Instruction Scheduling using Evolutionary Programming

ANJALI MAHAJAN\textsuperscript{1} M S ALI\textsuperscript{2} MAMTA PATIL\textsuperscript{3}
\textsuperscript{1}Department of Information Technology, G H Raisoni College of Engineering, Nagpur, India
\textsuperscript{2}Department of Computer Science and Engineering, College of Engineering, Badnera, India
\textsuperscript{3}Department of Computer Technology, G H Raisoni Polytechnic, Nagpur, India

Abstract: - Instruction scheduling is the problem of scheduling the assembly instructions output from the code generator to increase the efficiency of the final code. The instruction scheduling problem is mainly solved heuristically since finding an optimal solution requires significant computational resources and, in general, the problem of optimally scheduling instructions is known to be NP-Complete. In this paper, the specific problem of automatically creating instruction scheduling heuristics is addressed.

Key-Words: - compiler optimization, instruction scheduling, genetic programming, machine learning

1 Introduction

For many years, different heuristics have been developed to schedule instructions for various architectures. The majority of these heuristics have been list scheduling heuristics, though other scheduling techniques have been used. Developing good techniques for instruction scheduling is costly. It can require many months to develop the heuristics needed to create a good instruction scheduling for new processors. Moss et al. [4, 14, 16] proposed using machine learning to automate the task of generating a heuristic for scheduling. The task of testing and revising the set of rules needed to find a solution is perfectly suited for a machine learning approach when good data exists to learn good rules. It is also important to create heuristics that are fast, efficient, accurate and understandable. A deficiency with this approach is the overall complexity of the algorithm and the cost associated with executing the scheduler.

2 Related work

Moss et al. [4, 14, 16] discuss the feasibility of using machine learning to learn good instruction scheduling heuristics for basic blocks. They used several different kinds of machine learning techniques including function approximators, rule based learners and reinforcement learning. They used a small sample set of features taken from a hand-crafted compiler and produced a heuristic similar but slightly worse than the DEC heuristic [18] they were comparing against. One of the deficiencies associated with this work is the limited data set available to the learning algorithm as only small programs could be solved optimally and, therefore, only small blocks were used to form the data set. McGovern et al. [14] try to alleviate this problem by using reinforcement learning. However, this improvement further complicates the learning process as reinforcement learning is a much more difficult problem. On top of the data limitation, the small set of features used by Moss et al. reduces the power of the technique and relies heavily on the work of the hand crafted designer. We improve on this technique by using a much larger set of features along with a feature selection technique to improve the accuracy of our heuristic.

Beaty et al. [2] developed a technique for instruction scheduling using genetic algorithms and discriminative polynomials. He used a set of features taken from the Rocket compiler and used a genetic algorithm to weight these features. This technique while interesting suffers from several problems. First, the experimentation lacks transparency. While the set of features are reported, the specifics of the test data are not reported in terms of either numbers of blocks or source of blocks. Second, they state that they gain only a five percent improvement in accuracy over initial random solutions.

In recent work, Li and Olafson [12] learned heuristics for single machine job shop scheduling using decision tree learning. However, they created their training data using existing heuristics to
classify the instances they created. This means that some of their training instances would have been incorrectly classified. As a result, the heuristics that they learn are never better than the original heuristics used to label the data. Lee et al. [11] also present a job scheduling method that uses machine learning to learn release dates for jobs and schedule them using a genetic algorithm. This algorithm is quite expensive and requires several minutes to solve medium size problems. Correa et al. [6] used a genetic algorithm to learn schedules for parallel processing schedules. To increase the accuracy and decrease the cost of the technique, they used a list scheduling algorithm to seed the initial genetic population. However instead of using real data, they generated a set of synthetic problems designed to approximate real world DAGs.

Calder et al. [3] performed static branch prediction using neural networks and decision trees. They proposed that branches could be predicted by profiling a set of corpus data and extracting the features of that data to predict future branch execution without profiling. Their approach is similar to ours as they use a wide variety of features for the problem, selecting the best features using machine learning. However, they do not prune irrelevant features from their feature set and may be introducing inefficiencies to the process. Jimenez and Lin [10] also propose a machine learning technique for learning branch predictions. They use a perceptron learning method where branches exit profiles are learned using the perceptrons.

There are other papers in the literature that deal with machine learning and compilers. Stephenson et al. [19, 20] look at machine learning for register allocation and for building predicated hyperblocks. Monsifrot et al. [15] look at a simple learning technique to determine when to unroll loops. Similar to the paper by Monsifrot et al., Long et al. [13] present a paper which uses machine learning to perform loop optimization in Java.

3 Instruction Scheduling Problem

The instruction scheduling problem can be characterized as a set of instructions and a set of functional units [17, 9]. A functional unit can be defined as a unit within a processor assigned to complete as specific task. For example, most processors have a floating point unit that processes floating point instructions. The instructions must be scheduled on the functional units so that all constraints on the problem are satisfied. Two major constraints are issue width and precedence constraints. The issue width of a processor limits the number of instructions that can begin execution in any clock cycle. Precedence constraints ensure that certain operations are executed in a specific order. Modern processors typically have several different functional units that can process one or more different types of instructions. On top of this machine level parallelism, the addition of pipelining allows a new non-conflicting operation to execute in every clock cycle on a single functional unit.

The problem is modeled as a directed acyclic graph (DAG) where the nodes are the instructions and the edges represent the precedence constraints. The labels of the edges represent the latencies of the constraints. The latency of a constraint represents the number of cycles that must elapse before scheduling the succeeding instruction. Each DAG has an associated order as generated by the code generator and is equivalent to the id assigned to each node. Once a DAG has been constructed from the precedence relations, it is possible to determine the latest start time and earliest start time of each instruction. These properties are derived from the graph by tracing the paths through the graph and determining the maximum distances from the root and leaf nodes of the graph. The path that maximizes the distance between any two nodes is called the critical path.

Instructions scheduling is typically divided into two specific categories, local and global scheduling. Local scheduling, also known as basic block scheduling, is the scheduling of any sequence of code that executes consecutively without branch instructions, with the possible exception of call instructions. Global scheduling defines all other types of scheduling including those which consider the entire program. Many of the global scheduling techniques try to find a balance between the amount of parallelism available and the manageability of the code size. Programs can contain millions of instructions and thousands of branch instructions, especially in the presence of aggressive optimization techniques like loop unrolling, and it becomes difficult to apply techniques to the entire program efficiently.

In order to schedule a super block, we must define a cost function that we wish to minimize. For basic blocks, the cost function is simply the length of the schedule but for the super block scheduling problem a more complex function is used. The increased complexity comes from the existence of multiple exits allowing for different paths, and path lengths, out of a super block. Since there are multiple paths that need to be minimized, it is possible that
minimizing some paths may increase the path lengths for other paths. This leads to a situation where it is necessary to weight the paths by relative importance. This weighting is normally done by using profile data collected about the exits. Profile data is collected by compiling the program without scheduling and executing a standard reference set of data. For each exit, the frequency that the exit is taken is recorded and these values are converted into percentages. The weights, called exit probabilities, are used in determining the cost function. So instead of minimizing the path length, we minimize the weighted path lengths to all branches. The cost of a schedule $S$, which we wish to minimize, can be defined more formally as,

$$\text{cost}(S) = \sum_{b \in B} w(b) t_s(b);$$  \hspace{1cm} (1)

where $B$ is the set of all branches in the schedule $S$, $w(b)$ is the exit probability of branch $b$ and $t_s(b)$ is the time slot of $b$ in the given schedule $S$.

The paper proposes to develop a method for automated heuristic design for the global instruction scheduling using genetic programming as machine learning tool.

4 Machine learning

Machine Learning can be broken into different subfields including supervised, semi-supervised and unsupervised learning.

Supervised learning comprises the set of techniques where all learning is done from a data set that is labeled with the correct answer. Un-supervised learning attempts to determine structure within a problem without ever consulting a labeled data set. For learning the test sets are used. These sets are called the training set, the validation set and the testing set. The training set comprises the set of instances used to directly induce a classifier function. To improve the accuracy of the classifier, it is often useful to keep a set of instances, called the validation set, to decide when to stop the learning algorithm. The process of using the validation set to iteratively improve results is called cross-validation. The testing set is a set of instances kept aside until the final classifier is complete to test the accuracy of the classifier on unseen data.

The inclusion of the validation set allows the classifier to be refined in order to avoid the pitfall of over-fitting. The pitfall of over-fitting can be defined as the learning of a classifier function that performs better on the training set over another function that performs better on the entire testing set. In other words, a classifier is over-fit when that function performs better only on a specific set of training data and not on the general data.

5 Genetic Programming

Genetic Programming (GP) is a relatively new technique for machine learning applications. GP, like its ancestor, the Genetic Algorithm (GA), harnesses the power of evolution to solve problem. Starting with an initial population of randomly created programs, “individuals”, GP literally evolves a solution to the problem at hand.

Like natural evolution, the evolution that occurs in GP relies both on the genetic structure of the individuals that are under-going evolution, which allows the production of new individuals, and on same sort of selection pressure which makes it imperative for the individuals to constantly improve or face extinction.

Figure 1 shows the general flow of genetic programming in the context of our system. The algorithm begins by creating a population of initial expressions. The baseline heuristic over which we try to improve is included in the initial population; the remainder of the initial expressions are randomly generated. The algorithm then determines each expression’s level of fitness. In our case, heuristic that generated shorter schedule than others are fittest. To determine the fitness level of an expression, the algorithm compiles and runs a set of benchmarks using the expression as a heuristic’s priority function.

![Figure 1: general flow of genetic programming](image-url)
Once the algorithm reaches a user-defined limit on the number of generations, the process stops; otherwise, the algorithm proceeds by probabilistically choosing the best expressions for mating. Some of the offspring undergo mutation, and the algorithm continues. Unlike other evolutionary algorithms, which use fixed-length binary genomes,

5.1 Representation in Genetic Programming

Unlike traditional GA, GP uses program trees to represent individuals. GP’s expressions are variable in length and free-form. For example the LISP program \((+ 6(* 7 8))\) could be represented as in Figure 2. These program trees are made up of two fundamental building blocks, nodes and leaves. Nodes are simply functions such as + * which take one or more arguments, while the leaves are terminals, i.e. numbers or zero-argument functions. The first major step in any implementation of GP is to correctly identify the necessary functions and terminals and to ensure that any combination of them will result in a syntactically (although not necessarily functionally) correct program.

![Figure 2: A simple parse tree](image)

5.2 Survival of the fittest

An initial, random population of these individuals is created. Each of these is than evaluated, testing how suited they are to the particular task at hand. Typically, there are a number of test-causes from which each individual is assigned a score.

The next step in GP is to create the next generation in the population. Individuals are selected probabilistically from the previous (parent) generation to be either reproduced or crossed over. An individual is chosen for crossover, a second individual is also chosen, and these are crossed over, as described in the next section, to produce individual.

![Figure 3: Crossing over two parent trees by swapping sub-trees](image)

5.3 Crossover

Due to the nature of the representation scheme used by GP, it is possible to swap sub-trees from two individuals. This result in the creation of two new, syntactically correct individuals, as shown in Figure 3. Crossover is the exploration tool of GP. It is by crossing over good performing individuals the population as a whole slowly evolves until a “perfect” individual appears, i.e. an individual who fulfills all the requirements of a problem. Many individuals, who will be produced as the result of a crossover, do not perform well, and, in true Darwinian evolutionary style, they will be selected neither for crossover or reproduction in the following generation, and so die off.

5.4 Methodology

Compiler heuristics are often based on assumptions that may not be valid across application and architectural variations. Our system uses genetic programming to automatically search for effective heuristic. This technique is very powerful; a small change to the heuristic function can drastically improve (or diminish) performance.

We generate a given heuristic by executing the iterative framework of Figure 1 around the compiler and architecture. We replace the heuristic function that we wish to optimize with an expression parser and evaluator. This allows us to compile the benchmarks in our ‘training’ suite using the
expressions—which are heuristic functions—in the population. The expressions that create the fastest executables for the applications in the training suite are favored for crossover.

Our system uses total execution time to assign fitness. This approach focuses on frequently executed procedures, and therefore, may slowly converge upon general-purpose solutions.

Table 1. GP Primitives

<table>
<thead>
<tr>
<th>Boolean-Valued Function Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool1 and Bool2</td>
</tr>
<tr>
<td>(and Bool1 Bool2)</td>
</tr>
<tr>
<td>Bool1 or Bool2</td>
</tr>
<tr>
<td>(or Bool1 Bool2)</td>
</tr>
<tr>
<td>not Bool1</td>
</tr>
<tr>
<td>(not Bool1)</td>
</tr>
<tr>
<td>Real1 &lt; Real2</td>
</tr>
<tr>
<td>(lt Real1 Real2)</td>
</tr>
<tr>
<td>Real1 &gt; Real2</td>
</tr>
<tr>
<td>(gt Real1 Real2)</td>
</tr>
<tr>
<td>Real1 = Real2</td>
</tr>
<tr>
<td>(eq Real1 Real2)</td>
</tr>
</tbody>
</table>

Table 1 shows the GP expression primitives that our system uses. Careful selection of GP primitives is essential. We want to give the system enough flexibility to potentially find unexpected results.

The simplest way to compare instructions would be to use a greater than, equal to or less than scheme where each feature of an instruction is compared and given a value true or false. This comparison function is useful because it is both simple to understand and is inexpensive to compute for feature value pairs. However, with this method, it is clear that some information is lost.

It is possible to create scenarios where a feature means very little when it is only slightly greater than or less than a given comparison value but is very informative when the value varies wildly with the comparison value. Therefore, a more fine grained distinction of the feature values may lead to better classifiers.

Our system creates an initial population that consists of 50 randomly generated expressions; it randomly 'grows' expressions of varying heights using the primitives in Table 1 and features extracted as given in Table 2. Features are measurable program characteristics.

5. 5 Feature Construction

In order to determine which instructions to select, it is necessary to give the classifier function a good set of features. We used three types of features within the instruction scheduling domain: DAG related features, ready list related features and instruction level features. Features can be further separated into dynamic and static features. Dynamic features are those features that changed depending on the clock cycle and the previously scheduled instructions. Static features do not change and can be calculated prior to scheduling.

The DAG related features are those features that are the same for each instruction within a given DAG. They are static features and can be calculated prior to scheduling. The DAG related features include many counting features including the number of instructions, number of instructions per type, number of functional units, and number of edges in the graph. Also included in DAG related features are averages, maximums and minimums of any property of the graph.

Many of the features [5, 7, 8] used by the previously proposed, hand-crafted super block scheduling heuristics are derived from the fact there exist multiple exits within the super block. Another obvious addition is the profile information given about the multiple exits. This leads to new features like weighted estimates. Table 2 show all of the features considered for super block scheduling heuristics.
The earliest cycle when the instruction can begin to execute, relative to the current cycle

Weighted resource based distance to every branch, as given in equation (3)

Maximum of critical path distance to each branch, as given in equation (4)

Weighted maximum of critical path distance to each branch, as given in equation (5)

$$\text{weighted}_\text{cp}(i) = \sum_{b \in B(i)} \text{cp}(i,b) w(b)$$  \hspace{1cm} (2) \\
$$\text{weighted}_\text{estimate}(i) = \sum_{b \in B(i)} \text{rb}(i,b) w(b)$$  \hspace{1cm} (3) \\
where, \text{rb} is resource based distance.

$$\text{max}_\text{dist}(i) = \max \{(\text{Cp}(i,b))\}$$  \hspace{1cm} (4) \\
$$\text{weighted}_\text{max}_\text{dist}(i) = \max \{(\text{cp}(i,b)w(b))\}$$ \hspace{1cm} (5)

6 Experimental Setup

We modified MachineSUIF\[1\] compiler by replacing its base global instruction scheduling heuristic with our GP expression parser and evaluator. This allows MachineSUIF to read an expression and evaluate it based on the values of human-selected features that might be important for creating effective heuristic. Table 2 describes these features. The instruction schedule algorithm passes the features in the table as parameters to the expression evaluator. It is then subjected to a training phase during which time it finds an effective heuristic. After the training phase is completed, directly uses it to make its compilation decisions.

We train our algorithm to perform well on a broad spectrum of applications. The benchmarks used are as shown in table 3. Instead of creating a specialized heuristic for each benchmark, we find one that works well for all benchmarks in the training set. To create a training set, we divide the benchmarks in table 3 into 2 sets: a training set and a validation set.

Table 3. Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>applu</td>
<td>23456</td>
</tr>
<tr>
<td>art</td>
<td>482</td>
</tr>
<tr>
<td>eon</td>
<td>4787</td>
</tr>
<tr>
<td>equake</td>
<td>498</td>
</tr>
<tr>
<td>fpppp</td>
<td>24532</td>
</tr>
<tr>
<td>lucas</td>
<td>991</td>
</tr>
<tr>
<td>mcf</td>
<td>378</td>
</tr>
<tr>
<td>mesa</td>
<td>16890</td>
</tr>
<tr>
<td>mgrid</td>
<td>23168</td>
</tr>
<tr>
<td>parser</td>
<td>3879</td>
</tr>
<tr>
<td>swim</td>
<td>21402</td>
</tr>
<tr>
<td>twolf</td>
<td>7856</td>
</tr>
<tr>
<td>vpr</td>
<td>3565</td>
</tr>
<tr>
<td>wupwise</td>
<td>568</td>
</tr>
</tbody>
</table>

Figure 4 shows the result of applying a single best heuristic to the benchmarks in the training set. It shows the speedup over machineSUIF’s base heuristic when the training input is used. It yields an improvement. We then apply the heuristic to the benchmarks in validation set as shown in figure 5. Since the benchmarks in validation are not related to the benchmarks in training set, this is the measure of the heuristic’s generality. It also shows an improvement. In the case of swim, vpr the base heuristic outperforms the genetic programming generated heuristic.
7 Conclusion

We propose a automated technique using genetic programming for generating instruction scheduling heuristics and we show that this techniques can be successfully applied to the problem of super block scheduling as instruction scheduling is important for improving the efficiency of code in the presence of instruction level parallelism and pipelining. As the block size increase the heuristic has a chance to make mistake.

References:


