RC22198 (W0110-031) October 10, 2001
Computer Science

IBM Research Report

Interaction of a Superscalar Architecture and a Data Mining Application

Mathew S. Thoennes
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. , payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Introduction

As problem sizes continue to grow there is more and more interest in keeping the run times reasonable. Superscalar processors offer the opportunity to increase performance via parallelism in the hardware. Compiler technology has tried to help the user take advantage of this parallelism by restructuring the code to make it run more efficiently on modern processors. Although there has been tremendous success in compilers, there are limits as to what can be done without a deeper understanding of the algorithms used, implementation and runtime behavior. This paper looks at the aspects of decision tree construction. It focuses on C4.5 [1], which is a popular decision tree generator. First, classifiers in general are discussed. The difference between deductive and inductive reasoning is then considered. A model is introduced that is used to formulate the problem of classification. Shortcomings of the classical statistical method of n-way analysis of variance are examined and an overview of the general concepts of decision trees with details of the implementation of C4.5 is presented. Finally, a section of C4.5 is examined to illustrate an area where modifications to the code could result in greater utilization of the parallelism available in the hardware.

Classifiers

The modern world of business has become increasing interested in being able to predict the behavior of either a single individual or a group based on some prior knowledge that has been collected. This has lead to the creation of a new discipline called data mining. Data mining is interested in creating business intelligence that can be used to either influence the marketing of items or verify the behavior of an individual or group. A classic example involves the analysis of the shopping basket in a grocery store. By examining the contents of various baskets over time it was possible to identify common items that were purchased together with a high probability. It was determined that with a high probability that those men who purchased milk and diapers would also purchase beer. With this information in mind the grocery stores could rearrange the store to position other items in relation to these three items to increase sales. A second widely used area is validation of behavior. Products that have very high fraud rates such as cellular phones, phone calling cards and credit cards utilize these techniques to determine if the current transaction is consistent with previous behavior. Thus if a series of transactions matches a behavior which has a high probability of fraud or the current transaction is inconsistent with the normal activity the transaction can be flagged as potentially fraudulent and either rejected or flagged for further inspection.

The goal in data mining is to create a model that can be used to classify future instances of the problem. Classifier is the term that is widely used for these models. Machine learning has studied classifiers for a long time and thus it is not a new field. Data mining has taken this work and moved it into the business world and has broadened the interest in the field.

Deductive versus Inductive

We are interested in classifiers because of wanting to build a model that predicts the outcome of future instances of a problem given a sample of the past behavior. In classical statistics we take the deductive approach where you generate a hypothesis and then test it against the null hypothesis. From the theory of the system that we are looking at we would formulate a hypothesis that we believe characterizes its behavior. We might make the hypothesis that **X** causes **Y**. This hypothesis would then be tested against a null hypothesis to see if it is better. A null hypothesis for this example might be that the hypothesis is no more accurate than to randomly choose an outcome from all possible outcomes with equal probability. We could then generate distributions of the percentage of incorrect classifications for each of the hypothesis. Taking groups of samples and applying either the hypothesis or the null hypothesis generate the distributions. Given these distributions we could then determine whether we should reject the null hypothesis and obtain a boundary on the probability that we have falsely rejected the null hypothesis. The problem with this traditional approach is that many times we cannot generate a precise hypothesis given the data and our knowledge of the system.

The goal is to take an inductive approach to the problem of building a classifier. We want to allow the observed instances to drive the generation of the model and provide an ex post facto explanation. Propositions that specify a particular value for an input of an instance will be combined to create interrelated propositions. These form the basis that generates specific hypothesis in the future. Building the model has two phases. First we must generate the model given the data provided then we must test the model with new data to determine if the model is correct or to identify any deficiencies. So we need to develop a way to identify the variables responsible for the outcome and how they interrelate.

The Model

We need to describe the model that is used for the remainder of this paper. It is comprised of several components. A set of predictors describes each instance. A predictor can be an observable feature or may by synthesized from a number of observable features. The predictors form a vector of the form:

$$< x_1 \, x_2 \, x_3 \, \ldots\ldots \, x_n >$$

This vector is an element of the domain **X** which is comprised of all possible values of the vector $<x_1 \; x_2 \; x_3 \; \ldots\ldots \; x_n>$. There is a dependant variable that is the outcome. Y represents it. A number of observations of the form of the domain **X** are provided. These observations are denoted by $U_1 \; U_2 \; U_3 \ldots\ldots \; U_m$. A weight may be associated with each of the observations. The predictors may fall into three categories. They are independent, specifier or elaborator. Independent predictors have a direct effect on the dependent variable. Specifiers are conditionals that must be met for other predictors to affect the dependent variable. Elaborators moderate the effect of other predictors on the dependent variable.

Our goal is to find a function of the form:

$$\mathbf{Y} = \mathbf{f}(x_1\, x_2\, x_3 \,.....\, x_n)$$

This assumes that there exists a function that maps every instance to the correct result. Usually we find that the equation is of the form:

$$\mathbf{Y} = \mathbf{f}(x_1\, x_2\, x_3 \,.....\, x_n) + \mathbf{e}$$

Where we want to minimize **e**. A common definition of **e** is the sum of squares of the deviations of the predicted values from the actual values. So what is **e**? If we could perfectly predict **e** then we would know all of the factors that influence **Y**. The challenge is that we must collect all of the observation data for each instance, collect all the right data for each instance and collect enough instances. There also is a basic randomness in **e** and there will be measurement error. Given these issues we chose the simplest assumption that **e** is a random variable with a fixed mean and variance. If **e** does not meet these requirements then we must assume that we are either missing one or more predictors or the missing predictor is correlated with a predictor that is used and is thus preventing the randomness in **e**.

N-way Analysis of Variance

One traditional approach to finding the function **f()** is to use a n-way analysis of variance. The form of the equation is to calculate the deviation from the output mean for the sample data for each instance. A resulting equation is of the form:

$$\mathbf{Y} = \overline{\mathbf{Y}} + a_i + b_j + \,.....\, + \mathbf{e}$$

where

a_i is the ith category of predictor A
b_j is the jth category of predictor B
$\underline{\mathbf{Y}}$ is the observed value of an instance with $a_i\, b_j$ .....
Y is the output mean
e is the error

There are a number of problems with this approach:

1) $a_i\, b_j$ .... are assumed to be continuous normally distributed variables. In reality they may also be categorical or ordinal values.

2) Measurement errors in the predictors as well as in the output **Y**.

3) Non-linear relationships.

4) Inter-correlations may exist between predictors.

5) Interaction effects in which several predictors are required to describe a concept.

6) Logical priorities amongst predictors and causal chains.

There has been work on all of these issues but the major drawback is the additive assumption. This assumption limits us to modeling the causes of the effects to terms that can be linearly added together. For categorical or ordinal we can introduce 0/1 variables that represent each of the possible values. To handle the measurement errors we can tolerate a higher value for **e**. For some of the remaining issues we can introduce additional terms constructed of the predictors to represent the effects that exist between them. The question then is what additional terms do we need to add? If the number of predictors is small then we can try the strategy of exhaustively searching the space of all possible terms. We would then use the equation with the smallest **e**. But in the case that the number of predictors is not small then we must decide which terms to include.

This is the key problem with n-way analysis of variance. We must decide a priori which additional terms to add to account for the effects which are not accounted for in the assumption that the predictors are uncorrelated. Thus n-way analysis of variance requires that we have considerable knowledge prior to solving the problem.

Decision Trees

Decision trees represent a different approach to generating the classification function than n-way analysis of variance. While n-way analysis of variance strives to assign the variation from the output mean, decision trees incrementally builds a model that strives to improve a metric at each step. This metric could be reduction in total mean square error of the two partitions from the error of the combined set. It could be the increase in predictive power or the reduction in uncertainty. This is accomplished by recursively partitioning the known instances via a binary split on the predictor that has the best value of the target criteria. The partitioning will stop when the delta value of the target criteria fails to exceed some predetermined value. A framework of the algorithm for the decision tree is:

(1) Establish the measure that will be used in the split decision. Set the minimum value for this measurement for which a split will not occur.
(2) Calculate the criterion value for the instances in the current group. Partition the group using each predictor as the key to created a binary split. Calculate the value of the criteria for the resultant split groups. Pick the partition that yields the greatest improvement of the measurement criteria.
(3) If the delta between the original group and the new partitioned groups exceeds the threshold in step 1 then keep the partition and recursively apply step 2 to the new partitions.
(4) For each terminal node assign the class that is the majority class of all of the instances at that node.

This will result in a tree structure. Each of the interior nodes will contain a test on one of the predictors and the leaves will contain a classification. The **f()** is not as straightforward as in the case of n-way analysis of variance. To classify an instance we start at the root node of the tree that is created and follow the branch from each decision node for this instance. When we reach a leaf node then the instance is classified with the class of the leaf node.

The decision tree allows us to build a classification model with fewer assumptions than are required for the n-way analysis of variance approach. We are not required to account for interactions between the predictors by introducing additional terms prior to knowing the model. This allows us to explore the problem while making minimal assumptions. While the validity of n-way analysis of variance comes from statistics the validity of decision trees has drawn on empirical results. A substantial amount of work has been done taking well know problems which have been analyzed via statistical methods and comparing with the results obtained from decision trees.

The final question is what to use as the measurement criterion for evaluating the goodness of the partitioning. The standard measurement used is entropy.

Entropy

The extension of the concept of entropy to computer science is attributed to Claude Shannon. In developing his theory of communication, he adopted entropy as a term to represent a measurement of information. In the area of thermodynamics, entropy represents a measure of the amount of energy in a thermodynamic system. In computer science, Shannon used this term to represent a measure of uncertainty. Decision trees use finite-dimensional entropy as a measure of the increase in certainty that guides the recursive partitioning process. For our discussion here we will use the following notation. The discrete probability distribution associated with n possible outcomes is represented by $\mathbf{p} \equiv (p_1, p_2, \ldots\ldots, p_n)^T$ for each outcome $\mathbf{x} \equiv (x_1, x_2, \ldots.., x_n)$. The entropy is denoted by $S_n(\mathbf{p})$. Some of the defining axioms are [2]:

1) $S_n(\mathbf{p})$ should depend on all of the p_j's, $j = 1, 2, \ldots., n$.
2) $S_n(\mathbf{p})$ should be a continuous function of p_j, $j = 1, 2, \ldots., n$.
3) $S_n(\mathbf{p})$ should be permutationally symmetric. If the order of the p_j's are merely permuted, then $S_n(\mathbf{p})$ should remain the same.
4) $S_n(1/n, 1/n, \ldots., 1/n)$ should be a monotonically increasing function of n.
5) $S_n(p_1, p_2, \ldots\ldots., p_n) = S_{n-1}(p_1 + p_2, p_3, \ldots\ldots, p_n) + (p_1 + p_2)S_2(p_1/(p_1 + p_2), p_2/(p_1, p_2))$.

The family of functions of the form of $-k \Sigma p_j \ln p_j$ which k is a positive function and $0 \ln 0 = 0$ satisfies the defining axioms. Shannon chose the function $- \Sigma p_j \ln p_j$ to represent his concept of entropy. This formula has the following desirable properties [2]:

1) Shannon's measure is nonnegative and is concave in $p_1, \ldots.., p_n$.
2) The measure does not change with the inclusion of a zero-probability outcome.

3) The entropy of a probability distribution representing a completely certain outcome is 0, and the entropy of any probability distribution representing uncertain outcomes is positive.
4) Given any fixed number of outcomes, the maximum possible entropy is that of the uniform distribution.
5) The entropy of the joint distribution of two independent distributions is the sum of the individual entropies.
6) The entropy of the joint distribution of two dependent distributions is no greater than the sum of the two individual entropies.

The entropy for a particular split is calculated via Shannon's formula. To illustrate the calculation of the entropy of a split let us assume a split that involves a binary attribute (0,1), two classes (A, B) and eight instances. Assume that six of the instances have an attribute value of 0 and that four of those instances are class A while the remainder are class B. The remaining two instances have an attribute value of 1 and both belong to class B. The calculation of the entropy is:

$$\text{Entropy} = \text{Entropy}(4 - A, 2 - B) + \text{Entropy}(0 - A, 2 - B)$$

Which given the numbers would be:

$$\text{Entropy} = [-(4/6)\log_2(4/6) - (2/6)\log_2(2/6)] + [-(0/2)\log_2(0/2) - (2/2)\log_2(2/2)]$$

Decision trees usually use information gain as a metric when determining which attribute to use for a particular split. Information gain is calculated by using entropy values that have been calculated for the split. Information gain is simply the difference between the entropy of the group prior to the split minus the entropy of the proposed split. The goal is then to maximize the information gain, which is equivalent to picking the split that results in the lowest entropy.

As we recursively split the instances we must reexamine them each time. Since the instances being examined at any given level of the recursion are dependent on the attributes which were chosen prior to this level there is no way to pre-compute the entropy values for the splits.

C4.5

The program C4.5 is an implementation of the decision tree algorithm. It has been used widely in the area of machine learning and data mining. Flexibility and ease of use are two of the features that have resulted in this wide use. One of the features is that the structure of the data in the program is created dynamically via two files that provide the classes, predictors, allowed values of the predictors and the instance data. Predictors can be binary, ordinal or continuous. C4.5 uses entropy in the determination of the predictor to use for partitioning the set of instances at any given level in the tree. One of the deficiencies of the entropy calculation is that it tends to favor predictors with many values that cause a group of instances to be broken into a large number of subsets. Taken to an extreme we could consider using a social security number as a predictor that would

result in the group of instances being broken into subsets of one item each. The information gain of this split would be maximal but would not result in a useful split. To correct this bias, C4.5 normalizes the information gain by adjusting the apparent gain attributed to a test with many outcomes. An adjustment factor is introduced which is called the split information. Split information measures the information gain that is attributed to the split alone. It uses the same formula as entropy except that we are just calculating the information gain of splitting on the attribute regardless of what the class is. The gain calculated via entropy is then divided by the split information which results in the new metric called gain ratio. Our previous example would result in the following calculation:

$$\text{split info} = -\,(6/8)\log_2(6/8) - (2/8)\log_2(2/8)$$

The gain ratio is then:

$$\text{gain ratio} = \text{gain} / \text{split info}$$

One of the drawbacks from the flexibility of C4.5 is that the data structures are not pre-allocated prior to reading in the classes, predictors and instances. The data structures and elements are allocated as the input is processed. Pointers are widely used in the data structures and elements are allocated, as needed, using standard memory allocation routines. An example is the structure that represents the instance data. Figure 1 shows the structure.

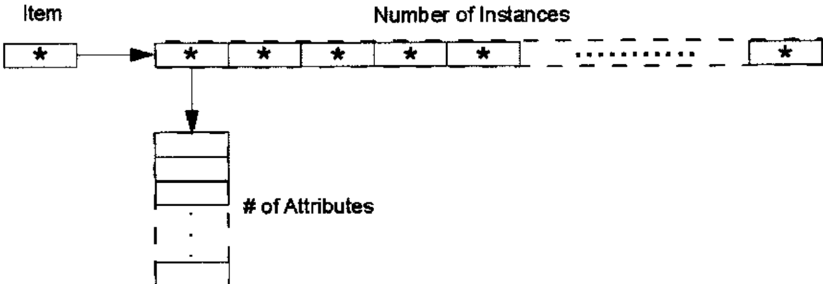

Figure 1.

An array of pointers is used to reference the data for each instance. Each entry in this array corresponds to an instance. Reallocating the array with a larger size as the instances are processed grows this array. Since the array is reallocated the elements remain contiguous in memory and thus can still be accessed as an array. The data for each instance is placed in an array of elements that are a union of an integer and floating point value. This array is then pointed to by an entry in the main array. Given that the structure that holds the data for an instance is allocated one at a time, the location of that structure in memory is not guaranteed.

C4.5 builds the tree through a divide and conquer methodology. This is the standard approach for decision trees. The number of branches for each level of the tree is not

binary but is the number of values for the attribute. The algorithm that is implemented in C4.5 is:

1) Calculate the gain ratio of all attributes that have not been used at a higher level of the tree. If the gain ratio of all available attributes is below the minimum value to allow a split, stop and return this node as a leaf node.
2) Select the attribute that has the highest gain ratio.
3) Partition the instance data on the chosen attribute and create a branch for each value of the chosen attribute with a new node containing the instances for that attribute value. Mark this attribute as used.
4) Recursively call this procedure for each of the new nodes. Upon return check to see if a leaf would be as good as the tree constructed from this node and if so collapse the tree into a leaf.

The algorithm requires that the instance data be continuously accessed to calculate the gain ratio at each level of the tree since the grouping of the instances is not known until the split has been decided. The use of an array of pointers to point to the instance data provides an easy mechanism to partition the list during the recursive calls. By simply moving the pointers in the array the list can be partitioned rather than requiring that the arrays containing the instance data be moved. After the tree has been built it is then pruned to remove excess structure that does not add to the classification accuracy of the tree.

Program and Processor Architecture Interactions

The implementation of C4.5 influences how the program runs on modern superscalar architectures. While there are certain fundamental details that are inherent in decision trees there are still some implementation details that can be explored to see if the performance can be improved.

Cache/Memory Interactions

The first area to examine is the interaction of the program with the caches and the memory. The speed of access to main memory is becoming a bottleneck as processors become faster. In the case of C4.5, the structure that holds the instance data is repeatedly used in the construction of the tree. Since the program is a recursive divide and conquer, the size of the partition of the instance data that is currently being worked on eventually shrinks to fit in the cache. But as problem size continues to grow the issue of more efficient dataflow prior to this becomes more important. Work has been done looking at ways to more efficiently handle the building of decision trees for problems that are out of core [3]. This work has utilized methods to summarize the instance data so that the first several levels of the tree can be built without having to examine the instance data. There is some error that is introduced by this method but for some applications it is acceptable given the decrease in the time to build a tree. Implementing a similar scheme at the level of the cache could have a similar impact.

Access of Instance Data

Another area to explore is the actual accessing of the instance data. Given the data, one could image a two dimensional array with the instance being one index and the individual attribute being the other index. As we examine the actual value of the attribute we would be accessing the array in a fixed pattern for the attribute value for each instance. If each instance only had a single attribute then since the array is contiguous in memory we would be taking advantage of data locality that would bring the next n instance's values into the cache as we load the cache line the current instance is in. If there is more than a single attribute we no longer get this advantage but we are still accessing the memory with a particular stride that is unique to this problem instance. For that case we could explore a prefetching mechanism to preload the data of the next instance via a stride value prior to its required use. Given that we have multiple execution and load/store units the goal would be to overlap the operations that use the current instance with the loading of the next instance. For prefetching or data locality to work we must have the array in either a contiguous or fixed structure where we can calculate the stride. Given the divide and conquer methodology of the tree building algorithm this presents a challenge due to the need to partition the instances during the recursive steps.

There are two ways to do this partitioning given the data structure of C4.5. A standard method, which could be used where the size of the data element is small, is to move the elements in the instance array to partition the array. Given that trees will normally be generated from problems with more than a few attributes, the cost of the copying would probably far exceed the gain from being able to access the data in an orderly manner. The second way to partition the instance array is to take the approach that is used in C4.5 where the array of instances points to the attribute data for each instance and the list can be partitioned by simply moving the pointers in the instance array. Since the structures that contain the attributes for an instance are allocated as they are added there is no guarantee of how the structure maps into the address space. But C4.5 may still benefit from prefetching. There are several caveats to prefetching. First there must be sufficient computational resources to calculate the addresses without interfering with the execution of the real work of the program. Secondly the memory bus and memory system need to have sufficient bandwidth so that the prefetching does not interfere. And finally the number of outstanding memory requests from the cache must be sufficient so that the prefetching requests do not lock up the cache and prevent real cache misses from being processed.

Processor Resource Utilization

Superscalars and Compilers

The design goal of a modern superscalar processor is to improve throughput by finding and utilizing instruction level parallelism, which is achieved by executing multiple instructions simultaneously. An example of a superscalar processor is the IBM Power 2 [4], which contains 2 load/store, 2 fixed point and 2 floating point units. During execution the instructions are examined to determine any dependencies that require that two instructions be issued in a particular order. An example is an addition followed by a store of the results. The dependency is that the store cannot be executed until the

addition is complete. The hardware examines the flow of instructions and attempts to issue as many simultaneous instructions as possible given the constraints and the available hardware. Modifying the dataflow may also eliminate some dependencies.

A significant amount of work has been done in compilers to take advantage of the multiple functional units in the modern superscalar processors. Code structure is examined during the compilation process and optimizations are performed. Restructuring the program to take advantage of the superscalar processor design is the goal. Modifications to the code must be safe. The definition of safe is that the changes cannot affect the correctness of the program as defined by the unmodified code. Limitations exist since the compiler is working with a static representation of the program. Actual data values and runtime behavior are not known at compile time. The compiler also does not have any insight into the algorithms or the high level design that is embedded into the program. Thus there are some real limitations on what can be done at compile time to improve the performance of the program.

Example Selection

The process of tree building goes through a number of phases. For C4.5 these are:

1) Read the class, attribute and instance data.
2) Recursively build the tree.
3) Prune the built tree.

Utilizing my previous performance work [5] the percentage of time spent in each section was measured. The largest test case was selected which contains 100,000 examples generated by the synthetic benchmark generator LED [6]. The percentage of the total runtime of the program for each phase was 19.3% for reading in the data, 22.5% for building the tree and .05% for pruning the tree. Initialization, output of data structures and text output accounted for the remaining runtime of the program.

After examination of the C4.5 code a section was chosen to study to see if it could be restructured to increase the efficiency of the use of the hardware parallelism. The section chosen is used to count the frequency of the output classes for the values of a particular attribute (predictor) and is part of the building of the tree. This section was chosen since it is called at each interior node of the tree for each of the attributes to be considered at this level of the tree and is 35% of the runtime of the tree-building phase.

An Example to Study

The section of code studied is part of the subroutine called ComputeFrequencies. The C source for the code is:

```
for (p = FP ; p <= LP ; p++) {
        Case = Item[p];
        Freq[Case.attribute][Case.class] += Weight[p];
}
```

The code walks through the instances totaling the number of classes for each of the possible values of the chosen attribute. This loop is a possible candidate for loop unrolling. Loop unrolling places multiple iterations of the original loop in a modified loop. Maximizing the amount of work done with the multiple functional units is the goal. From examining the potential data values for attribute and class, there exists a potential data dependency. Since it is not known a priori what the class or attribute value will be of an instance unrolling this loop may not increase the parallelism in the loop. If we place two original iterations in a modified loop, one of them may have to be held until the other completes due to a data dependency of both having the same location as a target.

With a little thought the traditional loop unrolling technique can be modified. This modification results in the following code:

```
p = FP;
/* start up code for odd no. of elements */
if ( ((LP – p) % 2) != 0) then {
        Case = Item[p];
        Freq[Case.attribute][Case.class] += Weight[p];
        p++;
}
/* process instances two at a time        */
for ( ; p <= LP ; p = p + 2) {
        Case = Item[p];
        Case2 = Item[p + 1];
        Freq[Case.attribute][Case.class] += Weight[p];
        Freq2[Case2.attribute][Case2.class] += Weight[p + 1];
}
/* combine the two arrays                 */
for (p = 0 ; p < MaxDiscrVal + 1 ; p = p + 2) {
        for (q = 0 ; q < MaxClass + 1 ; q++) {
                freq[p] [q] += freq2[p][q];
        }
}
```

The first section handles the case of an odd number of instances. The second loop is similar to the unmodified code except we are processing the instances two at a time. By introducing the second frequency array we have eliminated the potential for a data conflict that would require holding operations. The final loop is to combine the two arrays. There are no data dependencies between iterations of the second loop and thus it would be expected that the compiler would unroll this loop.

Looking Under the Hood

A high level programming language such as C hides much of the detail of the actual assembly code that is produced. In order to study the potential impact of the changes we need to look at the assembler code that is produced for the original and modified loop. The annotated assembler produced by the IBM xlC compiler for the original loop is:

104(SP) is the index of the attribute which is passed to the routine
108(SP) is the value of FP which is passed to the routine
112(SP) is the value of LP which is passed to the routine

```
__L2c:                     # 0x0000002c (H.10.NO_SYMBOL+0x2c)
        l       r3,T.22.Item(RTOC)      r3 loaded with location of the value of Item
        l       r3,0(r3)                r3 loaded with base address of array Item
        l       r4,68(SP)               r4 loaded with value of p
        rlinm   r4,r4,2,0,29            shift r4 left to calculate offset – 4 bytes per entry
        lx      r4,r3,r4                r4 loaded with address of instance data EA(r3+r4)
        st      r4,64(SP)               r4 stored in 64(SP)
        l       r3,T.26.Freq(RTOC)      r3 loaded with location of the value of Freq
        l       r3,0(r3)                r3 loaded with base address of array Freq
        lha     r5,104(SP)              r5 loaded with attribute chosen (integer)
        rlinm   r5,r5,2,0,29            shift r5 left to calculate offset – 4 bytes per entry
        lhax    r5,r4,r5                r5 loaded with attribute value EA(r4+r5)
        rlinm   r5,r5,2,0,29            shift r5 left to calculate offset – 4 bytes per entry
        lx      r3,r3,r5                r3 loaded with base for second index of Freq
        l       r5,T.30.MaxAtt(RTOC)    r5 loaded with location of the value of MaxAtt
        lha     r5,0(r5)                r5 loaded with value of MaxAtt
        cal     r5,1(r5)                r5 loaded with value of MaxAtt + 1
        rlinm   r5,r5,2,0,29            shift r5 left to calculate offset – 4 bytes per entry
        lhax    r4,r4,r5                r4 loaded with class value
        rlinm   r4,r4,2,0,29            shift r4 left to calculate offset – 4 bytes per entry
        lfsx    fp1,r3,r4               fp1 loaded with Freq[Case.attribute][Case.class]
        l       r5,T.34.Weight(RTOC)    r5 loaded with location of the value of Weight
        l       r5,0(r5)                r5 loaded with base address of array Weight
        l       r6,68(SP)               r6 loaded with value of p
        rlinm   r6,r6,2,0,29            shift r6 left to calculate offset – 4 bytes per entry
        lfsx    fp2,r5,r6               fp2 loaded with Weight[p]
        fa      fp1,fp1,fp2             fp1 loaded with fp1 + fp2
        frsp    fp1,fp1                 fp1 converted to single precision
        stfsx   fp1,r3,r4               fp1 stored in Freq[Case.attribute][Case.class]
        l       r3,68(SP)               r3 loaded with p
        cal     r3,1(r3)                r3 incremented by 1 (p + 1)
        st      r3,68(SP)               (p + 1) stored in 68(SP)
        l       r4,112(SP)              r4 loaded with LP
        cmp     0,r3,r4                 compare p with LP
        bc      BO_IF_NOT,CR0_GT,__L2c        branch if p is not greater than LP
```

The annotated assembler for the two loops in the modified code is:

104(SP) is the index of the attribute which is passed to the routine
108(SP) is the value of FP which is passed to the routine
112(SP) is the value of LP which is passed to the routine

First loop

```
__L2c:                     # 0x0000002c (H.10.NO_SYMBOL+0x2c)
        l       r3,T.22.Item(RTOC)      r3 loaded with location of the value of Item
        l       r4,0(r3)                r4 loaded with base address of Item
        l       r5,72(SP)               r5 loaded with value of p
        rlinm   r5,r5,2,0,29            rotate r5 left to calculate offset – 4 bytes per entry
```

```
lx      r4,r4,r5                r4 loaded with address of instance data EA(r4+r5)
st      r4,64(SP)               r4 stored in 64(SP)
l       r3,0(r3)                r3 loaded with base address of Item
l       r4,72(SP)               r4 loaded with base address of Item
cal     r4,1(r4)                r4 base address of Item incremented
rlinm   r4,r4,2,0,29            rotate r4 left to calculate offset – 4 bytes per entry
lx      r3,r3,r4                r3 loaded with address of instance data EA
st      r3,68(SP)               r3 stored in 68(SP)
l       r3,T.26.Freq(RTOC)      r3 loaded with location of the value of Freq
l       r3,0(r3)                r3 loaded with base address of Freq
l       r4,64(SP)               r4 loaded with offset for this instance in Item
lha     r5,104(SP)              r5 loaded with attribute chosen
rlinm   r5,r5,2,0,29            rotate r5 left to calculate offset – 4 bytes per entry
lhax    r5,r4,r5                r5 loaded with value of chosen attribute EA(r4+r5)
rlinm   r5,r5,2,0,29            rotate r5 left to calculate offset – 4 bytes per enty
lx      r3,r3,r5                r3 loaded with second base for index of Freq
l       r6,T.30.MaxAtt(RTOC)    r6 loaded with location of the value of MaxAtt
lha     r5,0(r6)                r5 loaded with value of MaxAtt
cal     r5,1(r5)                r5 incremented by 1 to get MaxAtt + 1
rlinm   r5,r5,2,0,29            rotate r5 left to calculate offset – 4 bytes per entry
lhax    r4,r4,r5                r4 loaded with class value
rlinm   r4,r4,2,0,29            rotate r4 left to calculate offset – 4 bytes per entry
lfsx    fp1,r3,r4               fp1 loaded with Freq[Case.attribute][Case.class]
l       r5,T.34.Weight(RTOC)    r5 loaded with location of the value of Weight
l       r7,0(r5)                r7 loaded with base address of Weight
l       r8,72(SP)               r8 loaded with value of p
rlinm   r8,r8,2,0,29            rotate r8 left to calculated offset – 4 bytes per entry
lfsx    fp2,r7,r8               fp1 loaded with Weight[p]
fa      fp1,fp1,fp2             fp1 loaded with fp1 + fp2
frsp    fp1,fp1                 fp1 converted to single precision
stfsx   fp1,r3,r4               fp1 stored in Freq[Case.attribute][Case.class]
l       r3,T.38.Freq2(RTOC)     r3 loaded with location of the value of Freq2
l       r3,0(r3)                r3 loaded with base address of Freq2
l       r4,68(SP)               r4 loaded with value of p + 1
lha     r7,104(SP)              r7 loaded with attribute chosen
rlinm   r7,r7,2,0,29            rotate r7 left to calculate offset – 4 bytes per entry
lhax    r7,r4,r7                r7 loaded with value of chosen attribute EA(r4+r7)
rlinm   r7,r7,2,0,29            rotate r7 left to calculate offset – 4 bytes per item
lx      r3,r3,r7                r3 loaded with second base for index of Freq2
lha     r6,0(r6)                r6 loaded with value of MaxAtt
cal     r6,1(r6)                r6 incremented by 1 to get MaxAtt + 1
rlinm   r6,r6,2,0,29            rotate r6 left to calculate offset – 4 bytes per enty
lhax    r4,r4,r6                r4 loaded with class value
rlinm   r4,r4,2,0,29            rotate r4 left to calculate offset – 4 bytes per entry
lfsx    fp2,r3,r4               fp2 loaded with Freq2[Case.attribute][Case.class]
l       r5,0(r5)                r5 loaded with base address of Weight
l       r6,72(SP)               r6 loaded with value of p
cal     r6,1(r6)                r6 incremented by 1 to get p + 1
rlinm   r6,r6,2,0,29            rotate r6 left to calculate offset – 4 bytes per entry
lfsx    fp1,r5,r6               fp1 loaded with Weight[p + 1]
fa      fp1,fp1,fp2             fp1 loaded with fp1 + fp2
frsp    fp1,fp1                 fp1 converted to single precision
stfsx   fp1,r3,r4               fp1 stored in Freq2[Case.attribute][Case.class]
l       r3,72(SP)               r3 loaded with p
cal     r3,2(r3)                r3 incremented by 2 to get p + 2
st      r3,72(SP)               r3 stored in 72(SP)
```

```
        l       r4,112(SP)              r4 loaded with LP
        cmp     0,r3,r4                 compare p with LP
        bc      BO_IF_NOT,CR0_GT,__L2c        branch if p is not greater than LP
```

Second loop

```
__L128:                 # 0x00000128 (H.10.NO_SYMBOL+0x128)
        cal     r3,0(r0)                r3 loaded immediately with 0
        st      r3,72(SP)               r3 stored in 72(SP) – p is set to 0
        l       r4,T.30.MaxDisrVal(RTOC)r4 loaded with  location of the value MaxDiscrVal
        lha     r4,0(r4)                r4 loaded with value MaxDiscrVal
        cal     r4,1(r4)                r4 incremented by 1 to get MaxDiscrVal
        cmp     0,r3,r4                 compare p and MaxDiscrVal + 1
        bc      BO_IF_NOT,CR0_LT,__L1cc       branch if p is not less than MaxDiscrVal + 1
__L140:                 # 0x00000140 (H.10.NO_SYMBOL+0x140)
        cal     r3,0(r0)                r3 loaded immediately with 0
        st      r3,76(SP)               r3 stored in 76(SP) – q is set to 0
        l       r4,T.42.MaxClass(RTOC) r4 loaded with the location of the value MaxClass
        lha     r4,0(r4)                r4 loaded with value MaxClass
        cal     r4,1(r4)                r4 incremented by 1 to get MaxClass + 1
        cmp     0,r3,r4                 compare q and MaxClass + 1
        bc      BO_IF_NOT,CR0_LT,__L1b0       branch if q is not less that MaxClass + 1
__L158:                 # 0x00000158 (H.10.NO_SYMBOL+0x158)
        l       r3,T.26.Freq(RTOC)      r3 loaded with the location of the value Freq
        l       r3,0(r3)                r3 loaded with base address of Freq
        l       r4,72(SP)               r4 loaded with p
        rlinm   r6,r4,2,0,29            rotate r4 left to calculate offset – 4 bytes per entry
        lx      r3,r3,r6                r3 load with second base for Freq
        l       r4,76(SP)               r4 loaded with q
        rlinm   r4,r4,2,0,29            rotate r4 left to calculate offset – 4 bytes per entry
        lfsx    fp1,r3,r4               fp1 loaded with Freq[p][q]
        l       r5,T.38.Freq2(RTOC)     r5 loaded with the location of the value Freq2
        l       r5,0(r5)                r5 loaded with base address of Freq2
        lx      r5,r5,r6                r5 loaded with second base of Freq2
        lfsx    fp2,r5,r4               fp2 loaded with Freq2[p][q]
        fa      fp1,fp1,fp2             fp1 loaded with fp1 + fp2
        frsp    fp1,fp1                 fp1 converted to single precision
        stfsx   fp1,r3,r4               fp1 stored in Freq[p][q]
        l       r3,76(SP)               load value of q
        cal     r3,1(r3)                r3 incremented by 1 to get q + 1
        st      r3,76(SP)               r3 stored in q
        l       r4,T.42.MaxClass(RTOC) r4 loaded with address of value of MaxClass
        lha     r4,0(r4)                r4 loaded with value of MaxClass
        cal     r4,1(r4)                r4 incremented by 1 to get MaxClass + 1
        cmp     0,r3,r4                 compare q and MaxClass + 1
        bc      BO_IF,CR0_LT,__L158   branch if q is less than MaxClass + 1
__L1b0:                 # 0x000001b0 (H.10.NO_SYMBOL+0x1b0)
        l       r3,72(SP)               r3 loaded with p
        cal     r3,1(r3)                r3 incremented by 1 to get p + 1
        st      r3,72(SP)               r3 stored in p
        l       r4,T.30.MaxAtt(RTOC)    r4 loaded with location of the value MaxAtt
        lha     r4,0(r4)                r4 loaded with the value MaxAtt
        cal     r4,1(r4)                r4 incremented by 1 to get MaxAtt + 1
        cmp     0,r3,r4                 compare p and MaxAtt + 1
        bc      BO_IF,CR0_LT,__L140   branch if p is less than MaxAtt + 1
```

A majority of the work in both the original and modified loop is related to address calculation for the arrays. The first loop of the modified code is not substantially different than the loop in the original code. In the modified loop the addresses for the location of the p and (p + 1) entries of Item are first calculated. The remaining part of the original loop is repeated twice with the second copy modified for Freq2 and (p + 1) thus the unrolling that was done in the modified loop is preserved in the assembler code. The compiler did not unroll the second loop in the modified code. Any parallelism in the loop is found by the hardware at runtime.

The modified code required additional modifications to C4.5. Freq2 has to be allocated and cleared prior to reaching the modified code. Additional code was added at the points in the code where this is done for the array Freq. It was possible to collapse the second loop of the modified code into an existing loop that follows this code.

Result from the performance measurements show that the modified code ran 2% slower than the unmodified code. The subroutine that contains the modified code ran 26% slower and since it is 8% of the total program this accounts for the 2% increase in overall execution time. Approximately twice the number of operations was executed in the fixed and floating point units for the modified subroutine. A good indication of the amount of parallel execution is the number of cycles per instruction (CPI). A lower number indicates that more parallelism was found. The CPI for the original code was 1.52 while it was 1.02 for the modified code. Unfortunately the more efficient use of the hardware did not yield a reduction in runtime.

One potential cause may be that the overhead cost of managing the second array outweighed the benefit of the additional parallelism. A second cause may be that the additional work of calculating the addresses for the second array is consuming resources that were previously used to start the next iteration of the loop. The next iteration of the loop can be speculatively executed if hardware resources are available and no data dependency exists.

A Second Attempt

Given the disappointing results of the first attempt to modify the code and the hypothesis that the overhead for Freq2 outweighed any increase in parallelism during execution, a different modification was made to the code. The loop was unrolled manually but the second array Freq2 was not introduced. This change eliminates the need to make other modification to the code. The source code for the change is:

```
p = FP;
/* start up code for odd no. of elements */
if ( ((LP – p) % 2) != 0) then {
        Case = Item[p];
        Freq[Case.attribute][Case.class] += Weight[p];
        p++;
}
/* process instances two at a time        */
for ( ; p <= LP ; p = p + 2) {
        Case = Item[p];
        Case2 = Item[p + 1];
        Freq[Case.attribute][Case.class] += Weight[p];
        Freq[Case2.attribute][Case2.class] += Weight[p + 1];
}
```

If the two instances both have the same attribute and class value then both instances will map to the same location in Freq. This data dependency prevents the execution of the statement for Case2. The data dependency in this example is read after write and is a true dependency that requires that the code be executed in the order it appears in the program to get the correct results. The second update must wait for the updated value from the calculation for Case. If this data dependency does not occur with high frequency then additional parallelism may be found during execution. The annotated assembler for the main loop is:

104(SP) is the index of the attribute which is passed to the routine
108(SP) is the value of FP which is passed to the routine
112(SP) is the value of LP which is passed to the routine

```
__L2c:                          # 0x0000002c (H.10.NO_SYMBOL+0x2c)
        l       r3,T.22.Item(RTOC)      r3 loaded with location of the value of Item
        l       r4,0(r3)                r4 loaded with base address of Item
        l       r5,72(SP)               r5 loaded with value of p
        rlinm   r5,r5,2,0,29            rotate r5 left to calculate offset – 4 bytes per entry
        lx      r4,r4,r5                r4 loaded with address of instance data EA(r4+r5)
        st      r4,64(SP)               r4 stored in 64(SP)
        l       r3,0(r3)                r3 loaded with base address of Item
        l       r4,72(SP)               r4 loaded with base address of Item
        cal     r4,1(r4)                r4 base address of Item incremented
        rlinm   r4,r4,2,0,29            rotate r4 left to calculate offset – 4 bytes per entry
        lx      r3,r3,r4                r3 loaded with address of instance data EA
        st      r3,68(SP)               r3 stored in 68(SP)
        l       r3,T.26.Freq(RTOC)      r3 loaded with location of the value of Freq
        l       r4,0(r3)                r4 loaded with base address of Freq
        l       r5,64(SP)               r5 loaded with offset for this instance in Item
        lha     r6,104(SP)              r6 loaded with attribute chosen
        rlinm   r6,r6,2,0,29            rotate r6 left to calculate offset – 4 bytes per entry
        lhax    r6,r5,r6                r6 loaded with value of chosen attribute EA(r5+r6)
        rlinm   r6,r6,2,0,29            rotate r6 left to calculate offset – 4 bytes per enty
        lx      r4,r4,r6                r4 loaded with second base for index of Freq
        l       r6,T.30.MaxAtt(RTOC)    r6 loaded with location of the value of MaxAtt
        lha     r7,0(r6)                r7 loaded with value of MaxAtt
```

```
cal     r7,1(r7)                  r7 incremented by 1 to get MaxAtt + 1
rlinm   r7,r7,2,0,29              rotate r7 left to calculate offset – 4 bytes per entry
lhax    r5,r5,r7                  r5 loaded with class value
rlinm   r7,r5,2,0,29              rotate r5 left to calculate offset - store in r7
lfsx    fp1,r4,r7                 fp1 loaded with Freq[Case.attribute][Case.class]
l       r5,T.34.Weight(RTOC)      r5 loaded with location of the value of Weight
l       r8,0(r5)                  r8 loaded with base address of Weight
l       r9,72(SP)                 r9 loaded with value of p
rlinm   r9,r9,2,0,29              rotate r9 left to calculated offset – 4 bytes per entry
lfsx    fp2,r8,r9                 fp1 loaded with Weight[p]
fa      fp1,fp1,fp2               fp1 loaded with fp1 + fp2
frsp    fp1,fp1                   fp1 converted to single precision
stfsx   fp1,r4,r7                 fp1 stored in Freq[Case.attribute][Case.class]
l       r3,0(r3)                  r3 loaded with base address of Freq
l       r4,68(SP)                 r4 loaded with value of p + 1
lha     r7,104(SP)                r7 loaded with attribute chosen
rlinm   r7,r7,2,0,29              rotate r7 left to calculate offset – 4 bytes per entry
lhax    r7,r4,r7                  r7 loaded with value of chosen attribute EA(r4+r7)
rlinm   r7,r7,2,0,29              rotate r7 left to calculate offset – 4 bytes per item
lx      r3,r3,r7                  r3 loaded with second base for index of Freq2
lha     r6,0(r6)                  r6 loaded with value of MaxAtt
cal     r6,1(r6)                  r6 incremented by 1 to get MaxAtt + 1
rlinm   r6,r6,2,0,29              rotate r6 left to calculate offset – 4 bytes per enty
lhax    r4,r4,r6                  r4 loaded with class value
rlinm   r4,r4,2,0,29              rotate r4 left to calculate offset – 4 bytes per entry
lfsx    fp2,r3,r4                 fp2 loaded with Freq[Case2.attribute][Case2.class]
l       r5,0(r5)                  r5 loaded with base address of Weight
l       r6,72(SP)                 r6 loaded with value of p
cal     r6,1(r6)                  r6 incremented by 1 to get p + 1
rlinm   r6,r6,2,0,29              rotate r6 left to calculate offset – 4 bytes per entry
lfsx    fp1,r5,r6                 fp1 loaded with Weight[p + 1]
fa      fp1,fp1,fp2               fp1 loaded with fp1 + fp2
frsp    fp1,fp1                   fp1 converted to single precision
stfsx   fp1,r3,r4                 fp1 stored in Freq2[Case.attribute][Case.class]
l       r3,72(SP)                 r3 loaded with p
cal     r3,2(r3)                  r3 incremented by 2 to get p + 2
st      r3,72(SP)                 r3 stored in 72(SP)
l       r4,112(SP)                r4 loaded with LP
cmp     0,r3,r4                   compare p with LP
bc      BO_IF_NOT,CR0_GT,__L2c    branch if p is not greater than LP
```

The assembler for this modification is very similar to the assembler for the first attempt. There is a minor difference in the assembler code since Freq2 in no longer used but the interesting difference is that compiler has utilized two additional registers. The performance of this version is very similar to the performance of the original loop. It appears that the overhead of clearing the Freq2 each time the subroutine was called accounted for the increase in runtime of the previous modification. This was confirmed with additional performance runs.

In all cases, it does not appear from examining the details of the performance runs that the functional units in the processor are running at their maximum capacity. Thus it appears that the maximum amount of parallelism is being extracted from this loop by the

Power2 architecture. It appears that the address calculations are limiting the amount of parallelism that can be found.

Future areas of potential work would include a study of modifications that could be made to the program and architecture to obtain additional parallelism in this loop. Reducing the number of address calculations may be possible if the existing instance data structure that uses pointers is replaced by an array data structure that could be accessed via fixed offsets. The impact to this change to the overall program would have to be studied since the cost of the partitioning of the instance array during recursion may exceed the gain in performance. Additional integer units in the processor architecture may provide additional resources for address calculation in the loop. Increasing the instruction fetch bandwidth would explore the possibility that the performance of the loop is limited by the availability of instructions to issue to the execution units. Finally the impact of increasing the bandwidth and the number of outstanding memory requests to memory to support the additional integer units and/or the increased instruction bandwidth would need to be studied.

Conclusions

This paper has explored potential areas where performance is influenced by the interaction of an application with the architecture of the system. The problem of classification is described along with several examples of its application. Building a model via an inductive approach is explored and contrasted with the approach of testing a model via a deductive hypothesis. Shortcomings are discussed of the classical statistical approach to building a model via n-way analysis of variance. An alternative approach using decision trees is introduced and the basic algorithm is described. Entropy is introduced for the measurement of information and an example is given. The program structure of C4.5, a decision tree generator, is discussed and the shortcomings of entropy are discussed. The modified version of entropy used by C4.5 is shown.

Finally three areas of interaction between program and processor architecture are examined. First is the interaction of the program with the cache and memory system. In C4.5, the number of instances and attributes will determine if the current level of the tree will fit in cache memory. Maximizing the data available in cache is the goal. Work related to out of core solutions to decision tree generation and potential application to an out of cache problem is discussed. The second area is the impact of the internal data structures on accessing the instance data. The pointer structure that is used in C4.5 simplifies the reorganization of the instances during the recursive partitioning but limits the ability to access the data in a structured manner. The third area is the impact that implementation has on performance. Two modifications are proposed to increase the amount of parallelism in a section of code. Performance measurements show the first modification increases the parallelism but resulted in longer runtimes. The second modification performed no better than the original code and verified that the overhead of the first modification was the source of the longer runtimes. Future work to further explore the interaction of the loop and the processor architecture is discussed.

The algorithm, structure and implementation have a large impact on the runtime characteristics of a program as shown in this paper. To maximize the performance of a program, details that affect performance must be examined throughout the implementation of a program. The final examples illustrate the difficulty in predicting the impact in actual performance from high-level decisions.

References

1) J. Ross Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1992.
2) S.-C. Fang, J. R. Rajasekera and H.-S. J. Tsao, *Entropy Optimization and Mathematical Programming*, Kluwer Academic Publishers, 1997.
3) J. Gehrke, R. Ramakrishnan and V. Ganti, "RainForest – A Framework for Fast Decision Tree Construction of Large Datasets", *VLDB '98, Proceedings of the 24th International Conference on Very Large Data Bases*, Morgan Kaufmann, 1998.
4) S.W.White and S. Dhawan, "POWER2: Next Generation of the RISC System/6000 family", *IBM Journal of Research and Development*, 38(1), pp. 493-502, 1994.
5) M. Thoennes and C. Weems, "Performance Characterization of Data Mining Application Via Hardware-Based Monitoring", *ITCOM 2001: Commercial Applications for High Performance Computing Proceedings*, SPIE, 2001
6) C.L. Blake and C.J. Merz, *UCI Repository of machine learning databases [http://www.ics.uci.edu/~mlearn/MLRepository.html]*, Irvine, CA: University of California, Department of Information and Computer Science, 1998.
7) John A. Sonquist, *Multivariate Model Building: The Validation of a Search Strategy*, Institute For Social Research – The University of Michigan, 1970.
8) Tom M. Mitchell, *Machine Learning*, WCB/McGraw-Hill, 1997.