Towards Retargettable Compilers — Feedback Directed Compilation Using Genetic Algorithms (Work in Progress)

Dr. Andy P. Nisbet, Andy.Nisbet@cs.tcd.ie
Department of Computer Science, Trinity College Dublin, Ireland

Abstract

The GAPS [6, 7, 8, 9, 10] research project investigated the use of genetic algorithm (GA) optimisation techniques in the compilation of programs for SPMD execution on parallel architectures, where an GA was used to control the application of high-level loop restructuring techniques to a program. Loop transformations are represented as mappings that are applied to each statement and its associated iteration space.

In this paper the **I**terative **C**ompilation **E**nvironment (ICE)[1], is described. ICE utilises GAPS techniques to investigate the architectural independence of GA techniques when compiling[2] ADI (1024x1024), TOMCATV (N=257,itact=5) and SHALLOW (N=512) benchmarks for i), Philips TM1300 VLIW processor, ii), Intel PIII and iii), Sun UltraSPARC-IIi. ICE improvements to GAPS allow a wider range of transformations to be applied by the GA which include padding of arrays and loop permutation, fission, fusion, unrolling and tiling. Further, ICE can use profile information to ensure that statements accounting for large portions of a program's total execution time have an enhanced probability of selection for transformation by the GA.

1 Introduction

The use of program transformations to improve program performance has been extensively studied for many years. Such work is based primarily on static anaylsis, possibly in conjunction with some profile information to determine the significant regions of the code, and runtime dependent control-flow. Many compiler research papers dealing with program optimisation focus on a particular performance problem on a particular target architecture and attempt to generate a transformation and a cost-benefit framework to determine when to apply the particular optimisation. Typically, the cost-benefit framework considers the new transformation in isolation and does not make any significant attempt to consider the effect of other program transformations on optimisation. The reasons for this shortcoming are, i), an infinite number of transformations can be

[1]This work is supported by Trinity College lectureship startup grant R01019.
[2]Apologies, we aim to present full results at the conference for all benchmarks and all architectures.

applied, and ii), determining the eventual total overhead for a particular target architecture is intractable in general. Thus, it is extremely difficult to integrate the piecemeal nature of compiler research into a global program optimisation framework. Further, the strategies employed by commercial/research compilers to perform global program optimisation are heavily dependent on the target processor architecture.

Compiler technology requires a strategy which can:–

- apply transformations in order to globally minimise all sources of overhead.
- perform global program optimisation for different processor architectures.

Such a strategy is possible with the use of genetic algorithm techniques. Section 2 describes what an GA is and how and they may be used to optimise compilation for different processor architectures. Section 3 describe the ICE project infrastructure. Section 4 describes the method and the actual experiments performed whilst section 5 discusses the results of the experiments. Section 6 and 7 discuss related work and conclusions/future work respectively.

2 Genetic Algorithm (GA) Optimisation

GAs iteratively seek better solutions to problems in the same way that evolution has optimised populations of living organisms for their habitat and environment. The process of reproduction creates new child organisms through the application of recombination and mutation operators to *encodings* of parent organisms. The processes of natural selection ensure that successful organisms reproduce more often than less successful ones. In a GA, an organism represents a potential solution in a problem domain. A *population* of organisms represents a subset of the possible solutions to a problem. Solutions are represented in a population as encodings. Feedback on the success/performance of a solution encoding in solving a problem is measured by a solution's *fitness* value which is calculated by an *evaluation* function. High fitness solution encodings are selected for reproduction more frequently than low fitness solutions in order to mimic natural selection. Thus, GAs can evolve good solutions to complex optimisation problems via the manipulation of solution encodings.

2.1 GA Optimised Compilation

GAs can be applied to compilation problems using three basic concepts:– i) An encoding describes a *legal ordered set of transformations*[3]. ii) Mutation and recombination reproduction operators describe mechanisms that alter encodings. iii) An evaluation function represents a target architecture plus the application program to be compiled. Evaluation functions calculate the total overhead of the application program when transformed by an encoding, and give high fitness values to encodings which yield low overheads and hence low execution time.

[3] Program semantics are preserved when code is generated by the application of a legal ordered set of transformations to a program.

2.2 Architectural Independence

GAs can be used to evolve transformations, which yield programs having low overheads on a particular target architecture. The GA itself requires no knowledge of the target architecture, all information concerning the target architecture can be encapsulated by the evaluation function. Thus, GA optimised compilation can be made *architecturally independent* if:–

1. Reproduction operators contain no architecture specific information.
2. The evaluation function is the measured execution time of the transformed program on the target architecture.

Retargetting an GA optimised compiler to a new processor architecture then consists of changing the evaluation function to measure the execution time of transformed programs on the new architecture.

3 ICE Infrastructure

The ICE approach has been implemented using the Omega Project's [1] Unified Transformation Framework (UTF) [2] in the *petit* research prototype compiler. The UTF was chosen because it offers a comprehensive set of transformations. The UTF provides two simple mathematical abstractions:– *time-mappings* and *space-mappings* which respectively and concisely specify i) the loop-restructuring transformations applied, and ii), the partition of computation for parallel execution[4]. These mathematical abstractions enable a sequence of traditional loop and partitioning transformations to be represented by a uniform data structure amenable to encoding within a GA. Further, they remove the need to check for sub-sequences of loop-restructuring transformations that are equivalent to an identity transformation.

3.1 Time-mappings

The total order in which statements and any iterations associated with a statement are executed by a program, are respectively described by a time-mapping T_p and an iteration-space I_p for each and every statement p in the program. The *iteration variables* $i_{p1}, \ldots, i_{pm_p}$ in the domain of a time-mapping T_p represent the number of loops m_p surrounding a particular statement p. A mapping T_p maps an iteration-space having $[i_{p1}, \ldots, i_{pm_p}]$ iteration-variables in the domain onto an iteration-space in the range $[f_{p0}, \ldots, f_{pn}]|C_p$ where the f_{pk} expressions are quasi-affine[5] functions of the iteration variables and symbolic constants, and C_p is an optional restriction on the domain of the mapping. Note, each of the positions in the range of a time-mapping $0, 1, ..., n$ are referred to as *levels*. The points in the new iteration spaces for each and every time-mapping range are then executed in lexicographic order as specified by the operator $\prec$ as defined in equation 1.

[4] Note, in ICE we merely restructure the code using time-mappings and do not utilise space-mappings.

[5] Quasi-affine expressions are affine functions plus integer division and remainder when dividing by a constant.

$$(x_1, \ldots, x_n) \prec (y_1, \ldots, y_n) \Leftrightarrow \exists m : (\forall i \; 1 \leq i < m \Rightarrow x_i = y_i) \wedge x_m < y_m \qquad (1)$$

The concept of a *group* is introduced in order to relate the structure and the number of loops in a transformed program to its associated time-mappings. Two statements p, q are said to be in the same *group* $g_p^k \Leftrightarrow g_q^k$ at level k if $\forall i : (0 \leq i \leq k) \wedge (f_{pi} \Leftrightarrow f_{qi})$ where f_{pi} and f_{qi} are both constants. The effect of two statements existing in the same group at a level k is that the execution periods of any loops specified by $\forall i : ((0 \leq i \leq k) \wedge ((f_{pi} \neq constant) \vee (f_{qi} \neq constant))$ will overlap. Note, although statements may be in the same group at levels $< n$, at level n each statement must be in a group that is distinct from that of other statements. Thus, time-mappings specify a total order for all statements in the restructured source code.

Legality of Time-mappings The iteration reordering program transformations specified by time-mappings are only legal if the original program semantics are preserved. This is true if, i) the restructured program performs exactly the same set of computations and, ii), the new ordering of iterations respects all control-flow and data dependences in the original code. If i is an iteration of statement p and j an iteration of statement q, and the dependence relation d_{pq} indicates that a dependence exists from i to j then $T_p(i)$ must be executed before $T_q(j)$, as in equation 2 where Sym is the set of all symbolic constants.

$$\forall i, j, p, q, Sym \; i \rightarrow j \in d_{pq} \Rightarrow T_p(i) \prec T_q(j) \qquad (2)$$

3.2 Transformation Encodings

A member of the population P_i is encoded by representations of a multidimensional time-mapping for each and every statement in the original program. Time-mappings are represented as:–

1. the levels at which an iteration variable is scheduled and the value of any alignment constant at that level.
2. the unrolling factor applied to an iteration variable to be unrolled, and the level at which the unrolled iterations are to be scheduled.
3. the tile factor and the level where a tiled iterator is inserted if an iteration variable is to be tiled.
4. the values of the constants at levels where no iteration variable or tiled iterators are scheduled.

Note, ICE forces even numbered levels of time-mappings to contain integer constant values whilst oddly numbered levels can contain an *iterator schedule* (this includes iterator variables and tiling/unrolling) with an alignment integer constant or the value zero. This simplifies the implementation of GA reproduction operators without restricting the transformations which can be expressed as time-mappings. ICE currently uses loop permutation and loop alignment decisions generated by *petit*. This means the order in which iteration variables appear in a time-mapping is determined by *petit*, however the GA can alter the positions at which an iteration variable and an alignment constant appear. Functionality which removes this restriction from ICE is currently under development.

3.3 Reproduction Operators

Reproduction operators are currently given equal and fixed selection probability. An operator and two individuals are randomly selected for reproduction in order to create two new encodings. ICE has implemented 6 different reproduction operators which come in normal and profile directed flavours.

Normal reproduction operators assign equal selection probability to all statements irrespective of their computational cost, whereas *profile-directed operators* bias statement selection probability in favour of statements having high measured computational costs in order to direct reproductive effort towards high cost statements. Statements are then selected according to a linearly normalised cost function which makes the probability of selecting the statement having the highest cost 10 times that of the lowest cost statement. ICE inserts timer calls protected by `#ifdef PROFILE ... #endif` preprocessor macros around loop nests and scalar statements. Profile information is then obtained by recompiling and rerunning the executable with `PROFILE` defined. Each loop nest has a separate timer value whose value is printed on program exit. The ICE framework associates each timer value with the group of statements enclosed by the timer's start and stop calls. Currently, the profile information[6] consists of cycles executed.

3.3.1 Group-based Mutation

This operator alters the grouping of a pair of parent P_i's statements in order to produce a new child C_i encoding having a different group structure. A second child C_j is created by another application of this operator to parent P_j. The alterations to group structure may change the number and structure of loops. in the transformed program.

The group-based mutation operator randomly selects an even numbered level k, and two statements p, q. The lexicographic ordering of $T_p^{k'}$ and $T_q^{k'}$ at level k is then evaluated where $T_p^{k'} = (f_{p0}, f_{p2}, \ldots, f_{p(k-2)}, f_{pk})$. A new and different lexicographic ordering for $T_p^{k'}$ and $T_q^{k'}$ is then randomly selected and enforced at level k.

3.3.2 Group-based Crossover

This operator combines information from two parents P_i, P_j and produces a single child C_i with the aim of creating new statement groupings. Child C_j is produced by a second application of the operator on P_j, P_i. An application of group-based crossover on P_i, P_j creates the child C_i such that the values of the f_{pk} constants are taken from

[6]Profile directed reproduction operators currently only utilise cycles executed, future work will exploit the use of CPU performance counter information obtained using PAPI [11].

P_i if $(k$ *is even* $) \wedge k \in K$ else they are taken from P_j whilst the iterator scheduling is taken from P_i. The $|K|$ is randomly chosen such that $1 \leq |K| \leq 2(n+1)/3$.

Note that in group-based crossover the structure of the groups for all statements at a particular level k are potentially altered whereas the group-based mutation operator directly alters the grouping of a statement-pair.

3.3.3 Iterator-based Crossover

Iterator-based crossover aims to fuse loop-nests and reorder statements having different nesting depths in order to reduce loop overhead and cache-misses. Iterator-based crossover combines iterator scheduling information from two parents P_i, P_j and produces a single child C_i. A second application of this operator on P_j, P_i creates child C_j. This operator randomly selects a set of statements $P : 1 \leq |P| \leq 2(|Stmts|+1)/3$ and imposes the iterator scheduling of statements $p : p \in P$ from P_i onto a child encoding. The total ordering of f_{pk}'s and f_{qk}'s (for even k) in the child, and the iterator scheduling for statements $q : q \notin P$ are inherited from the encoding of P_j.

3.3.4 Padding Mutation

Padding is a data storage transformation that seeks to alter the layout of elements of a matrix in memory. The padding mutation operator randomly selects an array variable, and one of its dimensions of a parent P_i, in order to produce a new child C_i encoding with the aim of inserting, removing, or altering the padding to a value in the range 1..32. A second child C_j is created by another application of this operator to parent P_j. Padding, in conjunction with loop tiling, and, or loop unrolling transformations can have a dramatic effect on the number of cache misses experienced by a program fragment.

3.3.5 Tile Mutation

This operator directly alters the iterator schedule of a parent P_i in order to produce a new child C_i encoding with the aim of introducing, altering or removing tiled loops. A second child C_j is created by another application of this operator to parent P_j. Loop tiling is generally used with the aim of improving memory locality especially within the data-cache. A statement and a corresponding iterator are randomly selected. Tiling an iterator involves randomly choosing a tile size between 2 and 64 and randomly deciding on a level to place the new tiled iterator. If the iterator is already tiled then a choice is randomly made to i), tile the iterator with a new randomly selected tile size and tiled iterator insert level, or to ii), remove the tile. If the iterator is already unrolled then the unroll is replaced with a tile.

3.3.6 Unroll Mutation

This operator directly alters the iterator schedule of a parent P_i in order to produce a new child C_i encoding with the aim of introducing, altering or removing unrolling. A second child C_j is created by another application of this operator to parent P_j. Unrolling replicates the body of a loop by the *unrolling factor* (u) and then iterates by a step $u \times o$ where o is the original step of the loop. Unrolling can improve perofrmance by, reducing loop overhead, increasing instruction level parallelism and by improving register, data-cache or TLB locality. A statement and a corresponding iterator are randomly selected. Unrolling an iterator involves randomly choosing an unroll factor between 2 and 16. If the iterator is already tiled then it is replaced by an unroll. If the iterator is already unrolled then a choice is randomly made to delete the unroll or to alter the unroll factor.

3.3.7 Population Initialisation

A fixed-size population can be initialised from a text file that describes the time-mapping representations for each member of the population, or by using the *petit* generated time mappings to initialise a single population member with random initialisation applied to the remaining population members. ICE saves the time-mappings of the current population in a unique file whenever a new legal individual is created. Expert programmers can use population initialisation from a file to control the seeding of the initial population, or to restart ICE from a particular point in its reproduction with a different population.

3.4 Evaluation Functions

ICE uses measured execution time as the evaluation function. Evaluation functions normally return a *fitness* value which GAs seek to maximise. In ICE we seek to minimise measured execution time. Thus, individuals having low measured execution time are given high fitness values.

4 Method

ICE currently optimises the application of data-array padding, loop-unrolling, loop-tiling, loop-distribution, loop-fusion and statement reordering transformations whilst loop-permutation[7] and iteration-partitioning decision are determined during a preprocessing step which invokes *petit*[8] [1]. ICE seeds one member of the population of 20 with encodings of restructuring transformations generated by *petit* whilst the remaining 19 members are randomly initialised. Note that illegal time-mapping encodings which do not preserve program semantics are allowed into the population. This removes the obligations that i) random initialisation must generate legal solutions, and ii), reproduction operators must always generate legal solutions. Further, the path in the solution space from a particular encoding to the optimal solution may be considerably shortened or only be reachable by intermediary illegal solutions.

For **legal** time-mapping encodings, the evaluation function actually compiles and then runs and measures the time taken to execute the program on the target CPU. If profile directed reproduction has been selected then the exectable is recompiled and

[7] Loop-permutation is equivalent to any combination of loop-interchange and loop-reversal .

[8] A prototype parallelising compiler.

reexecuted as suggested in section 3.3. **Illegal** time-mapping encodings are automatically given an execution time equal to the maximum possible value of a double without running any code. Linearly normalised fitness then determines the selection probability of individuals ranked using the evaluation function with selection probabilities ranging from 0.00555 (3s.f.) to 0.111 (3s.f.) in increments of 0.00555 (3s.f.). Thus, legal time-mapping encodings will always have a greater probability of selection for reproduction than illegal encodings. A steady-state reproduction policy uses the reproduction operators with equal and fixed selection probability. A new individual can then only be inserted into the population if its group structure and the iterator scheduling of the individual's encoding is unique within the current population. This aims to preserve population diversity by preventing the insertion of duplicate encodings into the population. However, it is still possible for equivalent transformations to have different solution encodings. ICE preserves elitism[9] by selecting individuals for deletion from the lower ranked half of the population. The elitist policy ensures that ICE will always produce code that is at least as good as that used to seed the intial population.

ADI, TOMCATV and SHALLOW (full results to appear in the final paper) compiler benchmarks have been used to evaluate the performance of ICE in compiling separately for TM1300, PIII and UltraSPARC-IIi. ADI is a direct method for the solution of PDEs. SHALLOW is a 2D 5 point finite-difference method for the solution of the Shallow Water equations which model a simplified fluid dynamics problem that is representative of more complex atmosperic models used in weather forecasting. TOMCATV is a SPEC benchmark. Figures 1, 2, 3, 4 and 5, 6 respectively plot the execution time in seconds returned by the evaluation function for every legal solution generated against the number of reproductions for ADI 1024×1024, SHALLOW 512×512 and TOMCATV $N = 257$ benchmarks on a TM1300 target. Figures 7 and 8 plot similar information for the ADI becnhmark on an Intel PIII target. Complete results for all benchmarks and all architectures will be presented at the conference.

5 Results

The preliminary results of ICE on ADI, TOMCATV and SHALLOW benchmarks are promising. Figure 9 tabulates the best execution time produced by ICE as a percentage of the original execution time for the benchmarks on the different CPU architectures. For ADI using normal operators on the TM1300, ICE produces a transformed program whose execution time is 35.6% of the original version. A preliminary inspection of the results suggests that the performance gains have been made through the elimination of cache misses and conflicts[10]. From the results, it can be seen that profile directed operators produce better solutions within fewer reproductions than normal (random) reproduction operators. However, the profiled operators do not always produce solutions that are better than the best produced using normal reproduction operators during 10000 reproductions. Further experimental work is required to complete the set of results and to evaluate if the incorporation of profiling information is causing premature convergence within the population.

[9] Elitism ensures that the best solution in a population is never deleted.

[10] The final paper will include a full breakdown on the reduction of cache misses by type.

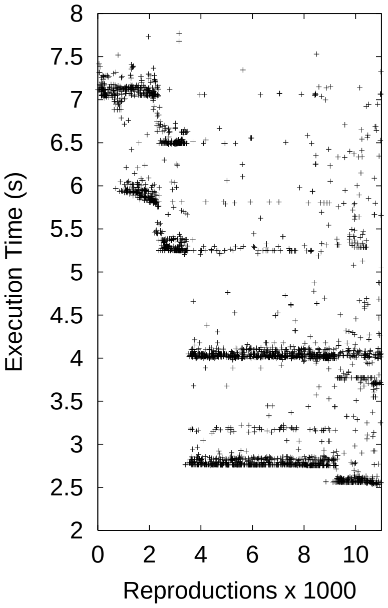

Figure 1: ADI Performance (normal operators TM1300).

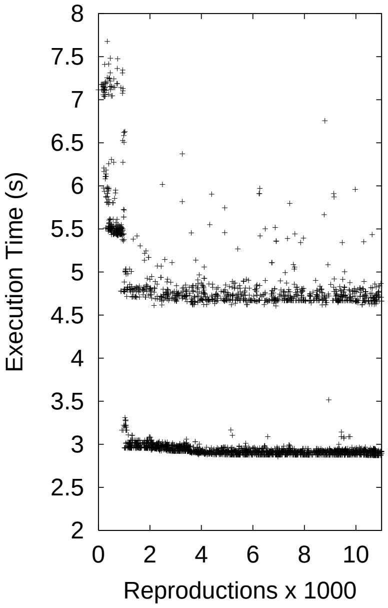

Figure 2: ADI Performance (profiled directed operators TM1300).

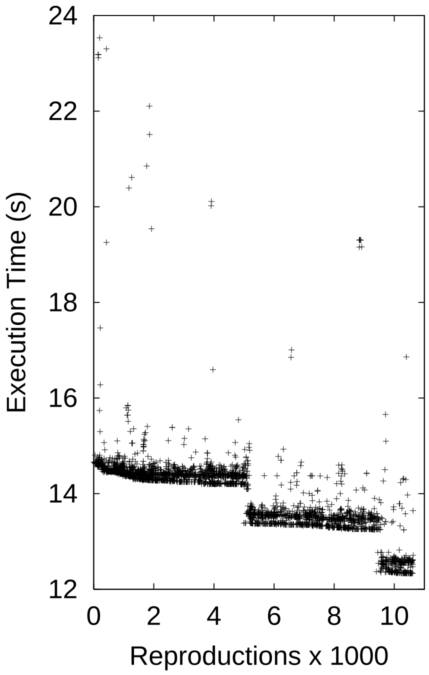

Figure 3: SHALLOW Performance (normal operators TM1300).

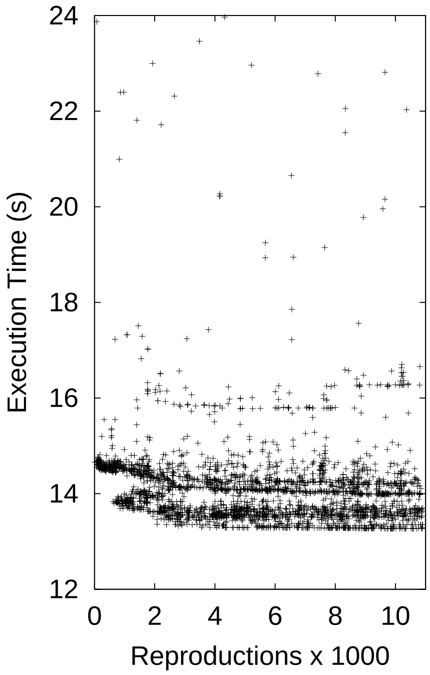

Figure 4: SHALLOW Performance (profile directed operators TM1300).

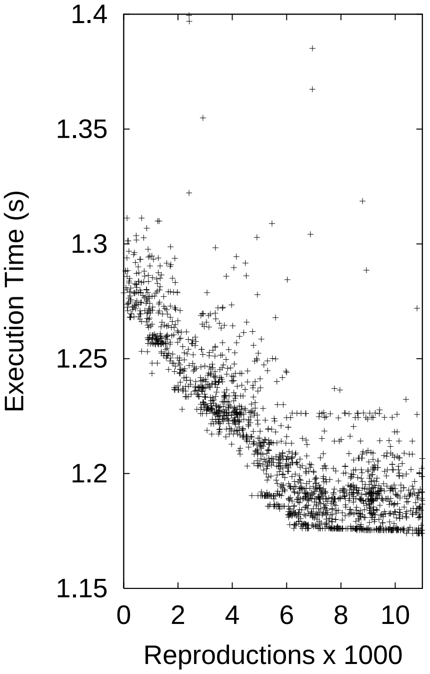

Figure 5: TOMCATV Performance (normal operators TM1300).

Figure 6: TOMCATV Performance (profile directed operators TM1300).

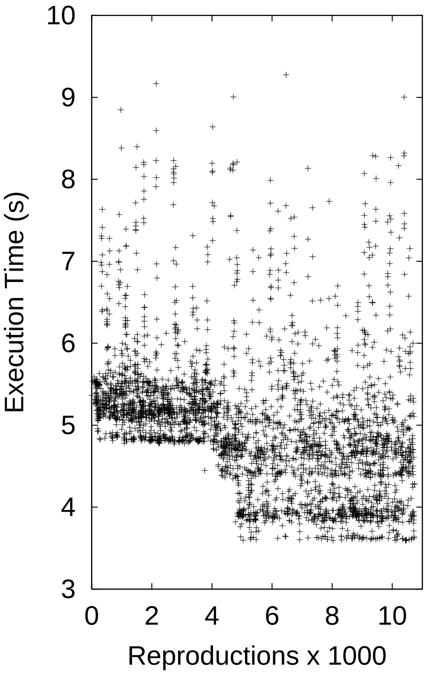

Figure 7: ADI Performance (normal operators PIII).

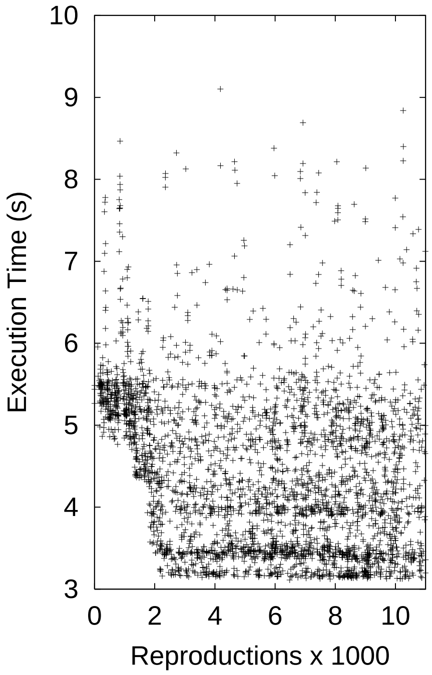

Figure 8: ADI Performance (profile directed operators PIII).

Benchmark	TM1300	PIII	UltraSPARC-II
ADI (normal)	35.6%	66.9%	
ADI (profiled)	40.0%	56.7%	
TOMCATV (normal)	91.8%		
TOMCATV (profiled)			
SHALLOW (normal)	84.1%		
SHALLOW (profiled)	90.1%		

Figure 9: Execution time as a % of Original.

6 Related Work

The iterative compilation approach of [3] target a small set of paramaterisable loop and data transformations (tiling unrolling and padding) for the optimisation of a a simple matrix multiplication application. Their iterative search for an optimal transformation focusses on sampling the cartesian transformation space at regular intervals, and then applying the appropriate transformations, executing the binaries and measuring the execution time. The points lying between the current global best and the average are added to an ordered queue. Iteratively, the points are dequeued and points within the neighbouring region are visited, again using a regular interval. This process continues until a specific number of points are visited. The authors point out that this approach may not be appropriate when transformations are not parameterisable.

[4] describes a method for the feedback directed selection of compiler switches. This applies fractional factorial design [5] in selection and characterization of compiler switches. A relatively small number of experimental runs help to determine the effects of various compiler optimisations and the positive and negative interactions between compiler switches. The key contribution of their work is in helping compiler developers to determine optimal compiler switches in a fraction of the time that is required by a brute force method.

7 Conclusions & Future Work

Iterative compilation appears (initially at least) to be a promising area of research. The preliminary ADI results on the TM1300 and PIII architectures suggest ICE can be retargetted to different CPU architectures and produce performance improvements. Full experimental data for all benchmarks and architectures is required to investigate the bold claims of the retarggetable nature of the ICE infrastructure, as is not yet clear if ICE is producing results of similar quality on different CPU architectures. On the ADI benchmark on the TM1300 and PIII the best performance improvements are made on the TM1300 architecture although this may be a feature of the TM1300 VLIW architecture being able to exploit more ILP than the PIII.

Future work should investigate the *hybridisation* of reproduction operators to exploit data-dependence and control-flow information in order to increase the rate of producing legal programs. Work is planned to study the effects of such hybridisation on i), convergence time, and ii), solution quality. An experimental study is also planned to investigate the effect of not allowing illegal solution encodings to enter the population. Ongoing work plans to

- investigate the performance of multiple ICE runs on the same benchmarks.
- integrate cache miss and other performance counter information into profile directed reproduction operators.
- investigate optimising for minimum executable size and, or, execution time.
- investigate the use of other iterative optimisation techniques such as simulated annealing in compilation.

8 References

[1] The Omega Project, Frameworks and Algorithms for the Analysis and Transformation of Scientific Programs,
http://www.cs.umd.edu/projects/omega/
[2] W.A. Kelly (1996), Optimization within a Unified Transformation Framework, PhD. Thesis, University of Maryland, USA.
[3] F. Bodin, T. Kisuki, P.M.W. Knijnenburb, M.F.P. O'Boyle, E. Rohou., Iterative Compilation in a Non-Linear Optimization Space *Workshop on Profile and Feedback Directed Compilation*, in conjunction with PACT98, Paris, France, 1998
[4] K. Chow, Y. Wu, Feedback-Directed Selection and Characterization of Compiler Optimizations, *2nd ACM Workshop on Feedback-Directed Optimization (FDO)*, Haifa, Israel, November 15, 1999
[5] G.E.P. Box, Statistics for Experimenters, An Introduction to Design, Data Analysis and Model Building, John Wiley and Sons, 1978
[6] A.P. Nisbet*,(1998) GAPS: Iterative Feedback Directed Parallelisation Using Genetic Algorithms, (poster) *Supercomputing 98*, November, Orlando Florida.
[7] A.P. Nisbet*,(1998) Iterative Feedback Directed Parallelisation Using Genetic Algorithms, *Workshop on Profile and Feedback-Directed Compilation*, in conjunction with *PACT98: International Conference on Parallel Architectures and Compilation Techniques*.
[8] A.P. Nisbet* (1998) GAPS: A Compiler Framework for Genetic Algorithm (GA) Optimised Parallelisation, (extended poster) *Proceedings of High-Performance Computing and Networking Europe*, [9] A.P. Nisbet* (1998), GAPS: Genetic Algorithm Optimised Parallelisation, *7th Workshop on Compiler for Parallel Computers*, pages 172-183, Linkoeping, Sweden.
[10] A.P. Nisbet, GAPS: A Genetic Algorithm Optimised Compiler Framework for HPC, (poster) European Parallel Tools Meeting, October, 1996, ONERA, Chatillon, France.
[11] PAPI Project Home Page http://icl.cs.utk.edu/projects/papi/.