. A substantial amount of information is supplied to the programmer without getting bogged down in the irrelevant details of function calling conventions, register usage, temporary variable usage and so on. However this information comes at a cost. Firstly there are some restrictions due to the design of DART:

1. A simplifying assumption is made about what kinds of values are significant.

Address	Source	Loads	FP ops	Stores
libm's exponential functions				
0x0410c092		0	15,100	0
CFFTF in the 1D inverse FFT				
0x0804dc3b	cfftf.f90:18	2,560	0	0
0x0804dc62	cfftf.f90:23	10,240	0	0
CFFTB1 in the 1D inverse FFT				
0x0804b32a	cfftb.f90:18	102,400	0	0
0x0804b351	cfftb.f90:23	409,600	0	0
CFFTI initialisation of the workspaces				
0x0804e071	cffti.f90:32	0	0	16
0x0804e0c9	cffti.f90:42	0	0	4
0x0804e0ce	cffti.f90:43	0	0	4
0x0804e104	cffti.f90:49	16	0	0

Table 6.13: DART denormal profile for patched 187 . facerec

DART assumes arrays of possibly dynamically allocated memory are where most denormals will come from and be written to, and these make up the marked memory ranges. There are cases where this assumption causes some odd results. For example, [Table 6.7](#) shows the writes of the denormal exponential values to the kernel array in 187 . facerec, but other methods were required to find which calculations generated those denormals.

2. Simplifying assumptions are made about the 'life cycle' of a value. It is assumed that a value that is written to a marked array is the result of a, hopefully small, DAG of floating point operations whose leaf nodes are loads from other marked locations. This is certainly true for the most part for both applications in this chapter. `jacobi` is simple enough that this completely covers all its behaviour, but 187 . facerec exhibits some anomalies. One of these is when the compiler creates temporary arrays to hold intermediate values. As long as this temporary array is used together with a marked array, reads from and writes to these arrays can be seen, but if a chain of more than one temporary array is used, and operations between two temporaries occur, these groups of intermediate operations are not reported to the end user. This can be seen in the transpose lines of [Table 6.6](#).
3. All important operations are assumed to operate on arrays. This is true of 187 . facerec and `jacobi` but may not be true in the general case. For example consider an application with a helper routine which calculates a value by some iterative method, and incidentally causes many denormal operations to

occur without generating a denormal result. DART will certainly notice these denormal values — it tracks any denormal value generated by a floating point operation, not just the ones with marked sources, however these denormals will only be reported in an instruction profile unless they are written to a marked array.

4. DART does not distinguish between denormal values that are generated by different pieces of code. If it had a means of doing this, for example by attaching a stack trace to each write, the problem of integer values being stored in a floating point workspace explored in [Sec. 6.5.7](#) would have become apparent more quickly.
5. Finally, as is inherent in a dynamic analysis tool, DART can only highlight denormal arithmetic found in a specific run of an application, it cannot identify potential denormal arithmetic that may occur with different data sets or under different scenarios.

Some of these issues could be resolved by refining how DART's reporting occurs. In most of the cases, the desired metadata is generated and propagated correctly, it simply is not reported to the user. A balance needs to be struck between overwhelming the user (and memory subsystem) with mountains of irrelevant data on the one hand, and missing important patterns of behaviour on the other. One possible extension to DART's metadata would be to identify common call-paths in an application by recording frequently occurring stack traces for each array.

Further to this, there are some implementation-level issues:

1. The primary issue is that DART is written in Valgrind and as such is subject to a floating point behaviour different to that of the x87 [FPU](#). Valgrind uses 64-bit registers internally for all its calculations, whereas the x87 uses 80-bit registers. As seen in [Sec. 6.4.3](#) this can cause profiling anomalies depending on the application.
2. Valgrind and shadow memory tracking exacts a significant performance overhead — on the order of 30–40×. This is not a significant problem for either of the applications in this chapter, as they can be easily run with small datasets, but may be an issue for other codes. The instrumentation of every single load and store causes a large part of the slowdown. It may be possible to apply heuristics to apply this instrumentation to a more limited set of instructions.

3. The current implementation of DART assumes that only 32-bit floats are used and is insensitive to the difference between floating point instructions used to copy data, and integer instructions to do the same task. The integer loads and stores correctly copy the tags, but arguably should be reported separately from floating point loads and stores as the integer loads and stores do not have any performance overhead when dealing with denormal data.

6.7 Summary

This chapter described the workings of two applications: one simple C application used for verification purposes, and another more complex Fortran benchmark which generates large numbers of denormals. It showed three ways of reporting the large volumes of tracking data generated by DART — instruction profiling, array heatmaps and origin tracking. One requirement raised in [Chapter 4](#) was the need to know the distribution of denormal values in the program's data sets. Heat maps provide this information. Another concern was to find the source of the denormal data and how it evolves over time. Origin tracking allows the denormal sources to be found, and by generating a sequence of heat maps and origin maps as the program runs the behaviour of the program can be tracked over time.

Finally a detailed analysis was performed on the two applications using both DIP from [Sec. 4.4](#) and DART. The results of both were compared, and using DART the source of denormals in the Fortran benchmark was isolated and removed. This removed the performance penalty caused by denormal arithmetic, and removed the fine grained variability which could be seen in the image processing code.

Conclusions

7.1 Summary

Over the past 60 years, continuing efforts on the part of hardware designers have produced a series of huge increases in CPU power and system capability. Modern high end systems consist of large clusters of compute nodes each containing several CPU cores with a significant degree of architectural complexity and internal state in each of them.

Application performance models are often used to guide the choice of hardware during the procurement process for a compute cluster. Once purchased and installed, middleware is used to schedule and allocate tasks to resources, and these task schedulers need to be able to estimate the resource usage of the tasks they manage to work effectively. In smaller systems, an inability to predict an application's runtime can hinder its use in time critical applications. Many performance monitoring and performance modelling toolkits have been written to address these concerns.

The task of performance modelling is made significantly more difficult by the increases in the internal complexity of CPUs. Performance modelling tools could once assume that counting instruction mixes and memory accesses was enough to characterise an application. Today, with large multilevel caches, extensive pipelines, and sophisticated out of order execution engines, the context in which an instruction runs is far more relevant to performance than the instruction itself. Approximate analytical models have been developed for some of these features, but not for others, and some tools sidestep the problem entirely by directly benchmarking application kernels. However none of these address the issue of data-dependent performance variability.

7.2 Contributions

In this thesis we have examined some of the existing performance modelling techniques and have shown some of their limitations. We have examined a large C++ based medical imaging application that exhibits more than an order of magnitude

of variation in runtime depending on the input data given to it. Because of how it is written, this application does not admit ready analysis using existing tools, but the source of runtime variability for the application turns out to be algorithmic in nature — the application does more work with some datasets than others. A performance model was built for this application based on running ‘probe tasks’ which use subsampled input images in order to predict the behaviour of the full scale tasks, making a hitherto unpredictable application much more predictable.

By contrast, other numerical applications we examined have a fixed pattern of execution, but still exhibit runtime variability. The variability in these applications arises from their occasional use of denormal arithmetic which incurs a severe performance penalty on some CPU architectures. Two tools were written to analyse this behaviour, DIP, a limited tool based on floating point exceptions, and DART, a more sophisticated tool based on dynamic binary instrumentation. DIP makes it possible to identify the amount of denormal arithmetic occurring in an application and the instructions responsible for them. However, these ‘proximate causes’ do not show the distribution of the denormals in an application’s data and provide insufficient information to find the origins of the denormals. DART, in contrast, makes it possible to trace the lifecycle of the denormal values, and by producing denormal read/write maps and denormal origin maps for an application’s principal data structures, allows the distribution and evolution of denormal values therein to be tracked. This information is used to isolate and remove the source of unnecessary denormal values for two applications, thus removing the slowdown caused by denormal arithmetic and producing more regular applications with less data-dependent runtime variability.

7.3 Future work

7.3.1 Generalising DART

The problem DART tries to solve can be envisaged on a more abstract level: Running an application takes a set of input values; and through several or many phases of processing, it generates intermediate values, finally producing an output. Each input value can be envisaged as following an unbroken path through memory and registers via CPU operations having its value transformed along the way until it reaches its final location and value.

If an intermediate value is used by more than one subsequent calculation, the path will branch into multiple paths. Similarly if multiple values are used to calculate

a new value (as in a binary, or n -ary operation), independent paths will join. If an intermediate value ends up not contributing to the output, that branch of the path terminates before the application finishes running. Each initial value can be given a unique label, and for every branch or join in a path, a new label can be created identifying both the new path, and where it came from.

Depending on the application's algorithms, some of these paths will have similar histories for at least some of their length, remaining in some sense 'close' to each other, and thus can be grouped together into bundles. These bundles may occur because of, for example, a phase of computation performing a similar calculation on all the members of an array.

A subset of all the possible values in this application history (the denormal values) cause performance problems on some hardware, and the problem at hand is to determine the past paths of those values when they are encountered in a running application; to identify the bundles they belong to; to characterise the bundles; and to identify the code responsible for each bundle.

DART tackles a subset of this problem. As described in [Sec. 5.6.3.2](#), it chops the full paths described above into pieces bounded by reads from a principal data structure in the application and terminating in writes to another principal data structure. The branches and joins within a path fragment are identified using the technique in [Sec. 6.2.4](#), and a primitive type of bundle characterisation is performed by instruction profiling, as in [Table 6.6](#), and array access profiling, as in [Table 6.7](#), [Table 6.8](#) and [Fig. 6.9](#).

To join these 'piecewise' bundles together into a full history currently requires considerable interpretation and understanding of the application's algorithms as is evidenced by [Sec. 6.5.3](#) and [Sec. 6.5.4](#). A more ambitious tool would automate some or all of this process, or would avoid the constraints of a piecewise entirely.

One possibility is to trace denormal operations backwards to the loads from the principal data structures, as is currently done, but then to permanently tag the values, and observe their evolution beyond the current boundary of a store to another principal data structure. To reduce the volume of reporting data, it might be possible to identify a single or limited number of paths that represent the overall behaviour of an entire bundle. Perhaps these paths always involve the same instructions, or each principal data structure load and store uses memory locations at fixed distances from each other. These paths could be reported individually along with a count of the bundle size, and the remaining paths silently discarded.

7.3.2 Observation and analysis

A more fundamental question is whether or not the observation of a single instance of a running application is enough to characterise the data flows within that application. As noted before, different inputs can trigger different paths of execution within an application. A single, or limited number of runs of an application may not exercise all the important paths of execution within an application, and these paths could have markedly different data flow characteristics.

Speculatively, it might be possible to apply data-flow analysis to the control flow graph of small fragments of an application to try to determine what ranges of inputs cause them to produce denormal values. Similarly, it might be possible to analyse if, when denormal data is fed into a fragment of the application, whether it is copied and grows in frequency or whether it dies away.

This type of analysis may have to approximate what could be a denormal-generating operation, and may have to measure these using probabilistic terms rather than strict binary predicates, as for all the basic arithmetic operators, if one input is held fixed, it is almost always possible to find a second input that will produce a denormal result.

Floating point representations

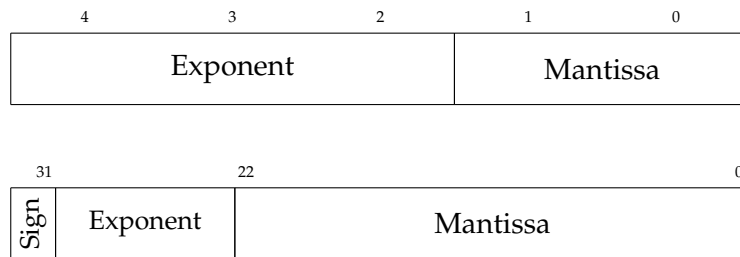


Fig. A.1: 5 bit minifloat compared to IEEE-754 float

This miniature floating point format uses 5 bits of storage; three bits for the exponent, and two bits for the mantissa, and the base is 2. The bias for the exponent is chosen to be -2 to allow the smallest non-zero number to be 1.

There is no sign bit, so only positive numbers can be represented.

Apart from the limited size of the fields, and the lack of a sign bit, it is similar in design to the IEEE-754 standard.

There are two choices of interpretation for the format. The first one uses the IEEE-754 convention of using the smallest possible exponent to indicating denormalized numbers, and the largest possible exponent to indicate the infinity and non-numeric values.

The second interpretation deviates from IEEE-754 by extending the range of valid exponents for normalized numbers to include the largest possible exponent. This removes the ability to represent Infinity or NaN, but increases the range of valid numbers.

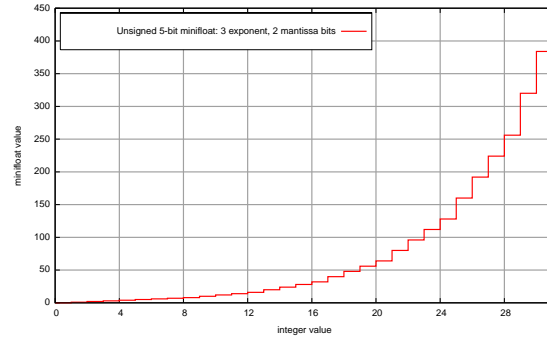


Fig. A.2: A 5 bit ‘minifloat’ format

Exponent bits	Mantissa bits	Interpretation	Value
Denormalized			
000	00	0	0
000	01	$0.01_2 \times 2^2$	1
000	10	$0.10_2 \times 2^2$	2
000	11	$0.11_2 \times 2^2$	3
Normalized			
001	00	$1.00_2 \times 2^2$	4
001	01	$1.01_2 \times 2^2$	5
001	10	$1.10_2 \times 2^2$	6
001	11	$1.11_2 \times 2^2$	7
010	00	$1.00_2 \times 2^3$	8
010	01	$1.01_2 \times 2^3$	10
010	10	$1.10_2 \times 2^3$	12
010	11	$1.11_2 \times 2^3$	14
011	00	$1.00_2 \times 2^4$	16
011	01	$1.01_2 \times 2^4$	20
011	10	$1.10_2 \times 2^4$	24
011	11	$1.11_2 \times 2^4$	28
100	00	$1.00_2 \times 2^5$	32
100	01	$1.01_2 \times 2^5$	40
100	10	$1.10_2 \times 2^5$	48
100	11	$1.11_2 \times 2^5$	56
101	00	$1.00_2 \times 2^6$	64
101	01	$1.01_2 \times 2^6$	80
101	10	$1.10_2 \times 2^6$	96
101	11	$1.11_2 \times 2^6$	112
110	00	$1.00_2 \times 2^7$	128
110	01	$1.01_2 \times 2^7$	160
110	10	$1.10_2 \times 2^7$	192
110	11	$1.11_2 \times 2^7$	224
Large or Infinity/NaN			
111	00	$1.00_2 \times 2^8/\text{Inf}$	256/Inf
111	01	$1.01_2 \times 2^8/\text{NaN}$	320/NaN
111	10	$1.10_2 \times 2^8/\text{NaN}$	384/NaN
111	11	$1.11_2 \times 2^8/\text{NaN}$	448/NaN

Table A.1: 5 bit unsigned minifloat values

Performance modelling in PACE

The following example demonstrates writing [CHIP³S](#) scripts and using the tools provided with PACE.

At the bottom of PACE's layered system are the hardware characterisations. These are created by a suite of benchmarking tools which, when run on idle machines, measure various properties of the hardware. As can be seen in [Fig. B.1](#), the hardware models are flexible and extensible.

```

config SunUltra10 {
  hardware {
    Tclk = 1 / 300,
    Desc = "SUN Ultra 10, U-SPARC II/300MHz, SunOS 5.8";
  }
  cache {
    L1_CAPACITY = 16 * 1024,
    L1_LINE_SIZE = 32,
    L1_ASSOC = 1,
    L1_READ_MISS_CYCLES = 11,
    ...
  }
  clc { /* C language characterisation */
    IFBR = 0.00113894
    CALL = 0.02192,
    LFOR = 0.0101134,
    ...
  }
  mpi {
    DD_COMM_A = 1024,
    DD_TSEND_B = 96.7032,
    DD_TRECV_B = 152.81,
    ...
  }
}

```

Fig. B.1: Excerpts from the `SunUltra.10.hmc1` hardware model

Since the example code is strictly sequential, the parallel template used ([Fig. B.2](#)) is extremely simple. It consists of a single `step` declaration, representing a sequential block of operations. The `confdev Tx` statement associates the primitive operations in a subtask's `compute` statements with the currently selected hardware model. This association means that when the runtime for the subtask is calculated it will use timings from the chosen hardware model.

While the [CHIP³S](#) compiler can be regarded as the core of PACE, other tools provided in the toolkit make PACE significantly easier to use. One of these tools, `capp`,


```

partmp async {
    var compute: Tx;

    option { nstage = 1, seval = 0; }

    proc exec init {
        step cpu {
            confdev Tx;
        }
    }
}

```

Fig. B.2: The `async.la` parallel template

parses C source code, and, with some user input, generates cflow blocks written in [CHIP³S](#).

As an example, it will be shown how to generate an application model for a simple C function ([Fig. B.3](#)). The function `blend` is a straightforward image processing routine¹ which alpha-blends two bitmaps together.

```

/* Composite (ie alpha blend) img2 on top of img1. img1 and img2 are
   w*h pixels in size, each pixel consists of an unsigned 8 bit (r,g,b)
   triple. Pixels are stored left-to-right and top-to-bottom.
*/
void blend(unsigned char *img1, unsigned char *img2, int w, int h, float alpha)
{
    float beta = 1.0 - alpha;
    int i, j, k;

    for(i = 0; i < h; i++) {
        for(j = 0; j < w; j++) {
            for(k = 0; k < 3; k++) {
                *img1++ = (*img1 * beta) + (*img2++ * alpha);
            }
        }
    }
}

```

Fig. B.3: `blend.c`

When `capp` is used on this source code, `capp` cannot determine the loop bounds for itself, so it prompts the user for appropriate values. `capp` produces the contents of `proc cflow blend`, and the user provides the rest of the subtask by hand. The results can be seen in [Fig. B.4](#). The cflow block characterises the performance of the function in terms of the image size, and the costs of computational primitives such as multiplication and pointer arithmetic. The user provided code links to the `async` parallel template, by assigning to `Tx` the cost of the sequence of operations generated by cflow `blend`. The `var numeric` statement exposes the variables `Width` and `Height` as parameters which can be changed by other subtasks.

¹It implements Porter and Duff's *over* operator [[PD84](#)].

```

subtask blend {
    include async;

    var numeric: Width, Height;

    link { async: Tx = blend(); }

    proc cflow blend {
        compute <is clc, FCAL, 4*FARL, FARF, AFDL, TFSL, SILL>;
        loop (<is clc, LFOR>, Height) h{
            compute <is clc, CMLL, SILL>;
            loop (<is clc, LFOR>, Width) {
                compute <is clc, CMLL, SILL>;
                loop (<is clc, LFOR>, 3) {
                    compute <is clc, CMLL, 3*POC1, 2*MCHL, 3*INLL, ACHL, TCHL>;
                }
                compute <is clc, INLL>;
            }
            compute <is clc, INLL>;
        }
    }
}

```

Fig. B.4: blend.la – the blend subtask

Finally, an application model is defined where the user-configurable variables `Width` and `Height` are exposed. The `link` section sets the number of processors available to 1 (this information is used by the parallel template code) and passes the appropriate input variables in to the blend subtask.

The `option` section sets the default hardware model used to be an Apple PowerMac G5. Characterisation data will be read from `AppleG5_2GHz.hmc1`.

The `proc exec init` section is where the evaluation of the model begins. In this case only the blend subtask needs to be evaluated.

```

application blend_app {
    include hardware;
    include blend;

    var numeric: Width = 320, Height = 240;

    link {
        hardware: Nproc = 1;
        blend: Width = Width, Height = Height;
    }

    option {
        hdruse = "AppleG5_2GHz";
    }

    proc exec init { call blend; }
}

```

Fig. B.5: blend_app.la – the application model

To create an executable application model, each of the [CHIP³S](#) scripts, which are stored as .la files, are compiled using the chip3s tool. This tool generates some intermediate C code which it then compiles into an object file. These object files are linked together with the [CHIP³S](#) runtime into an executable. The build process is represented by the Makefile in [Fig. B.6](#).

```
all: blend_app

blend_app: blend_app.o blend.o async.o
    chip3sld -o $@ $^

%.o: %.la
    chip3s -o $@ $<
```

Fig. B.6: Makefile

The executable can be run, and the variables (in this case Width and Height) can be modified. By default, the executable model outputs just a simple execution time. By using different command line options, individual processor usages and various trace and debug output can be produced.

A Linux/x86 LD PRELOAD denormal profiler

Below is included an annotated version of a simple floating point profiler for Linux on x86 platforms.

Assuming it is saved in a file called `denormprofiler.c`, it can be compiled and invoked as follows:

```
cc -fpic -shared denormprofiler.c -o ./denormprofiler.so
LD_PRELOAD=./denormprofiler.so mybinary
```

Normally, the profiling logs are written to stdout, but if the environment variable `DENORMPROF_LOG` is defined, denormal events will be written to the file `DENORMPROF_LOG.pid`, where `pid` is the process ID of the binary being profiled.

The logs consist of lines like the following

```
DENORM: 1233852477.093087 0xb8b2 0xb7e5292d 0xdd00
```

Where the first number is a POSIX timestamp, the second the FPU status word, the third the address of the instruction, and the fourth the first two bytes of the opcode of the instruction that caused the exception.

This raw information can be analysed to find denormal hotspots in the code.

denormprofiler.c

Headers for standard UNIX string, time and signal handling

```
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <signal.h>
5 #include <unistd.h>
6 #include <sys/time.h>
```

Headers for floating point and user mode context

```
7 #include <fenv.h>
8 #include <fpu_control.h>
```

```

9  #define __USE_GNU
10 #include <sys/ucontext.h>

```

GCC directive to mark init_denorm_profiler() to run on library load

```

11 void init_denorm_profiler(void) __attribute__((constructor));

```

Utility function to print a file contents into the log

```

12 FILE *profilelog = NULL;

13 static void copyfile(char *name)
14 {
15     char buf[128];
16     FILE *f = fopen(name, "r");
17     if (f == NULL) return;
18     while(!feof(f)) {
19         fgets(buf, 128, f);
20         fprintf(profilelog, "%s", buf);
21     }
22     fclose(f);
23 }

```

The exception handler proper. If a denormal exception occurred, it reports it, clears the denorm flag, switches off denorm trapping, and single steps the next instruction so it can complete.

```

24 #define FP (uc->uc_mcontext.fpregs)
25 #define EFLAGS_TF 0x100
26 void fpe_handler(int sig, siginfo_t *si, void *ptr)
27 {
28     ucontext_t *uc = (ucontext_t *)ptr;
29     unsigned int op;

30     /* If there is no a denorm, return immediately... */
31     if (FP->sw & _FPU_MASK_DM == 0) { return; }

32     /* Read the first 2 bytes of the opcode of the trapping insn */
33     op = ((unsigned char *)FP->ipoff)[0] << 8;
34     op |= ((unsigned char *)FP->ipoff)[1];

35     struct timeval tv;

```

```

36     gettimeofday(&tv, NULL);
37     fprintf(profilelog, "DENORM: %d.%06d 0x%04x 0x%08x 0x%04x\n",
38         tv.tv_sec, tv.tv_usec,
39         FP->sw,
40         (unsigned int)FP->ipoff,
41         op);
42     fflush(profilelog);

43     /* Disable denorm trapping */
44     FP->cw |= _FPU_MASK_DM;
45     /* Clear the denorm-occurred flag in the FPU status reg */
46     FP->sw &= ~_FPU_MASK_DM;
47     /* Enable single-stepping the next user-mode instruction */
48     uc->uc_mcontext.gregs[REG_EFL] |= EFLAGS_TF;
49 }

```

The trap handler is invoked after a single user-mode instruction has been completed. It turns back on the denormal trapping and disables single-stepping.

```

50 void trap_handler(int sig, siginfo_t *si, void *ptr)
51 {
52     ucontext_t *uc = (ucontext_t *)ptr;
53     /* Turn on denorm trapping again */
54     FP->cw &= ~_FPU_MASK_DM;
55     /* Disable single-step mode*/
56     uc->uc_mcontext.gregs[REG_EFL] &= ~EFLAGS_TF;
57 }

```

Initialisation called on library load. Open the log file, if any, report that the profiler has started; register the floating point and trap handlers, and enable denormal exceptions.

```

58 void init_denorm_profiler()
59 {
60     char tmp[32];

61     char *logname = getenv("DENORMPROF_LOG");
62     if (logname == NULL) {
63         profilelog = stdout;
64     } else {
65         sprintf(tmp, "%s.%d", logname, getpid());
66         profilelog = fopen(tmp, "a");
67     }

```

```

68     fprintf(profilelog, "Initializing denorm profiler on ");
69     sprintf(tmp, "/proc/%d/cmdline", getpid());
70     copyfile(tmp);
71     fprintf(profilelog, "\n");

72     /* We potentially want to unmask any of the bottom 5 bits
73      * in the x86 control word (ie Invalid, Denorm, Div-by-0,
74      * Overflow, Underflow).
75      * Specific to x86 and Linux.
76      */
77     fpu_control_t cw = _FPU_IEEE & ~ _FPU_MASK_DM;
78     _FPU_SETCW(cw);

79     struct sigaction action;

80     action.sa_sigaction = fpe_handler;
81     action.sa_flags = SA_SIGINFO;
82     sigemptyset(&action.sa_mask);
83     sigaction(SIGFPE, &action, NULL);

84     action.sa_sigaction = trap_handler;
85     sigaction(SIGTRAP, &action, NULL);
86 }

```

Integer Optimisations for FP on x86

D.1 Storage classes

GCC stores data in different sections of an executable and generates different DWARF debugging annotations depending on how the C data structures are declared. When the bounds can be determined at compile time, array sizes and loop iteration counts are used to trigger different code generation heuristics. The compiler optimisation level, specified on the command line, triggers other code generation heuristics. Some of the possibilities are listed here.

The storage classes for arrays of data are as follows:

- Global vs local
- Constant vs mutable
- Compile time vs static allocation

Not all of these combinations can be used together; for example it is impossible to have a dynamically allocated array of constant data. The possibilities are:

1. A global array of mutable data allocated at compile time.

```
double d[] = {1.1, 2.2, 3.3, 4.4, 5.5};
```

```
int main()
{
    d[2] = 42.0;
    return 0;
}
```

The array `d` is stored at a fixed offset in the `.data` section of the executable where it may be modified. Array accesses are performed relative to this address, for example a load may use `fldl 0x8049600(%eax,8)`.

2. A global array of constant data allocated at compile time.


```

const double d[] = {1.1, 2.2, 3.3, 4.4, 5.5};
int main()
{
    return 0;
}

```

This is similar to the global mutable compile time case, except the data is stored at a fixed offset in the `.rodata` section of the executable, and may not be modified.

3. A static local mutable array allocated at compile time.

```

int main()
{
    static double d[] = {1.1, 2.2, 3.3, 4.4, 5.5};
    d[2] = 42.0;
    return 0;
}

```

In C, a static local variable is one whose value is preserved after its function is exited. In implementation, it is identical to a global variable, except at compile time the variable is not visible outside the function that contains it. As with the global mutable compile time case, the data is in `.data` at a fixed offset.

4. A constant local array allocated at compile time.

```

int main()
{
    const double d[] = {1.1, 2.2, 3.3, 4.4, 5.5};
    return 0;
}

```

As with static local mutable arrays, these are implemented exactly as globals are, except the scope is restricted the containing function at compile time. The data is in the `.rodata` section at a fixed offset.

5. A local mutable array allocated at compile time.

```

int main()
{
    double d[] = {1.1, 2.2, 3.3, 4.4, 5.5};
    return 0;
}

```

These arrays have their contents set to their initial value every time they are entered. They are implemented by allocating space for the array on the stack at

function entry, and the fixed initial values are copied into the space on the stack using whatever method the compiler decides is the most efficient. Depending on the data involved, the initial data may be stored in the .rodata section, or may be encoded as a literal in several initialisation instructions. Accesses occur using stack-relative addresses, such as `fldl -0x38(%ebp,%eax,8)`

6. Mutable array allocated at run time.

```
int main()
{
    double *d = (double *)malloc(sizeof(double)*5);
    d[2] = 42.0;
    free(d);
    return 0;
}
```

A pointer to the base address of the array is stored on the stack. The storage is allocated at some arbitrary location in the heap by the memory allocator, and accesses occur using pointer arithmetic which the compiler can optimise in many ways depending on the context.

D.2 Assignment

Assignment of a floating point constant to an array element is usually optimised to a single integer move instruction (for 32-bit floats) and two move instructions (for 64-bit floats), as these are shorter and faster than assigning storage to the constant in .rodata and using a pair of `fld/fst` instructions. The `d[2] = 42.0` above might be optimised to

```
movl    $0x00000000,0x80495f0
movl    $0x40450000,0x80495f4
```

0x40450000 is the 32 most significant bits of the 64-bit IEEE-754 encoding of 42.0. 0x00000000 is the 32 least significant bits. For constants where one of the 32-bit words is 0, such as 42.0 above, this can be optimised even further:

```
xor     %eax,%eax
mov     %eax,0x8049654
movl    $0x40450000,0x80495f4
```

Assignment of variable data, i.e., `g[i] = f[i]` is performed using a `fldl` and `fstpl` pair for 64-bit floats, and a `mov` instruction for 32-bit floats.

The code for assignment of entire arrays, as occurs when initialising a local array, depends on the size of the array. With GCC, if the array is less than 3 64-bit floats in length, `fldl` and `fstpl` instructions are used on data stored in `.rodata`. If the array is less than 6 32-bit floats in length, integer `mov` instructions are used with the data stored as literal values in the instruction arguments. For longer 32- or 64-bit arrays, the copy is performed using an integer `rep movsl` loop, and the data is stored in `.rodata`.

If the data is stored inside a structure, and the structure is small, assignment is done using integer `mov` or `rep movsl` instructions. If the structure is larger, it is copied using `memcpy` which may use one of a number of optimisations depending on the CPU available and the alignment of the source and destination.

D.3 DWARF debugging information

DWARF is a tree based debugging format designed to be stored in sections of an ELF executable. The primary data structure in DWARF is the Debugging Information Entry (DIE), and is used to store information such as data types, array bounds, function entry points, function arguments, variable locations, and the mapping of code to source files. DIEs are stored in the `.debug_info` section of an ELF executable and may be nested. One instance of nesting is sub-program DIEs inside a compilation unit DIE. DIEs may also refer to other DIEs, which occurs when defining complex data types (e.g., an array of pointers to floats).

Each DIE has a *tag* which identifies its layout, and a set of *attributes* which label the information it contains. A number of DWARF *forms* are defined for the values an attribute can hold, including booleans, integers, strings, blocks of binary data, references to DIEs, and expressions. *Expressions* are calculations represented in a special bytecode which allow the location of a variable to be defined based on the contents of registers and memory. This bytecode is needed as many data types (such as Fortran array descriptors, and local variables) have locations which can only be determined at runtime. Since the same expressions tend to be used many times, rather than including them directly in an attribute, they are often stored in the ELF `.debug_loc` section and referred to by offset.

The internal layout of a DIE is defined by a set of structures in the ELF `.debug_abbrev` section. For each DIE, it specifies the attributes used, and each of their forms. Since DIEs with the same tags occur many times in the debugging information, this centralised definition of their layout in `.debug_abbrev` is a space saving measure.

An example of the debugging information is the short program below followed by the [DIEs](#) used:

```
int main()
{
    double *d = (double *)malloc(sizeof(double)*5);
    int i;
    for(i = 0; i < 5; i++) d[i] = 1.1 * i;
    d[2] = 42.0;
    for(i = 0; i < 5; i++) printf("%f\n", d[i]);
    free(d);
}
```

DIE: <0x1c3> DW_TAG_compile_unit

```
DW_AT_name      test.c
DW_AT_producer   GNU C 4.3.3
DW_AT_stmt_list  327
DW_AT_low_pc     0x8048434
DW_AT_high_pc    0x80484cd
DW_AT_language   1
DW_AT_comp_dir   /home/bfoley
```

DIE: <0x207> DW_TAG_base_type

```
DW_AT_name      int
DW_AT_byte_size 4
DW_AT_encoding   DW_ATE_signed
```

DIE: <0x22d> DW_TAG_subprogram

```
DW_AT_frame_base 0
DW_AT_sibling     <0x262>
DW_AT_name        main
DW_AT_type        <0x207>
DW_AT_low_pc      0x8048434
DW_AT_high_pc     0x80484cd
DW_AT_call_file   test.c
DW_AT_call_line   4
DW_AT_external    1
```

DIE: <0x249> DW_TAG_variable

```
DW_AT_call_line  5
DW_AT_call_file  1
DW_AT_name        d
```

```

DW_AT_location    DW_OP_fbreg -0x14
DW_AT_type        <0x262>

DIE: <0x255> DW_TAG_variable
  DW_AT_call_line  6
  DW_AT_call_file  1
  DW_AT_name       i
  DW_AT_location    DW_OP_fbreg -0x10
  DW_AT_type        <0x207>

DIE: <0x262> DW_TAG_pointer_type
  DW_AT_type        <0x268>
  DW_AT_byte_size   4

DIE: <0x268> DW_TAG_base_type
  DW_AT_name        double
  DW_AT_byte_size   8
  DW_AT_encoding    DW_ATE_float

```

As we can see, the source code is identified by the compile-unit [DIE](#) 0x1c3. This [DIE](#) names the source file, the compiler, and specifies the language (C is language number 1). The main function in subprogram [DIE](#) 0x22d is nested inside [DIE](#) 0x1c3. The type attribute in the subprogram [DIE](#) represents the return type of the function and refers to [DIE](#) 0x207 which holds the type. In this case it is a base type — a simple 32-bit signed integer.

We can also see main's two local variables nested inside this [DIE](#) as two variable [DIE](#)s. Each of them has a name, a type and a location. The `i` variable points to the same 0x207 [DIE](#) as the return type for main, since they both are 32-bit signed integers. The type for `d` is a little more complex — it is a pointer to a 64-bit float. Since data types in most programming languages can have qualifiers and hierarchical definitions, types are represented in DWARF using a tree of [DIE](#)s. `d` is defined as having the type of [DIE](#) 0x262 — a pointer to another type defined in [DIE](#) 0x268. [DIE](#) 0x268 is a base type that represents a 64-bit float.

The two local variables also illustrate the bytecode used by DWARF to locate a variable. In this case, both variables are local and have a constant offset relative to the stack frame base register.

A more complex example is one for a Fortran dynamically allocated array. The declaration below is for a complex 2D array, allocated at runtime, to be saved across function calls and whose bounds are unknown at compile time.

Complex(4), Allocatable, Save :: FImage (:, :)

This produces the following debugging information

DIE: <0x511b> DW_TAG_variable

DW_AT_call_line 31
DW_AT_call_file 1
DW_AT_name fimage
DW_AT_location DW_OP_addr 0x805adc0
DW_AT_type <0x5239>

DIE: <0x51f0> DW_TAG_base_type

DW_AT_name complex(kind=4)
DW_AT_byte_size 8
DW_AT_encoding DW_ATE_complex_float

DIE: <0x5239> DW_TAG_array_type

DW_AT_sibling <0x5274>
DW_AT_name array2_complex(kind=4)
DW_AT_ordering 1
DW_AT_allocated DW_OP_push_object_address DW_OP_deref
DW_OP_lit0 DW_OP_ne
DW_AT_data_location DW_OP_push_object_address DW_OP_deref
DW_AT_type <0x51f0>

DIE: <0x524f> DW_TAG_subrange_type

DW_AT_byte_stride DW_OP_push_object_address DW_OP_plus_uconst
0xc DW_OP_deref DW_OP_lit8 DW_OP_mul
DW_AT_lower_bound DW_OP_push_object_address DW_OP_plus_uconst
0x10 DW_OP_deref
DW_AT_upper_bound DW_OP_push_object_address DW_OP_plus_uconst
0x14 DW_OP_deref

DIE: <0x5261> DW_TAG_subrange_type

DW_AT_byte_stride DW_OP_push_object_address DW_OP_plus_uconst
0x18 DW_OP_deref DW_OP_lit8 DW_OP_mul
DW_AT_lower_bound DW_OP_push_object_address DW_OP_plus_uconst
0x1c DW_OP_deref
DW_AT_upper_bound DW_OP_push_object_address DW_OP_plus_uconst
0x20 DW_OP_deref

The location attribute in DIE 0x511b shows that the fimage array descriptor pointer is treated as a global variable. It is stored at the fixed address 0x805adc0 in .data.

The type of the variable is substantially more complex. From DIE 0x5239, we learn that it is an array type, and from the array DIE's type attribute, the elements in the array are complex values with 32-bit real and imaginary components.

The allocated attribute tells a debugger how to determine at runtime if the array has been allocated. This is done by dereferencing the first word in the array descriptor and comparing it with zero. Since the first word of the descriptor is the base address of the array data, this shows that the base address of the array is used to imply allocatedness and zero is an invalid address.

The array DIE has two subrange DIEs. These indicate the upper and lower bounds and the stride of each of the dimensions of the array. The rank of the array is the same as the number of subrange child DIEs. The stride of a dimension is the number of bytes an address has to be increased by to advance from index i to $i + 1$.

For the `gfortran` implementation, we happen to know that the array descriptor consists of three words — the base pointer to the data, an offset, and a word that describes both the array element data type and the rank of the array. These three words are followed by triples of words defining the stride, lower bound and upper bound for each dimension of the array. This can be seen in the bytecodes for each of the subranges of the array type DIE. The first subrange uses the 4th, 5th and 6th words in the descriptor (at offsets 12, 16 and 20), and the second subrange uses the 7th, 8th and 9th words.

We can see the `gfortran` array descriptors store strides as counted by elements, as the stride bytecodes multiply these values by 8 (the size of a 32-bit complex value in bytes) to get the size of the stride in bytes.

Valgrind Intermediate Representation and instrumentation

Here we will examine the following sequence of three instructions from the application 187.facerec described in [Sec. 6.1.2](#).

```
0x080507b6:  mov    8(%ebp),%edi
0x080507b9:  fmuls  (%edi)
0x080507bb:  fstps  -88(%ebp)
```

The first instruction loads a word from the address EBP+8 and stores it in the register EDI.

The second instruction reads a 32 bit float from the address in EDI, multiplies the topmost register in the floating point stack by it, and stores the result in the topmost register. This, in effect, treats EBP+8 as a pointer to a floating point value.

The third instruction stores the topmost register in the floating point stack as a 32 bit float at the address EBP-88 and then pops the value from the floating point stack (by updating the top of stack pointer). EBP-88 behaves as a temporary value on the current function's stack frame.

These three operations cause two loads (the pointer load, and the floating point value load), a multiply, and a store, along with a number of register accesses.

The instrumented IR is shown below. Valgrind inserts IMark statements before the IR for every instruction to inform tools of the instruction's address and length.

Each load is instrumented with a call to a load helper function, as is each store. GETs and PUTs (i.e., register reads and writes) are instrumented with corresponding GETs and PUTs for shadow registers. The exception to this is the parts of the guest state that are not directly accessible by the user and cannot store interesting data. Examples of this are FTOP and FPTAG which Valgrind uses to store the x87 stack state.

The instrumentation statements added by DART appear in ***bold italic***.

```
----- IMark(0x80507B6, 3) -----
t10 = GET:I32(EBP)           Read the value in EBP
t64 = GET:I32(s_EBP)       Read the tag for EBP
```


t9 = Add32(t10 ,8)	Add 8 to calculate the effective address
t11 = LDle:I32(t9)	Load from the effective address
t65 = DIRTY :: help_load(t9)	Call helper to read the tag for the address
PUT(EDI) = t11	Store value into EDI
PUT(s_EDI) = t65	Store tag into shadow EDI
----- IMark(0x80507B9, 2) -----	
t15 = LDle:F32(t11)	Load value at EDI. EDI previously set to t11
t66 = DIRTY :: help_load(t11)	Call helper to read the tag for the address
t50 = F32toF64(t15)	Valgrind internal convert to 64 bit float
t12 = GET:I32(FTOP)	Read the pointer to the top of the FP stack
t51 = GETI(FPREG:8xF64)[t12 ,0]	Read the value at the top of the FP stack
t68 = GETI(s_FPREG:8xI32)[t12,0]	Read the tag at the top of the FP stack
t52 = F64i{0x7FF8000000000000}	64 bit NaN constant
t53 = GETI(FPTAG:8xI8)[t12 ,0]	Get the x87 stack tag for top of stack
t54 = Mux0X(t53 , t52 , t51)	Set value to NaN if tag indicates stack is empty
t69 = Mux0X(t53,0,t68)	Tag for above. NaN is constant, so T = 0
t55 = MulF64(0, t54 , t50)	Multiply top of stack by value from EDI
t72 = DIRTY :: help_bin_fpop(t55,t69,t66)	Calculate tag for the multiply
PUTI(FPREG:8xF64)[t12 ,0] = t55	Write the result to the stack
PUTI(s_FPREG:8xI32)[t12,0] = t72	Write the tag to the top of stack
----- IMark(0x80507BB, 3) -----	
t23 = Add32(t10 , -88)	EBP previously read into t10 . Calculate EBP-88
t33 = GET:I32(FPROUND)	Get the FP rounding mode
t32 = And32(t33 ,3)	Keep the bottom two bits
t25 = F64toF32(t32 , t55)	Round top of stack (set to t55 previously) to 32 bit
DIRTY :: help_store(t23,t25,t72)	Update tag and perform any logging for the store
STle(t23) = t25	Perform the store
PUTI(FPTAG:8xI8)[t12 ,0] = 0	Update the x87 tag for the top of stack: no value
t35 = Add32(t12 ,1)	Increment the top of stack pointer (read into t12 previously)
PUT(FTOP) = t35	Update the top of stack pointer

187 . facerec denormal profile

Address	Source	Loads	FP ops	Stores
libgfortran3's complex transpose				
0x040a48a0		172,804	0	0
0x040a48a7		341,992	0	0
0x040a48ad		170,123	0	0
<i>Total</i>		<i>684,919</i>	<i>0</i>	<i>0</i>
libm's exponential functions				
0x0410c094		0	15,100	0
libc's memcpy				
0x041c9b76		604,000	0	0
CFFTB1 in the 1D inverse FFT				
0x0804b32a	cfftb.f90:18	102,400	0	0
0x0804b351	cfftb.f90:23	409,600	0	0
<i>Total</i>		<i>512,000</i>	<i>0</i>	<i>0</i>
CFFTF in the 1D inverse FFT				
0x0804dc3b	cfftf.f90:18	2,560	0	0
0x0804dc62	cfftf.f90:23	10,240	0	0
<i>Total</i>		<i>12,800</i>	<i>0</i>	<i>0</i>
Array copy in FFT2DB				
0x0805035e	fft2d.f90:172	245,051	0	0
0x08050360	fft2d.f90:172	0	0	488,423
0x08050362	fft2d.f90:172	488,021	0	0
0x08050365	fft2d.f90:172	0	0	488,021
0x08050377	fft2d.f90:172	243,372	0	0
<i>Total</i>		<i>976,444</i>	<i>0</i>	<i>976,444</i>
Transpose in FFT2DB				
0x080504f5	fft2d.f90:182	0	0	342,927
0x080504fb	fft2d.f90:182	0	0	341,992
<i>Total</i>		<i>0</i>	<i>0</i>	<i>684,919</i>
exp(x) in ComputeKernel				
0x080508c1	gaborRoutines.f90:199	0	0	15,100
GaborTrafo				
0x08051542	gaborRoutines.f90:110	604,000	809,520	0
0x0805154a	gaborRoutines.f90:110	0	615,920	0
0x0805154f	gaborRoutines.f90:110	0	488,021	0
0x08051554	gaborRoutines.f90:110	0	488,423	0
0x08051614	gaborRoutines.f90:110	0	0	488,423
0x08051616	gaborRoutines.f90:110	0	0	488,021
<i>Total</i>		<i>604,000</i>	<i>2,401,884</i>	<i>976,444</i>
Total		3,394,163	2,416,984	2,652,907

Table F.1: DART denormal profile for 187 . facerec, part I

Address	Source	Loads	FP ops	Stores
PassB4 in the 1D inverse FFT				
0x0804a64b	cfftb.f90:267	87,434	0	0
0x0804a650	cfftb.f90:267	89,105	173,544	0
0x0804a65d	cfftb.f90:269	88,411	0	0
0x0804a662	cfftb.f90:269	90,177	87,827	0
0x0804a66e	cfftb.f90:270	0	86,544	0
0x0804a673	cfftb.f90:271	86,795	0	0
0x0804a679	cfftb.f90:271	87,894	173,402	0
0x0804a687	cfftb.f90:273	89,626	0	0
0x0804a68c	cfftb.f90:273	87,635	0	0
0x0804a690	cfftb.f90:273	0	256,138	0
0x0804a69d	cfftb.f90:275	0	0	83,649
0x0804a69f	cfftb.f90:276	0	84,622	0
0x0804a6a3	cfftb.f90:276	0	0	84,622
0x0804a6a7	cfftb.f90:277	0	84,371	0
0x0804a6a9	cfftb.f90:277	0	0	84,371
0x0804a6ae	cfftb.f90:278	0	84,896	0
0x0804a6b2	cfftb.f90:278	0	0	84,896
0x0804a6bb	cfftb.f90:279	0	85,605	0
0x0804a6bd	cfftb.f90:279	0	0	85,605
0x0804a6bf	cfftb.f90:280	0	89,051	0
0x0804a6c5	cfftb.f90:280	0	0	89,051
0x0804a6c9	cfftb.f90:281	0	84,208	0
0x0804a6cb	cfftb.f90:281	0	0	84,208
0x0804a6ce	cfftb.f90:282	0	88,517	0
0x0804a6d0	cfftb.f90:282	0	0	88,517
0x0804a87d	cfftb.f90:287	402,490	0	0
0x0804a885	cfftb.f90:287	384,024	0	0
0x0804a889	cfftb.f90:287	0	761,494	0
0x0804a895	cfftb.f90:289	491,218	0	0

Table F.2: DART denormal profile for 187 . facerec, part II

Address	Source	Loads	FP ops	Stores
PassB4 in the 1D inverse FFT continued...				
0x0804a89d	cfftb.f90:289	327,814	0	0
0x0804a8a1	cfftb.f90:289	0	567,457	0
0x0804a8a5	cfftb.f90:289	0	60,013	0
0x0804a8aa	cfftb.f90:291	401,320	0	0
0x0804a8af	cfftb.f90:291	382,599	0	0
0x0804a8b3	cfftb.f90:291	0	761,493	0
0x0804a8bc	cfftb.f90:293	493,441	0	0
0x0804a8c7	cfftb.f90:293	328,764	0	0
0x0804a8c9	cfftb.f90:293	0	316,236	0
0x0804a8d6	cfftb.f90:294	0	594,298	0
0x0804a8df	cfftb.f90:295	0	0	279,828
0x0804a8e1	cfftb.f90:296	0	563,038	0
0x0804a8ea	cfftb.f90:297	0	0	279,603
0x0804a8ee	cfftb.f90:298	0	847,216	0
0x0804a8fb	cfftb.f90:301	0	281,956	0
0x0804a8ff	cfftb.f90:302	0	281,927	0
0x0804a91b	cfftb.f90:303	0	833,586	0
0x0804a926	cfftb.f90:303	0	0	282,079
0x0804a92a	cfftb.f90:304	0	290,076	0
0x0804a932	cfftb.f90:304	0	543,096	0
0x0804a93d	cfftb.f90:304	0	0	282,567
0x0804a94c	cfftb.f90:305	0	847,653	0
0x0804a954	cfftb.f90:305	0	0	282,695
0x0804a958	cfftb.f90:306	0	851,813	0
0x0804a962	cfftb.f90:306	0	0	284,834
0x0804a971	cfftb.f90:307	0	836,711	0
0x0804a979	cfftb.f90:307	0	0	282,122
0x0804a97d	cfftb.f90:308	0	837,850	0
0x0804a985	cfftb.f90:308	0	0	283,656
Total		3,918,747	11,454,638	2,942,303

Table F.3: DART denormal profile for 187. facerec, part III

APPENDIX G

nreg case study data

Experiment	Set 1	Set 2	Set 3	Set 4	Set 5	Avg.
light load						
FIFO, sequential						
Makespan	42,463 s	67,104 s	45,821 s	47,869 s	53,808 s	51,413 s
Idle time	52.62%	58.31%	62.61%	68.47%	49.78%	58.24%
FIFO, parallel						
Makespan	34,746 s	48,282 s	29,600 s	26,079 s	466,37 s	37,068 s
Idle time	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
TITAN						
Makespan	27,972 s	31,683 s	24,164 s	27,204 s	33,379 s	28,880 s
Idle time	4.26%	7.89%	10.39%	8.40%	6.27%	7.33%
medium load						
FIFO, sequential						
Makespan	69,952 s	92,259 s	85,053 s	87,765 s	98,435 s	86,692 s
Idle time	34.08%	49.76%	53.97%	61.84%	49.74%	50.50%
FIFO, parallel						
Makespan	79,613 s	80,017 s	67,618 s	57,853 s	85,399 s	74,100 s
Idle time	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
TITAN						
Makespan	57,069 s	58,114 s	54,483 s	48,332 s	64,326 s	56,464 s
Idle time	6.29%	7.11%	10.06%	7.21%	5.66%	7.20%
heavy load						
FIFO, sequential						
Makespan	171,145 s	155,330 s	141,049 s	153,092 s	178,035 s	159,730 s
Idle time	39.99%	45.65%	43.18%	47.62%	48.64%	45.05%
FIFO, parallel						
Makespan	177,268 s	145,785 s	138,397 s	138,481 s	157,859 s	151,558 s
Idle time	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
TITAN						
Makespan	125,236 s	119,656 s	108,909 s	115,611 s	109,823 s	115,847 s
Idle time	6.59%	8.93%	11.12%	8.88%	8.57%	8.76%

Table G.1: Comparing scheduling techniques with varying workloads

Experiment	Set 1	Set 2	Set 3	Set 4	Set 5	Avg.
40 tasks						
FIFO, sequential						
Makespan	63,724 s	72,124 s	61,339 s	72,343 s	59,609 s	65,827 s
Idle time	56.16%	63.28%	60.55%	69.51%	48.02%	60.00%
Probe 6×						
Makespan	46,333 s	41,733 s	42,133 s	43,733 s	44,466 s	43,680 s
Probe 5×						
Makespan	46,667 s	42,200 s	42,666 s	44,000 s	43,533 s	43,813 s
Probe 4×						
Makespan	47,600 s	41,400 s	42,400 s	41,266 s	45,733 s	43,680 s
Perfect						
Makespan	48,750 s	35,625 s	37,210 s	39,522 s	38,827 s	39,986 s
Idle time	14.35%	12.90%	20.32%	20.09%	16.94%	16.84%
80 tasks						
FIFO, sequential						
Makespan	102,908 s	97,589 s	125,160 s	104,288 s	116,471 s	109,283 s
Idle time	42.54%	44.69%	61.22%	52.33%	51.42%	50.96%
Probe 6×						
Makespan	91,467 s	97,266 s	92,133 s	95,000 s	96,400 s	94,453 s
Probe 5×						
Makespan	97,933 s	95,933 s	96,333 s	94,133 s	91,533 s	95,173 s
Probe 4×						
Makespan	98,533 s	93,066 s	94,266 s	89,066 s	91,666 s	93,319 s
Perfect						
Makespan	84,932 s	82,926 s	79,928 s	80,435 s	80,715 s	81,787 s
Idle time	16.93%	20.55%	19.66%	18.98%	19.09%	19.03%
160 tasks						
FIFO, sequential						
Makespan	168,188 s	205,291 s	187,179 s	177,874 s	182,488 s	184,204 s
Idle time	33.04%	49.47%	45.30%	42.31%	38.58%	42.08%
Probe 6×						
Makespan	215,867 s	213,200 s	193,600 s	208,466 s	208,466 s	213,133 s
Probe 5×						
Makespan	221,000 s	206,066 s	199,866 s	213,933 s	221,533 s	212,480 s
Probe 4×						
Makespan	212,600 s	208,600 s	193,933 s	211,933 s	213,333 s	208,080 s
Perfect						
Makespan	182,815 s	179,550 s	160,030 s	177,545 s	186,041 s	177,196 s
Idle time	20.77%	25.83%	21.42%	24.21%	23.32%	23.14%

Table G.2: Comparing different prediction models with varying workloads

187 . facerec runtimes

The following table shows the results of benchmarking 187 . facerec, both with and without the patch in [Sec. 6.5.8](#). The test is run on four machines of different ages and with different microarchitectures. The tests are performed by running 187 . facerec 6 times using the largest dataset provided and benchmarked by measuring the user time reported by the UNIX `time` utility. The results of the first test are discarded to allow for any warmup effects, and the remaining 5 results are averaged.

P4Xeon uses a 2.5 GHz Intel dual core Xeon. This dates from late 2002, is based on the Pentium 4 microarchitecture, has 512 kB L2 cache, 1024 kB L3 cache and was manufactured on a 130 nm process.

Opteron uses a 2.4 GHz AMD dual core Opteron 250. This dates from mid 2003, is based on the Sledgehammer microarchitecture, has 1024 kB L2 cache and was manufactured on a 130 nm process.

Yonah uses a 2.0 GHz Intel Core Duo T2500. This dates from early 2006, is based on the Pentium M microarchitecture, has 2048 kB L2 cache, and was manufactured on a 65 nm process.

Kentsfield uses a 2.4 GHz Intel Core 2 Quad Q6600. This dates from mid 2007, is based on the Core microarchitecture, has 2048 kB L2 cache for each pair of [CPU](#) cores, and was manufactured on a 65 nm process.

Experiment	Set 1	Set 2	Set 3	Set 4	Set 5	Avg.	Speedup
P4Xeon							
Unpatched	531.03s	535.86s	538.48s	536.30s	531.70s	534.67s	
Patched	398.31s	396.81s	397.87s	401.93s	405.38s	400.06s	25.2%
Opteron							
Unpatched	175.85s	178.93s	176.56s	172.94s	176.39s	176.13s	
Patched	164.98s	164.64s	168.28s	161.12s	162.74s	164.35s	6.7%
Yonah							
Unpatched	164.65s	166.25s	167.21s	166.57s	167.10s	166.36s	
Patched	132.85s	133.10s	132.37s	133.03s	132.09s	132.69s	20.2%
Kentsfield							
Unpatched	134.11s	134.19s	134.22s	133.48s	133.77s	133.95	
Patched	101.40s	101.49s	101.52s	101.33s	101.15s	101.38	24.3%

Table H.1: Runtimes of 187 . facerec on different architectures

Bibliography

- [AHKB00] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA*, pages 248–259, 2000. (Cited in [1.1.](#))
- [BA97] Douglas C. Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, UT Austin Technical Report, 1997. (Cited in [2.1.1.](#))
- [BA05] John Markus Bjørndalen and Otto Anshus. Lessons learned in benchmarking – Floating point benchmarks: can you trust them? Technical report, Department of Computer Science, University of Tromsø, Norway, 2005. (Cited in [6.1.](#))
- [BCGH05] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Scheduling Skeleton-Based Grid Applications Using PEPA and NWS. *The Computer Journal*, 48:369–378, March 2005. (Cited in [2.1.5.](#))
- [BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *In Proceedings of CGO’03*, pages 265–276, March 2003. (Cited in [5.](#))
- [BHS⁺95] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NS-95-020, NASA Advanced Supercomputing Division, December 1995. (Cited in [2.1.4.](#))
- [BK04] Stephen W. Boyd and Angelos D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Second International Conference on Applied Cryptography and Network Security*, June 8–11 2004. (Cited in [5.3.](#))
- [Bor03] Shekhar Borkar. Getting Gigascale Chips: Challenges and Opportunities in Continuing Moore’s Law. *ACM Queue*, 1(7):26–33, 2003. (Cited in [1.1.](#))
- [BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, pages 83–94, 2000. (Cited in [2.1.1.](#))
- [CFF⁺04] Karl Czajkowski, Don Ferguson, Ian Foster, Jeff Frey, Steve Graham, Tom Maguire, David Snelling, and Steve Tuecke. From Open Grid Service Infrastructure to WS-Resource Framework: Refactoring and Evolution. Technical report, IBM, 2004. (Cited in [2.2.2.](#))

- [CGH04] Muffy Calder, Stephen Gilmore, and Jane Hillston. Modelling the influence of RKIP on the ERK signalling pathway using the stochastic process algebra PEPA. In *Proceedings of the BioConcur Workshop on Concurrent Models in Molecular Biology*, August 2004. (Cited in 2.1.5.)
- [CK92] Martin Campbell-Kelly. The Airy tape: an early chapter in the history of debugging. *Annals of the History of Computing, IEEE*, 14(4):16–26, 1992. (Cited in 1.)
- [CKPN99] Junwei Cao, Darren J. Kerbyson, Efstathios Papaefstathiou, and Graham R. Nudd. Modelling of ASCI High Performance Applications using PACE. In *Proceedings of the UK Performance Engineering Workshop*, pages 413–424, July 1999. (Cited in Declarations, 1.2, and 2.1.2.)
- [Dug06] Adam Duguid. Coping with the parallelism of BitTorrent: Conversion of PEPA to ODEs in dealing with state space explosion. In *Formal Modeling and Analysis of Timed Systems, 4th International Conference (FORMATS 2006)*, pages 156–170, September 2006. (Cited in 2.1.5.)
- [FFK⁺97] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, pages 365–375, 1997. (Cited in 2.2.2.)
- [Gan] Ganglia monitoring system. <http://ganglia.sourceforge.net>. (Cited in 2.2.)
- [GHLR04] Stephen Gilmore, Jane Hillston, Kloul Leïla, and Marina Ribaudó. Software performance modelling using PEPA nets. *ACM SIGSOFT Software Engineering Notes*, 29(1):13–23, January 2004. (Cited in 2.1.5.)
- [GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a Call Graph Execution Profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982. (Cited in 5.1.)
- [Har99] John Harper. *Analytic Cache Modelling of Numerical Programs*. PhD thesis, University of Warwick, September 1999. (Cited in Declarations, 1.1, and 2.1.2.)
- [HH78] C. A. R. Hoare and C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978. (Cited in 2.1.5.)

- [HHL⁺03] Rolf A. Heckemann, Thomas Hartkens, Kelvin K. Leung, Yalin Zheng, Derek L. G. Hill, Joseph V. Hajnal, and Daniel Rückert. Information Extraction from Medical Images: Developing an e-Science Application Based on the Globus Toolkit. In *Proceedings of the 2nd UK e-Science All Hands Meeting*, 2003. (Cited in 3 and 3.)
- [HMS⁺09] Simon D. Hammond, Gihan R. Mudalige, Jonathan A. Smith, J. A. Herdman, A. Vadgma, and Stephen Jarvis. WARPP - A Toolkit for Simulating High-Performance Parallel Scientific Codes. *2nd ACM International Conference on Simulation Tools and Techniques*, 2–6 March 2009. (Cited in 2.1.3 and 2.1.3.)
- [HSMJ09] Simon D. Hammond, Jonathan A. Smith, Gihan R. Mudalige, and Stephen Jarvis. Predictive Simulation of HPC Applications. *23rd IEEE International Conference on Advanced Information Networking and Applications (AINA '09)*, 26–29 May 2009. (Cited in 2.1.3.)
- [HYH⁺04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D.T. Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Thirteenth International World Wide Web Conference (WWW2004)*, pages 40–52, May 17–22 2004. (Cited in 5.3.)
- [Jon09] Rick Jones. NetPerf – a network performance benchmark. <http://www.netperf.org>, 2009. (Cited in 2.2.)
- [JSK⁺06] Stephen A. Jarvis, Daniel P. Spooner, Hélène Lim Cho Keung, Junwei Cao, Subhash Saini, and Graham R. Nudd. Performance prediction and its use in parallel and distributed computing systems. *Future Generation Computer Systems*, 22(7), 2006. (Cited in 1.2.)
- [JWT04] Markus Kowarschik Josef Weidendorfer and Carsten Trinitis. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In *4th International Conference on Computational Science (ICCS 2004)*, June 2004. (Cited in 3.3.)
- [KHCN96] Darren J. Kerbyson, John S. Harper, Alexander Craig, and Graham R. Nudd. PACE: A Toolset to Investigate and Predict Performance in Parallel Systems. In *Proceedings of the European Parallel Tools Meeting*, Châtillon, France, October 1996. (Cited in Declarations.)
- [KPN98] Darren J. Kerbyson, Efstathios Papaefstathiou, and Graham R. Nudd. Application Execution Steering Using On-the-fly Performance Prediction. In *High-Performance Computing and Networking*, volume 1401 of LNCS, pages 718–727. Springer, 1998. (Cited in 1.2 and 2.1.2.)

- [Lad93] M. Lades. Distortion Invariant Object Recognition in the Dynamic Link Architecture. *IEEE Transactions on Computers*, 42(3):300–311, March 1993. (Cited in 6.1 and 6.1.2.)
- [Lam74] Leslie Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17(2), 1974. (Cited in 2.3.)
- [LCM⁺05] 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)*, pages 190–200, June 2005. (Cited in 5.)
- [MHSJ09] Gihan R. Mudalige, Simon D. Hammond, Jonathan A. Smith, and Stephen Jarvis. Predictive Analysis and Optimisation of Pipelined Wavefront Computations. *23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, 25–29 May 2009. (Cited in 2.3.)
- [Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980. (Cited in 2.1.5.)
- [MJSN06] Gihan R. Mudalige, Stephen A. Jarvis, Daniel P. Spooner, and Graham R. Nudd. Predictive Performance Analysis of a Parallel Pipelined Synchronous Wavefront Application for Commodity Processor Cluster Systems. In *IEEE International Conference on Cluster Computing*, 2006. (Cited in Declarations and 2.4.)
- [Mon07] Randall Monroe. Exploits of a Mom. <http://xkcd.com/327>, October 10 2007. (Cited in 5.3.)
- [MVJ08] Gihan Mudalige, Mary K. Vernon, and Stephen A. Jarvis. A Plug-and-Play Model for Evaluating Wavefront Computations on Parallel Architectures. *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, 14–18 April 2008. (Cited in 2.1.3, 2.3, 2.3, and 2.3.)
- [Nag] Nagios infrastructure monitoring system. <http://nagios.org>. (Cited in 2.2.)
- [NKP⁺00] Graham .R. Nudd, Darren J. Kerbyson, Efstathios Papaefstathiou, John S. Harper, Stewart C. Perry, and Daniel V. Wilcox. PACE: A Toolset for the Performance Prediction of Parallel and Distributed

- Systems. *The International Journal of High Performance Computing Applications, Special Issues on Performance Modelling*, 14(3):228–251, 2000. (Cited in 1.1, 1.2, and 2.1.2.)
- [NS05] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS '05)*, February 2005. (Cited in 5.3.1.)
- [NS07a] Nicholas Nethercote and Julian Seward. How to Shadow Every Byte of Memory Used by a Program. In *Proceedings of VEE 2007*, June 2007. (Cited in 5.2.)
- [NS07b] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of PLDI 2007*, June 2007. (Cited in 1.3 and 5.)
- [Ope] Paderborn Center for Parallel Computing OpenCCS. <https://www.openccs.eu/core>. (Cited in 2.4.)
- [Pan02] Vijay Pande. Atomistic protein folding simulations on the submillisecond timescale using worldwide distributed computing. *Biopolymers*, 2002. (Cited in 1.)
- [Pap95] Efsthios Papaefstathiou. *A Framework for Characterising Parallel Systems for Performance Evaluation*. PhD thesis, University of Warwick, September 1995. (Cited in Declarations.)
- [PD84] Thomas Porter and Tom Duff. Compositing digital images. *Computer Graphics*, 18(3):253–259, July 1984. (Cited in 1.)
- [PKP03] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *ACM/IEEE SC2003*, Phoenix, Arizona, November 2003. (Cited in 2 and 2.1.3.)
- [PKvdM96] Michael Pöttsch, Norbert Krüger, and Christoph von der Malsburg. Improving object recognition by transforming gabor filter responses. *Network: Computation in Neural Systems*, 7(2):341–347, 1996. (Cited in (document) and 6.2.)
- [RBH⁺04] A. L. Rowland, M. Burns, T. Hartkens, J. V. Hajnal, D. Rückert, and Derek L. G. Hill. Information eXtraction from Images (IXI): Image Processing Workflows Using A Grid Enabled Image Database. In *DiDaMIC Workshop - MICCAI*, pages 104–111, 2004. (Cited in 3.6 and 3.8.)

- [RSH⁺99] D. Rückert, L. I. Sonoda, C. Hayes, D. L. G. Hill, M. O. Leach, and D. J. Hawkes. Nonrigid Registration Using Free-Form Deformations: Application to Breast MR Images. *IEEE Transactions on Medical Imaging*, 18(8):712–721, August 1999. (Cited in [3.1.](#))
- [SCJ⁺04] Daniel P. Spooner, Junwei Cao, Stephen A. Jarvis, Ligang He, and Graham R. Nudd. Performance-aware Workflow Management for Grid Computing. *The Computer Journal*, 2004. (Cited in [2.4.](#))
- [SCT⁺02] Daniel P. Spooner, Junwei Cao, James David Turner, Hélène N. Lim Choi Keung, Stephen A. Jarvis, and Graham R. Nudd. Localised Workload Management Using Performance Prediction and QoS Contracts. In *18th Annual UK Performance Engineering Workshop*, July 2002. (Cited in [Declarations.](#))
- [SHM⁺09] Jonathan A. Smith, Simon D. Hammond, Gihan R. Mudalige, J.A. Davis, A.B. Mills, and Stephen Jarvis. hpsgprof: A New Profiling Tool for Large-Scale Parallel Scientific Codes. *UK Performance Engineering Workshop 2009*, 6 July 2009. (Cited in [2.1.3.](#))
- [SJC⁺03] Daniel P. Spooner, Stephen A. Jarvis, Junwei Cao, Subhash Saini, and Graham R. Nudd. Local Grid Scheduling Techniques using Performance Prediction. *IEE Proceedings of Computers and Digital Techniques*, 15(2):87–96, 2003. (Cited in [Declarations](#), [1.2](#), [1.2](#), [2.1.2](#), [2.4](#), and [2.4.](#))
- [SKD⁺03] Daniel P. Spooner, Hélène Lim Choi Keung, Justin R.D. Dyson, Lei Zhao, and Graham R. Nudd. Performance-based Middleware Services for Grid Computing. In *12th IEEE International Symposium on High Performance Distributed Computing*, pages 151–159, Seattle, USA, June 2003. (Cited in [Declarations.](#))
- [Smi02] Stephen M. Smith. Fast Robust Automated Brain Extraction. *Human Brain Mapping*, 17(3):143–155, November 2002. (Cited in [3.5](#) and [3.10.](#))
- [SN05] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX’05 Annual Technical Conference*, April 2005. (Cited in [5.2.](#))
- [Tan01] Andrew Tanenbaum. *Modern Operating Systems*. 2001. (Cited in [5.6.1.](#))
- [TCF⁺03] Stephen Tuecke, Karl Czajkowski, Ian Foster, Jeffrey Frey, Steve Graham, Carl Kesselman, Tom Maguire, Thomas Sandholm, David Snelling, and Peter Vanderbilt. Open Grid Services Infrastructure (OGSI) Version 1.0. Technical report, Global Grid Forum Draft Recommendation, June 27, 2003. (Cited in [2.2.2.](#))

- [TCPP98] MacFarland T., H. M. P. Couchman, F. R. Pearce, and J. Pichlmeier. A New Parallel P3M Code Very Large Cosmological Simulations. *New Astronomy*, 3(8):687–705, 1998. (Cited in 1.)
- [TLHKN02] James David Turner, Raul Lopez-Hernandez, Darren J. Kerbyson, and Graham R. Nudd. Performance Optimisation of a Lossless Compression Algorithm using the PACE Toolkit. Technical Report CS-RR-389, University of Warwick Research Report, 10 May 2002. (Cited in [Declarations](#) and 3.5.1.)
- [Tur03] James David Turner. *A Dynamic Prediction and Monitoring Framework for Distributed Applications*. PhD thesis, University of Warwick, May 2003. (Cited in 2.1.2.)
- [TWGS02] Valerie Taylor, Xingfu Wu, Jonathan Geisler, and Rick Stevens. Using Kernel Couplings to Predict Parallel Application Performance. In *Proceedings of HPDC 2002*, July 2002. (Cited in 2.1.4 and 2.1.4.)
- [TWL⁺01] Valerie Taylor, Xingfu Wu, Xin Li, Jonathan Geisler, Zhiling Lan, Mark Hereld, Ivan Judson, and Rick Stevens. Prophecy: Automating the Modeling Process. In *Third Annual International Workshop on Active Middleware Services*, 2001. (Cited in 2.1.4.)
- [TWS03] Valerie Taylor, Xingfu Wu, and Rick Stevens. Prophecy: An Infrastructure for Performance Analysis and Modeling of Parallel and Grid Applications. *ACM SIGMETRICS Performance Evaluation Review*, 30(4), March 2003. (Cited in 2.1.4.)
- [vH60] Sebastian von Hoerner. Die numerische Integration des n-Körper-Problemes für Sternhaufen. I. *Z. Astrophys*, 50:184–214, 1960. (Cited in 1.)
- [Wal64] Christopher Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, 13:114–117, February 1964. (Cited in 1.1.)
- [WFW93] Robert Wilson, Robert French, and Christopher Wilson. An Overview of the SUIF Compiler System. Technical report, Computer Systems Lab Stanford University, 1993. (Cited in 2.1.2.)
- [Wol96] George Woltman. The Great Internet Mersenne Prime Search. <http://www.mersenne.org/prime.htm>, 1996. (Cited in 1.)
- [Wol03] Rich Wolski. Experiences with Predicting Resource Performance Online in Computational Grid Settings. *ACM SIGMETRICS Performance Evaluation Review*, 30(4):41–49, March 2003. (Cited in 2.2.1.)

- [WS-03] Web Services Resource Properties. <http://www.ibm.com/developerworks/library/ws-resource/ws-resourceproperties.pdf>, 2003. (Cited in 2.2.2.)
- [WS-04a] Web Services Resource Properties. <http://www.ibm.com/developerworks/library/ws-pubsub/WS-PubSub.pdf>, 2004. (Cited in 2.2.2.)
- [WS-04b] Web Services Service Group. <http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ServiceGroup-1.2-draft-02.pdf>, 2004. (Cited in 2.2.2.)
- [WSH99] Rich Wolski, Neil Spring, and Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5–6):757–768, October 1999. (Cited in 2.2.1 and 2.2.1.)
- [WTS01] Xingfu Wu, Valerie Taylor, and Rick Stevens. Design and Implementation of Prophecy Automatic Instrumentation and Data Entry System. In *Proc. of the 13th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS2001)*, August 2001. (Cited in 2.1.4.)
- [ZBS01] Yongyue. Zhang, Michael Brady, and Stephen Smith. Segmentation of Brain MR Images Through a Hidden Markov Random Field Model and the Expectation Maximization Algorithm. *IEEE Transactions on Medical Imaging*, 20(1):45–57, 2001. (Cited in 3.5.)

Colophon

This thesis was produced on an Apple MacBook running Mac OS X. TextMate was used as the text editor and `pdflatex` was used to typeset the thesis. The $\text{T}_{\text{E}}\text{X}$ Live distribution proved to be a valuable source of $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ packages used throughout, and the thesis layout is based on a modified version of Mark Hadley's Warwick thesis style file `wnewthesis`.

Bibliographic information was maintained and processed using Bib $\text{T}_{\text{E}}\text{X}$. The figures were produced and processed using Gnuplot, ImageMagick, Multivalent, the Python Imaging Library, OmniGraffle, GraphicConvertor and Adobe Photoshop.

Source control was provided by Subversion which served as a much needed buffer for unwise editorial decisions, and saved the author from catastrophe on a number of occasions.