Machine Learning in Compilers

Hugh Leather

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2010

Abstract

Tuning a compiler so that it produces optimised code is a difficult task because modern processors are complicated; they have a large number of components operating in parallel and each is sensitive to the behaviour of the others. Building analytical models on which optimisation heuristics can be based has become harder as processor complexity increased and this trend is bound to continue as the world moves towards further heterogeneous parallelism. Compiler writers need to spend months to get a heuristic right for any particular architecture and these days compilers often support a wide range of disparate devices. Whenever a new processor comes out, even if derived from a previous one, the compiler's heuristics will need to be retuned for it. This is, typically, too much effort and so, in fact, most compilers are out of date.

Machine learning has been shown to help; by running example programs, compiled in different ways, and observing how those ways effect program run-time, automatic machine learning tools can predict good settings with which to compile new, as yet unseen programs. The field is nascent, but has demonstrated significant results already and promises a day when compilers will be tuned for new hardware without the need for months of compiler experts' time. Many hurdles still remain, however, and while experts no longer have to worry about the details of heuristic parameters, they must spend their time on the details of the machine learning process instead to get the full benefits of the approach.

This thesis aims to remove some of the aspects of machine learning based compilers for which human experts are still required, paving the way for a completely automatic, retuning compiler.

First, we tackle the most conspicuous area of human involvement; feature generation. In all previous machine learning works for compilers, the features, which describe the important aspects of each example to the machine learning tools, must be constructed by an expert. Should that expert choose features poorly, they will miss crucial information without which the machine learning algorithm can never excel. We show that not only can we automatically derive good features, but that these features out perform those of human experts. We demonstrate our approach on loop unrolling, and find we do better than previous work, obtaining XXX% of the available performance, more than the XXX% of previous state of the art.

Next, we demonstrate a new method to efficiently capture the raw data needed for machine learning tasks. The iterative compilation on which machine learning in compilers depends is typically time consuming, often requiring months of compute time. The underlying processes are also noisy, so that most prior works fall into two categories; those which attempt to gather clean data by executing a large number of times and those which ignore the statistical validity of their data to keep experiment times feasible. Our approach, on the other hand guarantees clean data while adapting to the experiment at hand, needing an order of magnitude less work that prior techniques.

Acknowledgements

I must profusely thank my adviser, Michael O'Boyle. He has been an invaluable source or advice and guidance throughout my studies.

I must also, of course, equally thank my wife, Janne, who has put up with me during what must have felt like long years.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Hugh Leather*)

Table of Contents

Chapter 1

Introduction

1.1 Machine Learning in Compilers

1.2 The Problem

1.3 Contributions

1.4 Structure

1.5 Summary

Chapter 2

Related Work

The first work to automate compiler optimization was in the area of iterative or feedback-directed compilation which searches the program optimization space. Many smart search heuristics have been developed Cooper et al. (2005); **?**); Pan and Eigenmann (2006); Triantafyllis et al. (2003). Cooper et al. Cooper et al. (1999, 2002, 2005) explore the optimization space using hill climbing and genetic algorithms. Other researchers have used analytical Yotov et al. (2003) or empirical models to explore the optimization space Agakov et al. (2006). Agakov et al. Agakov et al. (2006) built a model offline that is used to guide search. Haneda et al. Haneda et al. (2005) make use of statistical inference to select good optimizations. In contrast to our work, these approaches do not make use of predictive modelling and require a recompilation of the program for each step in searching the optimization space. On the other hand, our technique focuses on avoiding this search and recompilation by directly predicting the correct set of compiler settings with no profile runs.

More recently, there have been a number of papers aimed at using machine learning to tune individual optimization heuristics. One of the first researchers to incorporate machine learning into compiler for optimization were McGovern and Moss Mcgovern and Moss (1998) who used reinforcement learning for scheduling of straight-line code. Cavazos *et al.* Cavazos and Moss (2004) extend this idea by learning whether or not to apply instruction scheduling. Using the induced heuristic, they were able to reduce scheduling effort but were unable to reduce the total execution time for the SPECjvm8 benchmark suite.

Stephenson *et al.* Stephenson et al. (2003) is the work most similar to ours. They used genetic programming (GP) to tune heuristic priority functions for three compiler optimizations. In contrast we use GP to search over *feature* space rather than the

model space. They achieved significant improvements for hyperblock selection and data prefetching within the Trimaran's IMPACT compiler.

In the area of loop unrolling Monsifrot et al. Monsifrot et al. (2002) use a classifier based on decision tree learning to determine which loops to unroll. They looked at the performance of compiling Fortran programs from the SPEC benchmark suite using g77 for two different architectures, an UltraSPARC and an IA64. They showed an improvement over the hand-tuned heuristic of 3% and 2.7% over g77's unrolling strategy on the IA64 and UltraSPARC, respectively. Stephenson and Amarasinghe Stephenson and Amarasinghe (2005) went beyond Monsifort prediciting the actual unroll factor within the Open Research Compiler. Using support vector machines they show an average 5% improvement over the default heuristic..

In contrast to our work, these techniques used hand-selected features, the quality of which was not explicitly evaluated. To the best of our knowledge, this paper is the first to search and automatically generate features for machine learning.

Iterative compilation has been explored for some timeAlmagor et al. (2004); Bodin et al. (1998); Kulkarni et al. (2003) and efforts have been made to reduce the cost of itAgakov et al. (2006), albeit by reducing the number of versions that need to be searched to find the best. Machine learning approaches to compilers attempt to automatically tune heuristics and require large sets of data, captured through iterative compilationMoss et al. (1998); Monsifrot et al. (2002); Stephenson et al. (2003). These works, to the best of our knowledge, all use only fixed sized sampling plans.

Efforts to promote statistical rigor in execution time measurements have been madeGeorges et al. (2007); Blackburn et al. (2006). In these, a program version is run multiple times until either an estimate of inaccuracy is sufficiently small or some maximum number is reached. Each point in the optimization space is executed until we have a good estimate of its mean so the data is statistically valid. However, this effort does not take into account the relative merits of each point. A point that is clearly bad will be refined just as much as the most promising point in the space. Since their technique considers each point in isolation it can perform worse than an optimally chosen constant sized approach.

In Mytkowicz et al. (2009), the difficulties of avoiding measurement bias are described. The authors demonstrate that, even with a simulator, apparently innocuous modifications (such as sizes of irrelevant environment variables) can affect the performance of a program. They suggest that random changes must be made to the set up state so that multiple measurements are required. Even in simulators, previously a

haven of noise free data, correct measurements must handle noise.

Sequential analysis, however, has been used to reduce the cost of sampling in contexts from industrial processesWald (1947) to medical trialsWhitehead (1992). Our work is most similar to Maron and Moore (1994, 1997) wherein machine learning models are raced to find the best. Their work, however, relying on Hoeffding's inequalityHoeffding (1963), requires that the random variables under consideration are all bounded - which is not the case for run times. Moreover, their work only concentrates on removing poor performers, it does not consider the situation where some of the random variables are equivalent for practical purposes. To the best of our knowledge, our paper is the first to bring sequential sampling to iterative compilation.

Something here

2.1 Iterative Compilation

2.1.1 Automatic Conversion of Programs from Serial to Parallel using Genetic Programming – The Paragen System

In an early paper Walsh and Ryan (1995), Walsh et al. describe an interesting system to automatically parallelise Occam programs. Their approach uses Koza Koza (1990a) style Genetic Programming (GP) to build expressions representing a new version of the program to be parallelised. Non-terminals available to evolve with GP declare that children should be run either in parallel or in serial; terminals are statements from the original program.

Two simple GP non-terminals, **Do** and **DoAcross**, are provided to execute loops serially and in parallel, respectively. It is not at all clear from the paper how these nodes interact with GP system. Other nodes have no knowledge of the content of the program; these must at least know their iteration bounds and might well expect to be paired with their original program counterparts.

Flow dependencies across loop iterations cause a problem for Paragen. Each iteration is thus synchronised, presumably, significantly to the detriment of parallelisation.

Unlike previous approaches, Paragen performs no analysis on the correctness of the resulting programs. Statements from the original program may be replicated many times in individuals - indeed, were it not for a special rule intended to close a GP loophole (wherein the genetic breeding process makes use of some, unintended artifact of the fitness function), statements could also be entirely removed from the program.

Rather than providing guarantees of correctness, evidence is provided in the form of a number of test cases under which candidates must perform identically to the original program.

The fitness function in Paragen is multi-objective, one part being the highest correctness in terms of ‘hits’ and the other being highest parallelism. The Pygmy Algorithm is used for selecting individuals although no evidence is provided to suggest that this is a good approach.

This paper is interesting due to its unusual nature. However, no results are given, making analysis difficult. The lack of correctness guarantee makes this system unlikely to gain acceptance in the engineering community.

Since the virtual machine on which they execute is somewhat idealistic, it is far from clear that this technique can produce better results than more typical, analysis

based approaches to exploiting IPC.

2.1.2 Iterative Compilation in a Non-Linear Optimisation Space

Bodin et al.Bodin et al. (1998) iteratively search for optimal parameters to loop unrolling, tiling and padding for matrix multiplication to minimise execution time.

The paper is not interested in particular search algorithms, rather in the applicability of search techniques to the problem. Is this not the same as saying that they are interested only showing that the search space is not flat? Or is saying the search space isn't flat and we don't want to defend our search algorithm?

The authors describe that search parameters are different for different environments and lengths of search. Learning such paramters would be an interesting next step. Indeed, learning a search algorithm would be an challenging avenue.

2.1.3 Genetic algorithms and instruction scheduling

Beaty Beaty (1991) searches for optimal instruction schedules with Genetic Algorithms. In his paper instructions are ordered in a list, the scheduler then proceeds in a manner similar to topological sort, choosing ready instructions from the Data Dependency DAG (DDD). Whenever a choice is available, the instruction earliest in the ordered list is selected. The permuted lists are evolved by GA with the fitness function being the length, in cycles, of the schedule.

Beaty offers no results in this paper, making it a difficult one to analyse. However, this approach looks to be an interesting precursor for later papers.

2.1.4 Optimizing for Reduced Code Space using Genetic Algorithms

Cooper et al. Cooper et al. (1999) use Genetic Algorithms to choose optimisation sequences that most reduce code size. Their approach evolves lists of twelve optimisations chosen from a set of ten available in their compiler.

Comparing their code size against their compiler's default settings gives very favourable results. However, the default settings are intended to optimise time, not space, and so the comparison is not entirely fair. They also manually construct a sequence of optimisations that should be space conscious and compare their technique against that. Again the comparison is favourable for sequences learned targeting the individual program.

The GAs learn how to best optimise individual programs and the authors note that there is significant difference in the sequences selected for each benchmark. For each new program to be compiled the process must be repeated.

The authors validate that their GA is superior to the Monte Carlo technique. In this paper they found that to reach similar code sizes their GA took roughly half the number of program evaluations than did the Monte Carlo version.

2.2 Machine Learnt Compilation

2.2.1 Automatic Tuning of Inlining Heuristics

In Cavazos and OBoyle (2005) Cavzos et al. learn improved values for five integer constants inside of two of JikesRVM's inlining predicates. The inlining predicates are composed of a number of thresholds over characteristics of a method's bytecode. The characteristics are the callee size, inline depth and caller size. Tests against these thresholds lead to a fixed set of *if-then* rules determining the inlining policy of the JVM.

The thresholds are learned via a Genetic Algorithm (GA) to minimise running time, compilation time or a balance of both, with performance being taken as the geometric mean over the their chosen benchmarks.

The authors generate impressive speedups on their chosen benchmarks against the default constants in the JikesRVM. They also apply their technique to different architectures, demonstrating that they can learn appropriate values in different environments.

It is not clear how the original JikesRVM constants were derived. It is possible that a some effort may have gone in to choosing the most suitable values over some benchmarks. Were this the case, then the improvements seen in this paper may be a result of different environments and benchmarks.

No feeling for the computational effort required to learn these constants is given.

2.2.2 Meta Optimization: Improving Compiler Heuristics with Machine Learning

Stephenson et al. **?** demonstrate that machine learning can be successfully to optimise priority or cost functions inside compilers. They present a general framework allowing

easy learning of such functions through a standard, Koza Koza (1990a) style Genetic Programming (GP) system.

The learning phase in this system takes days of compute time.

In common with other papers, this approach requires the programmer to careful select features for the system to learn over.

The selection of GP terminals will have a significant effect on the outcome. Generally only summary information is provided. To some extent the power of GP seems to be under utilised since nearly all machine learning methods should be able to replace GP in this fashion. Providing not only summaries, but very fine grained functions and terminals might permit GP to succeed where other search techniques fail due to the explosive search space?

The results achieved in this paper are criticised in Agakov et al. (2006), which states that the tuned over optimisations are not well implemented and that the register allocation improvement is only 2%. However, as in Cavazos and OBoyle (2005) any increase is found automatically.

2.2.3 Using Machine Learning to Focus Iterative Optimization

Agakov et al., in Agakov et al. (2006), use machine learning to constrain search spaces for iterative compilation. They select a number of features to describe their benchmarks. They then model the effects of different optimisations sequences over programs with given features. This model is then consulted, when a new program needs to be compiled, to generate a set of optimisation sequences that may prove most profitable for subsequent iterative compilation.

In the paper, the authors find which optimisation sequences of length five from 14 (and later, 20 from 82) in SUIF lead to the best performance. They learn the correlation between thirty or so hand selected features and the performance of the optimisation sequences. The authors compare learning this correlation with two different models, the independent identically distributed model (IID) and Markov chains. The models are used to learn those sequences which lead to within 95% of the best achievable performance.

The choice of the 95th percentile leads to the question of what would happen if that value was changed. One may envisage in a stable search space that learning the very best sequence would be possible. Conversely there might be some threshold beyond which the search space restrictions are too tight to be useful. It would be interesting to

see the results as the percentile increases toward 100%.

The 33 features they learn over are reduced by PCA to only five dimensions. The most suitable training example to match a newly encountered program is determined by the nearest neighbour algorithm in this five dimensional space. On the oracles, this leads to only 75–80% of the performance.

These exciting results lead one to wonder if different machine learning algorithms would do better (different genetic operators perhaps); would different models provide better results (since model expense is only paid once); each run on a new program could feed back into the models; are there automatic ways of determining features?

2.2.4 Learning to Schedule Straight-Line Code

In Moss et al. (1998), Moss et al. apply machine learning to instruction scheduling heuristics. A priority function over instructions is learned and used in a greedy selection algorithm to choose the next instruction to schedule.

Inputs to the machine learning algorithm are hand selected. They are: number of instructions scheduled so far modulo two; instruction type; height in the DDD; delay until actual execution and whether the instruction can dual issue with the last instruction. Several different machine learning techniques are applied: decision trees, table lookup, ELF and neural network.

The results for each learning algorithm were very similar but significantly worse than the hand coded algorithm existing in the compiler. The authors state that increasing the amount of information available to learn from did little to increase performance in previous trials. These two facts suggest that either the existing algorithm was not priority based or that it had a great deal more information available to it. Assuming the former, one wonders if some strategy may be evolved that considers more than just the instruction in isolation, perhaps examining some neighbourhood of the instruction may increase performance.

2.2.5 A Machine Learning Approach to Automatic Production of Compiler Heuristics

Monsifrot et al.Monsifrot et al. (2002) use decision trees with boosting to learn when to unroll loops. The authors classify loops as worthy of unrolling according to features descibing the number of statements, arithmetic operations, if statements, array accesses and iterations in the loop. The classification is somewhat noisy, around 85 per cent.

It is difficult to determine whether the accuracy of 85% for the classification is good or bad. It would interesting to discover if this is due to insufficient examples or if additional features would disambiguate the classification. Since the authors indicate that some loop classes contain both positive and negative examples, the latter is suggested.

2.2.6 Hybrid Optimizations: Which Optimization Algorithm to Use?

Cavazos et al.Cavazos et al. (2006) use rule induction to choose between register allocation algorithms, Linear Scan (LS) and Graph Colouring (GC), in the Jikes RVM. In the context of a Just-In-Time compiler expensive optimisations, GC for example, must be applied only where the benefit justifies the cost. The authors thus attempt to learn the appropriate cost - benefit relation.

The authors select a number summaries over the control flow graph of each method as inputs to their learning algorithm. Rule induction automatically ignores irrelevant features. Methods are labelled according to which optimisation is better. Methods are labelled GC if the spill count is greater that that for LS by some threshold or if the cost is only worse than for LS by some threshold. This labelling is what is to be learned.

It is not clear how the labelling indicates a choice that would be useful in programs not from the benchmarks. We might express the runtime associated with each optimisation by a function, for each block, $f(t) = at + c$, where c is the compilation cost and a is the runtime cost per unit of time. We expect $a_{LS} > a_{GC}$ and $c_{GC} > c_{LS}$. The real choice will thus be heavily dependent on t, the time the program runs for. This is not information made available to the compiler. If their approach then reduces the real execution performance, is this due to a misestimate of the runtime (indirectly through the choice of thresholds) or due to any misclassifications? It would have been good to see how well an oracle would have performed.

The authors present one of the rules learned for particular threshold settings. Of 142 methods that should have been categorised as GC, only 64 were, 45%. However, of those 1844 that should have been LS, only 1815 were so chosen, 98%. No analysis is offered for the poor categorisation of GC methods.

2.2.7 Predicting Unroll Factors Using Supervised Classification

Stephenson and AmarasingheStephenson and Amarasinghe (2005) employ two different machine learning techniques to loop unrolling. They regard the loop unrolling problem as one of multi-class classification by limiting unroll factors to fall between

one and eight inclusive. So phrased, the use a Nearest Neighbour classifier and a Support Vector Machine to disinguish amongst cases. They provide 38 hand selected features to learn over.

The authors describe their attempts to reduce the feature space. They use Mutual Information Score (MIS) and a greedy algorithm to select a few of their features with the most important. The authors point out that fewer features are easier to learn over and are less likely to overfit the training data. They do not, however, show the consequences in this case.

Noting that one of the most common objections to using machine learning in compilers is that effort is moved from tuning heuristics to tuning feature computations, the authors believe that the next wave of compiler improvements will include generic feature extractors. They do not see these as automated tools, rather as frameworks "much like compiler infrastructures provide generic data flow analysis packages."

They learn the optimal factor 65% of the time and the first or second best for 79% of the time. They get 5% improvement in runtime for SPEC 2000. These impressive results, they say, were acheived in a few sectonds of compute time, compared to many years of human tuning in the original compiler. Despite also spending two weeks instrumenting the compiler and one week collecting data the total time is far better than in hand tuned cases.

2.3 Customisable Processors

2.3.1 Automatic Instruction-Set Extensions

In Pozzi and Ienne (2006); Biswas et al. (2006b) Pozzi et al. present a number of problems for instruction-set extensions and some solutions for them. The goal is to find some subsets of a Data Flow Graph (DFG) that can be turned into instruction to add to a base processor.

They have some constraints, such as maximum number of inputs and outputs the instruction may have and node typs that can't be present in the DFG (e.g. loads and stores). Additionally, candidate sub-graphs must be convex, meaning that there must be no paths between two member nodes containing a non member node.

The authors develop a number of mathematical problem statements describing both the identification of suitable subgraphs and selection amongst candidates based on some cost benefit function supplied to them.

The problems they are present are generally NP-hard in the worst case. Tackling these, the authors typically use exhaustive search, with pruning, claiming that frequently, in practice, they find solutions in reasonable time. It should be noted from their results, however, that on more than one occasion, they fail to find any solution at all.

The authors sort their graphs topologically. Thereafter the complete set of cuts may be represented as binary tree with left edges meaning to exclude the node from the cut and right edges meaning to include. By choosing topological sort, they are able to prune their search since once output constraints or convexity are violated, adding more nodes later in in the sort (and hence 'higher' in the DFG) will not fix the violation. Similarly, inclusion of a forbidden node cannot be fixed by adding more nodes. A more complex pruning can be performed over the input constraints by noting that edges leading to DFG inputs or to forbidden nodes cannot be removed by adding more nodes.

This pruning still leaves an exponential search in the worst case.

The authors compare their algorithm which may be viewed as an exhaustive search against some state-of-the art alternatives and against a genetic algorithm (GA). They quite often out perform the state-of-the art by a long way (particularly as the problem sizes increase), however, the GA lags by only a very little. It is quite possible that as the problem sizes continue to grow, their exhaustive solution will become intractible while GA will continue to be feasible. Indeed, for some problems, their approach is unable to find a solution.

Further speedups are shown permitting memory operations to be included. As a result, much larger DFG subsets can be accomodated. Vectors are identified in the source which can be transferred into and out of the new instruction unit with DMA. Local memories are provided so that the newly tranformed program will further benefit from reduced cache pollution and register pressure, as well as allowing much larger cuts to be transformed into hardware.

The I/O constraints imposed on new instructions are relaxed by pipelining. The idea is to ensure that in each stage of the pipeline the I/O constraints are met. Additional pipeline registers are inserted between edges in the DFG for the new instruction until the I/O constraints are met. The task is to choose these registers so as to minimise the number of pipeline stages (and with a tie to minimise the number of registers).

Their solution searches exhaustively the feasible register patterns to find the best one. They state that in pratical cases there are only a polynomial number of cases while in the worst case there are exponentially many.

Their results show that this approach is capable of greater speedups than the previous formulation when the I/O constraints are low. Of course at some point the I/O constraints exceed what is required to completely describe the new instruction in combinatorial logic. Once such a state has been reached, there is no performance benefit in pipelining.

2.3.2 Performance and energy benefits of instruction set extensions in an FPGA soft core

Biswas et al. Biswas et al. (2006a) consider adding *Ad-hoc Functional Units* to embedded processors, in particular the Xilinx MicroBlaze soft-core.

In this paper, the authors show that instruction set extension (ISE) can reduce power and energy requirements. This is a direct consequence of the speedup gained by the ISE and is directly proportional to it.

The authors also note that if communication overheads are non-zero then ISEs must replace larger subsections of code to outweigh this overhead, although this seems hardly surprising.

2.3.3 Datapath Synthesis

While approaches to instruction-set extension often assume that extensions will not share resources, Brisk and SarrafzadehBrisk and Sarrafzadeh (2006) attempt to reduce the area required by merging as many nodes of the extensions' DFGs as possible.

Custom instruction selection in the standard formulation is equivalent to the knapsack problem (once the candidates have been found) - maximise performance within some area constraint. However, this ignores the possibility of resource sharing.

The authors present the Minimum Area-Cost Acyclic Common Supergraph Problem. In the problem, a number of graphs are given and a supergraph with minimum area must be created such that each input graph is isomorphic to some portion of the supergraph. Intuitively, this graph will be a merging of input instructions. It remains to efficiently construct the supergraph and to deal with the bookkeeping for the instructions, now embedded.

Solving this problem is NP-hard. The authors note, however that the number of individual paths in a DAGs is typically no more than 3—V—. (The actual number of paths can be found in polynomial time with topological sort). Therefore they propose pairwise examination of paths in two DAGs and merging the greatest common sub

sequences. This is continued repeatedly until no further matches are possible and serves as an adequate heuristic.

Now, to manage the merging of nodes, multiplexers are inserted. When an instruction is to be executed, the multiplexers are set to open up the correct isomorphic sub-graph. The instruction then proceeds as before suffering only the cost of the multiplexer delay.

Insertion of multiplexers is straightforward for noncommutative operators. For the others, choices exist and poorly made choices will lead to deeper multiplexors and increased delay. The authors solve this balancing problem by reducing to an instance of the Maximum-cost Induced Bipartite Subgraph Problem, which is NP-hard.

The results show excellent reductions in area for minor performance reduction.

The non shared approach allows efficient pipelines. Here, however, if multiple instructions are issued to the combined graph there may well be overlap with pipeline stages overbooked. While this situation may be rare (one might expect these instructions to be executed in tight clusters), the authors do not explain how to overcome it.

2.3.4 Instruction Matching and Modeling

Parameswaran et al.Parameswaran et al. (2006) consider the situation where some number of instruction extensions have been placed in a library. These instructions are combinatorial in nature. They wish to find functionally equivalent DFGs in the input program so that they can replace them with calls to the more efficient library instructions.

A first step in this problem is to be able to tell if two combinatorial circuits are functionally equivalent. Binary Decision Diagrams (BDD) are the solution of choice for the authors. BDDs are generally applicable but can consume vast amounts of memory.

The design flow takes profiled C/C++ and dissects it into a number of segments, which are then converted into Verilog and compiled into combinatorial logic. Each is then compared to each library instruction via BDD to check equivalence (trivial filtering is performed to reduce the number of checks).

No mention is made of what will happen if the library instruction is equivalent to some subset of the segement. Since presumably enumerating all potention subsegments will be prohibitive, it is not clear how much benefit this approach will yield. Moreover, as only combinatorial circuits may be compared for equivalence, it is not

certain that significant speedups will be acheived.

2.3.5 Challenges to Automatic Customization

Nigel TophamTopham (2006) joins Fisher et al. (2006) taking issue with the practicalities of automatic instruction set extension. Drawing on his background working with ARC and their customisable processors, he notes that any company attempting ISEs must, due to the complexity of the hardware design process, still have a skilled hardware team employed. Even if an ISE is automatically selected, 'taping-out' is such a difficult and error prone task that experienced humans simply must be involved.

Topham argues, convincingly, that the speedups provided by the state of the art automatic ISEs are meager by comparison to what a human would produce. Algorithms designed for sequential execution in software are very different from those optimal for silicon. Automatic optimisers, then, are restricted to an order of magnitude or so improvements since they cannot undertake the drastic redesigns any hardware engineer would develop first.

The state of the art speedups in automatic ISEs acheive rarely as much as one order of magnitude, often only around a forty to one hundred per cent improvement. Humans, he points out, would typically expect to gain hundreds or thousands of times the base performance. Since those self same people must be involved in the automatic ISE process, he questions whether they would not just decided to write the ISEs themselves.

Moreover, the author gives examples were software optimisations (which certainly would have been applied to the code before resorting to the expensive process of taping out custom hardware) obscure the program to such an extent that current automated instruction-set extenders are given little opportunity to find improvements.

In Fisher et al. (2006), Fisher et al. present a similarly dismal view of the future of customisable processors. They point to their experiences at Hewlett-Packard building Custom-Fit Processors. While technically successful, their project had little impact on the industry for both political reasons and due to the long development cycle for silicon which so poorly matches the speed of software development lifecycles.

2.3.6 Automated Processor Configuration and Instruction Extension

Goodwin et al.Goodwin et al. (2006) describe the Tensilica Xtensa architecture. The architecture is customisable with a number of parameters: choosing between single and

multi-issue; permitting a additiona load/store unit; tuning the length of the pipeline; register file sizes; inclusion of multipliers and specialised units.

Additionally, flexible length instruction-set extensions (FLIX) can be intermingled with the existing code. These new instructions can be VLIW, SIMD or fused. The company provides a closed tool, XPRES which for candidate architecture/tool-chain pairs, each with a different performance/area trade-off. This space is searched to find optimal solutions. Very little is explained about the search process.

Application code is written in simple C-like form with annotations to disambiguate pointer aliasing and other mpediments to discovering ILP. This recognises one of Tohham'sTopham (2006) objections, that code will be heavily optimised for a sequential machine before resorting to hardware synthesis, but requires developers to now port their code to a new format.

2.3.7 Coprocessor Generation from Executable Code

From CriticalBlue, Taylor and StewartTaylor and Stewart (2006) describe their Cascade toolchain. Cascade differs from others in that its input is machine code destined for ARM processor. Clients select some hot function and ask the toolchain to generate a coprocessor capable of accelerating it.

The coprocessors created by Cascade are VLIW machines, given sufficient functional units for the task at hand. They remain programmable, ensuring processor viability during the lifetime of the product. The coprocessors operate via DMA, informing the main processor with interrupts that the function is complete.

One clever feature of the architecture is that datapaths can be customised so that function units can communicate directly, as well as via register files. This optimisation can then be controlled by software so when appropriate instructions are found. Moreover, this approach can reduce the number of ports needed in the register file.

While the fact that their tool uses machine code for input may seem attractive for developers who there after may feel they don't need to think about it and to managers, eyeing legacy code they cannot afford to rewrite, this comes at some cost. After compilation, a good deal of the semantics of a program are lost. The article shows only trivial examples of attempts to recapture this information, leaving only a cryptic statement that they do do more in their tool.

Critical Blue believe that the nature of computationally intensive kernels and the additional processing time available to their system offset somewhat the loss of seman-

tics.

2.4 Machine Learning

2.4.1 The genetic programming paradigm: Genetically breeding populations of computer programs to solve problems

In Koza (1990b), John Koza presents the original Genetic Programming (GP) system. Breaking from the typical linear, fixed-length Genetic Algorithms (GA) he searches for Lisp-based S-expressions to solve problems.

In this work he presents a large number of problems and shows how GP solves them. For example, in the Santa Fe Ant Trail, an ant must directed along a path in a square field to eat as many pieces of food as possible in as few steps. He shows an expression learned to achieve this goal:

```
(IF-SENSOR (ADVANCE)
(PROGN (TURN-RIGHT)
(IF-SENSOR (ADVANCE) (TURN-LEFT))
(PROGN (TURN-LEFT)
(IF-SENSOR (ADVANCE)
(TURN-RIGHT))
(ADVANCE))))
```

The essential genetic operators are introduced over trees, mutation and crossover. A typical mutations selects some sub tree, removes it and replaces it with a new, random tree. Simple crossover randomly chooses two sub-trees in the parents and swaps them.

This seminal work formed the basis for a huge amount of further work over the subsequent few years.

2.4.2 Evolution and co-evolution of computer programs to control independent-acting agents

In Koza (1990a), Koza et al. introduce hierarchical co-evolution. In co-evolution, two populations are evolved. The fitness function for the first population is determined relative to the performance of the second and vice verse. A biological example would be the arms race between a species of plant and insects that might feed on them.

The absolute fitness of individuals is never calculated; only fitness relative to the opposing population. This methodology is suitable for situations where the perfect strategy to compare against is not known, such as in chess, where bootstrapping an initial population is difficult.

The authors do not describe how their algorithm is significantly different from previous uses of co-evolution in GAs. It is possible that they are only pointing out that it can be used in GP successfully, too. Additionally, there is no mention made of the meaning of 'hierarchical' in their context. Possibly it only relates to the hierarchical nature of GP trees.

The bulk of the paper, however redescribes Genetic Programming (GP) and its performacnce on a a set of standard machine learning benchmarks.

2.4.3 Grammatical Evolution: Evolving Programs for an Arbitrary Language

Ryan et al. Ryan et al. (1998) introduce a novel version of Genetic Programming (GP), called Grammatical Evolution (GE). In typical, Koza style, GP the genotype and phenotype are identical, usually lisp-like expressions. By contrast, in GE the genotype is a list of choices to make in the expansion of sentence from a Backus Naur Form grammar (BNF).

Separating genotype from phenotype allows GE to use any search technique suitable for variable length integer lists, widening the appropriate choices considerably over GP. Additionally, due to the phenotype being any grammatically correct string of symbols, the system can easily produce program fragments in any language, making fitness testing trivial by comparison to the frameworks needed for non-lisp GP.

However, the genotype in GE is a linearisation of grammatical expansion choices. As such it suffers from a number of setbacks. There is a ripple effect due to the fact that position in the genome is not understood by cross-over or mutation. This means that small changes to the front of a genotype can easily cause massive changes to the interpretation of the remainder. As such, it is difficult to explore neighbourhoods of individuals.

Additionally, when more genetic material is required during the expansion of a rule than is available in the individual the genotype is wrapped. This often causes infinite expansions leading to many individuals that cannot be mapped to phenotypes.

Finally, equal weighting to production probabilities forces the system to be highly

sensitive to the grammar used. Two different grammars, both recognising the same language can radically alter the probabilities of various strings being produced. The user has little recourse in this system to overcome this drawback.

Unfortunately, while the authors demonstrate their system over standard symbolic regression problems, they do not present quantitative results, making this approach difficult to compare to ordinary GP.

2.4.4 No Coercion and No Prohibition, A Position Independent Encoding Scheme for Evolutionary Algorithms—The Chorus System

In an attempt to solve some of the problems inherent in Grammatical Evolution (GE)Ryan et al. (1998), Ryan et al. propose a new formulation in Ryan et al. (2002), the Chorus System. Chorus differs from GE in the mapping of genotypes to phenotypes. In GE each choice in the genome is decoded modulo the current number of choices in the grammar expansion. In Chorus, on the other hand, all genes are modulo the total number of productions. An initially empty concentration table is maintained. Each time a choice is encountered in the grammar expansion, the table is consulted and the production with the highest positive concentration is selected and its concentration reduced. If no such production exists, the genome is read, each gene incrementing the concentration table until a suitable production can be found. Wrapping is not used, individuals with insufficient genetic material are heavily penalised.

While the new approach introduces a much higher percentage of introns than in GE, it ensures that the absolute position of genes is irrelevant, only occasionally, the relative positions.

In the paper, the authors develop a complete schema notation based on regular expressions for their system.

Results are shown comparing against the standard GP benchmarks. Chorus is out performed by GE and occasionally by GP. The authors, however, expect that the high degree of introns contributes to this poor performance and expect that to be a benefit in longer trials.

2.5 Feature Selection and Generation

Features Selection assumes that the problem description language contains a superset of the features needed to pick out the target hypothesis. The simplest approach is to enumerate all sets of features and determine which yields the best machine learning performance. Whilst always providing optimal results, this exhaustive search is exponential in the number of features and hence, typically, impractical for all but the most trivial problems.

Heurstics are therefore employed. Common hill-climbing searches are Forward Selection (FS) and Backward Elimination (BE). The former greedily adds features while the latter removes them. Genetic Algorithms (GA) are often used when local minima trap the simpler greedy algorithms.

Feature Generation enriches the hypothesis language with new, derived features. Inductive generalisation is often used to create new features, particularly Constructive Induction (CI) Bensusan and Kuscu (1996).

When the hypothesis language contains a large number of redunant features, machine learning will be slow. Here feature selection can be used to reduce the complexity presented to the machine learning algorithm. In cases where the hypothesis language is insufficiently rich, feature generation can extend it with more useful information. Hybrid approaches prevent generated features being themselves redundant.

2.5.1 Robust Feature Selection Algorithms

In Vafaie and DeJong (1993) Vafaie and De Jong consider how to choose features for learning image classifications. Feature selection is important since it has a dramatic impact on the effectiveness of machine learning algorithms, not only in their efficiency, but also in the quality of the final results.

The auithors search for binary strings representing which features to provide to classification algorithm, AQ15 from a choice of 100. To evaluate these length 100 binary strings the AQ15 algorithm is trained over the selected features with a training set of input data. Then trained AQ15 is then tested against a test set to determine its recognition performance. A feature set are compared by the recognition performance they yield.

The problem is difficult for a number of reasons. The authors have no domain knowledge to guide the correct choice of features. Additionally, the feature sets are highly interactive producing many local minima.

The paper compares GA and sensitivity analysis (Sequential Backward Selection, SBS, which greedily removes features while doing so improves fitness). GA performed well when there are interactions between features and local minima but was inefficient when there were few interactions and local minima. In all cases, however, GA produced excellent quality results.

2.5.2 Constructive Induction using Genetic Programming

Constructive Induction (CI) pairs an attribute management system with selective learning system. If it detects that the learning system's performance is too low it determines that composite attributes are needed. Good composites are then searched for. Standard CI does not consider removing attributes.

Benusan et al.Bensusan and Kuscu (1996) use Genetic Programming (GP) to learn two attributes to add in their CI system. Their example is learning the four-bit parity function (notoriously hard for selective learning systems). The allow their GP system one type of function node, XOR, and search for expressions that allow a feed forward neural network to learn the parity function.

The authors compare their CI system to performance only with selective learners. The CI system us 100% accurate while the selective learners are 0% accurate.

It should be noted, however, that GP is perfectly capable of learning 4-bit parity all by itself, particularly when provided only XOR. While the authors' approach is certainly interesting, it would surely benefit from being applied to a real world example.

2.5.3 A Hybrid Approach to Feature Selection and Generation Using an Evolutionary Algorithm

Ritthoff et al.Ritthoff et al. (2002) present a combined feature selector and generator.

Their starting point is to have some problem that wish to learn with a large number of features and a set of generators which can combine those features to create new, composite features. They also have some machine learning algorithm (in the paper they use a Support Vector Machine (SVN) but any will do) and example data.

Their approach is to build a modified variable length Genetic Algorithm. Individuals contain a list of n strings describing features. Each of these is either an initial feature or a composite of features created by some applications of the generators. Additionally, an n bit vector describes which features will are selected and so will be provided to the SVN to learn over.

The standard genetic mutator (bit flipping in the selection vector) and one-point variable length crossover are augmented with a feature generator mutator. This takes an individual and adds some number of features generated from existing, selected features. That is to say, it might choose features x and y from an individual and add feature x * y to it. Many features may be added in one step and they may combine newly added features as well to permit arbitrarily complex features to be contructed. Only selected features may be combined in this way.

Evaluation of individuals tests the classification performance of an SVN trained over the example set with the given selected features.

Their results on artificial problems show that their combined approach clearly out performs learning over insufficient feature sets. That is not, perhaps surprising.

They also show learning over a large time series (5000 points) for estimating coefficients in chromatography experiments. They show that their hybrid approach is very successful at helping the SVN to learn these coefficients for different time series.

Design of the hypothesis language and generators will still influence the performance of the system. However, this approach allows the easy incorporation of domain knowledge since hand crafted features and generators are simple to add.

2.5.4 Automatic Feature Extraction from Large Time Series

Mierswa Mierswa (2004) uses Genetic Programming (GP) to learn features over time series data, particularly audio sequences.

Three genres of operation are provided to the GP system. These are:

- Transformations - which map each value in the input stream to a new value
- Functions - which aggregate a stream in to a fixed length vector
- Windowing - which applies Functions to moving windows over a stream to create a new stream, with fewer elements

Expressions of these functions are combined to create features to be used in the C4.5 Decision Tree algorithm and an SVN.

Experiments classify audio streams according to user preference or genre (for example, pop versus techno). The features learned are not obvious but it is not clear how successful these are compared to either human feature generation attempts or to, say a feature set consisting of a thousand random samples. Consequently, while the paper

demonstrates that features may be learned over infinite time series we have no clear idea how well it performs.

Chapter 3

Fine Grained Extensible Compiler

3.1 Introduction

Today's compilers do not provide the functionality required by machine learning techniques. All the main compilers were begun long before machine learning was shown to be useful to their goals and this has left a set engineering problems that must be overcome by any researcher who wants to explore the potential of machine learning in compilers. The researcher needs to affect the compiler across a wide range of granularities; sometimes forcing it compile code differently and sometimes extracting information about the program as it is compiled.

For control purposes, compilers mostly go no further than providing a range of global settings, typically through command line options or environment variable. These, however, do not allow the researcher to change the compiler's response to individual functions, loops or other constructs. Unless the current experiment is at the whole program level (or at least at the compilation unit level) then these settings will not be sufficiently fine grained. Occasionally, more capabilities are made available through source code annotations and extensions such as pragmas in C. However, being forced to modify the source code for large benchmarks is likely to be error prone and difficult to automate.

If we review the typical phases of a machine learning in compilers experiment and pay attention to the support required from the compiler we shall see how awkward the current situation is.

In the first phase, the researcher will need to collect iterative compilation data. He will have chosen some heuristic to replace and that heuristic will be focused on some set of entities in the compilers internal representation of the source code - for

example, functions, loops or basic blocks. Now if the researcher is lucky, the heuristic, which answers the question, "for entity, X, what settings should optimisation, O, use?" will take the form of a function call in the compiler. Consider, for example, the loop unrolling heuristic in GCC. It is a function call looking like this:

```
int decideUnrollTimes(loop* lp) {
    int times = /*the heuristic*/;
    return times;
}
```

The original heuristic, then, simply returns the number of times each loop should be unrolled, possibly answering none. [1]

Our researcher, glad to see a nicely encapsulated heuristic, decides that at each point in the iterative compilation only one loop per function will be unrolled, the rest will remain unchanged. In this way, he hopes to reduce the interactions between loops which might affect data gathering and to make counting the number of cycles in the changed loop takes easier to measure. Of necessity, then he will have to print out a list of which loops are in which functions so that he can plan the iterative compilation. In time honoured fashion, he adds some logging code to the heuristic:

```
int decideUnrollTimes(loop* lp) {
    int times = /*the heuristic*/;
    if(shouldPrintLoops) {
        print("unrollable-loop=%s,fn=%s\n",lp,lp->fun);
    }
    return times;
}
```

Now that he has a list of all the loops in the benchmarks, he can write a small driver which will create a compilation strategy for each point in the iterative compilation process. The driver outputs a list for each compilation point which gives how many times to unroll each loop. Now, he has to force the compiler to accept these overrides:

[1] In reality the heuristic is somewhat spread across the several functions. A small amount of refactoring allows it to look something like the pseudo-code presented here. Moreover, in GCC, the heuristic returns not only the number of times to unroll a loop, but also which of several different flavours of unrolling to perform

```
int decideUnrollTimes(loop* lp) {
    int times;
    if(shouldOverride) times = /*search in override file*/;
    else times = /*the heuristic*/;
    if(shouldPrintLoops) {
        print("unrollable-loop=%s,fn=%s\n",lp,lp->fun);
    }
    return times;
}
```

The researcher also adds cycle counting to each function with an unrolled loop in it (a potentially non- trivial task) and runs all of his iterative compilations. The profiles gathered from each run have to be associated with the point in the iterative compilation they belong to and recorded in a database. This, undoubtedly, requires some significant coding and careful organisation. It is not, however, just inside the heuristic, so we will come back to it later.

After all the iterative compilation data is safely stashed away in a database, the next step is to get some machine learning tool to learn a model from that data which can predict for new, unseen loops what the best unroll factor should be. The researcher must have a list of features for every loop that the machine learning tool will base the model upon. He adds yet another block of code to the once clean heuristic:

```
int decideUnrollTimes(loop* lp) {
    int times;
    if(shouldOverride) ...
    else /*the heuristic*/
    if(shouldPrintLoops) ...
    if(shouldPrintFeatures) {
        print(features_for_loop(lp));
    }
    return times;
}
```

Of course these features need to be uploaded to the database for use by the machine learning tool.

Finally, armed with a predictive model, our researcher can embed it back into the compiler, enabling anyone to use the better heuristic he has created. He adds more to the heuristic function:

```
int decideUnrollTimes(loop* lp) {
    int times;
    if(shouldUseML) {
        f = features_for_loop(lp);
        times = apply_model(f);
    }
    else if(shouldOverride) ...
    else /*the heuristic*/
    if(shouldPrintLoops) ...
    if(shouldPrintFeatures) ...
    return times;
}
```

In reality, once properly fleshed out, the code would be huge and quite likely the original heuristic would be lost in the mass of other code, none of which is the compiler's primary concern. That alone is a major cause for concern, but coupled with the fact that much of this baggage may be slightly different for different experiments and is is quickly clear why none of these hacks are likely to make it into any production compiler. The researcher is left reimplementing his modifications every time the compiler is upgraded; a situation that is slow, error prone and deeply frustrating.

These issues (and indeed several other, non-machine learning problems) would all be solved if the compiler had been built to be extensible. Extensible software allows external users (in this case, our hapless researcher) to adapt the behaviour of predefined points in the original software, all without changing a single line of the code in the original. A good extensibility library will permit heuristics to be replaced, reused or modified in clean fashion from outside the compiler. In our loop unrolling example, the researcher would have been able to use one extension to find out which his benchmarks have, another to force different loops to be unrolled according to his iterative compilation strategies and yet others to print loop features and to install the new heuristic. With extension capabilities the compiler would change from being an opaque black box to a fully customisable research compiler without compromising code quality, readability and maintainability.

The remainder of this chapter describes *libPlugin*, a powerful, feature rich extensibility library. Applied to GCC the compiler becomes perfect for machine learning research with minimal changes to the compiler's code. *libPlugin*is, in fact, completely independent of GCC, able to make any application extensible with very little work. However, the presentation here will focus on using the library for machine learning purposes and will, in particular, demonstrate how simple the above loop unrolling ex-

ample becomes when supported by *libPlugin*.

3.2 *libPlugin*

*libPlugin*is an extensibility library for C which makes providing extension capabilities for applications easy. The library is application agnostic but was specifically constructed to make GCC extensible. With it, the compiler's source code remains clean.

One of the main goals of the project is that absolutely minimal changes should required to GCC to support extensibility. As will be seen, when a fixed heuristic is converted to an extensible one the differences are almost unnoticeable. Indeed, the changes to GCC amount to some ten lines of code in the main function and often only one additional line of code per heuristic. The plug-in system is extremely simple to use without compromising power and flexibility.

There are two primary objects types in *libPlugin*; plug-ins and extension points.

3.2.1 Plug-ins and Extension Points

Plug-ins are modules encapsulating a areas of functionality for the compiler. Plug-ins provide services to each other through extension points; plug-ins use services of other plug-ins by extending those points. Moreover, only plugins can provide extension-points so that even core application services are bundled together into plug-ins, even though they might always be present and not optional.

In GCC, for example, there is a core plug-in which allows loop unrolling factors to be overridden on a perloop basis. This plug-in offers an extension point called `gcc-rtl-unroll-and-peel-loops.override` [2]. If a developer wants to force certain unroll factors for particular loops, they can write another plugin which extends the point, `gcc-rtl-unroll-and-peel-loops.override`.

3.2.2 Plug-in File Format

For each plug-in there is an XML description file. This file tells *libPlugin*everything it needs to know about the plug-in from its dependencies to the extension-points it

[2]GCC unrolls loops at different stages in the compilation process. The first but simplest is performed at a high-level while the code is still in an AST form (called GIMPLE). The more powerful optimisation operates on loops which have been converted to a lower level form, the Register Transfer Level (RTL). That optimisation performs both unrolling and peeling at once. Thus, the extension-point mentioned above is for the RTL level unrolling optimisation. A similar plug-in enables overriding for the GIMPLE level loops.

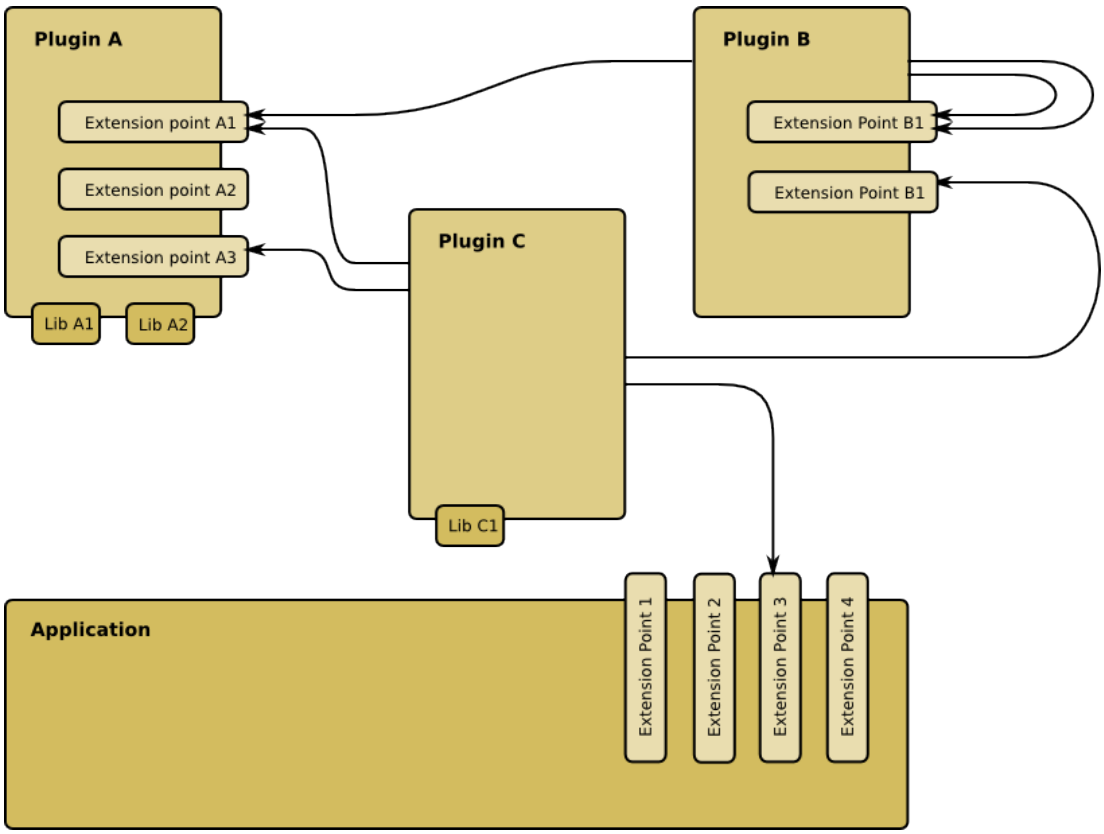


Figure 3.1: Plug-ins interact with each other, and the application. There can be lots of plug-ins, all working together. The system will manage plug-in dependencies and ensure that only necessary plug-ins and extension-points are created.
Extension-points can be extended multiple times and plug-ins can even extend their own extension-points. The application can also provide extension points for plug-ins to use by 'being a plug-in'; it puts it's own description file on the plug-in search path and appears to *libPlugin*to be just like any other plug-in.

provides and uses. Some plug-ins need no more than this description file, while others might need some C code to drive their functionality. In the latter case, the plug-in has some number of shared libraries in addition to the XML description.

The minimal plug-in specification is shown below. The plug-in will cause GCC to print `Hello, World!` to the standard output. This XML plug-in specification is either placed on a special plug-in search path or mentioned on the command line to GCC.

```
1 <?gcc version="4.3"?>
2 <plugin id="hello-world">
3     <extension point="message.start">Hello, World!</extension>
4 </plugin>
```

Each plug-in specification file must contain valid XML indicate what applications it should work with and contain a valid `<plugin/>` element. The simple file above is described, line-by-line, below.

In line 1, the plug-in declares that it applies to GCC with version at least 4.3 [3].

In line 2, the plug-in gives itself an identifier. Plug-ins use identifiers to declare that they depend upon each other and users also give these identifiers to load optional plug-ins from the command line. Plug-ins can, in fact, be anonymous, but it is considered good practice to always name them.

Line 3 says that this plug-in extends another plug-ins extension point. The identifier of that point is `message.start`. *libPlugin*will ensure that the plug-in providing that extension point is loaded and that only one such plug-in exists. In this case, the extension point is simple, and happens to be provided by the `message` plug-in. It will print the text contents of the extension to the standard output[4].

Line 4 ends the specification.

3.2.3 Plug-in Selections and Dependencies

Not all plug-ins are loaded by the system; the plug-in search path may include many plug-ins and several will be required only on some occasions. For example, there is a plug-in provided by default which will print a list of all passes each function goes through as GCC compiles it. That plug-in is useful for debugging purposes but would not be interesting during the majority of compilations. Conversely, if a developer has installed a plug-in which implements a useful optimisation, he may want a plug-in to be always present.

3.2.3.1 Lazy and Eager Plug-ins

In *libPlugin*there are several ways to indicate which plug-ins should be loaded during a compilation. The first is the distinction between *eager* and *lazy* plug-ins. Lazy plug-

[3] *libPlugin*is a library that can help to make any application extensible, not just GCC. Some plug-ins may work with more than application and can provide multiple directives to that effect - indeed some plug-ins work with all applications and can give a catch all directive of `<?plugin version="1.0"?>`. Each application will decide whether the version string is suitable for it and *libPlugin*will ignore any unsuitable plug-ins. This simple multi-application will, in the next version, allow plug-ins to target the linker, assembler or driver as well as different language specific parts of GCC. These capabilities, however, are not of particular interest to machine learning tasks and so will not be elaborated upon further.

[4] The `message` plug-in is quite powerful. It can send formatted text to different files and even sockets at the beginning and end of the application as well as in response to various events. This makes is very simple, for example, to log how all of the loops in benchmark were unrolled.

ins are not loaded unless they are required by some other mechanism. Eager plug-ins are always loaded.

A plug-in is marked as lazy by adding attribute `lazy="true"` to the plug-in XML specification file. All plug-ins without this are eager.

```
<plugin id="..." lazy="true">
    ...
</plugin>
```

3.2.3.2 Loading From Command Line

Users may inform *libPlugin*that they require certain plug-ins to be loaded via command line arguments. *libPlugin*will check that the plug-ins exist on the path and if they are marked as lazy plug-ins, then it will put them in the required set and cause them to be loaded.

To load a lazy plug-in from the command line, the user gives a comma separated list of plug-in identifiers. For example, with the command line below, GCC will log all passes each function goes through during compilation and will additionally print how the default unrolling heuristic would optimise each loop in the file `foo.c`[5].

```
gcc -plugins gcc-print-pases,gcc-print-unrollable-loops foo.c
```

3.2.3.3 Recursive Dependencies

Plug-ins need to depend upon the services provided by other plug-ins. *libPlugin*provides several ways to specify these dependencies.

The simplest dependency is implicit. When a plug-in extends an extension point, the owner of the extension point will be loaded (if not already). A plug-in can achieve the same effect programmatically in its life-cycle methods by simply getting the address of the extension point.

The second method is explicit. The plug-in'XML specification can contain a `<requires plugin="foo"/>"` to require plug-in `foo`. [6] This can also be achieved programmatically.

[5]Both of the plug-ins, `gcc-print-pases and gcc-print-unrollable-loops` are provided by default in the GCC implementation of *libPlugin*

[6]The `<requires/>` element also allows a specific range of plug-in versions to be given.

3.2.4 Plug-in Life-Cycle

Plug-ins have a defined life-cycle model which allows them to perform suitable initialisation and clean up tasks in the knowledge that services they require from other plug-ins will be operational.

The *libPlugin*plug-in manager goes through a number of phases. An example of the movement through these phases is given in figures 3.2 and 3.3.

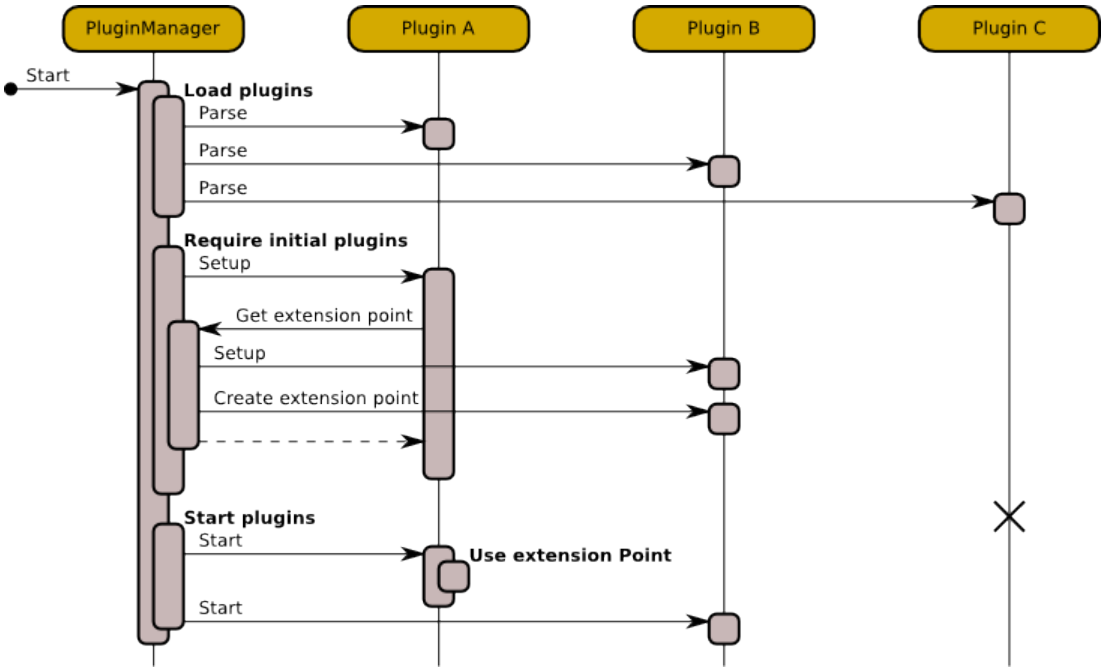


Figure 3.2: Plug-in start sequence in UML form.

First the XML plug-in specification documents for each plug-in on the search path are parsed.

Next any plug-in that is not marked as `lazy` or is required by the user's command line is marked as required. (In this example, only Plug-in A meets this criterion).

Required plug-ins then have their `setup` life-cycle method called. This method may cause other plug-ins to become required by, in this case, programmatically asking for the extension point of another plug-in, which is here provided by plug-in B and thus plug-in B subsequently becomes required. Plug-ins cannot use other methods extension points during this method (because the other plug-in may not be set up).

Once all required plug-ins have been set up, their `start` life-cycle method is called. Plug-in A can now use the extension point from plug-in B.

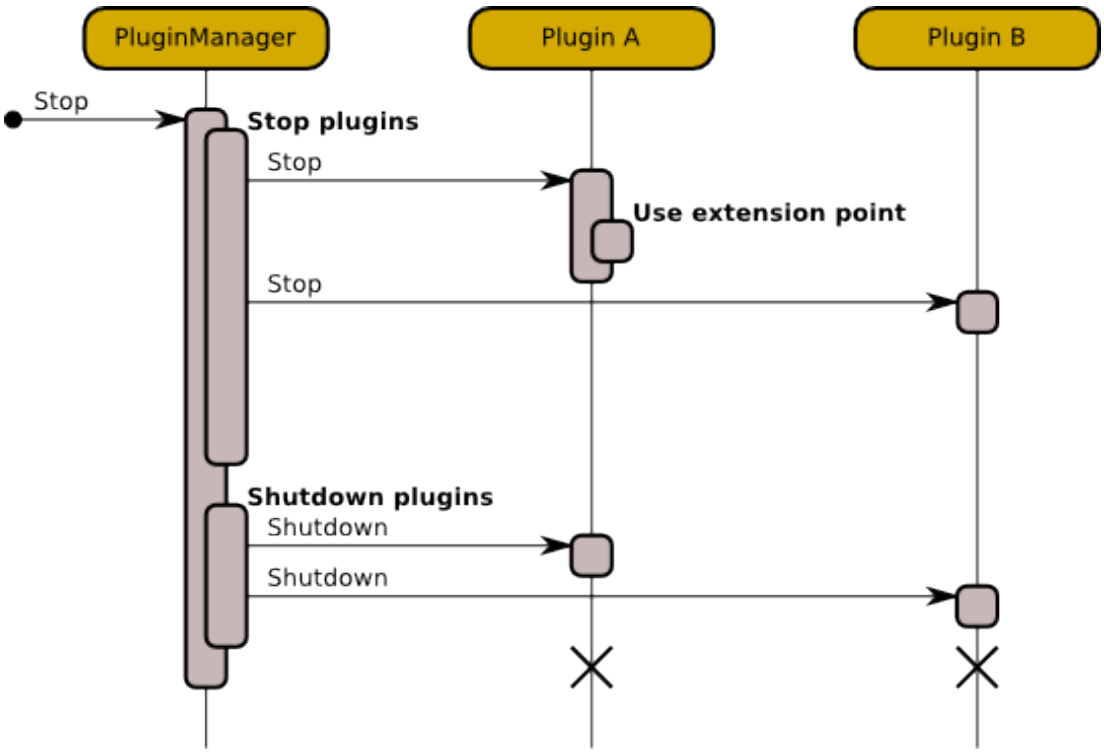


Figure 3.3: Plug-in stop sequence in UML form.

All live plug-ins have their `stop` life-cycle method called. During this phase, plug-ins know that other plug-ins are still active and can continue to use their extension points. This allows plug-ins to, for example, send statistics to a database plug-in, knowing that the database connection is still open.

After all plug-ins have stopped, their `shutdown` life-cycle methods are called. Plug-ins know that no other plug-ins will use their extension points so they can close any resources they might still have open.

3.2.4.1 Parsing

The first phase for the plug-in manager is to parse all of the plug-in XML specification files it can find in the plug-in search path. *libPlugin*will know which plug-ins exist and are eager. Additionally, it determines which extension points are provided by which plug-in.

3.2.4.2 Setup

During the setup phase, *libPlugin*will determine which plug-ins need to be required. It starts with the initial set of eager plug-ins and those which are specified on the command line. Then it require plug-ins from dependencies in the specification files of

already required plug-ins.

As each plug-in is processed its life-cycle `setup` method is called if it has one. This method might programmatically require other plug-ins. It may not, however, make use of any of the extension points of other plug-ins. This is because not all plug-ins will be properly setup during this phase.

For example, a database reporting plug-in might open a connection to the database in this phase.

3.2.4.3 Start

During the start phase, *libPlugin*runs the `start` life-cycle method of each plug-in than has one. From this point on, since all plug-ins are properly set up, plug-ins can use the extension points of other plug-ins.

In the database reporting example, other plug-ins might use this opportunity to create tables in the database.

3.2.4.4 Running

The system is initialised and all plug-ins have started. GCC then compiles the source files it has been asked to compile. Depending on the plug-ins that have been loaded, some of GCC's default behaviour will have been altered by the loaded plug-ins.

3.2.4.5 Stop

Once GCC has compiled the source files it was given, it stops loaded plug-ins. Every plug-in with a `stop` method will have that method called. Plug-ins may still use the extension points of other plug-ins and should keep their own usable.

If one plug-in, for example, has been collecting statistics during the running of GCC then it could now send them to a database plug-in, knowing that its extension points are still active.

3.2.4.6 Shutdown

Finally, before releasing all resources, *libPlugin*will call the `shutdown` life-cycle methods of any plug-ins that have them. This gives those plug-ins the opportunity to close any resources of their own which *libPlugin*cannot know about.

3.2.4.7 Life-cycle Methods

During the different phases, plug-ins might need to specify functions to be run. The plug-in XML specification format allows plug-ins to indicate which methods those are, if any. The plug-in attaches a shared library with the `<library>` element. Within that element the plug-in may use `<setup>`, `<start>`, `<stop>` or `<shutdown>` elements to show that it has life-cycle methods in the library.

Below, a simple plug-in will print a message during the start phase. First is the XML specification.

```
<plugin id="foo">
    <library path="foo.so">
        <start/>
    </library>
</plugin>
```

Next is some suitable code which should be compiled into library `foo.so`. [7] The function will print the identifier of the plug-in at start up.[8]

```
#include <stdio.h>
#include "libplugin/Plugin.h"

bool Plugin_start( Plugin* plugin ) {
    printf( "Plugin, %s, was started!\n", Plugin_getId( plugin ));
    return TRUE;
}
```

3.2.5 Extension Points and Extensions

Plug-ins bundle together related functionality into sensible units. The majority of the work, however, is performed at a finer grained level, that of extension points. At it's most simple an extension point is simply a named object which has an `extend` function with which other plug-ins can pass it snippets of XML. The XML can be arbitrarily complicated and include pointers to symbols from from the extending plug-in's shared libraries, so there is no limit to the power of the extension mechanism.

For example, GCC provides an extension point, `gcc-rtl-unroll-and-peel-loops.override`, which allows other plug-ins to override the default loop unrolling heuristic for any loops it chooses. If a plug-in includes the snippet below in its specification file, it will

[7] The predefined name `Plugin_start` can be changed by including a `symbol` attribute in the `<start>` element.

[8] There already exists a plug-in for printing formatted messages to various outputs.

extend that extension point, asking for loop two in function `bar` from file `foo.c` to be unrolled 10 times.[9]

```
<extension point="gcc-rtl-unroll-and-peel-loops.override">
    <loop
        main-input-file="foo.c" function="bar" loop="2"
        times="10"/>
</extension>
```

The implementation of the extension point itself is in two parts. First there must be some C function to accept any extensions. In pseudo-code it looks something like this:

```
list overrides = NULL;

bool overrideExtend(
    ExtensionPoint* self,
    Plugin* extender,
    xmlNodePtr specification
) {
    replace unroll heuristic with overrideUnroll;
    for each child in specification {
        append child to overrides;
    }
    return TRUE;
}

int overrideUnroll( loop* lp ) {
    spec = first element in overrides matching lp;
    if( spec == NULL ) return previous heuristic;
    else return spec.times;
}
```

The extension function remembers what overrides it is given. It also needs to replace the default unroll heuristic with something that will use the overrides when given. In fact the unrolling heuristic is represented by another extension point so it can be programmatically overridden.

This demonstrates one of the powerful aspects of the extension point system; the ability to compose different extension points to give layers of functionality. The first extension point allows low-level alterations to the the heuristic and another gives a high-level but less capable wrapper.

The plug-in must also declare the `gcc-rtl-unroll-and-peel-loops.override` extension point. It does this by putting the following in its XML specification:

[9] The unrolling override extension point provides more functionality than this.

```
<extension-point id="gcc-rtl-unroll-and-peel-loops.override">
    <extend symbol="overrideExtend"/>
</extension-point>
```

This declaration informs *libPlugin*that whenever another plug-in extends the extension point the `overrideExtend` function should be called.[10]

Although the extension mechanism is very simple to arrange it does require some coding of the extension function which invariably involves an amount of tedious XML processing[11]. Since one of *libPlugin*'s goals is to make extensibility as simple as possible, it offers are number of short cuts for defining powerful extensions for the most common cases with almost no code. The following sections describe the easy ways to create and use convenience extension points.

3.2.5.1 Events

Events are a common programming pattern that are extremely useful in an extensible compiler. Consider, for example, the case when a user would like to know what loops have been unrolled and with what unroll factor. They might choose to log this information to a file or to a database or to aggregate it in some other way as the loops are unrolled. If the compiler fires an event every time it unrolls a loop, then users can listen to those events and do anything they want. The compiler is thereafter free from worrying about whether is has supported every possible user interaction. *libPlugin*makes creating, firing and listening to events trivial. This section describes how that is done.

Event extension points allow one plug-in to publish events to other plug-ins. Plug-ins register their interest in listening to the event by extending the event extension point. Each extending plug-in will give a call back function to be invoked when the event occurs. The owning plug-in will give the system a function pointer (also of the same type) which will be replaced if any other plug- in extends this event. The replacement will call each of the call back functions from the listeners.

The process is best shown by example. Suppose that one plug-in would like to report an event called `something_happened` and parametrise this with a number and a string. First we see how the plug-in must write its C code to declare and to fire the event.

[10] In the actual plug-in the `overrideExtend` function is provided in a separate, optional shared library. There are other small differences in the real code which take advantage of more advanced *libPlugin*features.
It is also possible to create extension points by pointing to a factory method or programmatically in the `setup` life-cycle method of a plug-in.

[11] *libPlugin*is built on top of the open source `libXML2` library.

```
// Declare an empty function that is the same as firing
// the event to no listeners
void something_happened_empty(int number, char* string) {}

// Declare a function pointer for the event
void (*something_happened)(int number, char* string) =
    something_happend_empty;

// Later in the code, fire the event
something_happened(100, "it happened!");
```

The event is nothing more than a function pointer with the right prototype. The plug-in can call it whenever it needs to. Initially, the function it points to does nothing. If any other plug-in is listening to the event then *libPlugin*will have replaced the function pointer with a new function which informs all listeners of the event. The changes to the code are kept to the bare minimum. [12]

To tell *libPlugin*about the event, the plug-in adds this to its XML specification:

```
<event id="app.core.something-happened"
    signature="void f( int, char* )">
    <call symbol="something_happened"/>
</event>
```

The XML gives the event an identifier and tells *libPlugin*the name of the function pointer to replace and what the signature of the event is. We will see why the signature is necessary later.

Listening to the event is just as straightforward. The listening function is written with the same prototype as the event:

```
void handle_something_happened(int number, char* string) {
    printf("It happened! number=%d string='%s'\n", number, string);
}
```

And the listener function is declared to the system: [13]

```
<extension point="app.core.something-happened">
    <callback symbol="handle_something_happened"/>
</extension>
```

Now whenever the event is fired the call back is triggered.

[12] Often the function pointer will be initialised to `NULL` and a null check will be made before firing the event. This is more efficient and avoids computing needless arguments.

[13] Factory methods can also be used to create events and event handlers; both are objects, not just functions and the example declarations here show only the most convenient usage.

3.2.5.1.1 Event Handler Creators *libPlugin*goes much further. Plug-ins can provide event handling services to other plug-ins which often means that powerful effects can be achieved without writing any C code at all.

For example the `message` plug-in provides an event handling service which plug-ins can use to create an event handler which logs event information when the event is fired. The extension below has exactly the same effect as the previous, C-based one but requires no shared library. [14]

```
<extension point="app.core.something-happened"
            create="message.event-logger">
    <text>It happened! number=</text>
    <arg-print index="0" format="%d"/>
    <text> string='</text>
    <arg-print index="1" format="%s"/>
    <text>'</text>
    <br/>
</extension>
```

3.2.5.1.2 Dynamic Code Generation There has been a certain whiff of smoke and mirrors in the description of the working of the event extension points. In particular the issue of how *libPlugin*replaces the event function pointer so that it points to new function which will call the waiting listeners has been glossed over. In fact, this is strictly not possible using C. The listeners of two different events may have different signatures and one function cannot call both functions types with having them hard wired into it, so the 'call all listeners' function is impossible to write. Moreover, there is no function prototype suitable for that function which would allow it to be put at all the function pointers of the events. Not only can we not write the function in C, but we could not use it even then.[15]

Indeed the situation is worse because the actual function prototypes have to be changed; *libPlugin*is really an object-orientated system in C that allows users to stick with simple C prototypes whenever convenient and auto constructs the objects for them. This requires prototypes to be created that have a `this` pointer to the object and then the remainder of the original arguments. Again, due to C's lack of reflection this is not possible in C alone.

[14]There is coming a JavaScript plug-in which allows other plug-ins to run arbitrary scripts to be run in response to events.

[15]C's variable length arguments are not sufficient to solve this problem.

This difficulty is overcome by dynamically generating small thunk functions thunk that marshal arguments into and out of reflective arrays. When an event is listened to, *libPlugin*creates a function which has the same signature as the event. This function gathers the arguments into an array and passes it to a generic event dispatching function. That function in turn will call each of the listeners' call back functions which will generally require the arguments to unmarshalled again into the normal native argument type using another dynamically generated thunk.[16]

These efforts allow the user to make simple event handlers easily and naturally but also allows powerful generic event handlers can also be created. *libPlugin*takes care of all of the hard work.

3.2.5.2 Around Advice

One of the primary needs of machine learning in compilers is to be able to replace the default behaviour of an heuristic. To support this, *libPlugin*borrows a concept from aspect orientated programming (AOP). AOP allows developers to add *advice* to methods that have already been written. One form of this advice replaces the method with a new one which receives the same original arguments. The advice can perform any operation it desires but in particular can also, if it needs to, call the original method it replaced. In fact, these advices can be layered, with one method being advised multiple times, and each layer of advice can call the next one down if it wants to. The AOP formulation has been very successful and has been demonstrated in a large number of real world projects; it is also a perfect fit for the machine learning in compilers requirements.

*libPlugin*allows plug-in writers to specify that a function can be advised and for plug-ins to advise that function. The formulation is very similar to that of events with a few small changes; the functions involved may now have return types; the XML description of the advisable function uses an `<around>` tag rather than the `<event>` tag and there are additional API functions for advice to get a pointer to and call the next advice in the stack. Additionally, around advice can be built generically in the same way that event handlers can so that plug-ins do not always have to resort to C

[16]In fact the call back function of all event handlers accepts the reflective array version of the arguments together with a `this` argument. This allows generic event handlers such as the message event logger from the previous section to work. If the user provides a non-reflective function - which *libPlugin*allows believing that convenience for the user should be paramount - a thunk is dynamically created to unmarshall the arguments back into the native form and that thunk becomes the event handler's call back.

code to advise methods. For all of this to work, around advice requires the same kind of dynamic code generation as events.

Typically, around extension points do not exist in isolation. There is a more powerful concept called a *join point* which is more useful for defining replaceable heuristics. Join points contain an around extension point (and two events) within them and there are no practical advantages to using an around extension point by itself. Join points are discussed next.

3.2.5.2.1 API to Call Through the Advice Stack

The simplest way to advice a function is to write another function with the same prototype. When this advice is on the top of the advice stack (or called by higher level advice) it will completely replace the function it is advising.

So, if we had an original function,

```
int heuristic(int number, char* string) {
    return number * strlen(string);
}
```

We could completely replace it with a new function which did the same thing but added one to the answer.

```
int replacement_heuristic(int number, char* string) {
    return number * strlen(string) + 1;
}
```

Being unable to reuse the previous function (or rather the next advice down on the stack which might be just the original heuristic) is extremely annoying. *libPlugin*allows advice easy access to the next advice on the stack but only if they use one of the object-oriented forms of advice (recall that events were really object oriented to, they just appear to be functions because *libPlugin*generates convenience object for the user - the user could also have created the object himself).

To use the object oriented version there must be a `self` pointer as the first argument.

```
int replacement_heuristic(
    AroundAdvice* self,
    int number, char* string
) {
    return number * strlen(string) + 1;
}
```

We must also inform *libPlugin*in the plug-in XML that the advice function has this pointer as first argument:

```
<extension point="heuristic">
    <callback symbol="replacement_heuristic" type="non-static"/>
</extension>
```
17

From this `self` pointer we can get information about parameter types, extension and plug-in identifiers and much more. However, at present we are interested getting a function pointer we can call for the next advice on the stack. This we can do with the function `AroundAdvice_getCallNextFn`; it returns an untyped function pointer so we have to cast it to the proper type: [18]

```
int (*next)(int, char*);
next = (void (*)(int, char*))AroundAdvice_getCallNextFn(self);
```

The `next` function can then be called as normal. If the current advice is the only one on the advice stack then the next function will call the original function; otherwise it will call the next advice which can, should it need to, call its next advice and so on down to the original.

To add one to the original heuristic the code would then be:

```
int replacement_heuristic(
    AroundAdvice* self,
    int number, char* string
) {
    int (*next)(int, char*);
    next = (void (*)(int, char*))AroundAdvice_getCallNextFn(self);
    return next(number,string) + 1;
}
```

[17] *libPlugin*allows three convenience methods for automatically constructing the advice object from a single call back function. The default is without the `self` argument; in addition to the simple one with the `self` argument there is one which takes a `self` argument and a reflective array of the remaining arguments. This latter form is useful for generic advice.
The user could alternatively give a symbol which points to an advice object or to a function which creates advice objects. These forms give the user complete power at the expense of having to slightly write more code.
All of these conveniences are provided to events as well and at many other places in *libPlugin*, making it a very easy system to use and requiring as few lines of code as possible.

[18] This function will create the required dynamic thunk code if necessary and is cached thereafter. There are other functions to get the `next` function in different formats.
It is also possible for an advice object to demand to be the top advice on the stack. This is achieved by adding an attribute `selfish="true"` to the extension specification, but like most things in *libPlugin*can also be done programmatically. If any other advice tries to advise the function after a selfish advice has been applied an error is generated and the program stops.

3.2.5.3 Join-Point

Around extension points allow functions and heuristics to be modified. They have some small practicality limitations, however. A typical usage pattern for altering heuristics is that we may wish to receive an event when the function is first called with the arguments passed to it and another when the function terminates, this time with the return value as well as the arguments. These events are useful when some kind of reporting is required which does not override the behaviour of the advised function. We cannot simply have some around advice which performs the logging and then delegates to the next advice on the stack without altering arguments or return value since we cannot guarantee the ordering on the advice stack.

AOP solves this problem with a concept called a *join point* and *libPlugin*borrows that concept. A join point consists of exactly the two event extension points and an around extension point that we need as shown in figure **??**. When a join point is called, first all of the before event listeners are notified with the function's parameters. Then the top advice on the around advice stack is called with those parameters, which may or may not call further down the advice stack. Finally the after event listeners are notified with the return value from the top advice and the original parameters of the function call.

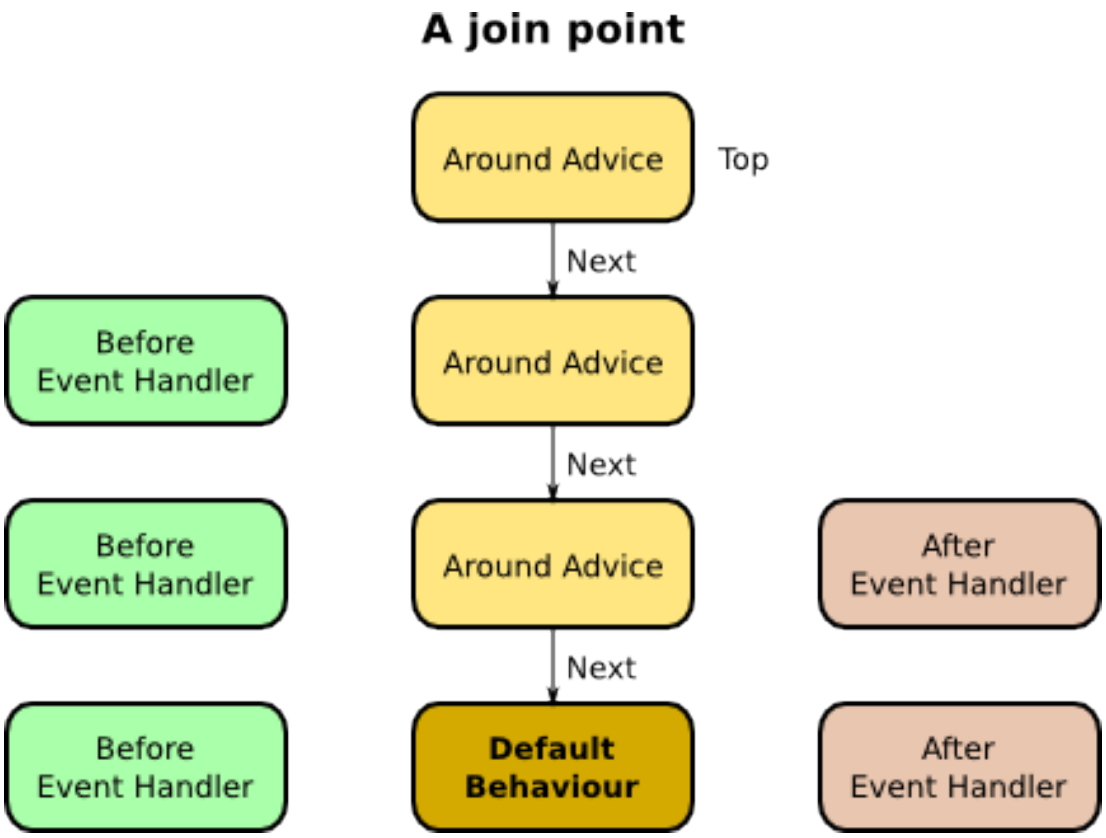
Creating and using a join point in *libPlugin*is very simple. For example, if the C code for the loop-unrolling heuristic is originally as our initial example:

```
1 int decideUnrollTimes(loop* lp) {
2     int times = /*the heuristic*/;
3     return times;
4 }
```

Then we can turn it into a extendible join point by converting it to function pointer instead. This requires only one additional line of code and all uses of the function remain exactly as they were; the compiler is not cluttered with ugly extensibility code. The heuristic now looks like this:

```
1 static int decideUnrollTimes_original(loop* lp) {
2     int times = /*the heuristic*/;
3     return times;
4 }
5 int (*decideUnrollTimes)(loop* lp) = decideUnrollTimes_original;
```

The join point needs to be declared to *libPlugin*in a plug-in XML specification file, giving an identifier for the join point, the prototype of the function so that dynamic



code can built for it and the function pointer to be replaced: [19]

```
<join-point id="decide-unroll-times"
    signature="int f(loop*)">
    <call symbol="decideUnrollTimes"/>
</join-point>
```

The join point is not in itself an extension point. Instead, a join point creates three extension points. If the identifier of the join point is `x`, then the first event extension point will have identifier, `x.before`, the around extension point will have `x.around` and the last event will have `x.after`. These extension points are then used as normal.

If no plug-in listens to one of the join point's events or places advice on the around stack then the function pointer, `decideUnrollTimes`, will still point to its original value, the function `decideUnrollTimes_original`. Only if necessary is any dynamic code constructed and the function pointer updated. In this way, just as for events and around extension points, there is practically no cost to making the compiler extensible.

[19] As with most things in *libPlugin*, this is just one of the convenient methods. There are other ways to achieve the same result which are useful if more power is required.

3.2.5.4 List

The simplest type of convenience extension point provided by the system is a list of values. The null terminated list can contain a pointer to any type of data and other plug-ins can append values to the list.

For primitive types (integers, floating point numbers and strings) the element to be appended can be given directly in the extending plug-in's XML description. More complex types are handles by either simply providing a symbol which points to the element in a shared library or by giving a factory

3.2.5.5 Hook

A hook allows a function's implementation in one plug-in to be replaced by another plug-in. A hook in the owning plug-in is simply a function pointer which another plug-in can overwrite. The owning plug-in can then call the hook whenever it needs to. Only one plug-in can extend a hook; the system will report an error if two plug-ins attempt to extend the same hook.

Hooks are extremely limited; for nearly all cases the join-point extension (see above), which is much more powerful should be used instead. However, there is a cost to join-points because no dynamic binding code needs to be generated. Hooks, on the other hand, provide extensibility at only the cost of an indirect function call. They should be used only when the hook will be called so often that the greatest efficiency is required.

All of these convenient methods for creating extension points means that the compiler can be 'marked up' for extensibility with a tiny amount of code which is hardly noticeable. The original purpose of the code remains uncluttered by extraneous additions just for the sake of extensibility.

Moreover, *libPlugin*takes great pains to ensure that extensibility is as efficient as possible, in particular in the case where the user does not wish to extend some part of the compiler. Consider, by comparison the situation in the original example where the heuristic was littered with extensibility code. This code tests whether each individual extension is required and these tests must be undertaken regardless of whether the user has requested the extension. Naïve implementations using shared libraries can look marginally cleaner but still scour through lists at each invocation, only to find that no extension for the particular point has been made. No extensibility library for C can

match *libPlugin*for it's power, simplicity, convenience and efficiency.

3.2.6 Machine Learning Plug-ins

*libPlugin*for GCC comes with a number of plug-ins that are useful for machine learning tasks. The following sections give a brief introduction to each.

3.2.6.1 Perfmon

The `gcc-perfmon` plug-in allows GCC to instrument functions it is compiling to count the number of cycles spent in each function. Cycle counts are very useful when performing iterative compilation on functions or smaller code elements as they give timings to very high precision when other techniques might be too coarse grained to time element. There can be a downside that occasional context switches will cause outliers in the data, however, these outliers are usually so far out that they are easily purged from the data.

When instrumenting a function `gcc-perfmon` will transform it from this:

```
fn() {
    fn_body;
}
```

Into something like:

```
static callcount_t fn_callCount = 0
static cyclecount_t fn_cycles = 0
fn() {
    fn_callCount++;
    cyclecount_t t0 = CYCLECOUNT();
    try {
        fn_body;
    } finally {
        cyclecount_t t1 = CYCLECOUNT();
        fn_cycles += t1 - t0;
    }
}
```

[20]

The cycle count for a function can also be paused when another function is called. By default the count is always paused when another function is called, but for some functions that may be inappropriate. Which functions to pause for can be completely under user control.

[20] In fact the code is slightly different for efficiency reasons and to support callee pausing. Also, the variable names are compiler temporaries that cannot clash with any names in the source code.

Code is also inserted into the current compilation unit to dump the gathered statistics when the program exits.[21] The statistics can be output to a file as XML, SQL or JSON. In all those cases an identifier for the current compilation can be added which is necessary when doing iterative compilation.

Additionally, the JSON format is compatible with CouchDB and open-source, loosely structured, document orientated database. These JSON data can be sent automatically to the database, meaning that every time the program is run the database is informed of that without further effort from any party. When sending to the database additional information is supplied such as the identity of the machine that program was run, the user name, dates and times, etc. The destination for the statistics is initially just the standard error stream, but this can be changed by a plug-in at compile time or even at run time by environment variables. [22]

The plug-in is a lazy plug-in, meaning that it must be explicitly invoked to be loaded by the system. The plug-in can be invoked by adding to the command line, `-plugins gcc-perfmon`, or by another plug-in extending one of its extension points.[23]

There are two join points in the plug-in, `gcc-perfmon.should-instrument-current-functio`n and `gcc-perfmon.should-pause-callee`, and an extension point, `gcc-perfmon.settings`. The latter extension point allows a simple specification of which functions to instrument and pause for, while the join points give programmatic control over the same.

3.2.6.1.1 `gcc-perfmon.should-instrument-current-function` This join point takes no arguments and returns true if the current function should be instrumented for cycle counting and false otherwise. By default, all functions are instrumented. Any plug-in wishing to alter this can advise this join point and look at GCC's `current_function` pointer to see what is being compiled.

3.2.6.1.2 `gcc-perfmon.should-pause-callee` To find out if a call should pause the cycle count, this join point takes the name of the function being called and a pointer to the AST object for the call. The join point returns true if the call should pause and false otherwise. By default is always returns true.

[21] This is done by adding a compilation destructor function.

[22] At compile time it can also be set on the command line or through environment variables. This is because *libPlugin*also supports full variable expansion mechanisms from those sources (and from inside plug-in specifications). Variable expansion is very useful for writing concise plug-ins but does not particularly aid machine learning goals, so is not discussed in this thesis.

[23] There are several other ways to cause a plug-in to be loaded, it as described in previous sections.

3.2.6.1.3 `gcc-perfmon.settings` The `settings` extension allows a very simple specification of which functions to instrument and which to pause for. The user gives wild-carded[24] lists of functions to instrument or to pause for. Each element of the lists either includes or excludes the functions it names and the elements processed in document order. A typical example might look like:

```
<?gcc version="4.3"?>

<plugin id="perfmon-example">
    <!-- Extend the settings and dump to file
     output.json -->
    <extension point="gcc-perfmon.settings" file="output.json">
        <!-- Say what to instrument -->
        <instrument>
            <!-- Initially, everything is instrumented, but
              if we only want to instrument a few functions,
              then we start by excluding everything -->
            <exclude main-input-file="*" function="*"/>
            <!-- Now list what to include, two from foo.c
              and all in bar.c -->
            <include main-input-file="foo.c" function="foo"/>
            <include main-input-file="foo.c" function="bar"/>
            <include main-input-file="bar.c" function="*"/>
        </instrument>
        <!-- Say what calls to pause on -->
        <pause>
            <!-- Initially, everything is paused -->
            <!-- We will not pause for transcendentals -->
            <exclude function="sin"/>
            <exclude function="cos"/>
            <exclude function="tan"/>
        </pause>
    </extension>
</plugin>
```
[25]

The extension point itself is built on top of the two join points from the plug-in and, to some extent, demonstrates how easy adding extensibility is with *libPlugin*.

[24] The wild cards used are POSIX glob patterns

[25] Match attributes where the pattern is a single * can be left out

3.2.6.2 Trace

The `gcc-trace` causes GCC to instrument the files it is compiling to create traces of each basic block as it is executed. These traces create very large data files but may offer interesting possibilities for machine learning.[26]

The plug-in instruments the code so that a print statement is placed at the beginning of each basic block. The print statement is directed to a file and spits out only an identifier number for the basic block followed by a new line. The file is the standard error stream by default but can be overridden by a plug-in or even at run time through environment variables. The trace files are normally so big that it is recommended to pipe them to a consuming program rather than store them directly.

Additionally, during the compilation the mapping of identifiers to basic blocks is written out to a file so that traces can be reverse engineered to find source file, function and line numbers for the basic blocks in the trace.

The plug-in allows the specification of which functions to trace in a very similar way to the methods for `gcc-perfmon` and so that is not further discussed here.

3.2.6.3 Command Line

Iterative compilation at the whole program level often involves simply searching through different command line options to find which produce the fastest program. In other cases, some benchmarks set various command line arguments which might conflict with the desired experiment, such as turning off loop unrolling when the experiment needs to investigate different unrolling strategies.

Although this changing the GCC command line is conceptually simple, the benchmarks used are not often compiled with a direct shell command to GCC. Instead, there is usually a make file involved and sometimes this can be almost impenetrable. Deconstructing the make file and re-engineering it if necessary is both time consuming and error prone. *libPlugin*solves this by allowing command line arguments to be altered from a plug-in. [27]

The plug-in is simply called `command-line` and is available to all *libPlugin*enabled applications, not just GCC. It contains only one extension point, `command-line.modify`.

[26] Some on going work is looking at how different input data sets can effect program performance and we hope that differences between execution traces may act as a proxy for features of the data.

[27] To be loaded a plug-in need only be eager and on the search path. That path can be set by environment variable. This makes command line modification easy and certain.

3.2.6.3.1 `command-line.modify` This extension point allows command line arguments to be removed and inserted before the program is run. [28]

Arguments are removed in sequences matching a wild card pattern. While one often wants to remove only a single argument at a time, there are situations when one argument indicates that next is a parameter for the first argument. Typically both must be removed at once. The following example removes all single arguments beginning with `-O` and double arguments where the first argument is `-I`.

```
<extension point="command-line.modify">
    <remove><arg>-O*</arg></remove>
    <remove><arg>-I</arg><arg>*<arg></remove>
</extension>
```

Arguments can be inserted at the front of the argument list.[29] Argument sequences can be added in the same fashion as they are removed. The example below inserts `-O3` at the front of the command line.

```
<extension point="command-line.modify">
    <insert><arg>-O3</arg></insert>
</extension>
```

The insertions and removals are processed in document order. A useful pattern for iterative compilation, therefore, is to first remove all of the arguments that will be iterated over and then insert the ones from the current search point.

3.2.6.4 Loop Unrolling

Loop unrolling is a well studied optimisation which nevertheless still has room for improvement. It is commonly targeted for machine learning experiments since researchers feel that if gains can be won against such a mature optimisation then they truly demonstrate the validity of their techniques.

In GCC there are two places where loop unrolling is performed. The first is in the high level AST, while the second is much later after the source has been lowered to RTL. The second is much more capable and has been around for much longer. It not only performs several flavours of unrolling but also manages loop peeling at the same time.

*libPlugin*offers a full featured plug-in to interact with the more powerful RTL unrolling optimisation. A similar plug-in is in development for the AST level code. This

[28] That is to say, before the program leaves the start phase of the *libPlugin*life-cycle model.

[29] This extension point does support insertion elsewhere, but plug-ins can programmatically alter the command line should they need more control

section discusses the former only in a plug-in called `gcc-rtl-unroll-and-peel-loops`. The plug-in provides several services related to printing unrolling information about loops and querying unrolling status and legal values, however, the focus here will only be on how to force the unrolling and peeling decision for individual loops.

The unrolling plug-in has one join point to programmatically control the unrolling decision and one extension point, built on top of that join point, that allows a simple, no C code specification to be given. The join point is called `gcc-rtl-unroll-and-peel-loops.decisi` and the extension is `gcc-rtl-unroll-and-peel-loops.override`. Only the `override` extension point is described here.

3.2.6.4.1 gcc-rtl-unroll-and-peel-loops.override This extension point allows the user to specify which loops should be unrolled and by how much. [30]

Extensions give a list of wild carded `<loop>` elements which match source file names, function names and loop numbers. Each of these `<loops>` gives the number of times the loop should be unrolled (or may say to use the default). [31] The elements are processed in document order until a match is found and that then gives the unroll factor for the loop. If no match is found the default heuristic is used. This ordered processing means that the most specific matching patterns should appear first.

For example, the plug-in specification below unrolls loop two in function `foo` ten times, all other loops in that function by the default heuristic and any loop any other function will not be unrolled.

```
1 <extension point="gcc-rtl-unroll-and-peel-loops.override">
2     <loop main-input-file="foo.c" function="foo" number="2" times="10"/>
3     <loop main-input-file="foo.c" function="foo" number="*" times="default"/>
4     <loop main-input-file="*" function="*" number="*" times="0"/>
5 </extension>
```

[32]

There is also a plug-in, `gcc-print-rtl-unroll-and-peel-loops` which will log a list of all the loops in each function. Information about whether the loop can be unrolled or peeled, and if so in what flavours and by how much may be included as

[30] It also allows the user to indicate which type of unrolling flavour to perform, whether to do loop peeling and what to do if the user gives an unroll or peel factor which is impossible for the loop and flavour. However, those capabilities, while useful for machine learning, would take a lot of space to describe without contributing much to the discussion.

[31] Strictly speaking, because of this extension point is built on top of `gcc-rtl-unroll-and-peel-loops.decision` join point, it can only guarantee to get the decision from the next advice down in the stack. Typically, however, this will just be the default heuristic.

[32] Match attributes where the pattern is a single * can be left out

well as what unrolling or peeled was actually performed. The log can be sent to various destinations, including a database, and in several formats.

3.2.6.5 Auto-Vectorisation

GCC has an optimisation to which tries to automatically vectorise loops. Vectorisation involves rewriting a loop to perform multiple iterations of a loop in one go. It is specifically supported by SIMD units, but might also be possible in some circumstances on the main CPU. The potential gains for successful vectorisation are large, but over zealous vectorisation can cause performance degradation.

*libPlugin*provides a plug-in to print information about vectorisation operations, to query various vectorisation aspects of loops and to control which loops are vectorised. It is a very similar plug-in to the loop unrolling plug-in. The essential difference is that rather than specifying the number of times the loop should be unrolled, the target is given. The target is unit that the code will be vectorised on (e.g. not at all, a SIMD unit, the main CPU, etc.)

3.2.6.6 Inlining

Function inlining replaces a function call site with the body of the function being called. Inlining is not always possible and not always desirable when it is possible.

There is a plug-in, `gcc-inlining`, in *libPlugin*which allows control of inlining. Again, the plug-in provides services to print information about inlining, query inlining characteristics (such as is a call site inlinable) and to control which call sites are inlined.

The most convenient way to control inlining is with the `gcc-inlining.override` extension point.

3.2.6.6.1 gcc-inlining.override This extension point allows inlining to be easily specified. The extensions consists of a list of wild carded `<call-site>` elements which match call sites within functions. When the compiler finds a call site it could inline it will scan the list of `<call-site>` elements until it finds a match and then use the inline value given in the element.

In the example below, all calls to function `bar` from function `foo` are inlined if possible; all calls to `blob` from `foo` will not be inlined and all other call sites in `foo` will use the default heuristic.

```
<extension point="gcc-inlining.override">
    <call-site main-input-file="foo.c" function="foo" callee="bar" inline="tr
    <call-site main-input-file="foo.c" function="foo" callee="blob" inline="f
    <call-site main-input-file="foo.c" function="foo" callee="*" inline="defa
</extension>
```

This extension point does not distinguish between two different call sites in the same function to the same callee. Sometimes, this will require resorting to programming via the underlying join point.

3.2.6.7 Pass Manager

Passes[33] in GCC are where it performs the majority of its work. The compiler contains some 180 different passes which transform the code down closer to machine code and apply all of the numerous optimisations that GCC supports.

GCC contains a tree of passes; some passes are really just containers for others. Each pass has a unique identifier [34], although a pass may appear more than once in the tree; for example, common sub-expression elimination can be applied more than once. Each pass has a `gate` function which determines if the pass should be applied. Typically gates will check to see if suitable command line options have 'turned on' the pass. The pass also has an execute function which does the pass' work. Finally, there is a state machine [35] which should allow a developer to know ahead of time which passes can executed and to build a path through the passes however they wish, essentially dispensing with the default tree. This state machine is in a poor state of repair, however, since typical testing does not exercise it.

*libPlugin*comes with a plug-in which supports machine learning experiments at the pass re-ordering level as well as ordinary compiler extensions. The plug-in, `gcc-pass-manager`, offers facilities to add new passes to those already in GCC. It permits passes to be forcibly turned on or off, overriding their gate functions. It also allows complete con-

[33] Other compilers may use the term 'phase' for GCC's 'pass'.

[34] In fact, in GCC 4.3 this is not the case, most passes are not named. The patch which adds *libPlugin*to GCC also ensures each pass has a unique name. In GCC 4.4 and above, passes are required to be named.

[35] The state machine is based on properties. An example property is "the function is in RTL form". The state of each function being compiled is described by nine properties which may either hold or not, giving 512 possible states for the function (in fact, not all property combinations are possible). Each pass lists the properties that must hold on a function for the pass to be able to execute on it. The pass also lists the properties that will become true or false as a result of running the pass on a function. With these it should be possible to plan a list of passes to apply which takes the function from the start state to the end state; i.e. to be suitable for sending to the assembler. This functionality would be very useful for machine learning on pass reordering and it is a shame that the state machine is out of date and incomplete.

trol over the pass tree on a per function basis.

There is also a plug-in, `gcc-print-passes` which will log a list of all the passes each function is run through while being compiled. The log can be sent to various destinations, including a database, and in several formats.

3.2.6.8 Features

Machine learning experiments require features to be given to the model learner and, once a model has been created, to the model to get its predications. There are many different features that can be thought of for any code block. Researchers often have different needs, some will ask for features at the basic block level, some only on functions, others on instructions; and even if targeting the same level of code element, researchers will come up with different features.

*libPlugin*comes some ten or so plug-ins that compute features for different code elements. Most have been created to the specifications of other researchers. Two of the feature sets, Stephenson's and Milepost's, are well known having been reimplemented from other papers; they are described below. One of the more simple feature plug-ins, built for a colleague, is also described.

All the feature plug-ins have a very similar interface and similar capabilities. By default, all the plug-ins are lazy. If one of the feature plug-ins is loaded it will by default write out features in an XML format to a file called `<plug-in name>.xml` and will compute features for all functions it encounters.[36] However, for all feature sets, the output format, destination and a filter set of included functions can be configured by extending the plug-in's extension points.

3.2.6.8.1 Stephenson's Features StephensonStephenson and Amarasinghe (2005) showed that machine learning could successfully outperform the heuristics built by human experts. His experiments targeted loop unrolling and so he needed features capable of describing loops to the machine learning tools. *libPlugin*has an implementation of features in plug-in `gcc-features-loop-stephenson`.

The full list of Stephenson's features is given in table 5.9.

3.2.6.8.2 Milepost Features The MilepostFursin et al. (2008a) project produced a standard set of features given in table **??**. The features are produced by writing out

[36]Some feature sets operate on smaller elements than whole functions. The features are grouped by function, however.

the whole AST to a Datalog database and then each feature is represented as a Datalog relation over that database. The features are at the function level only.

*libPlugin*has a plug-in, `gcc-features-function-milepost` which reimplements the Milepost features. The implementation is simpler, just written in C, and considerably faster than the original Datalog implementation.

3.2.6.8.3 Instruction Count Features The `gcc-features-basicblock-instructioncount` plug-in will create one feature for each type of AST or RTL node in each basic block or for each RTL.

3.2.6.8.4 Output Format *libPlugin*'s default feature plug-ins can all output the feature data in one of several formats. The current formats include XML, CSV, SQL and JSON. For example, if Stephenon's features were extended with the following format: [37]

```
<extension point="gcc-features-loop-stephenson">
    <output format="xml"/>
</extension>
```

[38] Then for each file compiled the features would print as XML like this:

```
<features type="stephenson" main-input-file="foo.c">
    <function name="bar" point="gcc-rtl-loop-init.execute.after">
        <loop number="1">
            <feature id="nesting.level" value="2"/>
            <feature id="num.ops" value="32"/>
            ...
        </loop>
    </function>
</features>
```

The output for other formats will be similar but if the format is not hierarchical (SQL and CSV) then the hierarchy is flattened. The hierarchy will also deeper or shallower depending on the granularity of the features set; for example, the Milepost features are function level, so the features are directly beneath the feature element and there are no loop elements.

The point at which the features were computed is described by the `point` attribute. This is discussed further in section 3.2.6.8.8. In this case, the features are computed

[37] For convenience, the output format and destination could be set by using command line arguments or environment variables through the plug-in variable syntax. This means that in the majority of cases, a plug-in does not need to be written.

[38] The extension point and the plug-in have the same name. Plug-ins and extension points inhabit different name spaces, so there is not conflict.

Figure 3.4: Milepost features

Number of basic blocks in the method
Number of basic blocks with a single successor
Number of basic blocks with two successors
Number of basic blocks with more then two successors
Number of basic blocks with a single predecessor
Number of basic blocks with two predecessors
Number of basic blocks with more then two predecessors
Number of basic blocks with a single predecessor and a single successor
Number of basic blocks with a single predecessor and two successors
Number of basic blocks with a two predecessors and one successor
Number of basic blocks with two successors and two predecessors
Number of basic blocks with more then two successors and more then two predecessors
Number of basic blocks with number of instructions less then 15
Number of basic blocks with number of instructions in the interval [15, 500]
Number of basic blocks with number of instructions greater then 500
Number of edges in the control flow graph
Number of critical edges in the control flow graph
Number of abnormal edges in the control flow graph
Number of direct calls in the method
Number of conditional branches in the method
Number of assignment instructions in the method
Number of binary integer operations in the method
Number of binary floating point operations in the method
Number of instructions in the method
Average of number of instructions in basic blocks
Average of number of phi-nodes at the beginning of a basic block
Average of arguments for a phi-node
Number of basic blocks with no phi nodes
Number of basic blocks with phi nodes in the interval [0, 3]
Number of basic blocks with more then 3 phi nodes
Number of basic block where total number of arguments for all phi-nodes is in greater then 5
Number of basic block where total number of arguments for all phi-nodes is in the interval [1, 5]
Number of switch instructions in the method
Number of unary operations in the method
Number of instruction that do pointer arithmetic in the method
Number of indirect references via pointers (”*” in C)
Number of times the address of a variables is taken (”&” in C)
Number of times the address of a function is taken (”&” in C)
Number of indirect calls (i.e. done via pointers) in the method
Number of assignment instructions with the left operand an integer constant in the method
Number of binary operations with one of the operands an integer constant in the method
Number of calls with pointers as arguments
Number of calls with the number of arguments is greater then 4
Number of calls that return a pointer
Number of calls that return an integer
Number of occurrences of integer constant zero
Number of occurrences of 32-bit integer constants
Number of occurrences of integer constant one
Number of occurrences of 64-bit integer constants
Number of references of a local variables in the method
Number of references (def/use) of static/extern variables in the method
Number of local variables referred in the method
Number of static/extern variables referred in the method
Number of local variables that are pointers in the method
Number of static/extern variables that are pointers in the method

before the `rtl-loop-init` pass is executed, which is the default for Stephenson's features.

3.2.6.8.5 Output Destination It is often convenient record the features to a different file than the default. All of the feature plug-ins allow this with the `file` attribute of the `<output>` element. The output format will be inferred if possible from the file name if the `format` attribute is not present.

In addition, the plug-ins all append the features to the file. This is convenient because benchmarks typically consist of more than one file and appending all the features means that the make file does not have to be altered. It is possible, however, to change this behaviour by using attribute `append="false"`.

```
<extension point="gcc-features-loop-stephenson">
    <output file="features.xml" append="false"/>
</extension>
```

Output can also be directed to a socket.

```
<extension point="gcc-features-loop-stephenson">
    <output host="server" port="12737"/>
</extension>
```

Multiple output destinations can be given and the plug-ins will send data to all of them.

3.2.6.8.6 Output to Database Recording data in a database is often more convenient than writing the data to file. *libPlugin*makes this automatic, relieving the researcher from the burden of doing it himself. Most of plug-ins that come bundled with GCC that report information about the compilation can direct their data to files, sockets or databases and the features plug-ins are no exception.

At present, only one database is supported, CouchDB. This database is document oriented meaning that data is loosely structured (in JSON) and is not relational. This style is excellent for researchers since it allows databases to quickly created and used without all of the headache associated with designing SQL tables and the difficulties that arise if the SQL layouts ever need to change. More databases may be supported at a later date.

To use the database, the plug-in must give server name or IP address, optionally a port, user name and password. The documents typically also need to be associated with some point in the iterative compilation space (otherwise the features will be meaning-

less). This can be done by giving `tag` attribute which will cause a field of the same name to be inserted into the document before sending it to the database. [39]

```
<extension point="gcc-features-loop-stephenson">
    <output
        db="couchdb" host="server" port="5984"
        user="hleather" pass="mypassword"
        tag="adpcm-compilation-1"/>
</extension>
```

3.2.6.8.7 Filtering functions It may be that a researcher will not be interested in features from all functions in all benchmarks. The features plug-ins can all specify a set of functions to include. The user gives wild-carded[40] lists of functions to get features for. Each element of the list either includes or excludes the functions it names and the elements processed in document order. A typical example might look like:

```
<extension point="gcc-features-loop-stephenson">
    <!-- Initially, all functions are included, but
     if we only want features for a few functions,
     then we start by excluding everything -->
    <exclude main-input-file="*" function="*"/>
    <!-- Now list what to include, two from foo.c
     and all in bar.c -->
    <include main-input-file="foo.c" function="foo"/>
    <include main-input-file="foo.c" function="bar"/>
    <include main-input-file="bar.c" function="*"/>
</extension>
```

3.2.6.8.8 When to Compute Features Features can often be computed at different points in the compilation. *libPlugin*allows features to be computed in response to any event. The `gcc-pass-manager` plug-in contains an event to mark the beginning and end of the execution of every pass and these are the typical events to which feature generation is attached although other events may be used. Each feature plug-in has a default event it will attach to if none is given. It may not be safe to attach to all events, for example, the function to get features for may not have be in the requisite state. At present, *libPlugin*does not sanity check the user's choice of event to see if it is suitable.

[39] There are also ways to add arbitrary JSON content into the document. This can be augmented with the `message` plug-in so that different information can be added for each function.

[40] The wild cards used are POSIX glob patterns

To choose the event to print features for, the user includes a `<when>` element with the identifier of the event's extension point. Multiple events can be given in each extension. [41] If different `tag` attributes or other data should be sent to the database on each event, then different extensions should be used. The `point` will be described in the features. In the following example, the simple basic block instruction histogram features are generated before and after loop unrolling:

```
<extension point="gcc-features-basicblock-instructioncount">
    <when point="gcc-rtl-unroll-and-peel-loops.execute.before"/>
    <when point="gcc-rtl-unroll-and-peel-loops.execute.after"/>
</when>
```

3.2.6.8.9 API All feature plug-ins have an API so that they can be used from other plug-ins. For instance, once a machine learning model has been built, it should be reinserted into the compiler. The model will only be able to make predictions on new benchmarks by seeing the features for those benchmarks. The API allows this to happen relatively easily.

3.2.6.9 Data Dump

*libPlugin*allows full AST or RTL data of functions to be dumped to files. This is useful in situations where the researcher is unsure about what feature should be used, when debugging features or when using automatic feature generation.

The plug-in, `gcc-dump-ast-or-rtl`, prints AST or RTL of functions according to the state the current function is in when the dump is called. The dump is sent only to files, not a database simply because of the volume of data produced. The dump is configured in much the same ways as the features plug-ins; the output file can be specified and events that cause the dump can be given, too, with the same syntax.

3.3 Using *libPlugin*for Machine Learning

We have seen how *libPlugin*makes the compiler easily extensible and have been introduced to a number of useful plug-ins for machine learning in GCC. We can now revisit the example from the introduction of this chapter and discover how the same

[41] Passes can appear more than once in the pass manager's pass tree. However, there will be only one event for the pass which will be fired multiple times per function if the pass is duplicated in the tree. This can mean that it is difficult to separate the features and to identify when the features were really generated. This problem will be investigated in the future.

task would be using *libPlugin*. Recall that in the example our researcher wanted to learn a new model to predict the best loop unrolling heuristics.

Our researcher's first task was to find what loops each benchmark contains. Ideally he would also get the range of unroll factors that can work for each loop. This task is easy enough, he simply loads plug-in, `gcc-print-rtl-unroll-and-peel-loops`, when compiling his code. He will also have to ensure that loop unrolling is enabled (it is not by default) which can be done by putting `-O3` on the command line, making sure that no other conflicting flags are there. Before *libPlugin*our researcher had to understand and alter all of the make files and build scripts of all of his benchmarks. Now he writes a small plug-in which loads the loop printing plug-in and changes the command line in one go:

```
<?gcc version="4.3"?>
<plugin id="find-unrollable-loops">
    <extension point="command-line.modify">
        <remove><arg>-O*</arg></remove>
        <remove><arg>-fno-unroll-loops</arg></remove>
        <insert><arg>-O3</arg></insert>
    </extension>
    <requires plugin="gcc-print-rtl-unroll-and-peel-loops"/>
</plugin>
```

If this plug-in is in the search path then invoking each benchmark's build script will be sufficifient. There will now be a file in the current directory, `gcc-print-rtl-unroll-and-peel-loo`
The file will contain entries like this:

```
<loop main-input-file="foo.c" function="bar" number="1">
    <unrollable type="simple"/>
    <unrollable type="stupid"/>
    ...
    <unrolled type="simple" times="8"/>
</loop>
```

These entries tell the researcher which loops are in the benchmark, whether they can be unrolled with different flavours and how the default heuristic was applied. Compared to the work previously required, our researcher has had to do almost nothing.

Now that he has a list of unrollable loops for each benchmark, the researcher can begin his iterative compilation. He writes a program to parse the unrollable loops file and spit out a plug-in for each point in the iterative compilation search space. This plug-in needs to unroll some loops and add cycle counting to every function.

```
<?gcc version="4.3"?>
<plugin id="<benchmarkname>-1">
    <!-- Still have to turn on unrolling -->
    <extension point="command-line.modify">
        <remove><arg>-O*</arg></remove>
        <remove><arg>-fno-unroll-loops</arg></remove>
        <insert><arg>-O3</arg></insert>
    </extension>

    <!-- Select the unrollings -->
    <extension point="gcc-rtl-unroll-and-peel-loops.override">
        <!-- Start with all loops not unrolled -->
        <loop times="0"/>
        <!-- Unroll one loop per function -->
        <loop main-input-file="foo.c" function="bar" number="2" times="2"/>
        ...
    </extension>

    <!-- Get cycle counts for each function -->
    <extension point="gcc-perfmon.settings">
        <output db="couchdb" host="server" tag="{plugin.id}"/>
    </extension>
</plugin>
```

Now every time the benchmarks are run the profiling data will be recorded to the database. Again, compared to life before *libPlugin*this is very little work. The researcher compiles each point in the space and runs the resulting programs.

With the iterative compilation done, the researcher needs to get features for each loop. He decides to use Stephenson's features to start with, compiling each benchmark with the following plug-in.

```
<?gcc version="4.3"?>
<plugin id="get-features">
    <extension point="gcc-features-loop-stephenson">
        <output db="couchdb" host="server" tag="<benchmarkname>"/>
    </extension>
</plugin>
```

The researcher now has all the information he needs stored away in his database.

The effort needed to achieve this was minimal compared to doing it without *libPlugin*and no changes had to be made to the compiler. He can now scan through the database and learn a model which predicts the best unroll factor for each loop. At this point he will write a plug-in (finally needing some C code) to insert his model into GCC.

```
int machinelearning_unrollTimes(loop* lp) {
    StephensonFeatures features;
    StephensonFeatures_compute(&features, lp);
    return /*Do ML stuff with features*/;
}
```

And he can insert this into GCC with a small plug-in specification:

```
<?gcc version="4.3"?>
<plugin id="machine-learning-unrolling">
    <library path="mlunroll.so"/>
    <extension point="gcc-rtl-unroll-and-peel-loops.decision.around">
        <callback symbol="machinelearning_unrollTimes"/>
    </extension>
</plugin>
```

The new plug-in can be packaged up and sent to any interested researcher who uses *libPlugin*. No users will need to change the compiler to try the new machine learning plug-in.

Chapter 4

Feature Grammars

Recent work has shown that machine learning can automate and in some cases outperform hand crafted compiler optimizations. Central to such an approach is that machine learning techniques typically rely upon summaries or *features* of the program. The quality of these features is critical to the accuracy of the resulting machine learned algorithm; no machine learning method will work well with poorly chosen features. However, due to the size and complexity of programs, theoretically there are an infinite number of potential features to choose from. The compiler writer now has to expend effort in choosing the best features from this space. A novel mechanism is developed to automatically find those features which most improve the quality of the machine learned heuristic. The feature space is described by a grammar and is then searched with genetic programming and predictive modelling.

This chapter describes how we design grammars to define the space of features, while the next explains how the feature space is searched for good features. This chapter is organised as follows. Section 4.2 delves deeper into the the problems of manually written features. Section 4.3 explains how a feature space is formulated. Then section 4.4 shows how features can be generated from the feature grammar and what problems are encountered during this process. Section 4.5 outlines the support required to actively search the feature space before concluding the chapter.

4.1 Introduction

Supervised machine learning needs features to work and to date, whenever machine learning has been applied to compilers, the features have been hand written by a human compiler expert. The expert wants to replace some heuristic with a machine learned

one and looks at all the data available when that heuristic must make its choices. From these data he will need to compute his features. The problem is that because the data is mostly trees and graphs of unbounded size there are an infinite number of possible summaries he can choose from for features. The expert can only try a few and each takes some effort to implement.

Not only is he faced by countless features to choose from, but features that are based on human intuition may not be the most successful because the interaction between features and a machine learning algorithm is complex. If the features do not represent all of the relationship between the program and the desired outcome they may not work sufficiently well with the machine learning algorithm. No machine learning tool will create quality predictions for new programs if there is little to learn from the input examples. In some ways the use of machine learning has pushed the problem from one of hand-coding the right heuristic to one of hand-coding the right features.

Previously, researchers in machine learning for compilers have manually created lists of features they believe reasonable[Cavazos and Moss (2004)]. Many such works use feature selection to remove redundant or unhelpful features. However, none have attempted to search through the feature space, generating entirely new features along the way or even acknowledged the existence of the space itself. In this work, on the other hand, an automated system is allowed to search through an infinite feature space to find features which most improve the machine learning algorithm's performance. In this approach the space of features is represented as a grammar where each sentence from the grammar represents one feature. The human is at last relieved from the burden of worrying about which features are important and which are not.

The main contributions of this chapter are the development of grammar based system to describe a space of features and of search mechanisms which combine the best of previous approaches without inheriting their drawbacks.

4.2 Manual Feature Creation

This section describes some of the problems that the compiler writer must be aware of when writing features by hand. Then the way features are used in previous works is shown to fail to help a machine learning tool to make a good prediction.

4.2.1 Difficulties with Human Created Features

Irrelevant features It is easy to conceive of features that will have no relevance to any optimisation task. Features such as 'the number of comments in the code' or 'the average length of identifiers' would not help a machine learning algorithm. These examples are so obvious that no human would suggest them but the same situation arises in more subtle cases where a sensible sounding feature simply has no bearing on the current optimisation.

More serious is the case when a feature is useful on its own but when added to an existing set of features does not show any additional improvement. This can happen when the all of the useful information in a feature is already present in the other features, for example if the feature is a linear combination of the others.

The compiler writer can somewhat mitigate the damage of irrelevant features by applying feature selection to weed out the unnecessary features. However, the features may also introduce noise which selection algorithms will fail to remove and which can mislead the machine learning tool.

Classification clashes Two distinct programs may have the same feature vector but different best values for the heuristic; a machine learning algorithm will predict at least one of them wrongly. This does in fact happen in practice as shown in Monsifrot et al. (2002). The presence of these clashes is a clear indication that the features are inadequate since they cannot distinguish examples from different classes. However, irrelevant features and noise can obscure classification clashes.

When clashes are found they place an upper bound on the accuracy of the models created by the machine learning tool. It is possible that adding other features may help to separate the features by adding other dimensions.

Classifier peculiarities A set of features that performs well for one machine learning algorithm might not be good for another (Kohavi and John (1997)). In other words features are not independent of the learning technology.

Beyond simple features Once the 'obvious' features have been written, inevitably they do not completely represent the relationship between the programs and the desired heuristic values. The expert must then choose which additional features to implement. The sheer number of choices can be huge and the expert will find that each stage, as they increase the complexity of their features they face the same difficult challenges

as they did before, only now multiplied due the larger set of features they must work with.

4.2.2 Motivating Example

```
for (i=0;i<EXP_TABLE_SIZE-1;i++) {
    l->SpotExpTable[i][1] =
        l->SpotExpTable[i+1][0] -
        l->SpotExpTable[i][0];
}
```

(a)

Method	Unroll	Cycles	Speedup	% of Max
Baseline	0	406,424	1.0000	0%
Oracle	11	328,352	1.2378	100%
GCC Default	7	418,464	0.9712	-12%
GCC Tree	2	392,655	1.0351	14%

(b)

Figure 4.1: Loop from MediaBench (a) and speedups using various schemes (b). GCC's default heuristic selects an unroll factor of 7 causing a slowdown. Using GCC features and machine learning, an unroll factor 2 is selected giving a small improvement.

This section demonstrates that selecting the right features can have significant impact on optimisation performance. Consider the loop in figure 4.1 selected from the `mesa` benchmark within *MediaBench*. If GCC's default loop unroll heuristic (labeled GCC Default in figure 4.1 (b)) is applied, it determines the best unroll factor is 7. When executed on the Pentium this achieves a slowdown of 0.97. However, if all loop unroll factors up to 15 are exhaustively evaluated, then the best unroll factor is found to be 11 resulting in a speedup of 1.24 as shown by the Oracle entry in figure 4.1 (b). If GCC's heuristic is replaced with a machine learning decision tree algorithm, whose features are the same information used by GCC's heuristic (as shown in figure 4.2 (a)), then it is possible to achieve a speedup of 1.04 or 14% of the maximum available. Figure 4.2(b) shows the path followed by the learned decision tree heuristic leading to the unroll factor of 2 being selected. It will be discovered in the next chapter, in section 5.3,

that by automatically searching a feature space, features which allow the full speed up to be selected can be found.

Name	Value
ninsns	10
av_ninsns	9
niter	6.14E17
expected_loop_iterations	49
num_loop_branches	1
simple_p	1

(a)

```
if( ninsns <= 63 )
 if( simple_p > 0 )
   if( num_loop_branches <= 3 )
     if( av_ninsns > 5 )
       if( niter > 6.1384926724882432E17 )
         if( expected_loop_iterations > 8 )
           if( niter <= 6.1428835034542899E17 )
             if( num_loop_branches <= 1 )
               unrollFactor = 2;
```

(b)

Figure 4.2: The path through the learned GCC tree heuristic (b) for the example in figure 4.1 and the features used in that path (a).

4.3 Defining the Feature Space

This section explains how feature grammars are used to describe a feature space. A toy compiler language is presented first and features are defined for it. It is shown how different areas of the feature space can be prioritised over others, in section 4.3.2, and how features are computed against the compiler's IR, in section 4.3.3. Implementation of advanced feature spaces that are difficult to define with a context free grammar alone is covered in 4.3.4

```
<feature> ::= "countNodesMatching(" <matches> ")"
<matches> ::= "isConstant" | "isVariable" | "isAnyType"
            | ("isPlus" | "isTimes")
              ("&& leftChildMatches(" <matches> ")")?
              ("&& rightChildMatches(" <matches> ")")?
```

Figure 4.3: A simple feature grammar. Sentences from the grammar are expressions which can be evaluated over statements from the toy language. Each feature counts the number of sub trees in the toy language statement which match a pattern.

4.3.1 Features for a simple language

A toy example is presented to show how the process works. The toy language allows only sets of assignment statements; the left hand side of each will be a variable name; the right will be an expression containing variables, constant integers, operators '+' and '*' and parentheses. How to parse this language is of no concern, therefore it is assumed that the ambiguity in operator precedence has been suitably dealt with. Only in the intermediate parse trees are interesting.

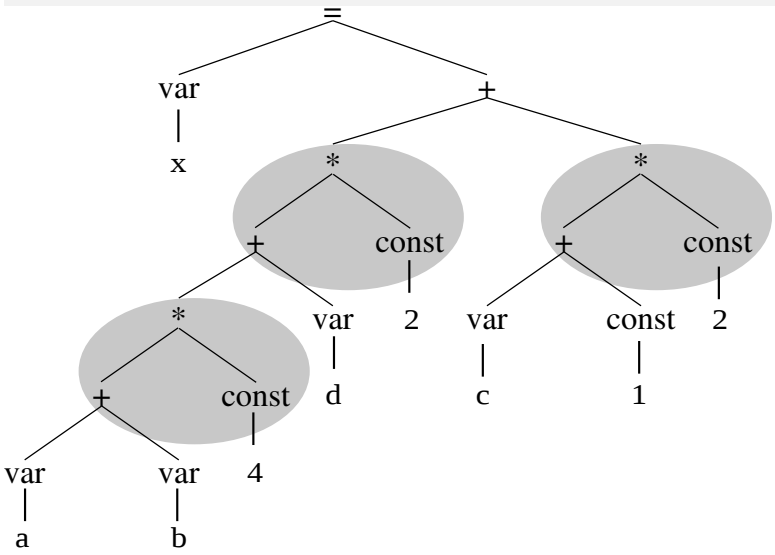
Example statements from the language are:

- a = 10
- b = 20
- c = a * b + 12
- d = a * ((b + c * c) * (2 + 3))

A human compiler expert, when thinking of features to be computed over expressions will most likely devise simple features like, *the number of '*' operators in the expression* or *the depth of the expression*. He will implement and test these features and with luck will discover that they are somewhat helpful to machine learning but in all likelihood do not perform as well as he had hoped. He will then, probably, create small variations of his original features. For example, to expand on his count of multiply nodes he may think that another feature could be to count those multiply nodes which have as their left child a '+' operator and whose right child is constant. He can add this feature to his set and try his machine learning tool again.

There are an infinite number of these features, however, as the compiler expert may choose to further refine his new feature by specifying the further children of the

```
countNodesMatching(
    isTimes &&
    leftChildMatches(
        isPlus
    ) &&
    rightChildMatches(
        isConstant
    )
)
```



x = ((a + b) * 4 + d) * 2 + (c + 1) * 2

Figure 4.4: An example feature from the grammar in figure 4.3. In the right hand of the figure is a sample AST from the tiny language, showing the matching sub structures.

The feature thus evaluates to three.

'+' node, restricting the set of values for the constant or any number of complex modifications that could be dreamed up. The expert will repeat this process, continually implementing and testing more features until either no further improvement in the machine learned model is found or he runs out of time to experiment.

The approach taken here is an automation of this labour intensive process. Since this system will explore the space of potential features, it must know what that space is. The space of features is described by means of a grammar where the language accepted by the grammar is some subset of an existing programming language. Each sentence from the grammar will be an expression which can compute the value of a feature when run over the compiler's internal representation of the current code section.

Figure 4.3 shows a simple grammar describing a set of such features in a pseudo-code style. These example features can be computed on expressions from the toy language and, in this case, each feature counts the number of sub-trees in the expression which match a pattern. Figure 4.4 shows an example feature from this grammar and an evaluation against a sample program fragment, showing the matching sub-trees. Applying this particular feature to this piece of code yields the value three.

Now that the limits of feature spaces can be defined, it can be shownhow to specify relative preferences for parts of the space.

4.3.2 Production Weighting

There are times when the researcher believes that some part of the feature space is more likely to useful than others. He would like to be able to influence the grammar to reflect his interest in different portions of the feature space. The grammar definition language allows productions to be weighted, changing the relative probabilities of productions. This makes the grammars probabilistic context free grammars (pCFGs) and allows the researcher to express their interest in some features over others.

Figure 4.5 shows a simple example where some productions are annotated with weights. Now, when choosing between productions for the `<matches>` rule, the first two will be ten times more likely than before. The researcher has changed the probability that some parts of the space will be explored compared to others.

Another example is shown in figure 4.6. The two grammars in that figure both describe the set of decimal digits, `"0"` to `"9"`. However, because of their construction they have very different preferences for different digits. In the first, figure 4.6(a), since the productions are chosen with a uniform probability, the digits will appear with equal

```
<feature> ::= "countNodesMatching(" <matches> ")"
<matches> ::=[weight=10] "isConstant"
            |[weight=10] "isVariable"
            | "isAnyType"
            |  ("isPlus" | "isTimes")
               ("&& leftChildMatches(" <matches> ")")?
               ("&& rightChildMatches(" <matches> ")")?
```

Figure 4.5: A small modification to the grammar of figure fig:simple-feature-grammar, showing weighted productions. In this example, the researcher has decided that features testing for constants or variables should be ten times more likely than others.

likelihood. In figure 4.6(b), however, the characters "8" and "9 are equally likely, but "7" is twice as likely as those, "6" is twice as likely as "7" and so on until "0", which will be chosen with probability 0.5 is 256 times more likely to be chosen than "8" or "9.

The grammars of figure 4.6 demonstrate that precise form of a grammar can have a dramatic effect on preferences for different parts of the space. Sometimes the most natural way of defining the grammar would lead an unacceptable bias for some features over others. It is too much to ask that the structure of the grammar be redefined to even out these issues, and instead weighting can be used to adjust the grammar back into what the researcher was looking for. In figure 4.7, the grammar from figure 4.6(b) has been weighted to have the same probabilities as the grammar in figure 4.6(a).

4.3.3 Feature Evaluation

The feature grammar defines a set of features, each of which is a sentence from the grammar. Once a feature has been extracted from the grammar it will need to be evaluated against the compiler's internal data to compute the feature values. A grammar is able to produce sentences which are a subset of some particular programming language. In this way an interpreter or compiler for the features already exists in the form of whatever compilers or interpreters there are for the underlying language or if one does not exist, it is at least an orthogonal problem.

To give a concrete example, figure 4.8 shows how the grammar from figure 4.3 (which was only presented in pseudo-code) might be implemented in C. Every sentence from the grammar is a valid C program which can be compiled with any standard

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

(a)

```
<digit> ::= "0" | <d1>
<d1>    ::= "1" | <d2>
<d2>    ::= "2" | <d3>
<d3>    ::= "3" | <d4>
<d4>    ::= "4" | <d5>
<d5>    ::= "5" | <d6>
<d6>    ::= "6" | <d7>
<d7>    ::= "7" | <d8>
<d8>    ::= "8" | "9"
```

(b)

Figure 4.6: Two different grammars for choosing a decimal digit. Both grammars recognise exactly the same language but they have significantly different preferences.

```
<digit> ::= "0" |[weight=9] <d1>
<d1>    ::= "1" |[weight=8] <d2>
<d2>    ::= "2" |[weight=7] <d3>
<d3>    ::= "3" |[weight=6] <d4>
<d4>    ::= "4" |[weight=5] <d5>
<d5>    ::= "5" |[weight=4] <d6>
<d6>    ::= "6" |[weight=3] <d7>
<d7>    ::= "7" |[weight=2] <d8>
<d8>    ::= "8" | "9"
```

Figure 4.7: A weighted grammar, similar in structure to that of figure 4.6(b) but which has the equal probability of choosing each digit just as in figure 4.6(a).

```
<feature>  ::=
      "#include <stdio.h>"
      "enum Type {CONSTANT, VARIABLE, PLUS, TIMES};"
      "struct Node {int type; Node* left; Node* right};"
      "bool match( Node* ast, Node* matcher) {"
      "   if(matcher == null) return true;"
      "   if(matcher->type != ast->type) return false;"
      "   return match(ast->left, matcher->left) && "
      "          match(ast->right, matcher->right);"
      "}"
      "int countNodesMatching(Node* ast, Node* matcher) {...}"
      "Node* readAST(char* filename) {...}"
      "int main(int argc, char* argv[]) {"
      "   Node* node = readAST(argv[1]);"
      "   printf('%d\n', countNodesMatching(" <matches> ");"
      "   return 0;"
      "}"
<matches> ::= "new Node(CONSTANT, null, null)"
           | "new Node(VARIABLE, null, null)"
           | "null"
           |  "new Node(" ( "PLUS" | "TIMES" ) ","
                          <matches> "," <matches> ")"
```

Figure 4.8: A version of the grammar from figure 4.3 in C. A few details have been elided for clarity. Any sentence from the grammar is a C program which can be compiled and run. When given an AST from the toy language, the C program will print the computed feature value.

C compiler. When run on an AST data file of a program from the toy language, the feature program will print out the computed value of its feature.

Features can be arbitrarily complicated, since typically the underlying language is Turing equivalent as long as the desired features can be expressed through the context free grammar. The next section deals with how to handle cases were the restriction to be context free is too limiting.

```
1 int feature(int A[]) {
2     int s = 0;
3     for(int i0 : A) {
4         for(int i1 : A) {
5             s += i0 + i1
6         }
7     }
8     return s;
9 }
```

Figure 4.9: A feature over arrays, written in Java. If the feature were generalised so that the loop nest could be of increasing depth, then this would not be representable as a CFG. CFG's do not support these semantics for sentence generation any more than they do for sentence parsing.

4.3.4 Semantic Actions

Context free grammars are subject to a number of limitations which restrict the types of sentences that can be created. For example, consider the feature in figure 4.9, which can be computed over arrays of integers. Suppose that the intention is to have features which allow deep nests of such for loops with a computing expression in the body of the innermost loop; the number of variables available to the innermost expression increases with every containing loop. This cannot be expressed with a CFG because a CFG cannot convey the semantics involved.

The situation has parallels in the world of program parsing. There a CFG is used to describe the syntax of a language and the semantics of the language are embedded as 'semantic actions' (Aho et al. (1986)) in the grammar. In parsing, these actions allow arbitrary code to be run during the parsing process; for example, symbol tables are updated and checked.

The system has a similar mechanism, allowing the grammars to produce more complicated features. Semantic actions are embedded in the grammar and whenever a production is selected the actions are run in the appropriate place. These semantic actions are snippets of arbitrary code which can update state and print values. For example, figure 4.10 shows a grammar which generalises the feature from figure 4.9. The first action in line 1 initialises a depth counter to zero; this action is executed before any rules are expanded. The next action is in line 7 which prints the current variable name and increments the depth. The final action, in line 13 prints a random variable name

```
1  {int depth = 0;}
2  <feature> ::= "int feature(int A[]) {"
3                "    int s = 0;"
4                <nest>
5                "    return s;"
6                "}"
7  <nest>    ::= "for(int " {print("i" + depth); depth++} ": A) {"
8                <nest>
9                "}"
10             | "s += " <expr>
11 <expr>    ::= <expr> " + " <expr>
12             | <var>
13 <var>     ::= {print("i" + random(0 to depth-1))}
```

Figure 4.10: A feature grammar with semantic actions, generalising the feature from figure 4.9. Semantic actions are found between braces.

from those available.

Semantic actions can be placed on entry to or exit from the whole grammar, rules and productions as well as inside productions as shown by the example.

Another place where semantic actions are useful is in setting production weights. Recall that each production in a pCFG can have a weight to modify the relative probabilities of a rule's productions. A previous example showed that the weights could be set to constant values, but in fact, they can be calculated from arbitrary code. If, for example, the loop nest should become more likely to terminate the deeper it got and not to be deeper than 5, the following weights could have been put on the productions at lines 7 and 10:

```
7  <nest>    ::=[weight=depth>5 ? 0 : 1] "for(int " {print...}
10             |[weight=depth * 2]  "s += " <expr>
```

These additional capabilities need only be used sparingly but allow extremely detailed and powerful control over the features that can be produced.

4.4 Generating Features from Grammars

This section discusses how features can be randomly created from a grammar and the design challenges that creates. The main challenge involves ensuring that sentence creation finishes because it is quite possible to construct infinitely recursive grammars.

How to expand a feature from the grammar is described in section 4.4.1. The causes of infinitely recursive grammars are covered in 4.4.2. The way to solve the problems of infinite recursion is discussed in 4.4.3 and then the consequences of that solution are talked about in section 4.4.4.

4.4.1 Feature Expansion

Now that there is a grammar describing the space of features, any number of features can be generated from it. One need merely start at the root rule of the grammar (which, in the case of the grammar in figure 4.3, is rule `<feature>`) and expand any non-terminals in it. Whenever there is a choice of production to expand they are chosen from randomly using roulette wheel selection so that the probability of choosing each production is proportional to its weight. By continuing until there are no more non-terminals left in the sentence there will be a finished feature.

For the example in figure 4.4, a derivation is given in figure 4.11. Step one starts with the root rule of the feature grammar - placing a single non-terminal as the current sentence. In step two the non-terminal is replaced by the only possible rule, leaving still only one non-terminal to be replaced. In step three the `<matches>` non-terminal is replaced; there are five productions and the last is randomly selected; the non-terminal is replaced with value of the production giving two non-terminals to replace. Finally, in step four the remaining `<matches>` non-terminals are replaced.

4.4.2 Problems of Recursion

Whilst ambiguities cause problems in parsing sentences from grammars, sentence production suffers from infinite recursion. Perhaps the simplest example of this is in the grammar in figure 4.12 which obviously produces an unending string of '*a*'s. Attempting to generate a sentence from this grammar will lead to some discomfort. The grammar definition language allows embedded actions, similar to semantic actions in parser generators, which allow such problems to be manually broken - by, for example, imposing a depth limit on the recursion. However, in practice such grammars are

```
1. <feature>
2. "count-nodes-matching(" <matches> ")"
3. "count-nodes-matching(
   is-times &&
   left-child-matches(" <matches> ") &&
   right-child-matches(" <matches> "))"
4. "count-nodes-matching(
   is-times &&
   left-child-matches(is-plus) &&
   right-child-matches(is-constant))"
```

Figure 4.11: Derivation of the example feature from figure 4.4.

```
<A> ::= <A> "a"
```

Figure 4.12: Infinitely recursive grammar. The only sentence the grammar recognises is an infinite sequence of as. Attempting to generate a sentence from this grammar will never finish because there will always be a non-terminal <A> left un-replaced.

unlikely to be seen.

More subtle recursion issues are caused probabilistically. Consider the grammar in figure 4.13. The language it recognises consists of odd numbers of consecutive 'a's. However, if the two productions are chosen from uniformly at random, then it is likely to get very long strings. The probability that a string of n non-terminals, $AAA...A$, will contain fewer non-terminals after each is expanded once is given by $I_{1/2}(2n/3, n/3+1)$, where I is the regularised incomplete beta function. As strings contain more non terminals they become increasingly likely to grow at each expansion. To solve these issues, the grammar system allows productions to be weighted as described next in section 4.4.3.

4.4.3 Avoiding Runaway Sentence Expansion with Production Weights

Explosive sentence lengths can be avoided by changing the probabilities of different productions. If, in the example from listing 4.13, the weight of the first production had been less than one third of the weight of the second production, then the expected length of a sentence after replacing each non-terminal once would be less than the original; the sentence expansion would not explode.

```
<A> ::= <A><A><A> | "a"
```

Figure 4.13: Probabilistically recursive grammar. At any point in the expansion of the sentence it is possible for the all non-terminals to be replaced with terminals. However, since the two productions are chosen with equal probability and the first production generates so many recursive non-terminals, there will most likely be an explosion of non-terminals and the sentence will become longer and longer.

Deciding the appropriate weights for productions is not generally possible to do analytically. This is due to the presence of semantic actions (see section 4.3.4) which introduce arbitrarily complex code into the grammar expander. However, as can be seen from figure 4.14, once weightings create a non-explosive grammar there is relatively small sensitivity to the weight values. Thus, it is quite easy to be conservative with weightings, the grammar will not suffer much for it; trial and error produces acceptable results with very little time or effort.

4.4.4 Short Sentence Bias

As described in the previous section, the grammar must weight productions to prevent runaway, explosive sentences. A consequence of this is that grammar system is biased towards short sentences. Figure 4.14 shows the sentence length bias for the grammar in listing 4.13. It can be seen that short sentences are produced, even when the productions are weighted close to the threshold at which run away expansion begins. For this grammar, the vast majority of sentences will be shorter than ten characters long.

Creating grammars that give rise to mostly short sentences has some drawbacks. In essence, by avoiding the grammar exploding one ends up preferring short sentences and cannot explore much of the grammar. Often, after generating a few thousand random sentences, the system is typically recreating sentences that have already been seen; exploration effectively stops.

Overcoming the bias toward short sentences is a side effect of searching the feature space, which will come be covered in the next section 4.5.

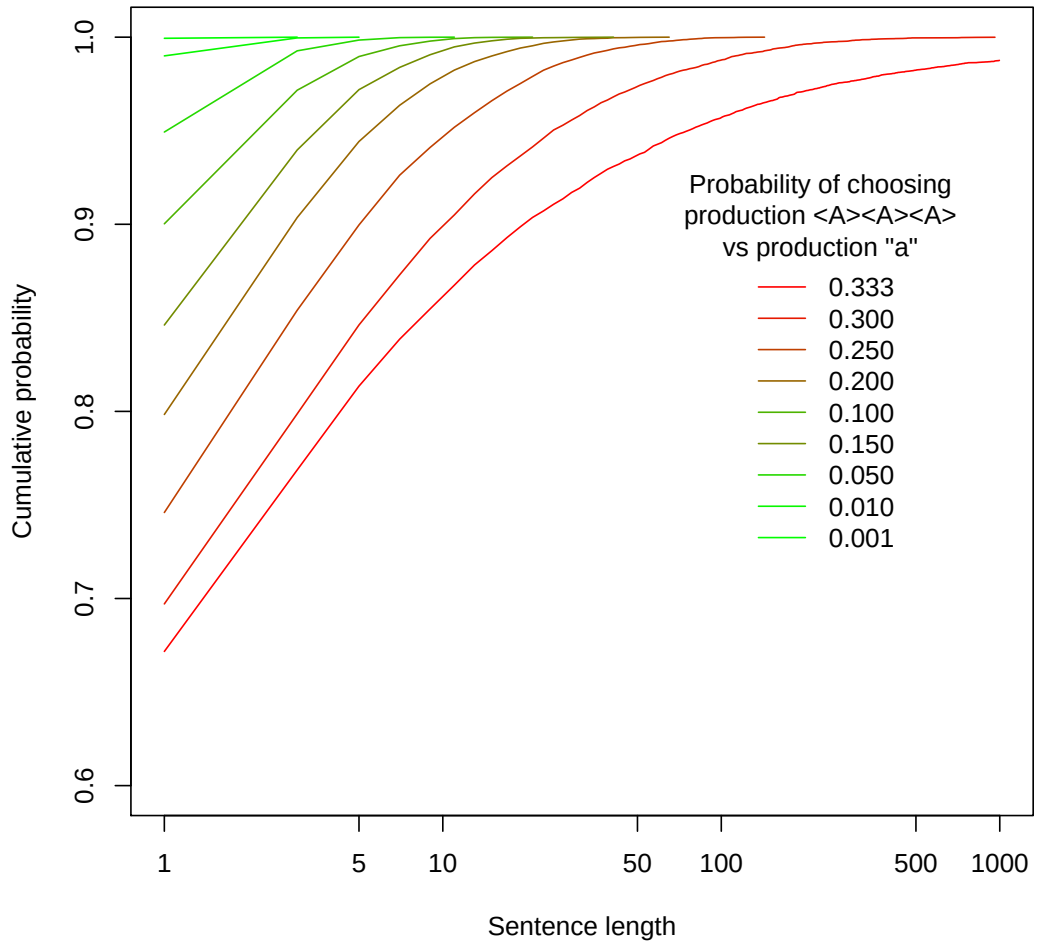


Figure 4.14: Cumulative probability of getting a sentence of a given length when expanding grammar `<A> ::= <A><A><A> | "a"` with different weights for the two productions. Even as the weights approach the beginning of runaway sentence generation (which first happens when production `<A><A><A>` is on third as likely as production `"a"`) there is significant bias towards short sentences.

The graphs also help to explain why setting weights by trial and error is so easy. Being conservative with the weights does not have a large effect on the sentence length; they will always be short for feasible weights.

4.5 Support for Searching the Feature Space

The simplest way to search the feature space is to randomly generate features as described in section 4.4. Thousands of features can be created in a matter of seconds which are then trained and evaluated. The bias towards short sentences, however, (see section 4.4.4) means that this approach will be practically limited to a small portion of the complete feature language accepted by the pCFG. Early experiments found that the amount of the feature space that could be reached was much smaller than expected.

One might try to use semantic actions to alter the bias of the feature space as the search begins to saturate its exploration of short features. This, however, is difficult to arrange, placing a heavy burden on the writer of the grammar to add large amounts of fragile code that are not directly concerned with describing the feature space. One could also arrange the weights of productions to favour long sentences. However, this quickly leads to runaway sentences in the feature generator. Even if the generator is rigged to bail out when a sentence becomes too large the generator then spends most of time creating features that fail or that have already been seen before.

Fortunately, the problem of short sentence bias is solved as a side effect of different search techniques than the naïve random approach. Suppose there are some feature to start with, chosen from the biased space. Small modifications can be made to it; possibly shortening it, maybe lengthening it. The feature will no longer be bound by the imposed bias of the pCFG (the bias will now only inform where the search should start, not limit the scope of that search).

The next section, 4.5.1, describes the trees that are searched over. Discussed, in section 4.5.2 is how the trees are repaired after modification; which is required for search operators which modify the trees and are themselves introduced in section 4.5.3. Finally, in section 4.5.4 discusses the differences between our system and other, older systems.

4.5.1 Choice Trees

The entities that are searched over in the system are the trees of choices that are made during the construction of a sentence, or feature. For each rule there are some number of productions that must be chosen between and depending on that choice, further choices may be made if the selected production contains other non-terminals to be expanded. These choices can be arranged as a tree where each node in the choice tree corresponds to a non-terminal in the parse tree.

A choice tree encodes the choices that were made during the generation of a feature or sentence from a pCFG. Each sub-tree describes the choices made for the expansion on a single rule and its children. Each node contains the random bits that were used to select the production from set of productions for one rule during the expansion. For these purposes, it is assumed that the grammar is written in Backus Naur Form; i.e. that each production consists only of terminals and non-terminals, without grouping, Klene stars or other extensions. The random bits will be used to perform a roulette wheel selection of the productions so that the probability of selecting each is proportional to its weight.

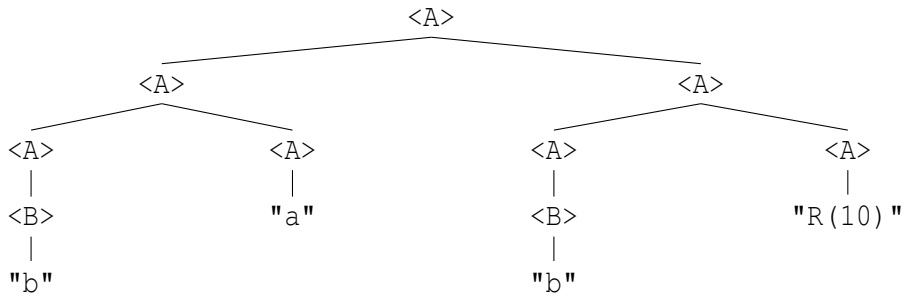
An example choice tree is given in figure 4.15. A simple grammar is given in the first block, 4.15(a). Note that the last production of the first rule contains a semantic action which requests random bits. Next, in 4.15(b), is a possible derivation of sentence `babR(10)` starting from `<A>`, where in each step one non-terminal is replaced with a production. Finally, in 4.15(c), a choice tree for the preceding derivation . The first `<A>` rule has four choices so a random number is generated, `0xdc` (for simplicity these are assumed to be just one byte long). This is used for roulette wheel selection among `<A>`'s four choices (since all productions have unit weights roulette selection is equivalent to the modulo function) to pick the first production, `<A>` $\rightarrow$ `<A><A>`. Subsequent nodes in the tree represent the remaining replacements of non-terminals. In the cases where the `<B>` non-terminals are replaced there is no choice; denoted by `X`. Note that the last leaf node corresponds to the production with a semantic action, so the node has one set of random bits for the production choice and one set as used by the semantic action. The choice tree itself consists only of the nodes and their random bits.

Choice trees can be used in two complimentary fashions; one records the choices made during the expansion of a feature and once recorded, it can be used to replay those choices to recreate the exact same feature as before. During the recording any random bits are remembered and sub-trees created according to the rules and productions of the grammar. When replaying, the source of random bits is replaced by those previously recorded.

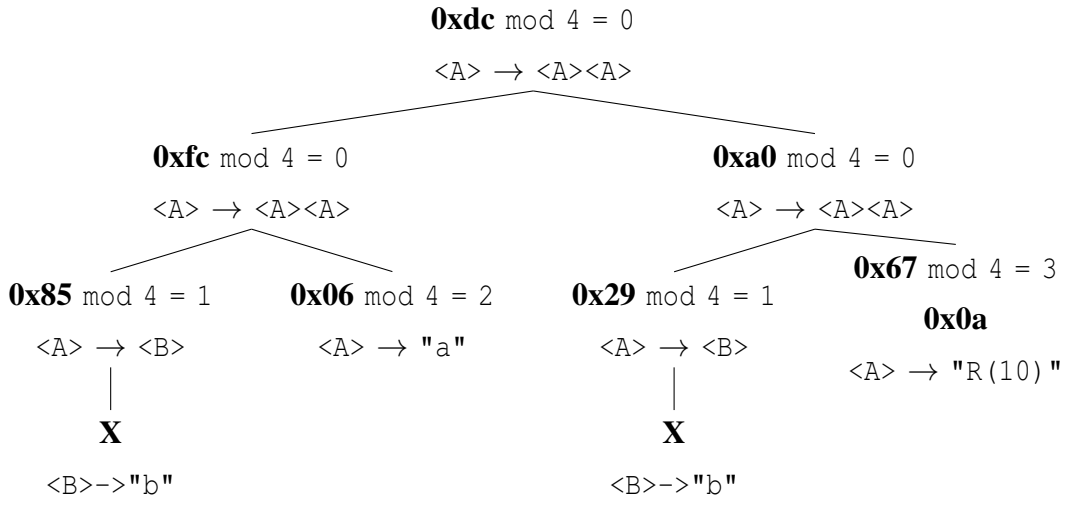
It should be noted that semantic actions (see section 4.3.4) can make use of random bits. This means that additional bits may be required when a particular production is chosen or during the evaluation of a production's weight. These bits are simply appended to the random bits used to choose the production during recording and are delivered up during replay.

```
1  <A> ::= <A><A> | <B> | "a" | {print("R("+randomInteger+")")}
2  <B> ::= "b"
```

(a)



(b)



(c)

Figure 4.15: An example choice tree. A simple grammar is shown in (a) . A possible derivation of sentence `babR(10)` is given in (b). (c), is a choice tree for the derivation in (b).

4.5.2 Repairing choice trees

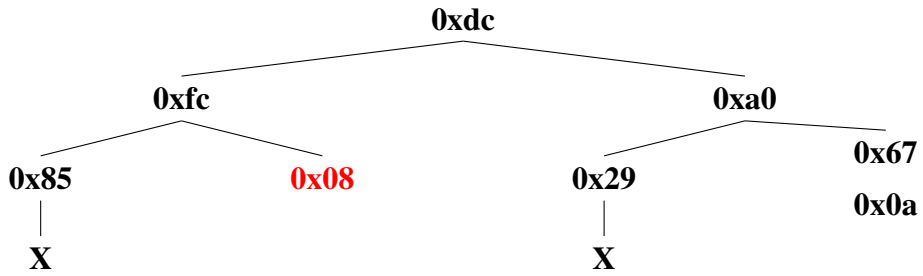
The search operators in the following section perform modifications to the choice trees. They may change a choice tree so that there are insufficient random bits to complete the replaying of the tree to create a feature. This section describes how these trees are repaired so that they are always valid; no search operation, no matter how drastic can create an invalid tree.

The mechanism for repairing trees is as follows. During playback, whatever random bits are present in the tree in the correct places are made use of. However, if at any point there should be missing information will cause a change from playback mode to creation mode until the required information is made up. In this way, reading a choice tree can alter the tree as a side effect, but at no point is the tree starved of information.

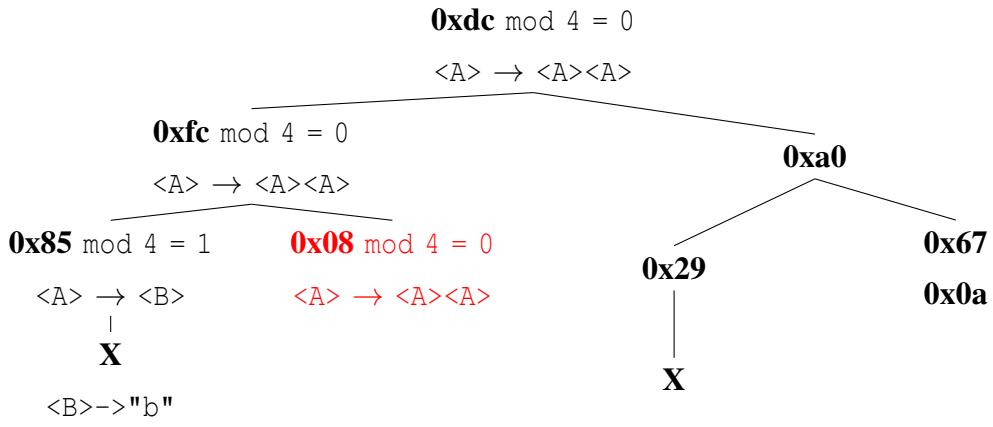
Figure 4.16(a) shows the choice tree from figure 4.15 that has been mutated so that the random bits of one node have changed (in red). The system begins to replay the tree (b), expanding the root non-terminal making choices according to the data in the tree. This proceeds just as normal until the mutated node is encountered. At this point the random bits in the selected node choose a different production from the one in the original tree; the `<A>` becomes `<A><A>` not `"a"`. Now, if the tree were complete there should be two child nodes beneath the mutated node, which are missing. To solve this issue the system begins to randomly create any nodes it needs, building a new sub-tree, rooted at the mutated node (c). Once the returned from repairing the mutated node, the system begins replaying just as normal, yielding a complete feature and updating the tree so that it is now complete and has remembered the new information needed to repair the tree.

It may also be found that modifications to a choice tree increase the information in the tree beyond what is needed. This can happen if bits which would select a node with many children are changed so that the node will only have one child, or additional bits are added to a node that are not used for the selected production. This does not damage the choice tree, the additional data simply has no effect on the feature produced from the tree.

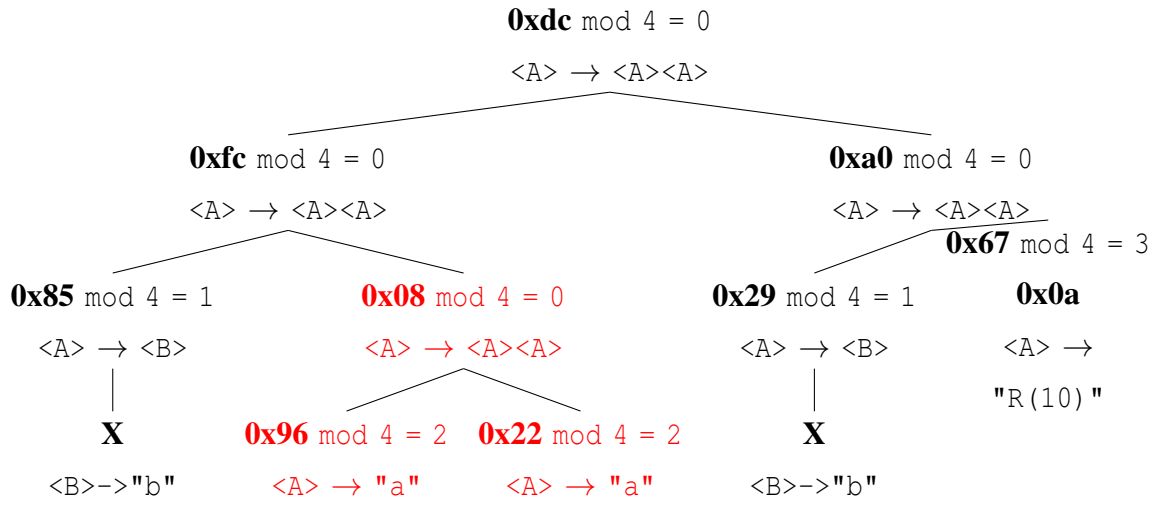
The system offers two modes of replaying a choice tree; in the first any extraneous data are not removed from the tree, in the second, extra bits and nodes are pruned from the tree. In some search techniques (especially genetic algorithms) allowing genetic information to contain redundant information is considered good practice since it is believed to mirror the biological counter parts. Indeed, in genetics, redundant data are



(a)



(b)



(c)

Figure 4.16: Repairing a choice tree. In (a) the choice tree from figure 4.15 has one node mutated. Replay begins reading the choice tree in (b) as normal until the mutated node is reached when a different production is chosen. The new production needs more data than the tree contains, so that is generated and recorded in the tree (c). There after the remains of the tree are unchanged.

given their own term, introns. Both capabilities are provided in the system.

Now that it has been seen that no change to choice trees can damage them beyond repair, the next section looks at the types of search operators the system provides over choice trees.

4.5.3 Search Operators

The system offers several search operators suitable for genetic programming, hill climbing or other techniques. Operators on choice tree nodes include:

- Deletion - The sub tree will be replaced with a new random tree during replay.
- Change random bits - The node may now select a different production; this may cause some children to be missing or irrelevant.
- Shuffle children (several variants) - If the children are from unrelated non-terminals then this might equivalent to deleting all the children and recreating them (although with some pre-specified random bits). If children are related, such as children of many binary operators, then their orders may be changed.
- Crossover sub-trees - Two sub trees from two choice trees are swapped over. This allows information to be shared across choice trees as is required for genetic programming techniques. There are two main variants of this operator: the first is oblivious, it chooses any sub trees from the two choice trees; the second is more targeted - it first expands both trees, remembering which the names of the rules that each node in the trees relates to. Then it prefers to select two sub-trees which relate to the same rule. Both of these main variants have parameters specifying likelihoods for sub-trees to be chosen, based for example on the depth or node count of the sub-tree.

4.5.4 Comparisons to Other Systems

The grammar system allows expressions and programs which are used for features to searched over. Other systems allow programs to be searched over, too. Perhaps the most famous is Genetic Programming (GP, Koza (1990a)). GP, as described in the Related Work chapter, builds an expression tree where each node must have the same return type. This is very restrictive, giving difficulties when there are different types in the system, like booleans, floats and integers. GP also suffers from serious problems

extending to anything other than expression trees; defining functions and loops, for example, have to be handled as special cases which quickly become unwieldy. GP does, however, offer locality for for changes to the expression trees; a modification in one part of the tree affects only that part of the tree, not others.

Most similar to our approach is Grammatical Evolution (GE, Ryan et al. (1998)). Here, a grammar is used giving the same flexibility as we do and also avoiding the problems of GP. However, because the GE codon stream is read sequentially, changes to it that occur during search destroy locality. The ideal would be to have a change in any node affect only the sub-tree rooted at that node. This is not the case in GE, a change early in the codon stream can make every other node subsequently generated be completely different. This causes problems for GE searches since the majority of mutations cause massive damage and the search degenerates into a random search (which is not the case with GP). Our system, based on choice trees suffers no such problems. Any change to a sub-tree in the choice tree modifies only the corresponding sub-tree in the resulting parse tree. In this way, our system combines the good searchability of GP with the expressive power of GE.

4.6 Summary

This chapter has shown how manually writing features is error prone and difficult and how the space of possible features is infinite. A method of describing families of features by using grammars was introduced and numerous associated problems explored and resolved. Finally, the elements essential to searching the feature space were described. The next chapter explains how to search the feature space in practice and details an experiment that demonstrates the effectiveness of the system.

Chapter 5

Searching for Features

The last chapter showed the failings of human derived features for machine learning in compilers and explained how families of features can be built from grammatical descriptions. In this chapter we search the space of features for good features which most improve the machine learning algorithm's performance. Our system is allowed to range over an infinite space of features with a smart genetic programming methodology. Although genetic programming has been used before to search over the model space (Stephenson and Amarasinghe (2005)), this is the first time it has been used to generate features.

We evaluated our technique on an extensively studied problem: loop unrolling (Monsifrot et al. (2002)). Loop unrolling is an optimisation performed by practically every modern compiler and we study its effect in a widely used open-source compiler, GCC. Furthermore, machine learning has been successfully applied to loop unrolling (Stephenson and Amarasinghe (2005)) allowing direct comparison. Given the mature nature of this problem, it should be a challenging task for a new technique to show additional improvement.

The remainder of this chapter is organised as follows. Section 5.1 presents an overview of our system explaining the major components and how the features grammars from the last chapter are used within it. Then in section 5.2 the feature grammar uses for GCC experiments is laid out, followed in section 5.3 by a demonstration of the need to generate features. Our experimental setup, methodology and results are presented in sections 5.4, 5.5, and 5.6 respectively. This is followed by some concluding remarks.

The main contribution of this chapter is an automated feature grammar generation system and its application to loop unrolling in GCC.

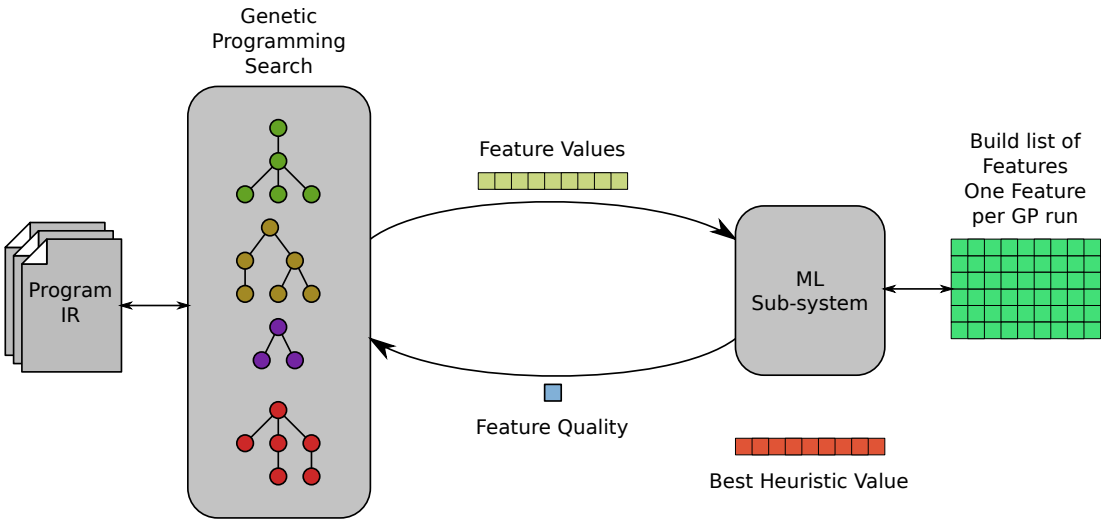


Figure 5.1: Overview of the system.

5.1 Overview

This section presents a high-level overview of our system, illustrated in figure 5.1. The system is comprised of the following components: *training data generation, feature search* and *machine learning*. The training data generation process extracts the compiler's intermediate representation of the program (as described in section 5.2) plus the optimal values for the heuristic we wish to learn. Once these data have been generated, the feature search component explores features over the compiler's intermediate representation (IR) and provides the corresponding feature values to the machine learning sub-system. The machine learning sub-system computes how good the feature is at predicting the best heuristic value in combination with the other features in the base feature set (which is initially empty). The search component finds the best such feature and, once it can no longer improve upon it, adds that feature to the base feature set and repeats. In this way, we build up a gradually improving set of features.

5.1.1 Data Generation

In a similar way to existing machine learning techniques, we must gather a number of examples of inputs to the heuristic and find out what the optimal answer should be for those examples. Each program is compiled in different ways, each with a different heuristic value. We time the execution of the compiled programs to find out which heuristic value is best for each program. Due to the intrinsic variability of the execution

times on the target architecture, we run each compiled program several times to reduce susceptibility to noise (see section 5.4).

We also extract from the compiler the internal data structures which describe the programs. The feature search component will generate summaries of these data as candidate features. Typical data will be the abstract syntax tree, looping structures, use-def chains, etc. Whatever information can be extracted should be recorded, including whatever analysis are performed by any existing heuristics. This process is described in detail in the previous chapter, section 5.2.

5.1.2 Feature Search

The feature search component maintains a population of feature expressions, represented as choice trees (described in section 4.5.1). The expressions come from a family described by a grammar derived automatically from the compiler's IR. Evaluating a feature on a program generates a single real number; the collection of those numbers over all programs forms a vector of feature values which are later used by the machine learning sub-system. The grammars and their use are discussed in section 4.3.

The search component uses a genetic search over choice trees using the search operators described in section 4.5.3. These allow genetic mutations and matings of the choice trees. A population of choice trees is kept and each is sorted according to a fitness function. After the trees are ranked, a new population of the same size is created. Each member of the new population is created by mutations and matings or, rarely, by just simple copying of the members of the previous population. The selection process to determine which individuals will participate in creating the next generation is tournament selection. In tournament selection a small number of individuals are random picked and from these one is chosen such that the probability of it being the best is highest and the probability for each below it in the ranking is exponentially lower. This ensures that fitter individuals are more likely to contribute their genetic material to the new generation.

The fitness function, which allows the search component to compare the quality of any two choice trees, is the speed-up that would be obtained by using a machine learning model in place of GCC's unrolling heuristic. The model is built from the new feature and the previously fixed set of features, as described in the next section, 5.1.3.

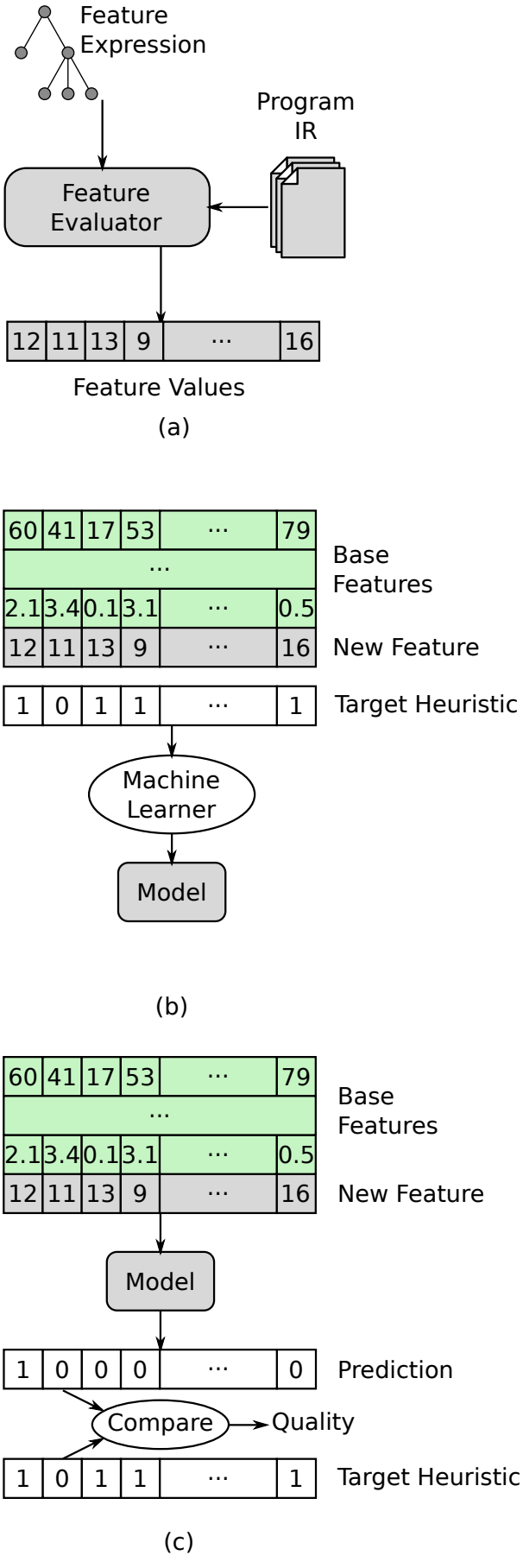


Figure 5.2: Machine learning used to determine the quality of a feature.

5.1.3 Machine Learning

The machine learning sub-system is the part of the system that provides feedback to the search component about how good a feature is. As mentioned above, the system maintains a list of good base features which is initially empty. It repeatedly searches for the best next feature to add to the base features, iteratively building up the list of good features. The system stops when it has failed to add a new feature that improves the results. The final output of the system will be the list good features at the end of the search.

Figure 5.2 shows the details of the process.

To evaluate the quality of a new feature we first compute the feature values across all programs (as shown in figure 5.2(a)). The program data might be the IR for loops of a number of benchmarks and a feature might be *the depth of loop times the number of basic-blocks*. A runtime system computes the feature expression on each program datum, yielding a vector of the resulting numbers and the feature values.

We then combine this feature with the previous base features and ask a machine learning algorithm to learn a model that can predict the target heuristic value (shown in figure 5.2(b)). Any machine learning tool can be used, however, since the tool will have to learn a model every time we need to compute the fitness of a feature it must be fast; we might compute the fitness of several million features during our search process. Some of the state of the art machine learning tools, like Support Vector Machines, are very slow, sometimes by two orders of magnitude compared to simpler tools like decision trees. In our work, then, we have always use the faster tools. However, since, like many search problems, ours could be trivially parallelised, the slower tools need not be ruled out in the long term.

Finally, we test the model's quality and report this back to the search component (shown in figure 5.2(c)). In our work we used the speed-up of the prediction to be the fitness of the model for a feature. Prediction accuracy could have been used, amongst other possibilities, but we found that in this case the system may concentrate on improving accuracy for small loops at the expense of those that dominate the execution time. Improving speed-up is a much closer fit to the goal of compiler optimisation.

5.1.3.1 Cross Validation

Cross validation is used in machine learning to avoid over fitting to the training set. Over fitting means that the predictive model may be very good for the points in the

training set by poor for any other points. A typical cross validation scenario partitions the input data into a number of sets (say ten). One set is kept out and called the *test* set, the remainder (typically $\frac{9}{10}^{\text{ths}}$ of all the data) is called the *training* set. A model is trained on the training set and it predicts values for the test set. This is repeated for each partition so that each partition is the test set once and predictions are created for all of the input data. The total prediction (albeit composed from different models) is then evaluated for quality.

We have two levels of cross validation. The outer level is the more typical and is used to determine how well our feature generator works This level means that the program data is partitioned, providing a training set to the whole feature search 5.2(a-c), and complete sets of features are created for each partition. The models learned from those sets are used to predict values for the test sets and the whole prediction is used to evaluate how good our method is. This is no different from most machine learning experiments.

The inner level, on the other hand, is used to prevent the feature generator over fitting while it is searching for features. The step shown in 5.2(b) is actually cross validated (we use ten-fold cross validation, so ten models are created for the ten training set/test set pairs). The multiple models combined, build the prediction in 5.2(c) over the several test sets.

At no point will the inner level see the outer level's test set. The inner level's training and test sets come from the outer level's training set only.

5.1.3.2 Parsimony

In practice, the system uses additional information to the quality metric described above. It happens that the feature expressions learned by genetic programming can quickly become very long. Two features can have the same results but have different lengths (for example, if one feature is `loop.depth`, a more complicated, equivalent feature is `loop.depth+(11)loop.depth`. Such simple equivalences are optimised out of our system, but complex examples still exist unoptimised).

In order to address this problem, we adopt the well know genetic programming methodology of rewarding parsimony. If the objective function computed by the machine learning tool for two features gives the same value then we determine that whichever feature expression is shorter is the better.

5.2 Grammar for Loops in GCC

This section shows one of the grammars created. This is a grammar for generating features over loops at the register transfer language of GCC (RTL) and is used in the experiments of chapter 5. The grammar contains many tens of thousands of productions because of the large number of different data-types in GCC's internal representations. The grammar is automatically created by observing the types of data that GCC uses internally; doing this is essential because of the grammar's size but also prevents the grammar needing to be hard coded, allowing modest changes GCC without requiring an engineer to rewrite the grammar.

An overview of the system is shown in figure 5.3. The system builds three main outputs from observing GCC's internal representation of the benchmarks; fast code representing the grammar, a custom feature evaluation language and a compact version of the benchmark data. Subsequent sections describe the major components of the grammar generator.

5.2.1 Data Generation

The first stage of the system extracts GCC's internal data structures for each of the benchmarks into XML files. *libPlugin*provides a plug-in to perform this data extraction.

In RTL, instructions are in an algebraic form with a treed, list-of-lists representation. Each node in the RTL may have some number of attributes. The RTL representation of the loops is extracted, augmented to include the structure of the basic blocks in the loop and the RTL instructions contained within their blocks. Also export is any information GCC can compute at that time, such as estimated block frequencies, loop depths, and so on. A flavour of the data is given in figure 5.4.

5.2.2 Structure Analysis

The hierarchical data produced follow a number of relational rules. For example, loops contain basic blocks as children and they in turn contain instructions. Those relationships are never violated. It would be foolish to create a feature like "*count the number of basic blocks which contain three loops*" since that can never be non-zero.

The grammars constructed are automatically derived from the structural rules of the data to ensure that such impossible features are never generated, improving the

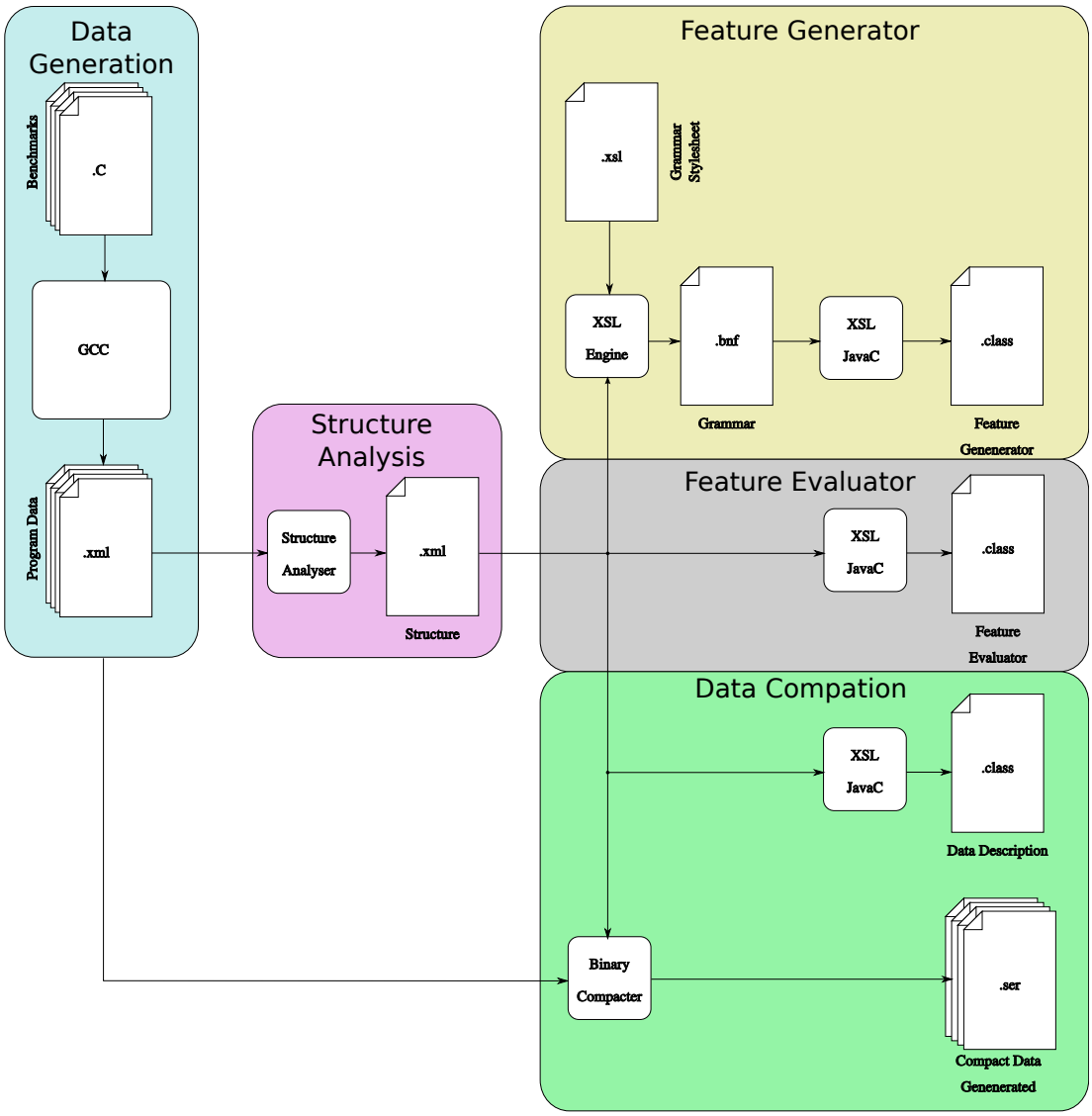


Figure 5.3: Overview of the grammar creation system for RTL loops in GCC.

```
<loop name="UTDSP.fft_1024.fft.3" ... insns="28" expected-iter="11">
   <basic-block index="10" ... frequency="9100" loop-depth="3">
      ...
      <insn ...>
         ...
         <set ...>
            <reg ... mode="SF">
               <int>112</int>
               ...
            </reg>
            <mult ... mode="SF">
               <reg mode="SF">
                  <int>94</int>
                  ...
               </reg>
               <reg volatile="true" mode="SF">
                  <int>84</int>
                  ...
               </reg>
            </mult>
         </set>
      </insn>
```

Figure 5.4: XML representation of RTL. Many of the attributes and values made available are removed for clarity. The example shows part of the third loop from the function fft in UTDSP benchmark, fft 1024. One instruction inside one of the basic blocks is shown which sets the value of one register to the multiplication of two others.

efficiency of the search. Since the rules that GCC's internal data structures follow are not immediately derivable in a machine readable manner from the GCC source code, a simpler approach is taken of examining the XML data files to find the observed structure within.

The XML data files are iterated through and the number of each type of node in the XML is collected; how many children each node type has is recorded; a histogram of child types for each child slot in each node type is built, as is a histogram of attribute values for each node type.

5.2.3 Data Compaction

The XML data produced by *libPlugin*is extremely detailed and because of XML's verbose nature the data files come to several gigabytes in total. Processing these files in their raw form to explore features is very memory intensive and typically causes the machine to thrash. To improve performance, the data files are converted into a compact binary format using the structure document as a guide. The structure document declares all different types of nodes, their attribute names and value and the children each node can have.

The system first creates a Java class that represents each type of node. Each attribute maps to a field of a suitable type and the node's children are packaged as an array. The structure document is used as source data so that the system knows what Java classes are needed. Next, the XML program data files are read and each is converted to objects of the Java classes and serialised to files. The resulting data is much smaller than the original XML, can be loaded into memory all at once and is much faster to compute feature values over.

5.2.4 Feature Evaluator

Although feature evaluation could be performed by an existing scripting language, generic scripting languages were found to be far too slow for searching over. Compiling features into C programs and Java programs was also tried. In C, the effort of spawning GCC and then spawning the feature program also proved to slow. For Java, the compilation could be performed in process but the resulting classes, after being loaded into memory, tended not be garbage collected, so eventually all memory was exhausted.

Instead, a custom feature interpreter was created. There was no need to create a

parser for the interpreter, since the feature grammar can produce ASTs directly, not just strings. The interpreter supports:

- Getting the types of nodes
- Getting attribute values of nodes, including determining if attributes are missing
- Logical, comparison and arithmetic operations
- Aggregators i.e. summations, finding extrema, etc.
- Iterating over children, descendants
- Standard control flow operations
- Pattern matching

Runtime support for feature evaluation is also computed from the structure document.

The next section describes how features are created that make use of this interpreter.

5.2.5 Feature Generator

This structural information is then used to mechanically create a grammar. Because they are automated and not hard coded, they are easy to update in response to changes in the compiler. The grammars, being machine generated, are quite large; several hundreds of kilobytes long. The transforms used in practice all make sure that trivially impossible features (or rather, uninteresting features which do not relate to possible structures) are never created. They also automatically set production weights to ensure grammars do not suffer from the probabilistic recursion issues described in subsection 4.4.2.

Figure 5.5 shows a pared down snippet of the grammar, giving an example of some of the feature expressions that can be created. The full grammar has many more functional capabilities and is also tuned to the structure of the RTL. This bit of the grammar says how some numerical values can be computed on node from the data: a numeric can be a binary expression of two numerics; it may be the numerical value of an attribute; it may aggregate numerical values of selected children; it might count the number of children matching come criterion; it might simply delegate to numeric of a child. The selection used in the numeric might be a logical combination of other matchers; it might be the result of comparing two numerics; it may check the type of the node; it

may test the value of an attribute. This part of the grammar is replicated many times for each bit of structural data. There are also many other functional concepts encoded in the grammar in similar ways.

```
1   <numeric> ::= <numeric> ( "+" | "-" | "*" | "/" ) <numeric>
2               | <value-of-an-attribute>
3               | ( "sum" | "min" | "max" | "avg" )
4                 "(for-each-child-that(" <match> "do" <numeric> "))"
5               | "count-children-matching(" <match> ")"
6               | "on-child" <random> "do" <numeric>
7   <match>   ::= <match> ( "or" | "and" | "xor" ) <match>
8               | "not(" <match> ")"
9               | <numeric> ( "<" | ">" ) <numeric>
10              | "is-type(" <node-type> ")"
11              | <attribute> "=" <value>
```

Figure 5.5: A simplified subset of the automatically generated grammar.

5.3 Motivating Example Reprise

In section 4.2.2 is presented an example for which naïve features failed to achieve much of the potential speed-up available. If, instead, our technique is used to search for the best set of features and train a decision tree over those then the best unroll factor of 11 can be automatically selected, giving the maximum speed-up for the loop in figure 4.1. The path of the decision tree selecting the unroll factor of 11 is also shown in figure 5.6(b) and the features touched are shown in figure 5.6(a). This example shows that while performance of the heuristic can be improved by a machine learning approach, it may ultimately be limited by the features used. By searching the space of features, features better suited to the learning task at hand can be found.

5.4 Experimental Setup

In this section we briefly describe the experimental setup, how the training data was generated and the steps taken to ensure accuracy of measurement.

Name	Value	Feature
f0	6.14 E17	`get-attr(@num-iter)`
f1	308	`count...!is-type(wide-int) ..`
f2	2	`count...is-type(basic-block)...`
f3	5	`max... is-type(basic-block) .`
f4	4	`count... is-type(array_type)`
f5	0	`count... is-type(le) && ...`

(a)

```
if( f2 <=4 )
 if( f5 <= 0 )
   if( f0 > 8206 )
     if( f1 > 168 )
       if( f0  > 6.1E17 )
         if( f2 <= 3 )
           if( f1 <= 1247 )
             if( f4 > 1 )
               if( f3 > 4 )
                 if( f3 <= 6 )
                   unrollFactor = 11;
```

(b)

Method	Unroll	Cycles	Speedup	% of Max
Baseline	0	406,424	1.0000	0%
Oracle	11	328,352	1.2378	100%
GCC Default	7	418,464	0.9712	-12%
GCC Tree	2	392,655	1.0351	14%
Our Technique	**11**	**328,352**	**1.2378**	**100%**

(c)

Figure 5.6: The path through the learned heuristic (b) for the example in figure 4.1 and the features from our scheme used by that path (a). In (c), the speed-up using our features is compared to that from other methods.

5.4.1 Compiler Setup

To demonstrate the applicability of our approach, we have applied it to loop unrolling within GCC 4.3.1. Loop unrolling is an extensively studied optimisation and there exists prior workMonsifrot et al. (2002); Stephenson and Amarasinghe (2005) with which to compare. We extended the compiler to allow unroll factors to be explicitly specified for each loop in a program.

5.4.2 Benchmarks

We took 57 benchmarks from the MediaBench, MiBench and UTDSP benchmark suites. Those benchmarks from the suites which did not compile immediately, without any modification except updating path variables, were excluded.

5.4.3 Platform

These experiments were run on a single unloaded, headless machine; an Intel single core Pentium 6 running at 2.8 GHz with 512 Mb of RAM. All files for the benchmarks were transferred to a 32 Mb RAM disk to reduce IO variability.

5.4.4 Generating Training Data

In order to learn the best unroll factor we need to generate training data where we know the best unroll factor for each of the training loops. To find this we took each loop, one at a time, and unrolled it by different factors, zero to fifteen. This gave a compiled program for which all but one loop has the default unroll factor as determined by GCC's default heuristic. We executed each of these versions of the program a number of times, in each case recording the number of cycles required to execute the function containing the loop that had been altered. We compiled without inlining to increase the independence of loops. In total we gathered data for 2778 loops.

5.4.5 Measurement

One of the difficulties in evaluating the performance of compiler optimisations is the impact of noise on the measured results. For each differently compiled variation of a benchmark we ran that version of the program at least one hundred times. We applied a standard statistical technique to reduce the effects of noise: applying a log transform

and removing outliers outside the 1.5 x IQR (interquartile range). The best unroll factor for each loop was determined as that with the lowest average (across the 100 runs) cycle count.

5.5 Experimental Methodology

This section outlines the methodology used when applying our feature search technique to the problem of loop unrolling in GCC.

5.5.1 Searching for Features

Our feature generator searches for one feature at a time. It prefers features which, in combination with the features selected by previous steps, most improve the performance of a machine learning tool. The genetic search for each feature consisted of a population of one hundred individuals. Each was allowed to run until fifteen generations produced no improvement in the best feature of the population or a maximum of two hundred generations, whichever came first. Search for new features to add was stopped when either two and a half thousand total generations were reached or when we failed to find an improving feature five times.

5.5.2 Cross-validation and Machine Learning

We split the loops into ten groups keeping one group out for testing so that we can perform ten-fold cross validation. Loops that are used for generating features and later learning a model are *never* used to evaluate the model. Final evaluation is always on *unseen* loops.

The machine learning algorithm used to find the quality of the features was a simple C4.5 decision tree Monsifrot et al. (2002), selected for its speed. When a feature was evaluated, we trained a decision tree on eight of the remaining nine loop partitions, called the training set. We then asked the decision tree to predict the unroll factors for loops in the remaining, ninth part, called the *internal validation* set. This was then used to determine the speedup attained by those unroll factors.

5.5.3 Search, Training and Deployment Cost

It took our system two days to learn the best set of features and model for this problem. Although this is a significant amount of time, it is a one off activity that it is performed "at the factory" and would be easily parallelised. If we consider the amount of time it takes for a compiler writer to develop a good heuristic, this cost is in fact small.

It is theoretically possible for our system to produce extremely computationally expensive features, increasing compile time. The system forces these feature evaluations to time out, giving them at most two seconds to evaluate over all loops. If a feature times out it is discarded and cannot contribute to the gene pool. We find that the pressure for simpler features means that features rarely time out. The features selected by our system for unrolling have no significant impact on GCC's execution time.

5.6 Results

This section evaluates our technique when applied to loop unrolling, demonstrating that it outperforms existing approaches. We first show the maximum benefit available from loop unrolling across the benchmark suite and to what extent GCC is able to achieve this. We then compare our approach against GCC's and a start-of-the-art machine learning schemes. This is followed by a brief analysis of the results.

5.6.1 Maximum Performance Available: evaluating GCC's heuristic

In order to determine how well our technique and others perform, we first conduct a limit study. As described in section 5.4 we exhaustively enumerated loop unroll factors up to 15 for each loop, recording the best setting for each. We then ran each benchmark with the best unroll factors set and recorded the speedup. The bars labeled oracle in figure 5.7 show the maximum achievable speedups compared to no unrolling for our benchmarks.

What is immediately obvious is that the impact of loop unrolling varies dramatically across benchmarks with an average speedup of 1.05. For some benchmarks such as `adpcm` from *MediaBench* no unroll factor has an impact on performance. In the case of `security_sha` from *MiBench*, however, there is a potential speedup of 1.28. What we would like is a scheme that is able to exploit this potential: delivering speedups when they are available and not slowing the program down otherwise.

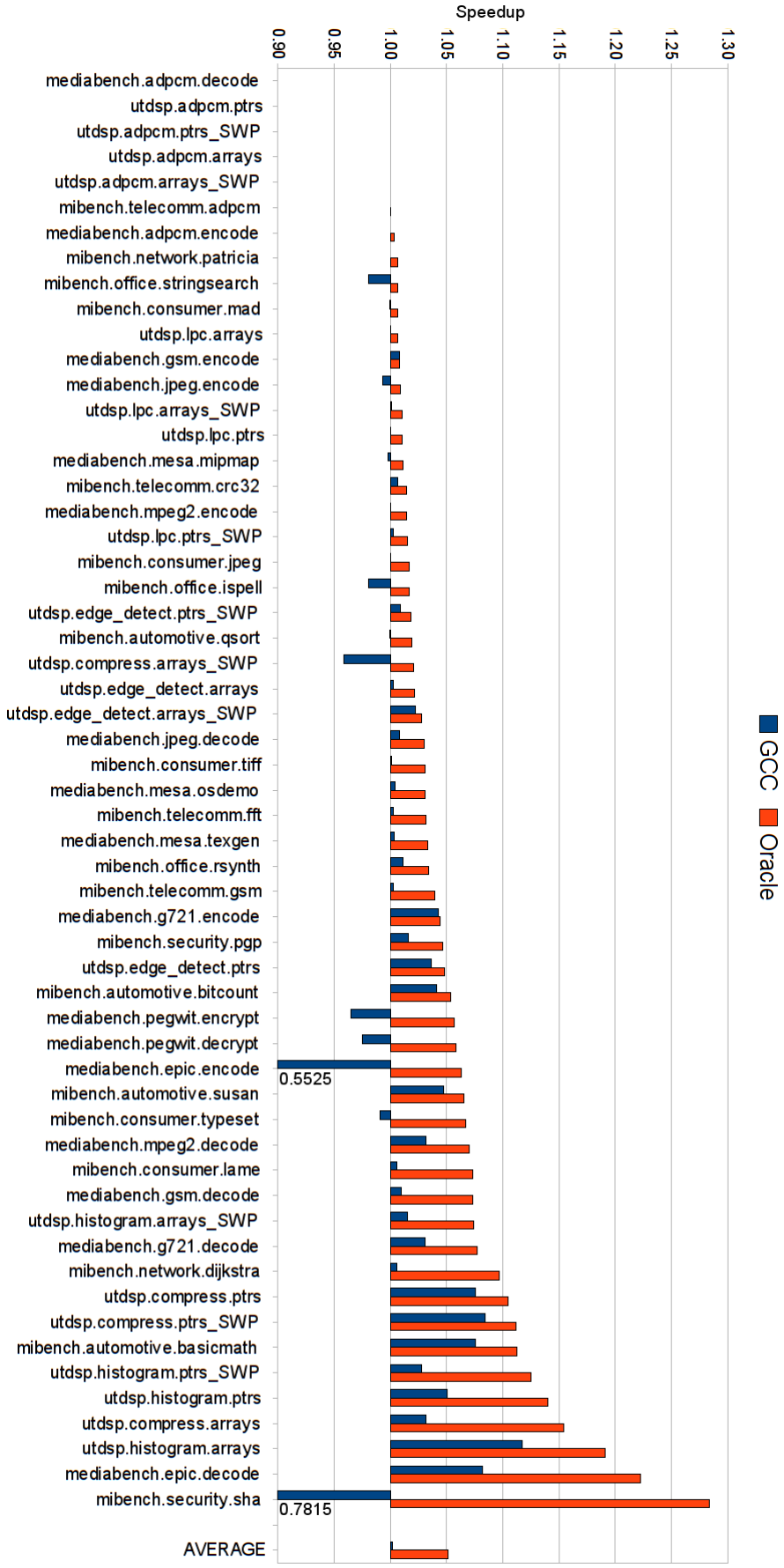


Figure 5.7: Speed up of the unroll factors chosen by GCC's default heuristic and the speed up of the best possible unroll factor - the oracle.

If we now consider the set of bars in figure 5.7 labeled GCC, we see the performance of GCC across the same benchmark suite. In some case cases it is able to achieve speedup, 1.12 on `histogram.arrays` from *UTDSP*, yet in the case of `security_ sha` which has the biggest potential for performance gains, it delivers a large slowdown of 0.78. In fact it slows down 12 of the benchmarks the worst being `epic_encode` from *MiBench* where the slowdown is 0.55. This demonstrates the difficulty compiler writers have in developing a portable optimisation that delivers performance gains.

5.6.2 Our Approach

Given the potential performance available from loop unrolling and GCC's poor performance, we here demonstrate how our approach improves upon that.

5.6.2.1 Comparison with GCC and Oracle

The bars in figure 5.8, labelled `Our`, show the speedups of our approach across the benchmark suite. On average, we are able to achieve 76% of the maximum available. In those benchmarks where there is large potential speedup available such as `security_sha` we are able to achieve a speedup of 1.21 compared to GCC's 0.78. In fact if we concentrate on the benchmarks where there is significant speedup available ($>$1.10 speedup) we are able to achieve 82% of the maximum. Thus, we have a technique that on average delivers over 75% of the maximum speedup available. This is achieved entirely automatically and compares favorably with the 3% achieved by the hand-crafted GCC heuristic.

5.6.2.2 Comparison with a state-of-the-art ML technique

Although our technique performs well, this may be due to the particular machine learning algorithm rather than carefully generating the correct features. In this section we evaluate an alternative state-of-the art scheme (`stateML`) based on a support-vector machine (SVM) described in Stephenson and Amarasinghe (2005). We implemented this technique within GCC using the features described in Stephenson and Amarasinghe (2005) and shown in table 5.9. This model was trained and evaluated using cross-validation in exactly the same manner as ours. The bars labeled ' stateML' in figure 5.8 represent the performance achieved using this technique. On average it

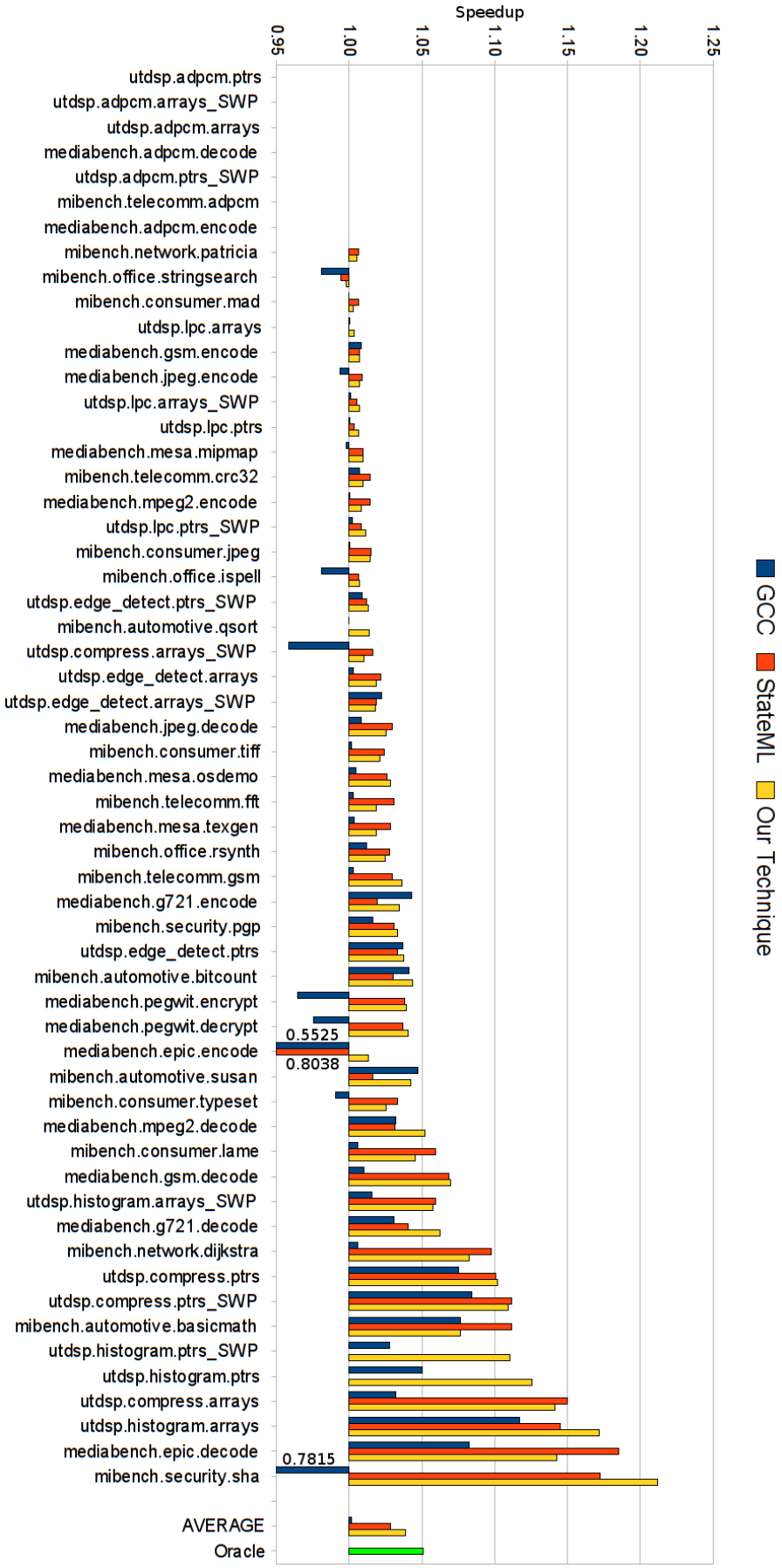


Figure 5.8: Comparison of speed ups between GCC's heuristic (`GCC`) , our technique (`Our`) and the state-ofthe-art ML approach (`stateML`). GCC achieves an average of 3% of the performance available, stateML achieves 59% of the performance available while our approach achieves 76% of the maximum performance available.

achieves 59%, outperforming GCC, which given that this was achieved automatically is significant. However, this is still short of the maximum achievable.

For the SVM method Schlkopf and Smola (2001) we used the "one-vs-all" approach where we learn K different classifiers (one for each unroll factor) each trained to distinguish the examples in a specific class from the examples in all the remaining classes. At prediction time, when a new loop is presented, the classifiers are executed and the class (unroll factor) with the largest ouput is selected. In our experiments we have used the Gaussian Radial Basis Function (RBF) kernel:

$$k(\vec{x},\vec{x}') = \exp\left(-\frac{|\vec{x}-\vec{x}'|^2}{2\sigma^2}\right), \tag{5.1}$$

with $\sigma = 1$ and we have set the upper bound parameter (C) of the SVM to 10.

5.6.2.3 Using Decision Trees for GCC and the stateML Features

Although we have shown that our approach is superior to both the default heuristic in GCC and the stateML technique, it may be argued that both alternative schemes have good features, that (i) GCC just has a poorly implemented heuristics and (ii) that the stateML may not be best suited to loop unrolling within GCC given that is was developed within a different compiler setting.

We therefore applied the same machine learning procedure based on decision trees using both GCC' s and stateML's features, the results of which are shown in figure 5.10. Thus each of the 3 different approaches, GCC, stateML and our technique share the same machine learning model, differing in only their choice of features. Using this approach, the GCC based features (labeled GCC Tree) are able to achieve 48% of the maximum performance available, a significant improvement over the 3% available from the default heuristic. The stateML features (labeled stateML Tree) slightly worsen from 59 to 53% of the maximum available. Combining the two sets of features, however, GCC and stateML, has no further impact on performance.

These results show that machine learning does work but is limited to approximately half of the maximum performance available. By searching for the best features in tandem with learning a heuristic, we have been able to automatically improve this performance to 76%.

The loop nest level
The number of operations in loop body
The number of floating point operations in loop body
The number of branches in loop body
The number of memory operations in loop body
The number of operands in loop body
The number of implicit instructions in loop body
The number of unique predicates in loop body
The estimated latency of the critical path of loop
The estimated cycle length of loop body
The language (C or FORTRAN)
The number of parallel “computations” in loop
The maximum dependence height of computations
The maximum height of memory dependencies of computations
The maximum height of control dependencies of computations
The average dependence height of computations
The number of indirect references in loop body
The minimum memory-to-memory loop-carried dependence
The number of memory-to-memory dependencies
The trip count of the loop (-1 if unknown)
The number of uses in the loop
The number of definitions in the loop.

Figure 5.9: The stateML features

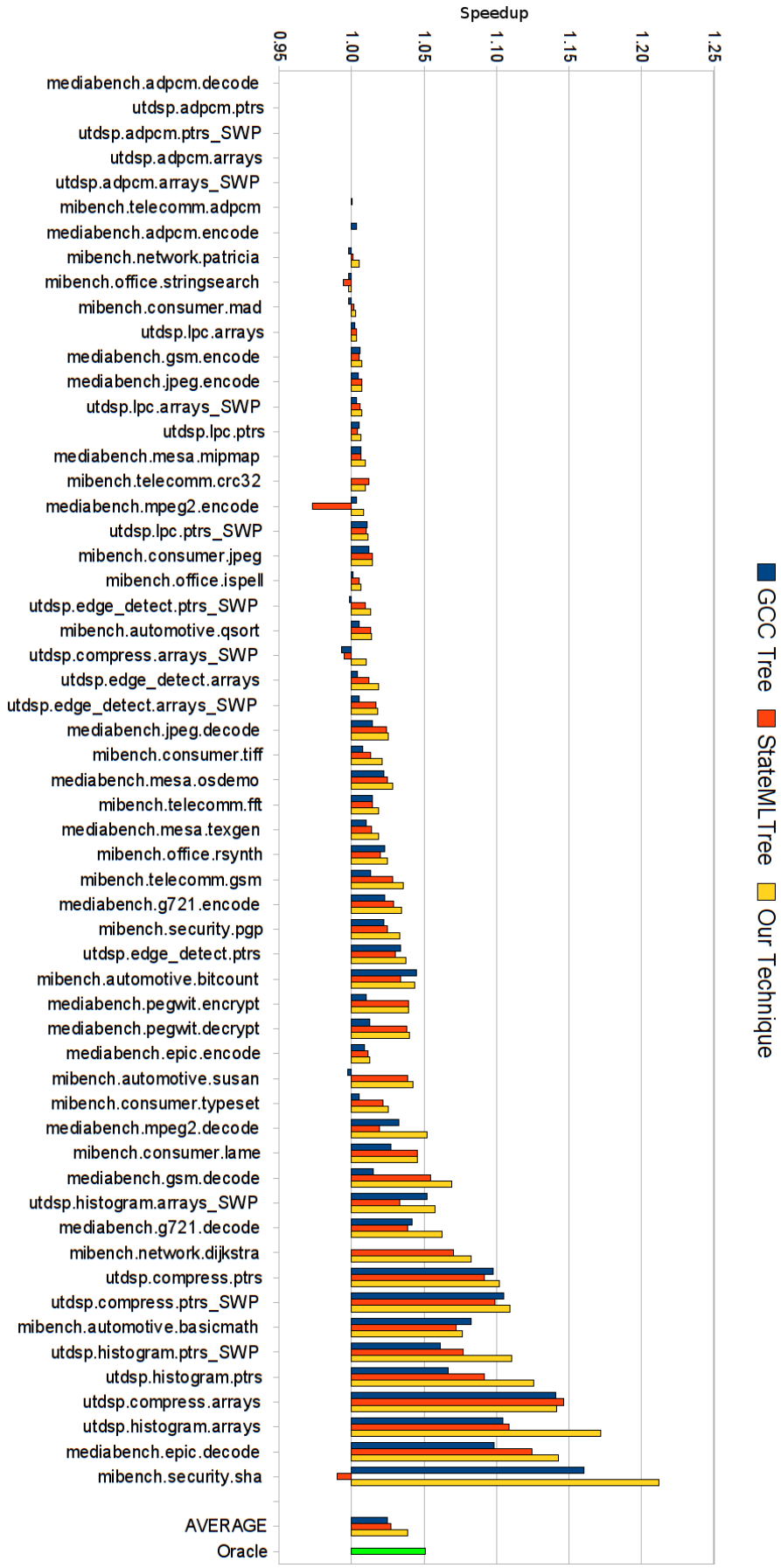


Figure 5.10: Speedup using a decision tree as the learning technique for GCC' s, stateML compared to our technique. Keeping the machine learning algorithm identical shows the relative merits of the feature sets.

5.6.3 Best features found

Figure 5.11 presents the first six out of thirty features found by our system in one fold. The speedup that feature attains, when combined with previous features is given, as is the translation of that speedup into a percentage of the maximum possible. We also show how much more of the maximum speedup each consecutive feature brings.

A few of the expression elements from the best features are explained below.
count(s) returns the number of elements in sequence, s
filter(s,m) filters sequence, s, removing any not matching expression m
sum(s,e) takes the sum of expression e applied to each member of sequence s
is-type(t) determines if the current node is of type t
/, //** and *[n]* are the children, descendants and particular child of the current node, respectively.

The first most important feature computes the loop's number of iterations, clearly, there is no point unrolling a loop more times than it has iterations. The remaining features are less obvious and are unlikely to be picked by a compiler writer demonstrating the strength of our approach. The features display elements which appeal to the intuition, but are, nonetheless, complicated and it is difficult to explain exactly why some elements are present. This is an artifact of the objective function which attempts, when adding a feature to find the most helpful feature for the machine learner, but makes no effort to find features whose rationale can be understood by a human. The meaning of the features is given in 5.12.

5.7 Summary

In this chapter we have developed a new technique to automatically generate good features for machine learning based optimizing compilation. By automatically deriving a feature grammar from the internal representation of the compiler, we can search a feature space using genetic programming.

We have applied this generic technique to automatically learn good features for loop unrolling within GCC. Our technique automatically finds features able to achieve, on average, 76% of the maximum available speed-up, dramatically outperforming features that were manually generated by compiler experts before. In this chapter our approach focusses on the RTL representation of the loop. Our system is generic, however, and is easily extended to cover different data structures within any compiler. Future work, will investigate exploring different feature spaces for new optimisations.

Number	Speedup	% of Max	Improvement	Feature
1	1.01971	38.63%	38.63%	get-attr(@num-iter)
2	1.02665	52.22%	13.59%	count(filter(//*, !(is-type(wide-int) || (is-type(float_extend) && [(is-type(reg)]/count(filter(//*,is-type(int))))) || is-type(union_type))))
3	1.03089	60.52%	8.30%	count(filter(/*, (is-type(basic-block) && (!@loop-depth==2 || (0.0 > ((count(filter(//*, is-type(var_decl))) - (count(filter(//*, (is-type(xor) && @mode==HI))) + sum(filter(/*, (is-type(call_insn) && has-attr(@unchanging))), count(filter(//*, is-type(real_type)))))))) / count(filter(/*, is-type(code_label)))))))))
4	1.03353	65.70%	5.18%	max(filter(/*,(is-type(basic-block) && !(@loop-depth==3 && @may-be-hot==true))), count(filter(/*,(is-type(insn) && /[5][(is-type(set) && /[0][(is-type(reg) && !@mode==DF)])]))))
5	1.03448	67.55%	1.86%	count(filter(//*,is-type(array_type)))
6	1.03503	68.62%	1.07%	count(filter(//*,(is-type(le) && !has-attr(@mode))))

Figure 5.11: Best features found by our feature search in one fold

Feature 1 Get the 'number of iterations if constant' attribute of the loop

Feature 2 Count the number of nodes at any depth in the loop which are
neither `wide-int` nodes
nor `float_extend` nodes for which
the first child is not a `reg` node with the count of `int` nodes beneath that `reg` node is zero
nor `union_type` nodes

Feature 3 Count the number of `basic-block`s for which
the `loop-depth` was not 2 or
(number of `var_decl` nodes in the block minus
the number of `xor` nodes in the block with `mode` attribute `HI` minus
the sum, over all `call_insn`s with an `unchanging` attribute of
number of `real_type` nodes
divided by the number of `code_label`s in the block)
is less than zero

Feature 4 For each `basic-block` that is not both of depth 3 and hot, compute
the number of instructions which have child 5 that
is a `set` and its first child is
a `reg` whose `mode` attribute is not `DF`.
Then take the maximum over all `basic-blocks`

Feature 5 Count the number of `array_type` nodes at any depth in the loop

Feature 6 Count the number of `le` nodes at any depth in the loop that do not have a `mode` attribute

Figure 5.12: Meanings of features from figure 5.11.

Chapter 6

Reducing the Cost of Iterative Compilation

The last two chapters showed how the compiler writer could be freed from having to think about what features to use in their machine learning set ups. This chapter frees the human from one more hurdle; efficient, iterative compilation.

Although iterative compilation produces excellent results, the costs can be prohibitive for ordinary use. Machine learning techniques (Agakov et al. (2006); Stephenson and Amarasinghe (2005); Monsifrot et al. (2002); Moss et al. (1998)) have been used to solve this problem. Heuristics are tuned 'at the factory' so that thereafter the optimisation space does not need to be searched. Machine learning has successfully tuned heuristics for embedded applications that out-perform their expert derived counterparts. The training data for the machine learning tools, however, must be generated by large scale iterative compilation. The compute time to profile all the different versions of the training benchmarks can be on the order of weeks or months (Fursin et al. (2008b)).

Each variation of a program must be run multiple times because of noise in performance measurements; everything from the other processes running on the machine or the state of the file system to the temperature of the computer can have an effect. This becomes more of a problem as the granularity of the measurements becomes finer. When individual functions and loops are measurement targets, the noise to signal ratio can be significant (Monsifrot et al. (2002)).

Different approaches are taken to circumvent the noise problem. In some instances, researchers have chosen a fixed sample size plan, running each program version a constant number of times without observing how the results are shaping up as they go

(Agakov et al. (2006); Bodin et al. (1998)). The hope is that the constant number of runs is sufficiently large to yield good results but not too large to waste effort. Often there is no analysis presented as to whether this number of runs is truly sufficient or if it is too many; confidence intervals and standard error bars rarely feature on performance graphs.

It may be tempting to use simulation to overcome noise, but not only are simulators slow and incompletely accurate, they are also subject to measurement bias Mytkowicz et al. (2009). To overcome that bias random variations in set up must be effected and simulations run multiple times; the result is noise in the measurements, just as there is noise for direct execution.

What is needed is a technique which provides statistically rigorous results in the presence of noisy data and simultaneously reduces the cost of searching a large space of optimisation settings. To the best of our knowledge, there is little prior work in this area. The closest to our work are (Georges et al. (2007); Blackburn et al. (2006)) where the authors recommend statistical rigour. They examine each point in the compiler optimisation space in isolation and propose performing executions until an estimate of the sample's inaccuracy is tolerably small or some maximum number of executions is reached. While this gives accurate measurements, it does not reduce the total number of executions needed for the whole optimisation space. Indeed, we will show that their approach can require more executions than a perfectly selected constant sized sampling plan.

An algorithm for selecting the number of times to execute each program version is called a sampling plan (Wetherill and Glazebrook (1986)). Sequential sampling plans (Wald (1947)), used in medical trials (Whitehead (1992)), adjust the sample size dynamically as data is gathered. These sequential systems adapt to ensure both that sample sizes are large enough for good results and that sampling stops when enough data have been collected.

This chapter develops an algorithm which:

- Determines a subset of the optimisation settings or program versions which are 'better' than all others, to some user supplied significance level.
- Minimises the number of runs required, dropping poorly performing versions early and finishing when sufficient data has been gathered for a decision.
- Provides statistically rigorous results.

- Allows more points in the compiler optimisation space to be examined.

Our algorithm 'races' different program versions, allowing those performing poorly to fall by the wayside with minimal sample size, whilst those fighting for the winning position are allowed more rein, increasing their sample size until either one wins or several draw.

We show that our adaptive, sequential sampling plan can dramatically reduce the number of runs needed to find the best program version. In this way, a greater part of the optimisation space can be explored than would otherwise be possible.

The remainder of this chapter is organised as follows. The next section presents examples showing the problem of noisy measurements. Section 6.2 gives an overview of our algorithm for the adaptively managed sampling plan and section 6.3 gives an in depth description of the technique. Then, section 6.4 shows the set up for our experiments, the results of which are given in section 6.5. Finally, section **??** concludes the chapter.

6.1 Motivation

Performance measurements on real systems are invariably noisy, an issue which becomes more problematic as the granularity of the measurement becomes finer. Figure 6.1 shows a typical distribution of cycle counts taken from function `run_length_encode_zeros` in the `MediaBench` `epic-encode` program. There is a minimum amount of work that the program must do, so there is a lower bound to the run time. On the other hand, there is no clear upper bound and the distribution features a long tail. The long tail of the distribution extends well beyond the median point and outliers from that tail, if included, can throw out the mean of small samples.

6.1.1 Confidence Intervals

Attempts to measure performance must be aware of the shape of its distribution. Certainly, a single observation will be insufficient to be confident that we have a good approximation for the performance; it might be nowhere near the mean. As the sample size grows, containing more and more observations of real measurements, we can be progressively more confident that the sample mean models the true mean of the distribution. As the sample size tends to infinity the difference between the sample mean and the true mean tends to zero.

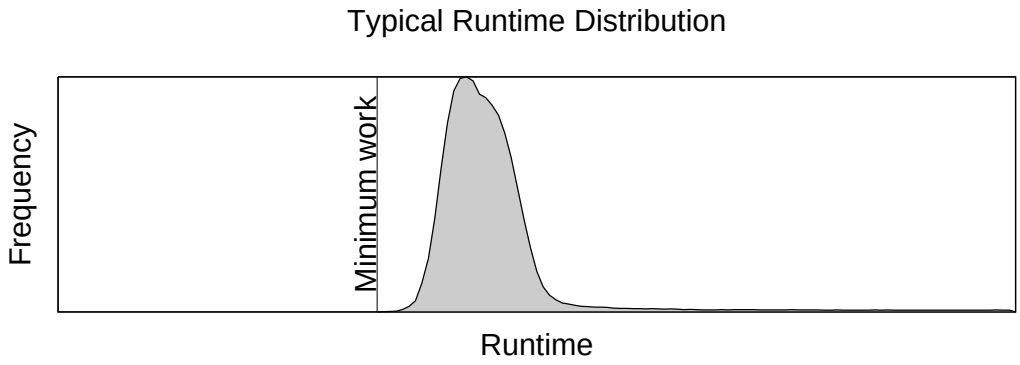


Figure 6.1: Typical distribution for the run time of a program. All programs have a minimum amount of work to achieve, giving a lower bound to the distribution. The shape of the distribution may vary but very often long tails can be observed. If small samples include observations from the tail then means can be thrown off. Without statistically rigorous techniques such situations will not be detected.

Confidence intervals can be used to assess whether samples are sufficiently large. These statistical ranges show where the likely value of the true mean falls. A confidence interval always contains the sample mean and extends for some distance from it in each direction. As the number of observations in a sample increases the width of the confidence interval decreases; we become more confident that we can pin down the true mean to be closer to the sample mean.

Confidence intervals are also parametrised by a probability or confidence coefficient. A confidence interval with a confidence coefficient of 99% indicates that we are 99% sure that the true mean is inside the interval; only in 1% of trials should the true mean fall outside. Higher significance levels require wider confidence intervals; conversely, if only low confidence is demanded the interval can be very narrow.

The difficulty in doing experiments with performance measurements is deciding how many observations are needed for each sample so that a confidence interval around the mean is sufficiently small. Typical, fixed size sampling plans require that this number of observations be fixed before any data is actually gathered and before any estimates of the noise are available.

The next section shows an example in which confidence intervals can prove or disprove the adequacy of different sample sizes.

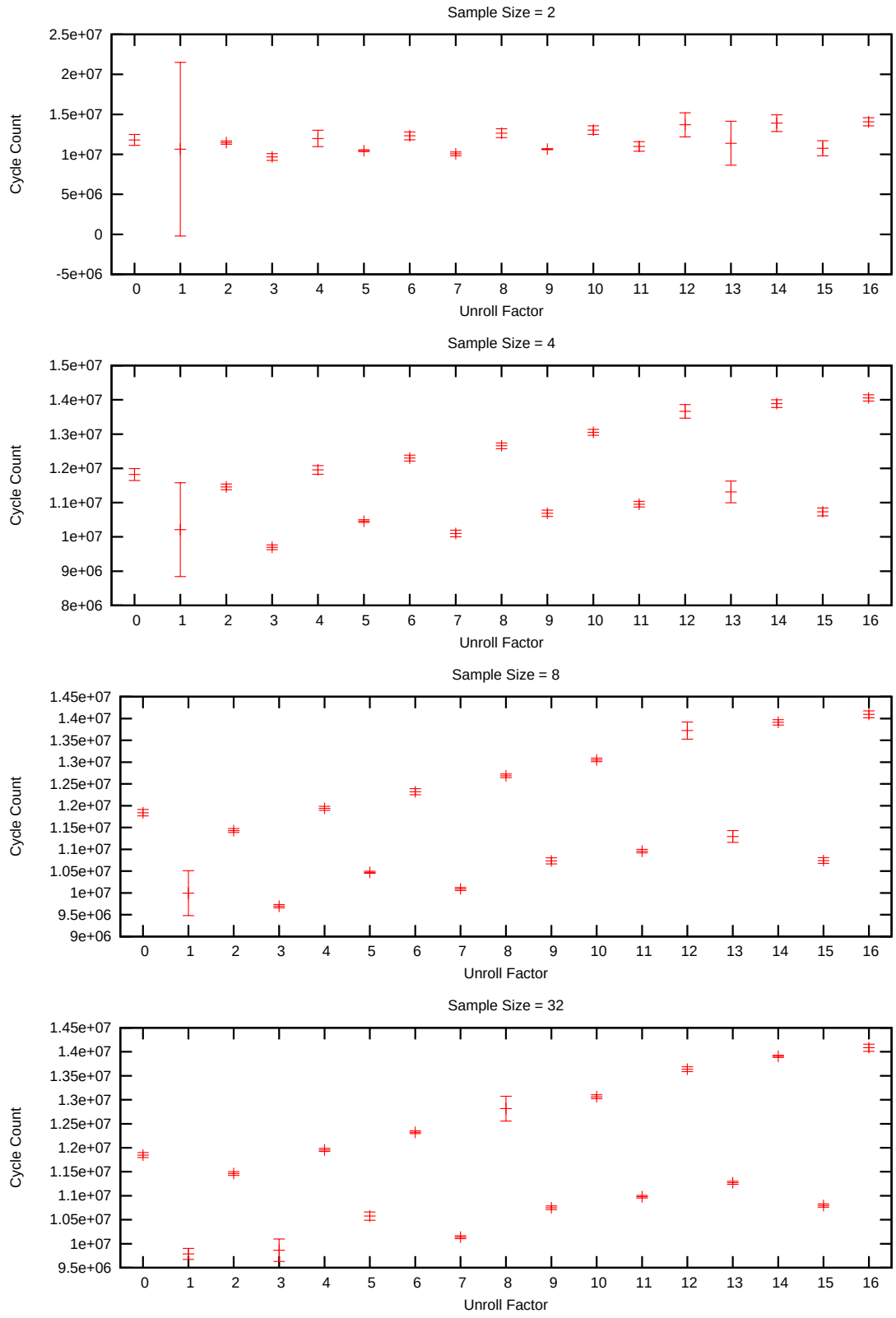


Figure 6.2: Confidence intervals at different sample sizes. Data is from function `run_length_encode_zeros` in `MediaBench epic-encode`. Cycle counts are shown for different unroll factors of a particular loop. Only when the sample size is 32 per unroll factor does an unambiguous winner emerge. The confidence interval is 95%. Note that the cycle count axis changes in each sub-figure.

6.1.2 Choosing a Sufficiently Large Sample Size

Figure 6.2 shows the number of cycles used by a loop in the function `run_length_encode_zeros` in the `MediaBench` `epic-encode` program. The loop was unrolled different numbers of times, to see which unroll factor most improved performance. For each unroll factor, the program was run a certain number of times (2, 4, 8 or 32) and the number of cycles was recorded. We plot, in each of the four graphs, the mean of the samples together with their 95% confidence intervals (note that the axes change between figures).

When the sample size is only 2 (top left graph in figure 6.2) we cannot say which unroll factor is the best. We can already be sure that some unroll factors are doing badly (for example, factor 4 is bettered by factor 9). However, the confidence intervals for some factors are so wide that we cannot be certain which has the lowest mean. With a constant sized sampling plan we have found that our sample size was too small.

As the sample sizes increase the confidence intervals become narrower. By the time we have sample sizes of 32 (bottom right graph in figure 6.2), we see that the complete interval for unroll factor 8 is lower than all the others and we thus find that factor to be the best. 32 executions of the program for each unroll factor are sufficient to tell which one to choose.

However, this simple, constant sized sampling plan does more work than necessary. Looking at the graph for sample size 4, we can see that the majority of the unroll factors were worse than factor 8; for all but factors 2, 9 and 16 the confidence intervals lay completely outside[1] the one for factor 8. If we had stopped executing those factors after sample size 4 and continued to 32 for the remaining 4 factors, we would only have executed each unroll factor an average of 7 times, a 78% reduction in the cost of sampling.

6.1.3 Choosing When to Stop Sampling

For some programs there will be no clear winner between two different versions. Alternatively, the difference might be so small compared to the noise that a huge sample size might be required to separate the program versions. In such a case we would like to stop early to avoid wasting effort. Consider the graph in figure 6.3; this graph shows the 95% confidence intervals of a different loop in `MediaBench` `epic-encode`. Again, the loop is unrolled different amounts, from 0 to 16, but this time the sample size is

[1] We do not advocate performing statistical tests visually in this fashion. Rather, bona fide Student's t-tests, ANOVA, etc. should be used.

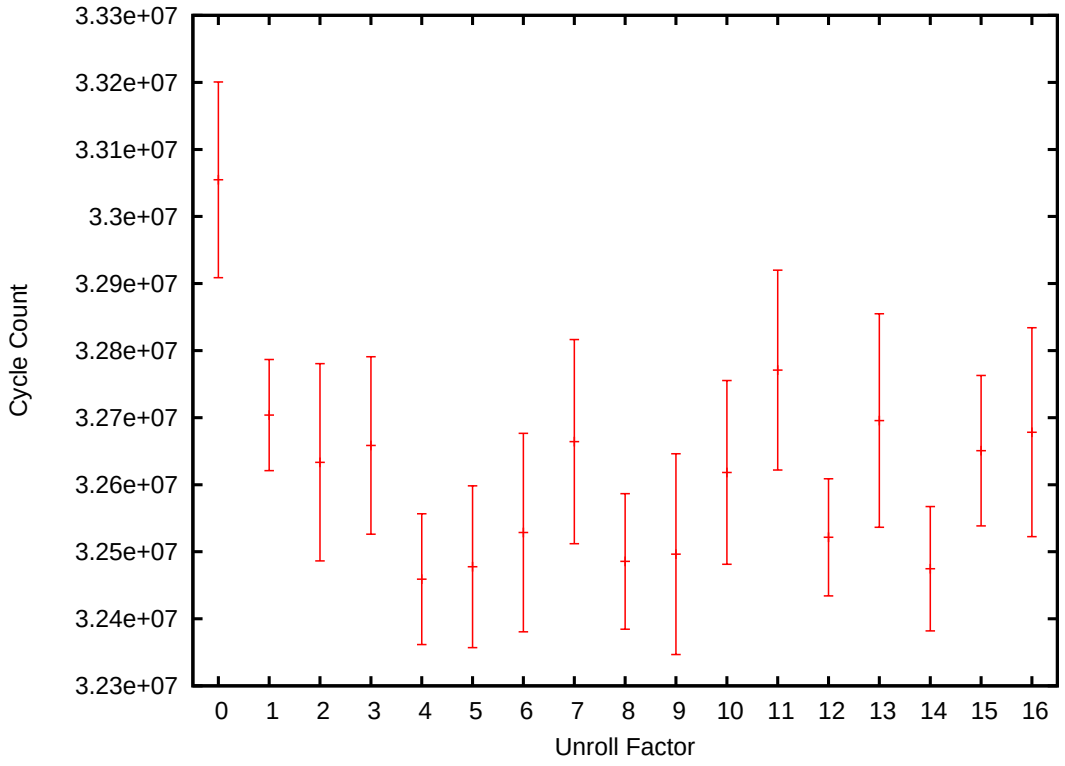


Figure 6.3: 95% Confidence intervals at sample size of 1000 for a loop in function `internal_filter` of benchmark `MediaBench epic-encode`.

1000. Even with this huge increase in sample size, only a few unroll factors (0, 1 and 11) can be excluded because their confidence intervals are completely disjoint to the one for unroll factor 4.

However, if we look at the worst case for unroll factor 4 and compare it to the best case for the other unroll factors we find a ratio of no more than 1.0065. In other words, if we chose factor 4 as the best factor, we could be fairly confident that if we are wrong it would be by not much more than 0.65%. The user, searching for the best program version might consider such a small error acceptable and agree that we need not execute the programs more times.

The situation is likely to be different for each program. In some cases, a small, constant sample size will suffice, in others much larger sample sizes must be taken. The user cannot, in general, know ahead of time how large the sample size should be.

This chapter presents a mechanism by which the sample sizes are adaptively managed to ensure statistically valid results while at the same time drastically reducing the number of executions times needed to select the best program version.

6.2 Method

This section presents our sequential sampling method. The essential idea of our algorithm is that we:

1. Maintain a sample for each program version
2. Determine which versions are worse than any other - these are losers
3. Finish if there is only one non losing version or the non losers are close enough to each other
4. Increase the sample size by one in each non loser
5. Repeat from step 2

The algorithm 'races' program versions to find out which will win - i.e. have the best performance. Poorly performing versions are knocked out of the race while potential winners continue, increasing their sample size. The moment we find there is either one clear winner or that the front runners are all good enough, we stop.

Figure 6.4 shows a pictorial example of our algorithm. In the left hand side of the first panel, (a), the samples are initialised for each program version. The right hand side shows the confidence intervals for each sample. Each version is, at this point a potential candidate to be in the winning set. We ensure that enough observations are in each sample for our statistical tests to work since they require a minimum sample size; little can be said statistically about a single observation.

In panel (b) the samples have been tested to see if any can already be identified as clear losers. A number of statistical tests are run (described in section 6.3.3) and any version shown to be worse than one of the other versions is taken out of the race. In the right hand pane, the bottom limits for the confidence intervals of two samples (marked red) can be seen to be above the top limits of other samples. This indicates that we are confident that the red samples are worse than the others and that even if we took more observations we are sure that will not change. The corresponding program versions are removed the candidate set (becoming white in the left hand side of the panel).

Since the remaining set of potential candidates contains more than one version we perform another set of tests to see if the versions are all approximately equal (detailed in section 6.3.4). At this point in the example there is not enough data to call the versions equal.

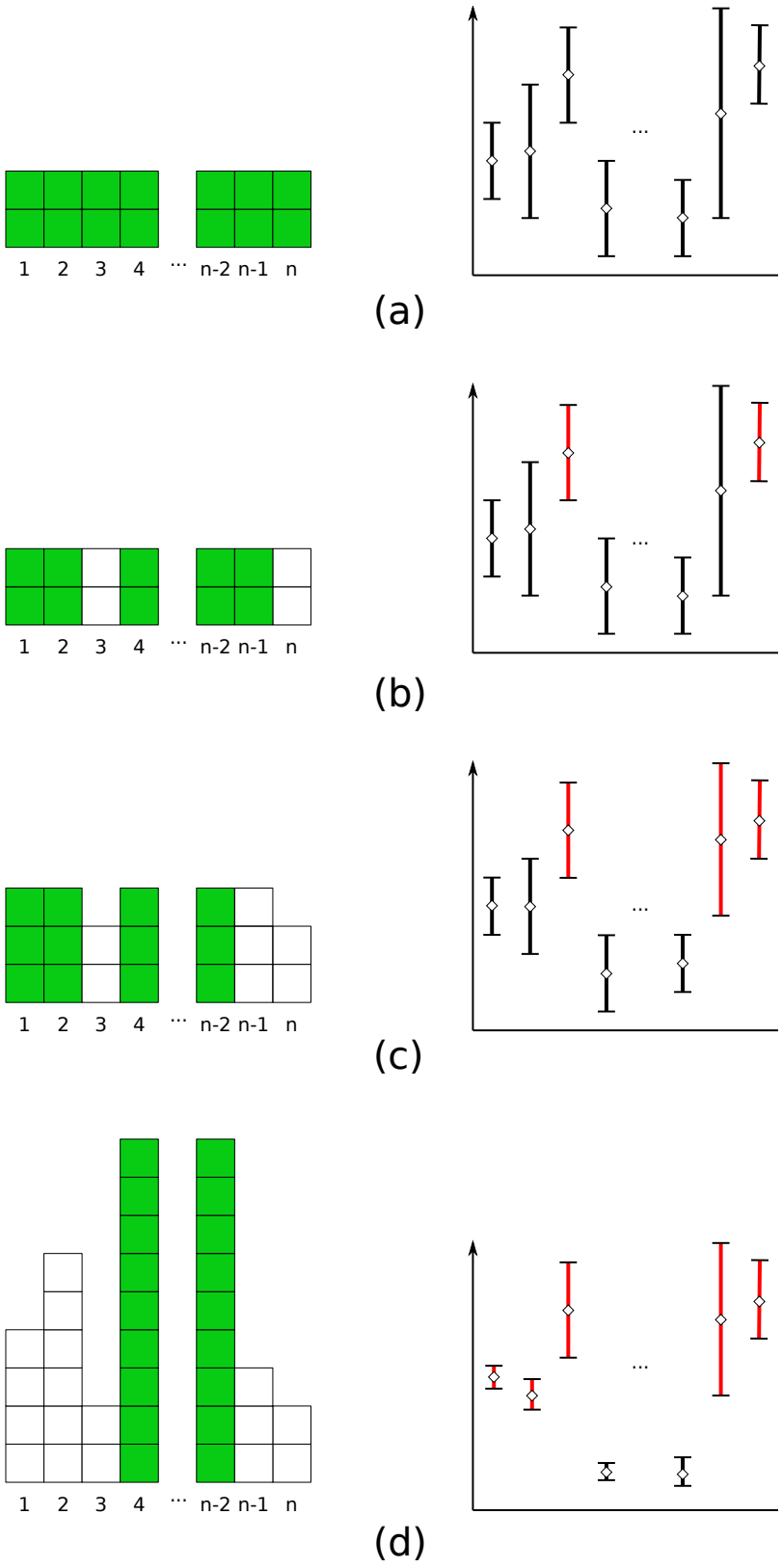


Figure 6.4: Several steps from our algorithm. In (a) initial samples are taken. In (b), versions that are clear losers are dropped. In (c), remaining versions are not deemed equivalent so the samples for them are grown by one. In (d), several steps have been run and the two remaining versions are found to be sufficiently good; the algorithm terminates.

In panel (c) another observation is added to each sample (as shown in the left hand side) and the process is repeated. As sample sizes grow, the amount of information available about each version increases, typically leading to the confidence intervals shrinking (as shown in the right hand side); statistical tests become more able to make decisions about the relative merits of the different versions. This may lead to more program versions becoming losers. In this panel, the second last version is knocked out of the race.

In the last panel, (d), the algorithm has run for several steps and discarded all the program versions but two. It has decided the remaining two program versions are sufficiently similar to consider them equal, so it returns them both.

6.2.1 Student's T-Tests

Our algorithm makes heavy use of statistical testing to determine which means are unlikely to be in the winning set.

We use a student's t-test ((Student)) which is a statistical test that can determine if the means of two samples are significantly different. It may be that differences observed in the sample means are due to the sample sizes being too small, rather than because the true means themselves are different. A t-test can only be used to check that the means are different; if it does not declare the means to be different, that does not necessarily mean that the means are the same, it could also be that the sample sizes are too small to verify the difference.

The significance level, α, of a t-test indicates the probability that the test will assert a difference in the means when none in fact exists (called a Type-I error). The lower this value the more confident we can be that a stated difference is real.

The t-test computes a 't-statistic' over the samples and compares this to a point on the cumulative distribution function (CDF) for the t-distribution (a probability distribution at the heart of the t- test). The point on the CDF parametrised by the probability, α, and an estimate of the number of degrees of freedom in the samples. The t-test normally indicates that the means are significantly different if the t-statistic is greater than the given point on CDF.

A t-test makes assumptions about the shape of the distribution and prefers it to look as normal as possible. There are also different variations on the t-test depending upon the exact use and what additional assumptions can be made. For example, if the sample sizes are equal or the variances are guaranteed to be equal then stronger

tests can be used. There are also alternatives to the t-test, for example the Mann-Whitney U test (Mann and Whitney (1947)), which make different assumptions about the distributions under test. These alternatives, however, are generally not preferred if the t-test is applicable.

6.3 Algorithm Details

This section describes the algorithm and its component parts in depth. Pseudo-code is presented in Algorithm 1.

Our algorithm begins with a set of all the program versions, C. For every version we maintain a sample which consists of the run time values we have taken so far for the corresponding version; these are S_c. In reality, we only need to record the sufficient descriptive statistics to determine the mean, confidence intervals and perform statistical tests, we never need to remember the complete list of run time observations and so the amount of space required by the algorithm is linear in the number of program versions.

The loop in lines 3 to 8 forms the bulk of the algorithm. First we remove any version that is provably worse than any other. Then we terminate if there is only one candidate left or all the remaining candidates are equal. Line 6 increases the sample size for remaining versions. Finally we terminate if some user defined limit is reached, allowing a hard boundary to be imposed on the total number of times each program version will ever be executed.

The loop does not remember which program versions were losers from iteration to iteration. This means that a version which is found to be a loser in one iteration has the opportunity to re-enter the race later on. It can happen that a version deemed promising early on turns out to be less so once more information about it has been gathered. We found that reconsidering losers provided lower error rates.

A detailed description of the subroutines used by the algorithm follows.

6.3.1 Initialisation

At the beginning of the algorithm the sample sets are initialised to have two observations, the minimum necessary to make statistical inferences with a t-test. If other statistical tests are used, the initial number of observations may have to be different.

Algorithm 1 Pseudo code for our algorithm

1. $C \leftarrow \{c; c \text{ is a compilation strategy}\}$
2. $\forall c \in C, S_c \leftarrow \{sampleRuntime_c(), sampleRuntime_c()\}$
3. **for ever do**
4. $\quad C' \leftarrow C - losers(C, S)$
5. $\quad$ **if** $|C'| = 1$ **or** $candidatesEqual(C', S)$ **then return** C'
6. $\quad \forall c \in C', S_c \leftarrow S_c \cup sampleRuntime_c()$
7. $\quad$ **if** $sampleThresholdReached(C', S)$ **then return** C'
8. **end for**
9. $sampleRuntime_c()$
10. $\quad x \leftarrow$ execute strategy c and record runtime
11. $\quad$ **return** $\ln x$
12. $losers(C, S)$
13. $\quad$ **return** $\{c \in C; \exists d \in C, d \neq c, S_d <_{\alpha_{LT}} S_c\}$
14. $candidatesEqual(C', S)$
15. $\quad B \leftarrow \{b \in C'; \mu_b \leq \mu_c, \forall d \in C'\}$
16. $\quad$ **return** $\bigwedge_{b \in B, c \in C'; b \neq c} S_b =_{\alpha_{EQ}, \varepsilon} S_c$
17. $sampleThresholdReached(C, S)$
18. $\quad$ **return** $\bigvee_{c \in C} |S_c| \geq MAX\_SAMPLE\_SIZE$

6.3.2 Sampling the Run time

The cycle count for the current program version is measured by the function *sampleRuntime*. It assumes that there is some mechanism to profile the program to calculate the measurement.

Of special note here is that we take the natural logarithm of the run time. The reason for this is that run time distributions are both skewed and often suffer from outliers; applying a log transform is a common way to make the distribution look more 'normal' and to reduce the effects of outliers (Bland and Altman (1996)). Having distributions which are closer to a normal distribution frequently improves the accuracy of statistical tests. More general transformations, such as a Box-Cox transform (Box and Cox (1964)) of which the logarithm is a special case, could be used instead, but we found adequate results from the simple logarithm.

6.3.3 Weeding Out Losers

The algorithm needs to determine which versions are unlikely to be in the final winning set which is handled by the function, *losers*. The set of losers consists of any version, c, for which there is another version, d, that looks to be better performing. The 'better performing' test is a relation, $<_{\alpha_{LT}}$, over samples, (S_d, S_c). Intuitively we want $S_c <_{\alpha_{LT}} S_d$ whenever it appears from that the true mean of S_c is worse than the true mean of S_d to some confidence level. This relation is not a total ordering since if we do not have enough observations to be confident, then neither $S_c <_{\alpha_{LT}} S_d$ nor $S_d <_{\alpha_{LT}} S_c$ will hold.

To determine membership of the relation, $<_{\alpha_{LT}}$, we first check that the mean, μ_d, of S_d is less than the mean, μ_c, of S_c. If that is the case then we perform a student's t- test to discover if the difference in the means is significant to some user supplied significance level, α_{LT}.

Since we cannot be certain that the variances are equal and since also the number of observations in each sample may be different, we use Welch's t-test (B.L.Welch (1947)) where the t statistic is:

$$t = \frac{\mu_d - \mu_c}{\sqrt{s_d^2/n_d + s_c^2/n_c}}$$

and μ_i, s_i^2 and n_i are the sample mean, variance and size of the i^{th} program version, respectively.

The degrees of freedom are estimated by the Welch-Satterthwaite equation (Satterthwaite (1946)):

$$d.f. = \frac{\left(s_d^2/n_d + s_c^2/n_c\right)^2}{\frac{\left(s_d^2/n_d\right)^2}{(n_d-1)} + \frac{\left(s_c^2/n_c\right)^2}{(n_c-1)}}$$

As the testing proceeds, any version removed is not used to compare against subsequent versions in the tests for this iteration. In this way the algorithm guarantees that at least one version survives the *losers* function.

6.3.4 Finding the Winners

Stopping when enough data has been gathered is important since we may find that some program versions are either identical to each other or so nearly so that the compiler writer is content with several winners. Increasing the sample sizes once we know this is a waste of effort and, left unchecked, may continue indefinitely.

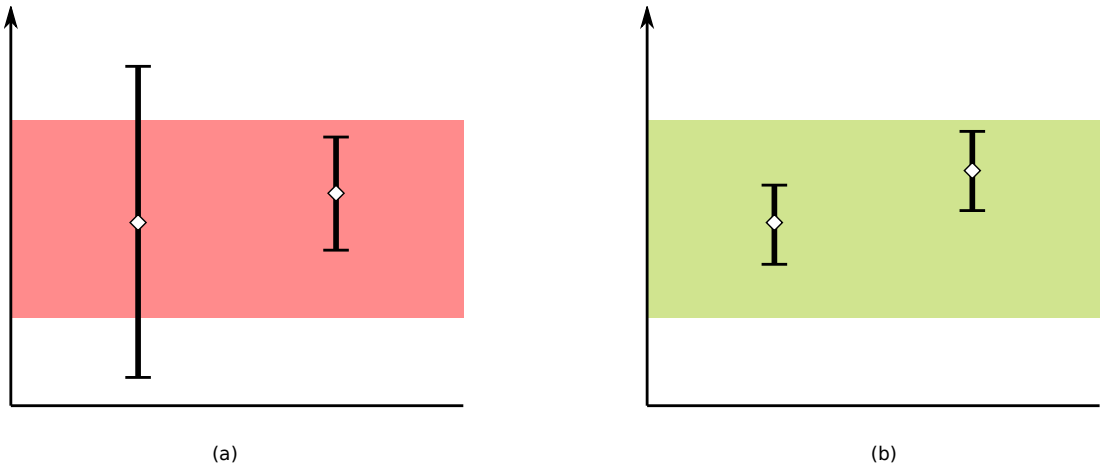


Figure 6.5: Equivalence testing with indifference regions. In (a), the confidence intervals of two samples are shown together with an indifference region. The samples are not equivalent since at least one confidence interval is outside the region. In (b), the confidence intervals are completely inside the region so the samples are considered equivalent.

In the happy case that one version has beaten all others, we can return that single winner. If more survive then we check to see if they are all sufficiently close together for the compiler writer, decided by function $candidatesEqual$.

Detecting that two distributions are approximately equal is the domain of equivalence testing (Wellek (2003)). The archetypal equivalence test is based on Westlake intervals (**?**) wherein a confidence interval is formed for the difference between two means and if that interval is completely contained within some 'indifference region' about zero then the distributions are considered equal as shown in figure 6.5. Indifference regions cannot be selected analytically, it is up to the compiler writer to express when distributions are equivalent.

In our case, the Westlake interval would require the indifference region to be specified as a difference of an absolute number of cycles. It is more natural, instead, to describe the indifference region as an upper bound on the ratio of means. Our justification for this is that speed ups and slow downs are important in compiler fields, but rarely are absolute cycle counts. We might consider a speed up of 1.001 to be uninteresting, regardless of whether it represents one million or ten billion cycles; conversely a speed up of 1.5 is likely exciting with similar disregard for the number of cycles in the difference.

We expect compiler writers to be interested in the version returned with the lowest

mean (the rest of the set we expect to be only of cursory interest, except perhaps to machine learning tools). We use this information to tune our equivalence test to the problem. Since the compiler writer will choose the one from the non losing set with the lowest mean, we wish to ensure, to some confidence, that none of the other non losers could have provided much of a speed up compared to that choice.

We can determine the most available speed up between two program versions by taking a confidence interval for each sample:

$$upper = \mu + t_{(\alpha_{EQ},n-1)}\sqrt{s^2/n}$$

$$lower = \mu - t_{(\alpha_{EQ},n-1)}\sqrt{s^2/n}$$

where t is the student's t upper percentage point value for a given user supplied probability, α_{EQ}, and μ, s, and n are as before.

Now, if the program version with the lowest mean is b, and we have another version, c, we can calculate the worst case speed up of c over b by comparing the upper end of b's confidence interval against the lower end of c's interval. However, we must remember that we initially transformed our observations with a logarithm transform. If we simply compare the interval ends directly we will not be describing speedups; we must apply the inverse transform, first (note that it is this transformation which prevents us using Fieller's theorem (E.C.Fieller (1954); M.A.Creasy (1956)) for the confidence interval of the ratio of two means). The speedup is:

$$worstspeedup_{b,c} = \frac{e^{upper_b}}{e^{lower_c}} = e^{(upper_b - lower_c)}$$

This shows us the speed up in the worst case where the true value of the mean for version b is at the upper end of its confidence interval and the true mean for c is at the lower end of its interval.

The compiler writer, having already given a significance level, α_{EQ}, must also now specify an indifference region, ε. This gives the maximum worst case speed up for the equivalence test. Putting this together, we now have a relation, $=_{EQ,\varepsilon}$, used in line 16, over pairs of samples such that:

$$(S_b, S_c) \in =_{EQ,\varepsilon} \leftrightarrow 1 + \varepsilon < worstspeedup_{b,c}$$

Other stopping conditions are possible, but we believe that this estimate of the nearness of the run times closely matches the requirements of iterative compilation. If the current experiment at hand needs a different stopping condition it should be easy to adjust our algorithm to it.

6.3.5 Limiting Total Sample Size

Our algorithm also gives the compiler writer the opportunity to place hard limits on the total sample size through the constant, *MAX_SAMPLE_SIZE* in function *sampleThresholdReached*. This fixed limit makes ours a restricted sampling plan (Wetherill and Glazebrook (1986)); other methods exist to ensure closed sample boundaries (P. and M.J.R. (1957)) and may be worth considering, although we have found that the restricted plan is quite adequate.

Having a hard sample size limit allows the compiler writer to choose how keen they are for correct results. A large limit permits the algorithm to expend more effort disambiguating difficult cases. Feedback is given when the limit is reached so that in those cases the compiler writer can either accept the best estimate so far from the algorithm or reject, deciding that further effort is not warranted. With a combination of this limit and the two significance levels, the compiler writer can tune the breadth and accuracy of the space they wish to explore.

6.4 Experimental Setup

In this section we briefly describe the experimental set up. We performed two different experiments; the first was to find the best unroll factor for loops, while the second was to find the best compiler flags for whole benchmarks.

6.4.1 Experiments

6.4.1.1 Loop Unrolling Experiment

Loop unrolling has been targeted in a number of previous machine learning and iterative compilation works (Monsifrot et al. (2002); Stephenson and Amarasinghe (2005)). Being a fine grained optimisation, there is a wide range of variation in noise to signal ratios in different loops.

6.4.1.2 Compiler Flags Experiment

Finding the best compiler flags has also been widely explored in both iterative compilation and machine learning (Fursin et al. (2008b)). The very long data gathering phases, often equating to months of compute time (Fursin et al. (2008b)), make efficiency of paramount importance.

6.4.2 Compiler Setup

For both experiments we used GCC 4.3.1. In the first we extended the compiler to allow unroll factors to be explicitly specified for each loop in a program. In the second we altered GCC to accept command line arguments externally, regardless of the benchmarks' makefile. This allowed us to force different compilation flags to be used.

6.4.3 Benchmarks

6.4.3.1 Loop Unrolling Experiment

For the loop unrolling experiment we took 22 embedded benchmarks from the MediaBench and UTDSP benchmark suites. Those benchmarks which did not compile immediately, without any modification except updating path variables, were excluded.

6.4.3.2 Compiler Flags Experiment

For the compiler flags experiment we added the MiBench suite, extending the number of benchmarks to 57. Again, we excluded those which did not immediately compile.

6.4.4 Platform

These experiments were run on an unloaded, headless machine; an Intel dual core Pentium 6 running at 2.8 GHz with 2Gb of RAM. We used a fast machine so that we could gather sufficient data for the naïve, constant sized sampling plans.

6.4.5 Data Generation

6.4.5.1 Loop Unrolling Experiment

For loop unrolling, we selected each of the 230 loops in the benchmarks and unrolled them different number of times. Each loop was unrolled between 0 and 16 times inclusive, giving a total of 17 different program versions per loop. Only one loop was modified at any one time, meaning that a program with ten loops would be compiled 170 times. This allowed each loop to be considered in isolation.

The current loop being changed was instrumented to record the cycle count before and after the loop. Each different version of a program was run 1000 times.

6.4.5.2 Compiler Flags Experiment

For the compilation flags experiment, we took 86 of GCC's flags and generated random collections of them. Each flag had a 5% chance of being set in each collection. Each benchmark was compiled using the flags from each new collection. If that produced a different binary than had already been seen for the benchmark, then the binary was run at least 100 times with the cycle count recorded.

On some benchmarks, particularly small ones, the compiler would generate identical binaries for many different flag collections. Thus, the number of different points in the compiler optimisation space varied across the benchmarks, from 34 for `mibench.telecomm.adpc` to 288 for `mediabench. pegwit`.

6.4.6 Failure Rate

We need a method to compare the different sampling plans. We want to ensure that the result of a sampling run chooses a program version which, if it is not the best, is slower than the best by no more than a given amount. If the version chosen by a sampling run is further from the best, then we call the run a failure, else we call it a success. This allows, by repeated running of the sampling plan, to generate mean failure rates and hence compare the quality of different plans; a better plan will have a lower mean failure rate.

Specifically, we desire that the true mean of the performance of whatever program version is finally selected by a plan should be sufficiently near to the true mean of the best possible version. The true mean is estimated by taking the mean of the full data set. We distinguish the estimated true mean as $\overset{*}{\mu}_i$ for the i^{th} program version, as opposed to the sample mean, μ_i.

If some sampling plan selects a version, c, and the best possible version according to the complete data set is b, then we compare ratio of the estimated true means, $\overset{*}{\mu}_b / \, overset * \mu_c$, which gives the slowdown caused by choosing version c over version b. If the ratio is greater than $1 - \theta$, for some positive θ, then we call the trial a success, otherwise it is a failure. In our experiments we fix θ to 0.5%, which means that, to be successful, a sampling run must choose a program version whose true mean is no more than 0.5% slower than the true mean of the best possible version.

6.4.7 Techniques Evaluated

6.4.7.1 Profiled Races

To evaluate our approach, we explored different values of the parameters which define our algorithm. Both the significance level for the less than test in the *losers* function, α_{LT}, and $alpha_{EQ}$, the significance level for the equality test in $candidatesEqual$, were allowed to range over the set $\{0.0001, 0.005, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5\}$. The set of significance levels gives a broad spectrum of significances from extremely high significance at 0.0001 to very low significance at the other end. For example, when $\alpha_{EQ} = 0.5$, the confidence intervals for two samples, even small ones, will likely be very narrow, so the algorithm will be very generous considering if two versions are equivalent. Conversely, if $\alpha_{LT} = 0.0001$, then the confidence interval for two samples will be wide unless either the samples are large or there is a very small standard deviation; the algorithm will be very sure before deciding that one program version outperforms another.

In all cases, the threshold, θ, which determines how far from the best is acceptable in function $candidateEqual$, is set to 0.5%, matching the boundary for failure. The *MAX_ SAMPLE_SIZE* constant is set to the maximum amount of data available in each experiment; in day-to- day use this constant may not be so large. Each parameter setting was run 100 times to determine the mean failure rate and average sample size for those values.

6.4.7.2 Constant Sized Sampling Plan

The first technique we compared against is a straight forward constant sized sampling plan. Here a fixed number of observations is taken of each program version's runtime or cycle count. Again, we ran plans for each sample size 100 times to generate mean failure rates.

6.4.7.3 JavaSTATS

The second method for comparison is the statistically rigorous approach, JavaSTATS (Georges et al. (2007); Blackburn et al. (2006)). JavaSTATS runs each program version until an estimate of the sample's inaccuracy is sufficiently small. The inaccuracy metric is a confidence interval divided by the sample mean. This metric provides a unit-less indicator of the accuracy of the current sample; a value near to zero is an accurate

sample. As the sample size grows to infinity the metric generally approaches zero.

Several parameters are required: α, the significance level for the confidence intervals; *theta*, the threshold at which the metric indicates an accurate sample and minimum and maximum sample sizes. We allowed the significance level and threshold to both range over the set $\{0.0001, 0.005, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5\}$. The minimum and maximum sample sizes were set at 2 and the maximum size of the data respectively, just as for our own algorithm.

6.5 Results

For each of our two experiments we compared average sample sizes and mean failure rates of the different techniques with their different parameter values.

6.5.1 Loop Unrolling Experiment

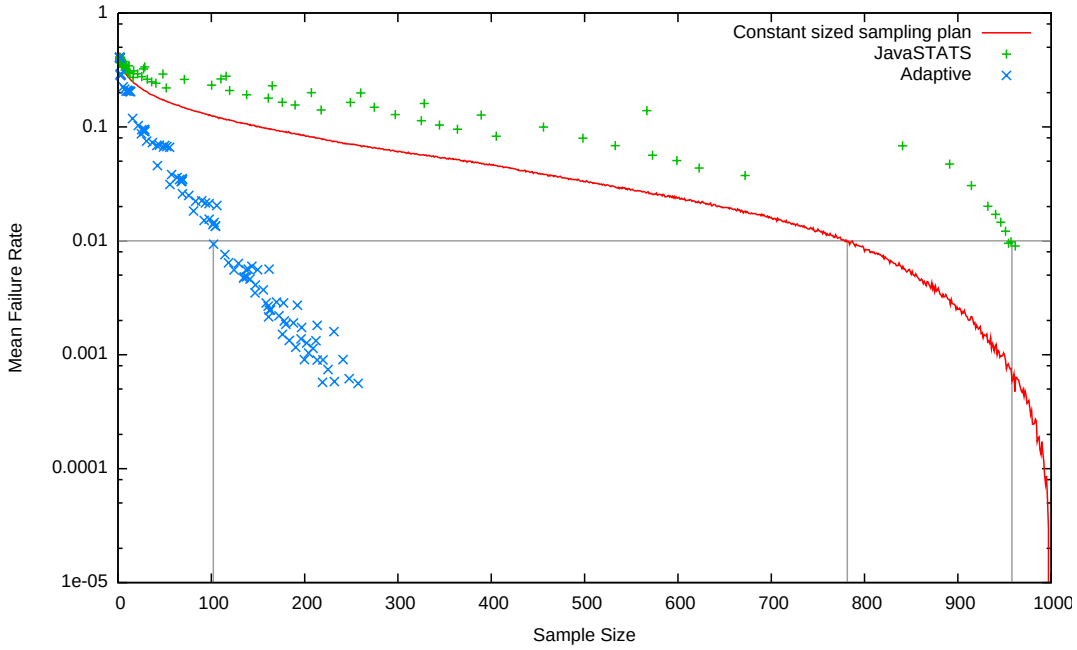
Figure 6.6(a) shows the performance of the different techniques for the loop unrolling experiment. A horizontal line indicates a 1% mean failure rate, an arbitrarily chosen point of comparison.

6.5.1.1 Constant Size Sampling Plan

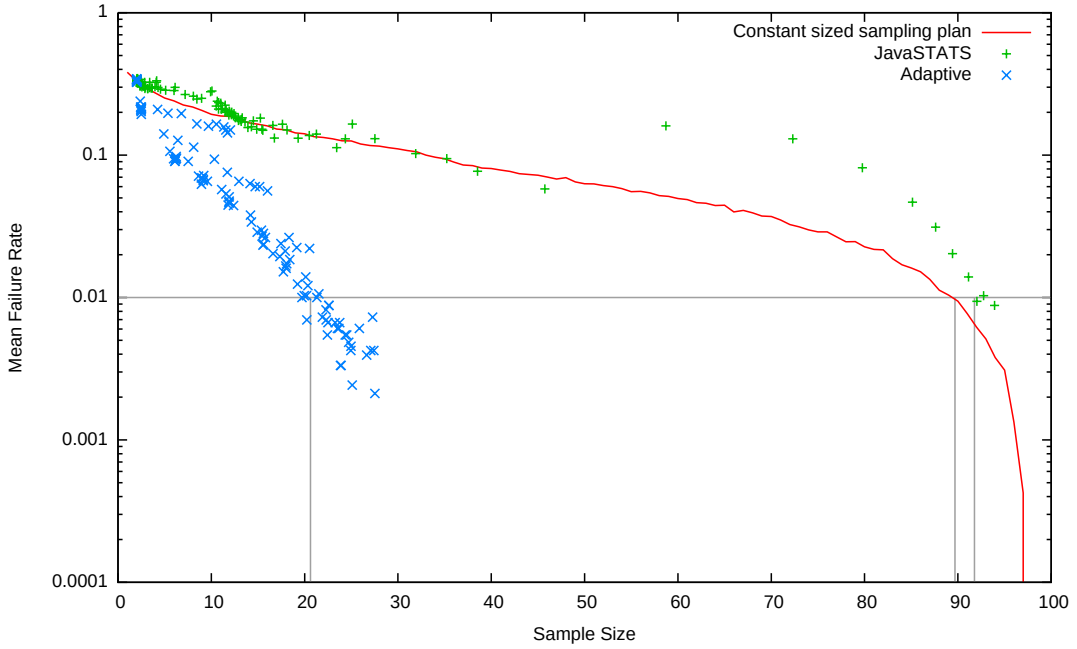
On average, a sample size of 780 or more is needed per unroll factor to achieve a failure rate less than 1%. In practice the size will have to be greater since the compiler writer cannot have prior, perfect knowledge of how many observations are needed in the samples.

6.5.1.2 JavaSTATS Sampling Plan

To manage the 1% failure rate, JavaSTATS needed tight settings of $\alpha = 0.0005$, $\theta = 0.0001$. At these settings, the average sample size was 957, nearly all the data and worse than the constant plan. We attribute this to the fact that it will sometimes fail early, damaging the mean failure rate; punitive settings are needed to compensate, which while stopping the early failures also force large sample sizes when no early failure has occurred. On the other hand, with such aggressive settings, the compiler writer gets good results without perfect knowledge of the right sample size as is required for the constant sampling plans.



(a) Loop Unrolling. At the 1% failure rate, an adaptive plan with $\alpha_{LT} = 0.02$ and $\alpha_{EQ} = 0.02$ needed only 102 samples compared to an optimal fixed sample plan of 780 samples, a reduction of 87%. JavaSTATS required 956 samples to dip below the 1% failure rate, our reduced that by 89%.



(b) Compilation Flags. At the 1% failure rate, an adaptive plan with $\alpha_{LT} = 0.002$ and $\alpha_{EQ} = 0.01$ needed only 21 samples compared to an optimal fixed sample plan of 90 samples, a reduction of 76%. JavaSTATS required 92 samples to dip below the 1% failure rate, our reduced that by 77%.

Figure 6.6: Comparison between our method, constant sized sampling and JavaSTATS (Georges et al. (2007)). Mean failure rates are shown against average sample sizes. A failure is when the given number of samples fails to find a program version that is within 0.5% of the cycle count of the best version. Intersection with a failure rate of 1% is shown.

The many points of our adaptive sampling plan show results with different values of α_{LT}, the significance level for the *losers* function, and α_{EQ}, the significance level for the *candidatesEqual* function, for these, θ, the equivalence threshold, is always 0.5%.

Since each program version in the compiler optimisation space is considered independently of the others, JavaSTATS will spend as much effort producing accurate estimates for the poor ones as for the best. JavaSTATS brings only statistical rigour, not efficient iterative compilation search.

6.5.1.3 Profile Races

Our algorithm performed very well compared to both of the other plans. At $\alpha_{LT} = 0.02$ and $\alpha_{EQ} = 0.02$ our adaptive algorithm first dips below an average failure rate of 1%, getting a failure rate of only 0.94% for an average sample size of only 102. This is 87% less than for the fixed size plan and 89% less than JavaSTATS.

Even applying very strict significance levels of $\alpha_{LT} = 0.0001$ and $\alpha_{EQ} = 0.0001$, our algorithm attained a tiny mean failure rate of just 0.056% for a small increase in sample size to only 257.

6.5.2 Compiler Flags Experiment

Figure 6.6(b) shows the performance of the different techniques for the compiler flags experiment. The graph looks very similar to that of the loop unrolling experiment.

6.5.2.1 Constant Size Sampling Plan

To achieve a failure rate less than 1%, an average sample size of 90 was needed per program version.

6.5.2.2 JavaSTATS Sampling Plan

Once more, JavaSTATS did slightly worse than the constant plans. At settings of $\alpha = 0.001$, $theta = 0.0001$ the failure rate dipped below 1% with an average sample size of 92. Again, it is still better to use JavaSTATS than the constant plan since it does not require knowing the perfect sample size ahead of time.

6.5.2.3 Profile Races

At $\alpha_{LT} = 0.002$ and $\alpha_{EQ} = 0.01$ our algorithm gets a failure rate of less than 1%, needing only 21 observations on average in each sample. This is 76% less than the constant plan and 77% less than JavaSTATS.

Again, cautious use of very strict significance levels, $\alpha_{LT} = 0.0001$ and $\alpha_{EQ} = 0.0001$, is not costly. With these values, our algorithm needs a sample size of just 27 and gets a mean failure rate of 0.021%.

6.5.3 Parameter Sensitivity

Figure 6.7 shows a few contours for different fixed values of the α_{LT} and α_{EQ} significance levels, demonstrating the role those parameters play in controlling the sample size and failure rates. The contours in the figure are for the loop unrolling experiment, but are very similar to those in the compilation flags experiment.

The points with the highest mean failure rate are those with high values of α_{LT} and α_{EQ}. When these are 0.5, for example, the algorithm never increases the samples more than the minimum since at that level confidence intervals are very narrow. The points with the lowest failure rate are those where α_{LT} and α_{EQ} are also the smallest, as is expected since they demand more confidence before either discarding versions or determining equality.

6.5.4 Individual Cases

In the previous section we showed that our adaptive algorithm significantly out-performs a simple, constant sized sampling plan. In this section we show how our sequential sampling plan performs on a number of individual cases, giving a flavour of what to expect.

Figures, 6.8 to 6.11, show the behaviour of our technique over particular examples of the loop unrolling data set. The settings of α_{LT} and α_{EQ} are both 0.02, the point at which the failure rate is 1%.

In each figure, the left hand graph shows the mean cycle count after all data (1000 executions) are considered; the variability of the data is shown with error bars at one standard deviation.

The right hand graph shows how our sequential sampling plan performs, averaged over 100 simulations. The average sample size for each unroll factor is shown together with a one standard deviation error bar to indicate variability.

6.5.4.1 Low Variability, Single Winner

The first example, figure 6.8, shows a scenario where the variability in the cycle count (left hand graph) is very small and there is a single program version which significantly

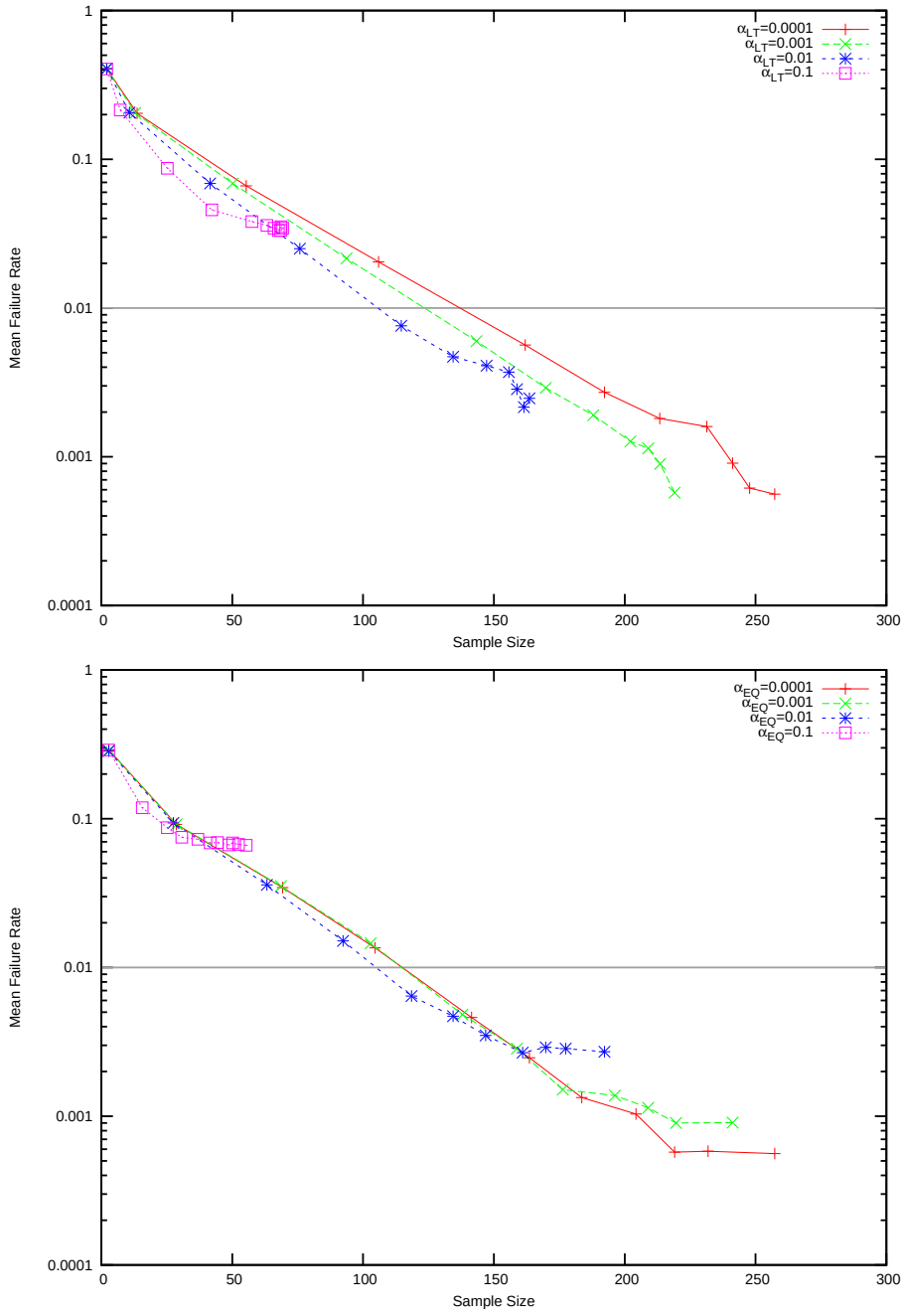


Figure 6.7: Parameter sensitivity. Contours of the two significance levels, α_{LT} and α_{EQ}, are shown for the loop unrolling experiment in figure 6.6. α_{LT} is the the significance level for the less than test in the $losers$ function; α_{EQ} is the significance level for the equality test in $candidatesEqual$

outperforms all of the others. The algorithm excludes the poor versions, often without needing to execute them beyond the minimum number; it only needs to execute from the best three perhaps once more. The average sample size was 2.15, compared to a minimum sample size of 2. This example shows how the well the algorithm handles easy cases.

6.5.4.2 Low Variability, Multiple Winners

The next example, figure 6.9, is only a little harder. Unroll factors 1 and 3 are very close together but the others are poor by comparison. The poorly performing versions are removed very quickly and focus is left on the two remaining candidates. These two are, on average, found to be equivalent after 35 executions each. The average sample size was 6.05.

6.5.4.3 High Variability, Multiple Winners

In figure 6.10, the different versions are more difficult to distinguish. The algorithm needs larger samples to come to a conclusion, but still is able to reduce efforts on poorer versions. For this problem, average sample size was 161.8 with no failures in 100 simulations. On average, the algorithm returned 8 of the 17 unroll factors in the winning set.

6.5.4.4 Sample Exhaustion

Finally, figure 6.11, shows a much harder case. Here some of the versions could not be culled and, at the same time, could not be proved to be equal. In 97% of the sampling runs the algorithm terminated because the maximum sample size limit was reached. The average number of samples was 454.9 and again there were no failures.

6.6 Summary

In this chapter we have shown that using fixed sized sampling plans can have unintended consequences for performance measurement and iterative compilation. Too small a sample can generate incorrect results. Noisy data can make a small sample appear to have a promising mean where a larger sample would give a very different

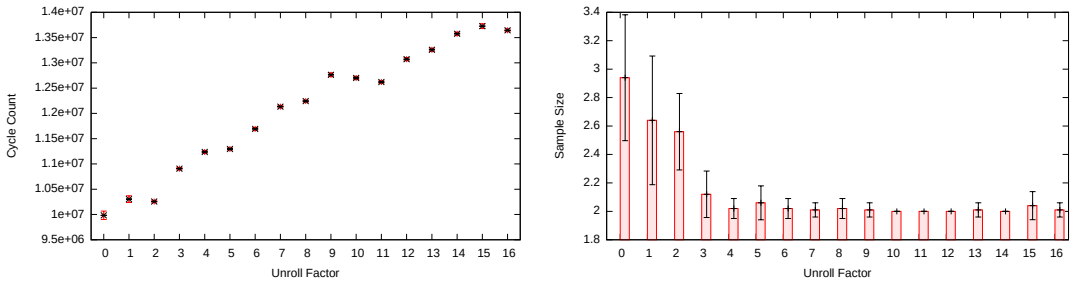


Figure 6.8: A loop from `MediaBench gsm-encode`, function `Gsm_preprocess`. If the winner is clear very early on, then very small sample sizes will result.

Error bars are, in both graphs, one standard deviation.

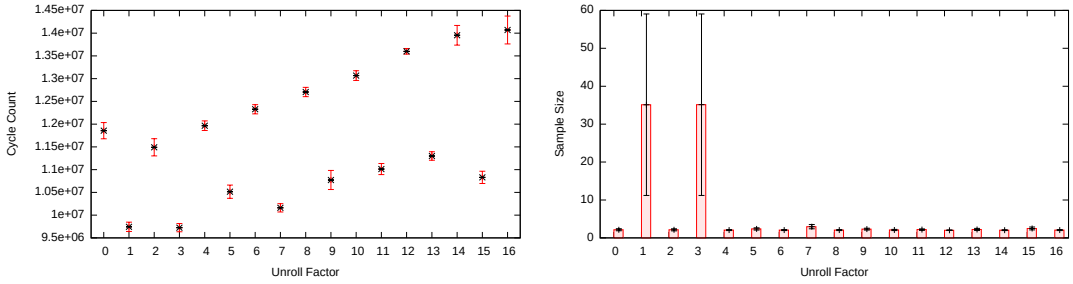


Figure 6.9: A loop from `MediaBench adpcm-decode`, function `adpcm_decoder`. Poorly candidates are quickly discarded and effort focused on the remaining set.

Error bars are, in both graphs, one standard deviation.

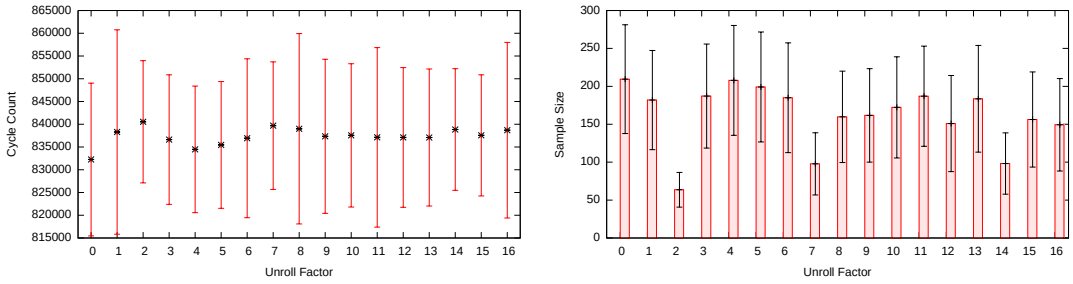


Figure 6.10: A loop from `MediaBench jpeg-encode`, function `emit_eobrun`. When the relative noise is large more samples must be taken.

Error bars are, in both graphs, one standard deviation.

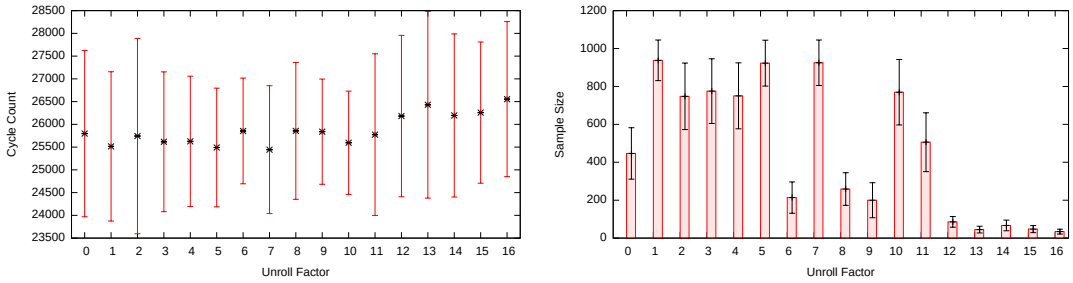


Figure 6.11: A loop from `UTDSP fft_1024`, function `main`. Sometimes the noise will case the sample limit restriction to be reached.

Error bars are, in both graphs, one standard deviation.

answer. It is not enough to only look at the means of performance measurements, the statistical significance of the results must be taken into account.

Too large a sample wastes excessive work since program versions are executed an unnecessary number of times. Often, some program versions could be discarded early but a fixed sampling plan is oblivious to these opportunities. With fixed sampling plans, the user must choose, ahead of time, how many runs each version needs to get good data. This is very difficult to do without already having data to examine.

We have provided an algorithm which automatically adapts to the requirements of the problem at hand. In cases where there is little noise and a clear winner is visible early the algorithm will take very few samples. When particular versions require larger sample sizes to disambiguate them the algorithm does just that. Finally, the algorithm terminates when versions are equivalent so that no further work is done trying to tell identical versions apart.

We applied our technique to finding the best loop unrolling factor for a number of loops from different benchmarks and also to finding the best compiler flags for whole programs. Some loops and programs generated noisy data while others had relatively clean data. Our method was able to adapt to these differences, choosing different sample sizes in each case. We reduced the cost of iterative compilation by between 76% and 87% compared to a fixed sized sampling plan. Compared to JavaSTATS (Georges et al. (2007)), we reduced the cost by between 77% and 89%.

There are other possible formulations of the algorithm, using different criteria to remove versions and decide when to stop. We will explore these in the future.

Chapter 7

Conclusion

Bibliography

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

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Almagor, L., Cooper, K. D., Grosul, A., Harvey, T. J., Reeves, S. W., Subramanian, D., Torczon, L., and Waterman, T. (2004). Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 231–239, New York, NY, USA. ACM.

Beaty, S. J. (1991). Genetic algorithms and instruction scheduling. In *MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture*, pages 206–211, New York, NY, USA. ACM Press.

Bensusan, H. and Kuscu, I. (1996). Constructive induction using genetic programming. In Fogarty, T. and Venturini, G., editors, *ICML'96, Evolutionary computing and Machine Learning Workshop*.

Biswas, P., Banerjee, S., Dutt, N., Ienne, P., and Pozzi, L. (2006a). Performance and energy benefits of instruction set extensions in an fpga soft core. In *Proceedings of the 19th International Conference on VLSI Design*, pages 651–656. IEEE.

Biswas, P., Dutt, N., Ienne, P., and Pozzi, L. (2006b). Automatic identification of application-specific functional units with architecturally visible storage. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 212–217. EDAA.

Blackburn, S. M., McKinley, K., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). The dacapo benchmarks: Java benchmarking development and analysis. In *OOPSLA 2006, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*.

Bland, J. M. and Altman, D. G. (1996). Transforming data. *BMJ (Clinical research ed.)*, 312.

B.L.Welch (1947). The generalization of 'student's' problem when several different population varlances are involved. *Biometrika*, 34:28–35.

Bodin, F., Kisuk, T., Knijnenburg, P. M. W., O'Boyle, M. F. P., and Rohou, E. (1998). Iterative compilation in a non-linear optimisation space. In *Workshop on Prole 14 and Feedback-Directed Compilation, in Conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

Box, G. E. P. and Cox, D. R. (1964). An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, 26(2):211–252.

Brisk, P. and Sarrafzadeh, M. (2006). *Datapath Synthesis*, chapter 10, pages 233–255. Morgan Kaufmann.

Cavazos, J. and Moss, J. E. B. (2004). Inducing heuristics to decide whether to schedule. *SIGPLAN Not.*, 39(6):183–194.

Cavazos, J., Moss, J. E. B., and O'Boyle, M. F. (2006). Hybrid optimizations: Which optimization algorithm to use? *Compiler Construction*.

Cavazos, J. and OBoyle, M. F. (2005). Automatic tuning of inlining heuristics. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 14. IEEE Computer Society.

Cooper, K. D., Grosul, A., Harvey, T. J., Reeves, S., Subramanian, D., Torczon, L., and Waterman, T. (2005). Acme: adaptive compilation made efficient. *SIGPLAN Not.*, 40(7):69–77.

Cooper, K. D., Schielke, P. J., and Subramanian, D. (1999). Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9.

Cooper, K. D., Subramanian, D., and Torczon, L. (2002). Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23:2002.

E.C.Fieller (1954). Some problems in interval estimation. *Journal of the Royal Statistical Society*, 16:175–185.

Fisher, J. A., Faraboschi, P., and Young, C. (2006). *Customizing Processors: Lofty Ambitions, Stark Realities*, chapter 3, pages 39–55. Morgan Kaufmann.

Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Barnard, P., Ashton, E., Courtois, E., Bodin, F., Bonilla, E., Thomson, J., Leather, H., Williams, C., and O'Boyle, M. (2008a). Milepost gcc: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*.

Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Barnard, P., Ashton, E., Courtois, E., Bodin, F., Bonilla, E., Thomson, J., Leather, H., Williams, C., and OB́oyle, M. (2008b). Milepost gcc: machine learning based research compiler. In *Proceedings of the GCC DevelopersŚummit*.

Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76.

Goodwin, D., Leibson, S., and Martin, G. (2006). *Automated Processor Configuration and Instruction Extension*, chapter 6, pages 117–143. Morgan Kaufmann.

Haneda, M., Knijnenburg, P. M. W., and Wijshoff, H. A. G. (2005). Automatic selection of compiler options using non-parametric inferential statistics. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, Washington, DC, USA. IEEE Computer Society.

Hoeffding, W. (1963). Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30.

Kohavi, R. and John, G. H. (1997). Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324.

Koza, J. R. (1990a). Evolution and co-evolution of computer programs to control independent-acting agents. In Meyer, J.-A. and Wilson, S. W., editors, *From Animals to Animats: Proceedings of the First International Conference on Simulation*

of Adaptive Behavior, 24-28, September 1990, pages 366–375, Paris, France. MIT Press.

Koza, J. R. (1990b). The genetic programming paradigm: Genetically breeding populations of computer programs to solve problems. In Soucek, B. and the IRIS Group, editors, *Dynamic, Genetic, and Chaotic Programming*, pages 203–321. John Wiley, New York.

Kulkarni, P., Zhao, W., Moon, H., Cho, K., Whalley, D., Davidson, J., Bailey, M., Paek, Y., and Gallivan, K. (2003). Finding effective optimization phase sequences. *SIGPLAN Not.*, 38(7):12–23.

M.A.Creasy (1956). Confidence limits for the gradient in the linear functional relationship. *Journal of the Royal Statistical Society*, 18:64–69.

Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics*, 18:50–60.

Maron, O. and Moore, A. W. (1994). Hoeffding races: Accelerating model selection search for classification and function approximation. In *Advances in neural information processing systems 6*, pages 59–66. Morgan Kaufmann.

Maron, O. and Moore, A. W. (1997). The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11:193–225.

Mcgovern, A. and Moss, E. (1998). Scheduling straight-line code using reinforcement learning and rollouts. In *In Proceedings of Neural Information Processing Symposium*. MIT Press.

Mierswa, I. (2004). Automatic feature extraction from large time series.

Monsifrot, A., Bodin, F., and Quiniou, R. (2002). A machine learning approach to automatic production of compiler heuristics.

Moss, E., Utgoff, P., Cavazos, J., Precup, D., Stefanović, D., Brodley, C., and Scheeff", D. (1998). Learning to schedule straight-line code. In Jordan, M. I., Kearns, M. J., and Solla, S. A., editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press.

Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! In *ASPLOS '09: Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems*.

P., A. and M.J.R., H. (1957). Interpretation of χ^2 tests. *Biometrics*, 13:113–115.

Pan, Z. and Eigenmann, R. (2006). Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332, Washington, DC, USA. IEEE Computer Society.

Parameswaran, S., Henkel, J., and Cheung, N. (2006). *Instruction Matching and Modeling*, chapter 11, pages 258–280. Morgan Kaufmann.

Pozzi, L. and Ienne, P. (2006). *Automatic Instruction-Set Extensions*, chapter 7, pages 145–183. Morgan Kaufmann.

Ritthoff, O., Klinkenberg, R., Fischer, S., and Mierswa, I. (2002). A hybrid approach to feature selection and generation using an evolutionary algorithm.

Ryan, C., Azad, A., Sheahan, A., and O'Neill, M. (2002). No coercion and no prohibition, A position independent encoding scheme for evolutionary algorithms—the Chorus system. In Foster, J. A., Lutton, E., Miller, J., Ryan, C., and Tettamanzi, A. G. B., editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278, pages 131–141, Kinsale, Ireland. Springer-Verlag.

Ryan, C., Collins, J. J., and O Neill, M. (1998). Grammatical evolution: Evolving programs for an arbitrary language. In Banzhaf, W., Poli, R., Schoenauer, M., and Fogarty, T. C., editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391, pages 83–95, Paris. Springer-Verlag.

Satterthwaite, F. E. (1946). An approximate distribution of estimates of variance components. *Biometrics Bulletin*, 2:110–114.

Schlkopf, B. and Smola, A. J. (2001). *Learning with Kernels: Support Vector Machines Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA.

Stephenson, M. and Amarasinghe, S. (2005). Predicting unroll factors using supervised classification. In *CGO '05: Proceedings of the international symposium on*

Code generation and optimization, pages 123–134, Washington, DC, USA. IEEE Computer Society.

Stephenson, M., Amarasinghe, S., Martin, M., and O'Reilly, U.-M. (2003). Meta optimization: Improving compiler heuristics with machine learning.

(Student), W. (1908). The probable error of a mean. *Biometrika*, 6:1–25.

Taylor, R. and Stewart, D. (2006). *Coprocessor Generation from Executable Code*, chapter 9, pages 209–232. Morgan Kaufmann.

Topham, N. (2006). *Challenges to Automatic Customization*, chapter 8, pages 186–208. Morgan Kaufmann.

Triantafyllis, S., Vachharajani, M., Vachharajani, N., and August, D. I. (2003). Compiler optimization-space exploration. In *Journal of Instruction-level Parallelism*, pages 204–215. IEEE Computer Society.

Vafaie, H. and DeJong, K. (1993). Robust feature selection algorithms. *Proc. 5th Intl. Conf. on Tools with Artifical Intelligence*, pages 356–363.

Wald, A. (1947). *Sequential Analysis*.

Walsh, P. and Ryan, C. (1995). Automatic conversion of programs from serial to parallel using genetic programming – the Paragen system. In D'Hollander, E. H., Joubert, G. R., Peters, F. J., and Trystram, D., editors, *Parallel Computing: State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Belgium*, volume 11, pages 415–422, Amsterdam. Elsevier, North-Holland.

Wellek, S. (2003). *Testing Statistical Hypotheses of Equivalence*. CRC Press.

Wetherill, G. and Glazebrook, K. (1986). *Sequential methods in statistics. 3rd ed.* London: Chapman and Hall.

Whitehead, J. (1992). *The Design and Analysis of Sequential Trials*. Ellis Horwood.

Yotov, K., Li, X., Ren, G., Cibulskis, M., DeJong, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P., and Wu, P. (2003). A comparison of empirical and model-driven optimization. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 63–76, New York, NY, USA. ACM.