A Machine Learning Approach to Automatic Production of Compiler Heuristics

Antoine Monsifrot, François Bodin, and René Quiniou

IRISA-University of Rennes France
{amonsifr,bodin,quiniou}@irisa.fr

Abstract. Achieving high performance on modern processors heavily relies on the compiler optimizations to exploit the microprocessor architecture. The efficiency of optimization directly depends on the compiler heuristics. These heuristics must be target-specific and each new processor generation requires heuristics reengineering.
In this paper, we address the automatic generation of optimization heuristics for a target processor by machine learning. We evaluate the potential of this method on an always legal and simple transformation: loop unrolling. Though simple to implement, this transformation may have strong effects on program execution (good or bad). However deciding to perform the transformation or not is difficult since many interacting parameters must be taken into account. So we propose a machine learning approach.
We try to answer the following questions: is it possible to devise a learning process that captures the relevant parameters involved in loop unrolling performance? Does the Machine Learning Based Heuristics achieve better performance than existing ones?

Keywords: decision tree, boosting, compiler heuristics, loop unrolling.

1 Introduction

Achieving high performance on modern processors heavily relies on the ability of the compiler to exploit the underlying architecture. Numerous program transformations have been implemented in order to produce efficient programs that exploit the potential of the processor architecture. These transformations interact in a complex way. As a consequence, an optimizing compiler relies on internal heuristics to choose an optimization and whether or not to apply it. Designing these heuristics is generally difficult. The heuristics must be specific to each implementation of the instruction set architecture. They are also dependent on changes made to the compiler.

In this paper, we address the problem of automatically generating such heuristics by **a machine learning approach**. To our knowledge this is the first study of machine learning to build these heuristics. The usual approach consists in running a set of benchmarks to setup heuristics parameters. Very few

D. Scott (Ed.): AIMSA 2002, LNAI 2443, pp. 41–50, 2002.
© Springer-Verlag Berlin Heidelberg 2002

papers have specifically addressed the issue of building such heuristics. Nevertheless approximate heuristics have been proposed [8,11] for *unroll and jam* a transformation that is like unrolling (our example) but that behaves differently.

Our study aims to simplify compiler construction while better exploiting optimizations. To evaluate the potential of this approach we have chosen a simple transformation: loop unrolling [6]. Loop unrolling is always legal and is easy to implement, but because it has many side effects, it is difficult to devise a decision rule that will be correct in most situations.

In this novel study we try to answer the following questions: is it possible to learn a decision rule that selects the parameters involved in loop unrolling efficiency? Does the Machine Learning Based Heuristics (denoted MLBH) achieve better performance than existing ones? Does the learning process really take into account the target architecture?

To answer the first question we build on previous studies [9] that defined an abstract representation of loops in order to capture the parameters influencing performance. To answer the second question we compare the performance of our Machine Learning Based Heuristics and the GNU Fortran compiler [3] on a set of applications. To answer the last question we have used two target machines, an UltraSPARC machine [12] and an IA-64 machine [7], and used on each the MLBH computed on the other.

The paper is organized as follows. Section 2 gives an overview of the loop unrolling transformation. Section 3 shows how machine learning techniques can be used to automatically build loop unrolling heuristics. Section 4 illustrates an implementation of the technique based on the OC1 decision tree software [10].

2 Loop Unrolling as a Case

The performance of superscalar processors relies on a very high frequency[1] and on the parallel execution of multiple instructions (this is also called *Instruction Level Parallelism* –ILP). To achieve this, the internal architecture of superscalar microprocessors is based on the following features:

Memory hierarchy: the main memory access time is typically hundreds of times greater than the CPU cycle time. To limit the slowdown due to memory accesses, a set of intermediate levels are added between the CPU unit and the main memory; the level the closest to the CPU is the fastest, but also the smallest. The data or instructions are loaded by blocks (sets of contiguous bytes in memory) from one memory level of the hierarchy to the next level to exploit the following fact: when a program accesses some memory element, the next contiguous one is usually also accessed in the very near future. In a classical configuration there are 2 levels, L1 and L2 of cache memories as shown on the figure 1. The penalty to load data from main memory tends to be equivalent to executing 1000 processor cycles. If the data is already in L2, it is one order of magnitude less. If the data is already in L1, the access can be done in only a few CPU cycles.

[1] typically 2 gigahertz corresponding to a processor cycle time of 0.5 nanosecond

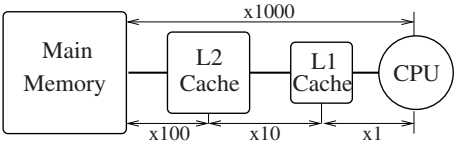

Fig. 1. Memory hierarchy

Multiple Pipelined Functional Units: the processor has multiple functional units that can run in parallel to execute several instructions per cycle (typically an integer operation can be executed in parallel with a memory access and a floating point computation). Furthermore these functional units are pipelined. This divides the operation in a sequence of steps that can be performed in parallel. Scheduling instructions in the functional units is performed in an *out-of-order* or *in-order* mode. Contrary to the *in-order*, in the *out-of-order* mode instructions are not always executed in the order specified by the program. When a processor runs at maximum speed, each pipelined functional unit executes one instruction per cycle. This requires that all operands and branch addresses are available at the beginning of the cycle. Otherwise, functional units must wait during some delays. The processor performance depends on these waiting delays.

The efficiency of the memory hierarchy and ILP are directly related to the structure and behavior of the code. Many program transformations reduce the number of waiting delays in program execution.

Loop unrolling [6] is a simple program transformation where the loop body is replicated many times. It may be applied at source code level to benefit from all compiler optimizations. It improves the exploitation of ILP: increasing the size of the body augments the number of instructions eligible to *out-of-order* scheduling. Loop unrolling also reduces loop management overhead but it has also some beneficial side effects from later compiler steps such as common sub-expression elimination. However it also has many negative side effects that can cancel the benefits of the transformation:

- the instruction cache behavior may be degraded (if the loop body becomes too big to fit in the cache),
- the register allocation phase may generate spill code (additional load and store instructions),
- it may prevent other optimization techniques.

As a consequence, it is difficult to fully exploit loop unrolling. Compilers are usually very conservative. Their heuristics are generally based on the loop body size: under a specific threshold, if there is no control flow statement, the loop is unrolled. This traditional approach under-exploits loop unrolling [5] and must be adapted when changes are made to the compiler or to the target architecture.

The usual approach to build loop unrolling heuristics for a given target computer consists in running a set of benchmarks to setup the heuristics parameters. This approach is intrinsically limited because in most optimizations such as loop unrolling, too many parameters are involved. Microarchitecture characteristics

(for instance the size of instruction cache, ...) as well as the other optimizations (for instance instruction scheduling, ...) that follow loop unrolling during the compilation process should be considered in the decision procedure. The main parameters, but not all (for instance number instruction cache misses), which influence loop unrolling efficiency directly depend on the loop body statements. This is because loop unrolling mainly impacts code generation and instruction scheduling. As a consequence, it is realistic to base the unrolling decision on the properties of the loop code while ignoring its execution context.

3 Machine Learning for Building Heuristics

Machine learning techniques offer an automatic, flexible and adaptive framework for dealing with the many parameters involved in deciding the effectiveness of program optimizations. Classically a decision rule is learnt from feature vectors describing positive and negative applications of the transformation. However, it is possible to use this framework only if the parameters can be abstracted statically from the loop code and if their number remains limited. Reducing the number of parameters involved in the process is important as the performance of machine learning techniques is poor when the number of dimensions of the learning space is high [10]. Furthermore learning from complex spaces requires more data.

To summarize the approach, the steps involved in using a machine learning technique for building heuristics for program transformation are:

1. finding a loop abstraction that captures the "performance" features involved in an optimization, in order to build the learning set,
2. choosing an automatic learning process to compute a rule in order to decide whether loop unrolling should be applied,
3. setting up the result of the learning process as heuristics for the compiler.

In the remainder of this section, we present the representation used for abstracting loop properties. The next section shows how to sort the loop into winning and loosing classes according to unrolling. Finally, the learning process based on learning decision trees is overviewed.

3.1 Loop Abstraction

The loop abstraction must capture the main loop characteristics that influence the execution efficiency on a modern processor. They are represented by integer **features** which are relevant static loop properties according to unrolling. We have selected 5 classes of integer features:

Memory access: number of memory accesses, number of array element reuses from one iteration to another.

Arithmetic operations count: number of additions, multiplications or divisions excepting those in array index computations.

Size of the loop body: number of statements in the loop.

Control statements in the loop: number of if statements, goto, etc. in the loop body.
Number of iterations: if it can be computed at compile time.

In order to reduce the learning complexity, only a subset of these features are used for a given compiler and target machine. The chosen subset was determined experimentally by cross validation (see Section 4). The quality of the predictions achieved by an artificial neural network based on 20 indices was equivalent to the predictive quality of the 6 chosen features.

Figure 2 gives the features that were selected and an example of loop abstraction.

```
do i=2,100
  a(i) = a(i)+a(i-1)*a(i+1)
enddo
```

Number of statements	1
Number of arithmetic operations	2
Minimum number of iterations	99
Number of array accesses	4
Number of array element reuses	3
Number of if statements	0

Fig. 2. Example of features for a loop.

3.2 Unrolling Beneficial Loops

A learning example refers to a loop in a particular context (represented by the loop features). To determine if unrolling is beneficial, each loop is executed twice. Then, the execution times of the original loop and of the unrolled loop are compared. Four cases can be considered:

not significant: the loop execution time is too small and therefore the timing is not significant. The loop is discarded from the learning set.
equal: the execution times of the original and of the unrolled loop are close. A threshold is used to take into account the timer inaccuracy. Thus, the loop performance is considered as invariant by unrolling if the benefit is less than 10%.
improved: the speedup is above 10%. The loop is considered as being **beneficial** by unrolling.
degraded: there is a speed-down. The loop is considered as a degraded loop by unrolling.

The loop set is then partitioned into equivalence classes (denoted loop classes in the remainder). Two loops are in the same loop class if their respective abstractions are equal.

The next step is to decide if a loop class is to be considered as a positive or a negative example. Note that there can be beneficial and degraded loops in the same class as non exhaustive descriptions are used to represent the loops. This is a natural situation as the loop execution or compilation context may greatly influence its execution time, for instance due to instruction cache memory effects. The following criterion has been used to decide whether a class will represent a positive or a negative example:

1. In a particular class, a loop whose performance degrades by less than 5% is counted once, a loop that degrades performance by 10% is counted twice. A loop that degrades performance more than 20% is counted three times.
2. if the number of unrolling beneficial loops is greater than the number of degraded loops (using the weights above), then the class represents a positive example, else the class represents a negative example.

3.3 A Learning Method Based on Decision Trees and Boosting

We have chosen to represent unrolling decision rules as decision trees. Decision trees can be learnt efficiently from feature based vectors. Each node of the decision tree represents a test checking the value(s) of one (or several) feature(s) which are easy to read by an expert. This is not the case for statistical methods like Nearest Neighbor or Artificial Neural Network for instance, which have comparable or slightly better performance.

We used the OC1 [10] software. OC1 is a classification tool that induces oblique decision trees. Oblique decision trees produce polygonal partitionings of the feature space. OC1 recursively splits a set of objects in a hyperspace by finding optimal hyperplanes until every object in a subspace belongs to the same class.

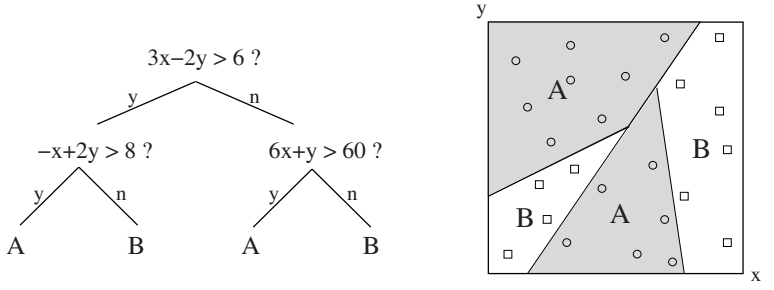

Fig. 3. The left side of the figure shows an oblique decision tree that uses two attributes. The right side shows the partitioning that this tree creates in the attribute space.

A decision tree example is shown in Figure 3, together with its 2-D related space. Each node of the tree tests a linear combination of some indices (equivalent to an hyperplane) and each leaf of the tree corresponds to a class. The main advantages of OC1 is that it finds smaller decision trees than classical tree learning methods. The major drawback is that they are less readable than classical ones.

The classification of a new loop is equivalent to finding a leaf loop class. Once induced, a decision tree can be used as a classification process. An object represented by its feature vector is classified by following the branches of the tree indicated by node tests until a leaf is reached.

To improve the accuracy obtained with OC1 we have used boosting [13]. Boosting is a general method for improving the accuracy of any given algorithm.

Boosting consists in learning a set of classifiers for more and more difficult problems: the weights of examples that are misclassified by the classifier learnt at step n are augmented (by a factor proportional to the global error) and at step $n+1$ a new classifier is learnt on this weighted examples. Finally, the global classification is obtained by a weighted vote of the individual classifier according to their proper accuracy. In our case 9 trees were computed.

4 Experiments

The learning set used in the experiments is made of loops extracted from programs in Fortran 77. Most of them were chosen in available benchmarks [4,1]. We have studied two types of programs: real applications (the majority comes from SPEC [4]) and computational kernels. Table 1 presents some characteristics of a loop set (cf section 3.2).

Table 1. IA-64 learning set.

Number of loops	1036
Discarded loops	177
Unrolling beneficial loops	233
Unrolling invariant loops	505
Unrolling degraded loops	121
Loop classes	572
Positive examples	139
Negative examples	433

The accuracy of the learning method was assessed by a 10-fold cross-validation. We have experiment with pruning. We have obtained smaller trees but the resulting quality was degraded. The results without pruning are presented in Table 2. Two factors can explain the fact that the overall accuracy cannot be better than 85%:

1. since unrolling beneficial and degraded loops can appear in the same class (cf section 3.2) a significant proportion of examples may be noisy,
2. the classification of positive examples is far worse than the classification of negative ones. Maybe the learning set does not contain enough beneficial loops.

To go beyond cross validation another set of experiments has been performed on two target machines, an UltraSPARC and an IA-64. They aim at showing the technique does catch the most significant loops of the programs. The g77 [3] compiler was used. With this compiler, loop unrolling can be globally turned on and off. To assess our method we have implemented loop unrolling at the source

Table 2. Cross validation accuracy

	UltraSPARC		**IA-64**	
	normal	boosting	normal	boosting
Accuracy of **overall** example classification	79.4%	85.2%	82.6%	85.2%
Accuracy of **positive** example classification	62.4%	61.7%	73.9%	69.6%
Accuracy of **negative** example classification	85.1%	92.0%	86.3%	92.3%

code level using TSF [2]. This is not the most efficient scheme because in some cases this inhibits some of the compiler optimizations (contrary to unrolling performed by the compiler itself). We have performed experiments to check whether the MLB heuristics are at least as good as compiler heuristics and whether the specificities of a target architecture can be taken into account. A set of benchmark programs were selected in the learning set and for each one we have:

1. run the code compiled by g77 with -O3 option,
2. run the code compiled with -O3 -funroll options : the compiler uses its own unrolling strategy,
3. unroll the loops according to the result of the MLB heuristics and run the compiled code with -O3 option. The heuristics was learned for the target machine from learning set where the test program was removed.
4. unroll the loops according to the result of the MLB heuristics learnt for the **other** target machine and run the compiled code with -O3 option.

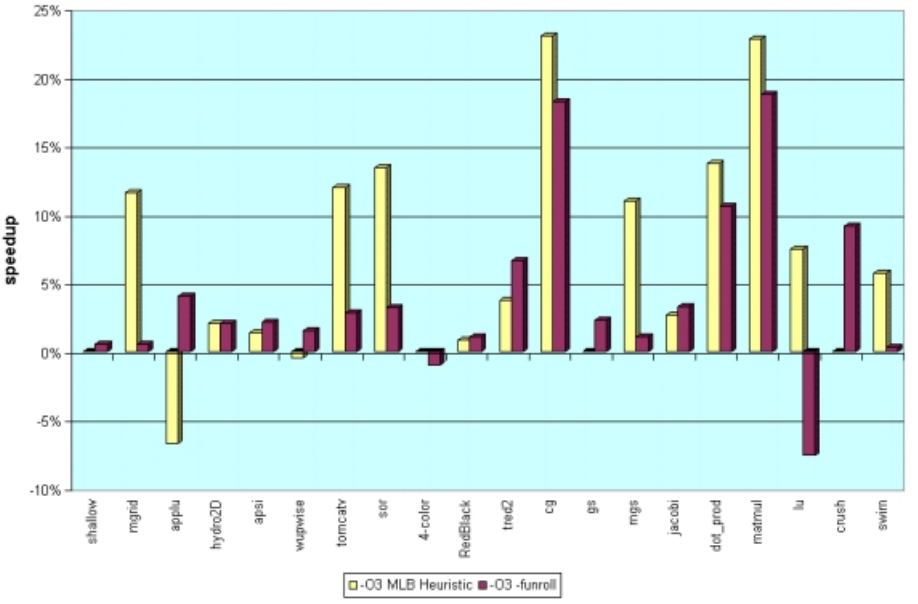

Fig. 4. IA-64 : -O3 is the reference execution time.

The performance results are given in Figure 4 and Figure 5 respectively for the IA-64 and UltraSPARC targets.

The average execution time of the optimized programs for the IA-64 is 93.8% of the reference execution time (no unrolling) using the MLB heuristics and 96.8% using the g77 unrolling strategy. On the UltraSPARC we have respectively 96% and 98.7% showing that our unrolling strategy performs better. Indeed,

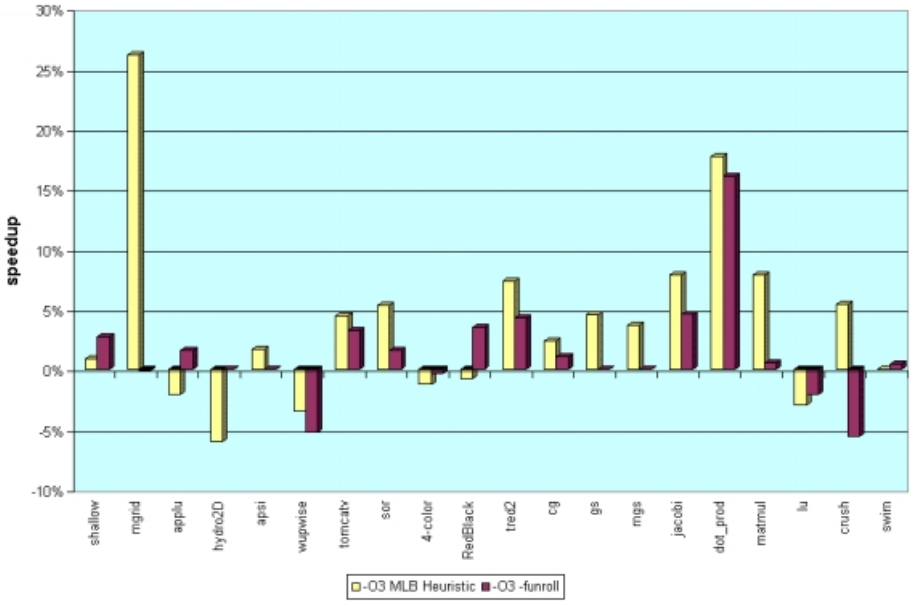

Fig. 5. UltraSPARC : -O3 is the reference execution time.

gaining a few percent on average execution time with one transformation is significant because each transformation is not often beneficial. For example, only 22% of the loops are beneficial by unrolling on IA-64 and 17% on UltraSPARC.

In the last experiment we exchanged the decision trees learnt for the two target machines. On the UltraSPARC, the speedup is degraded from 96% to 97.9% and on the IA-64 it is degraded from 93.8% to 96.8%. This shows that the heuristics are effectively tuned to a target architecture.

5 Conclusion

Compilers implement a lot of optimization algorithms for improving performance. The choice of using a particular sequence of optimizations and their parameters is done through a set of heuristics hard coded in the compiler.

At each major compiler revision, but also at new implementations of the target Instruction Set Architecture, a new set of heuristics must be reengineered.

In this paper, we have presented a new method for addressing such reengineering in the case of loop unrolling. Our method is based on a learning process which adapts to new target architectures or new compiler features. Using an abstract loop representation we showed that decision trees that provide target specific heuristics for loop unrolling can be learnt.

While our study is limited to the simple case of loop unrolling it opens a new direction for the design of compiler heuristics. Even for loop unrolling, there are still many issues to consider to go beyond this first result. Are there better abstractions that can capture loop characteristics? Can hardware counters

(for instance cache miss counters) provide better insight on loop unrolling? How large should the learning set be? Can other machine learning techniques be more efficient than decision trees?

More fundamentally our study raises the question whether it could be possible or not to quasi automatically reengineer the implementation of a set of optimization heuristics for new processor target implementations.

Acknowledgments. We would like to gratefully thank I. C. Lerman and L. Miclet for their insightful advice on machine learning techniques.

References

1. David Bailey. Nas kernel benchmark program, June 1988. `http://www.netlib.org/benchmark/nas.`
2. F. Bodin, Y. Mével, and R. Quiniou. A User Level Program Transformation Tool. In *Proceedings of the International Conference on Supercomputing*, pages 180–187, July 1998, Melbourne, Australia.
3. GNU Fortran Compiler. `http://gcc.gnu.org/`.
4. Standard Performance Evaluation Corporation. `http://www.specbench.org/`.
5. Jack W.Davidson and Sanjay Jinturkar. Aggressive Loop Unrolling in a Retargetable, Optimizing Compiler. In *Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 59–73. Springer, April 1996.
6. J. J. Dongarra and A. R. Hinds. Unrolling loops in FORTRAN. *Software Practice and Experience*, 9(3):219–226, March 1979.
7. IA-64. `http://www.intel.com/design/Itanium/idfisa/index.htm`.
8. A. Koseki, H. Komastu, and Y. Fukazawa. A Method for Estimating Optimal Unrolling Times for Nested Loops. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks*, 1997.
9. A. Monsifrot and F. Bodin. Computer Aided Hand Tuning (CAHT): "Applying Case-Based Reasoning to Performance Tuning". In *Proceedings of the 15th ACM International Conference on Supercomputing (ICS-01)*, pages 196–203. ACM Press, June 17–21 2001, Sorrento, Italy.
10. Sreerama K. Murthy, Simon Kasif, and Steven Salzberg. A System for Induction of Oblique Decision Trees. *Journal of Artificial Intelligence Research*, 2:1–32, 1994.
11. Vivek Sarkar. Optimized Unrolling of Nested Loops. In *Proceedings of the 14th ACM International Conference onSupercomputing (ICS-00)*, pages 153–166. ACM Press, May 2000.
12. UltraSPARC. `http://www.sun.com/processors/UltraSPARC-II/`.
13. C. Yu and D.B. Skillicorn. Parallelizing Boosting and Bagging. Technical report, Queen's University, Kingston, Ontario, Canada K7L 3N6, February 2001.