More on Conjunctive Selection Condition and Branch Prediction

CS764 Class Project - Fall 2002

Jichuan Chang and Nikhil Gupta
{chang,nikhil}@cs.wisc.edu

Abstract

Traditionally, database applications have focused on the parameters of the disk devices to optimize performance. As the memory and caches get bigger, a larger fraction of the data can be stored in memory and caches. For applications where substantial data can be present in the memory, other optimizations which consider processor and memory characteristics can also be applied effectively.

In this report, we re-examine the effect of branch predictions on conjunctive selection conditions, for both current and future CPU designs. Our simulation results show that branch prediction does degrade the performance significantly, especially in future processors, which will have higher arithmetic computation bandwidth and branch misprediction penalty. No optimizer will be needed for choosing amongst conjunctive selection plans, since the strategy using no branches will always perform better.

I. Introduction

Traditionally, database applications have focused on the performance parameters of the disk devices to optimize performance. Some examples of such parameters are the high cost of random I/O compared to sequential I/O, large size of disk blocks, etc. The applications try to exploit these by processing the data in physical order and clustering related data into disk-block sized units. However as the memory and caches are getting bigger and bigger on present day computer systems, a larger fraction of the data can be stored in memory and caches. In some cases even the entire database can be stored in memory. For these applications where substantial data can be present in the memory, other optimizations which consider processor and memory characteristics can also be applied effectively.

[1] presents one such approach for designing efficient algorithms, giving consideration to the characteristics of the CPU and the memory hierarchy. It focuses on one commonly used database operation, namely applying a conjunction of selection conditions to a set of database records. It shows that the branch misprediction penalty can have a significant impact on the performance of an algorithm. [2] re-evaluates the algorithm proposed in [1] by means of an architectural simulator. It shows that while on the current processors (like Pentium III and Pentium 4), the new algorithm improves performance, the algorithm will not be effective on future processors. We first

re-evaluate the work on the real hardware (Pentium III) and then by means of simulations, make predictions for future processors.

The rest of the paper is structured as follows. In Section II we provide a brief overview of Superscalar Processors and Branch Prediction. In Section III we discuss Conjunctive Selection Conditions. Section IV discusses the methodology of our experiments. The results are presented in Section V and we conclude in Section VI.

II. Superscalar Processor and Branch Prediction

A typical modern superscalar processor [3], [4], [5] has the following hardware organization [6]. The major parts of the microarchitecture are: instruction fetch and branch prediction, decode and register dependence analysis, issue and execution, memory operation analysis and execution, and instruction reorder and commit. Underlying this organization is a pipelined implementation.

The instruction fetch phase supplies instructions to the rest of the pipeline. It fetches multiple instructions per cycle from the cache memory. During the decoding, renaming and dispatch phase, the instructions are removed from the instruction fetch buffers, examined, and control and data dependence linkages are set up for the remaining pipeline phases. Also the instructions are distributed to buffers associated with hardware functional units for later issuing and execution. During the execution and issue phase a check is made for the availability of the input operands of the instruction and the ready instructions are issued for execution. The memory operations (loads and stores) are handled in the memory operation analysis and execution stage. Finally, in the commit phase the effects of the instruction are allowed to modify the logical processor state. The purpose of this phase is to implement the appearance of a sequential execution model, even though the actual execution is very likely non-sequential.

The presence of branches in the program produces complications in the instruction fetch stage. A certain number of instructions are fetched from the address denoted by the program counter. In the default case the program counter is incremented by the number of instructions fetched and then the instruction are fetched from this new value of program counter. However since branch instructions redirect the flow of control, the fetch mechanism must be redirected to fetch instructions from the branch target in case a branch instruction is encountered. Often, when a branch instruction is fetched, the data required to make the branch decision is not yet available. One option is to wait for the required data to become available. However for performance considerations, the outcome of a conditional branch is predicted using a branch predictor.

At some later time, the actual branch outcome is evaluated. If the prediction made earlier was incorrect, then the instructions need to be fetched from the correct path. Also, if the instructions were processed speculatively, they need to be removed the pipeline and their effects should be undone.

Thus there is a cost associated with a branch misprediction because the work done on the instructions fetched from the wrong target is useless. In a superscalar processor multiple instructions are fetched, issued and executed per cycle. Thus each misprediction can cause a lot of instructions to be squashed. As the length of the pipeline

increases (Pentium 4 has 20 stages), it increases this cost further. The number of stages between when the branch prediction was made and when the real outcome is known will be larger. So a misprediction will affect an even larger number of instructions.

Various types of branch predictors have been proposed in the literature [7], [8]. The basic idea behind the working of a dynamic branch predictor is as follows. As the program executes the predictors store information regarding the past history of branch outcomes - either the specific branch being predicted, other branches leading up to it, or both. This information is stored in a *branch history table* or *branch prediction table*, which is indexed with the address of the branch instruction being predicted. The actual evaluation of the branch causes the history to be updated and this history is consulted for making each branch prediction. There is a basic tradeoff between the accuracy of a predictor and the resources consumed by the predictor. The more history information that a branch predictor stores, the more likely it is to make an accurate prediction. At the same time however it requires more hardware resources to store this history information.

III. Conjunctive Selection Conditions

[1] addresses the impact of the delay induced by a branch misprediction on a specific database operation. The database operation considered is applying a conjunction of selection conditions to a set of database records. The goal is to obtain the records which satisfy the conjunction of conditions in an efficient way. There are two assumptions made in the work. The first is that the database fits in the main memory. The second assumption is that the cost of processing the selections is a significant cost of the overall query, so that its optimization helps the entire query. This assumption is likely to be true since the selections usually form the intitial steps of a more complex query and hence are applied to a large number of records.

The *selectivity* of a condition applied to a table is defined as the ratio of the records in the table that satisfy that condition. This definition is the same for both a single condition or a conjunction of conditions. The selectivities are assumed to be independent so that the selectivity of a conjunction of conditions is obtained by multiplying the selectivities of the individual conditions.

The basic problem is as follows. We have a large table stored as a collection of arrays, one array per column. The column datatypes are assumed to have fixed length. The arrays are numbered `r1` through `rn`. Suppose the conditions to be evaluated are `f1` through `fn`, where each `fi` operates on a single column `ri`. We wish to evaluate these conditions on this table and return pointers or offsets to the matching rows.

A simple way to code the selection operation applied to all records will be as follows. The array `answer` will store all the results. The variable `j` is assumed to be initialized to 0 in all the following examples.

```
/* Algorithm Branching-And */
for(i=0; i<number_of_records; i++) {
      if(f1(r1[i]) && f2(r2[i]) && ... && fn([rn[i]))
```

```
            answer[j++] = i;
}
```

The work done by this implementation depends on the selectivity of the intitial conditions. When `f1(r1[i])` is zero, no further work is done for record `i`. However the potential problem with this implementation is that the equivalent assembly language code has `n` conditional branches. If the initial functions are not very selective, then the system may execute many branches. The closer each selectivity is to 0.5, the higher the probability that the corresponding branch will be mispredicted and hence will suffer the misprediction penalty as explained earlier. An alternative implementation proposed in [1] is to use logical-and (`&`) in place of `&&` as follows:

```
/* Algorithm Logical-And */
for(i=0; i<number_of_records; i++) {
      if(f1(r1[i]) & f2(r2[i]) & ... & fn([rn[i]))
            answer[j++] = i;
}
```

The above code fragment uses logical-ands (`&`) in place of branching-ands (`&&`). So, in the corresponding assembly code there is only one branch. Thus the branch misprediction penalty is expected to be lower. However, if `f1` is selective then this strategy might perform poorly because the computation is always done for `f1` through `fn`. Even this single branch can be avoided by using the following code:

```
/* Algorithm No-Branch */
for(i=0; i<number_of_records; i++) {
      answer[j] = i;
      j += (f1(r1[i] & f2(r2[i] & ... & fn(rn[i]));
}
```

[1] proposes the above three methods for performing the conjunctive selection and compares their performance. It also proposes a framework in which plans are chosen from a space of plans in which each plan is a combination of these three basic techniques. It has developed a cost model that takes branch misprediction into account and a cost-based optimization technique using dynamic programming to choose the best plan. It shows that significant performance gains can be achieved by using these optimized plans. In this paper we do not consider the optimization, rather we compare the optmized plans and the limit of speedup. We re-evaluate the results presented in [1] on actual hardware and through simulations, to explore the expected performance of these strategies on future processors.

IV. Methodology

In order to compare the various strategies for conjunctive selections conditions we use two methods - comparison on the actual hardware and through simulation. For the comparison on actual hardware we used a machine whose details are given in Table I.

Processor	Pentium III (Katmai) 550 MHz
Cache	512K L2 Cache
OS	Linux 2.2.19
Compiler	GCC 2.95.3 (Optimization flag: -O2)

TABLE I

Machine Details

To compare the execution times of two programs on real hardware, we compare the cycles taken by each to execute. The number of cycles required for execution is obtained by using the time-stamp counter provided in Pentium processors. The time-stamp counter keeps a count of the number of cycles that have elapsed since start-up. It is a 64-bit MSR (model specific register) that is incremented every cycle. To access this counter `rdtsc` instruction [9] is provided. To measure the branch prediction accuracy on the real machine we use the counters supplied in the Pentium machines. By default, the counters cannot be read or programmed from user mode. If the PCE bit in control register 4 is set, then the counters can be read from user mode, but they still must be set from kernel mode. For this we used the *pmc device driver* module [10] which sets this PCE bit at module load time to allow subsequent user mode programs to read the values of the counters with the `rdpmc` instruction.

To get an idea of the performance of various strategies of conjunctive selection on different types of processors we use the Simplescalar [11] simulator. This allows us to use a powerful branch predictor, and vary the parameters of our interest namely the number of pipeline stages (and hence the branch misprediction penalty) and the number of ALUs. Among various other things, the simulator keeps track of the number of cycles taken for execution, the total number of branches encountered and the number of branch mispredictions.

For the experiments we used a table which had 4 columns. The data in the table was generated on the basis of the selectivity values used for a particular experiment. We report the selectivity values used along with the results in the next section. In the default case the data in each column was arranged randomly. To see the effects of arranging the data in an interesting manner, we also report some results when the data in each column is sorted. Again we mention whether the data is sorted or not when we report the results.

V. Results

In this section we present the results of various experiments we performed on the actual hardware and on the simulator.

A. Branch Predictor Comparison

Fig. 1 compares the branch predictor performance of the Pentium III predictor with the Yags predictor. The x-axis shows the selectivity of the columns and on the y-axis is the misprediction rate. The figure shows that the Yags predictor performs better (i.e suffers lower mispredictions) than the Pentium III predictor for all the selectivity values. The figure also shows that when the selectivity value is very low or when the selectivity value is very high, then the misprediction rate is low (the branches are predictable). On the other hand when the selectivity value is in between, the predictability is low. As expected, the trend shown by both the predictors is same.

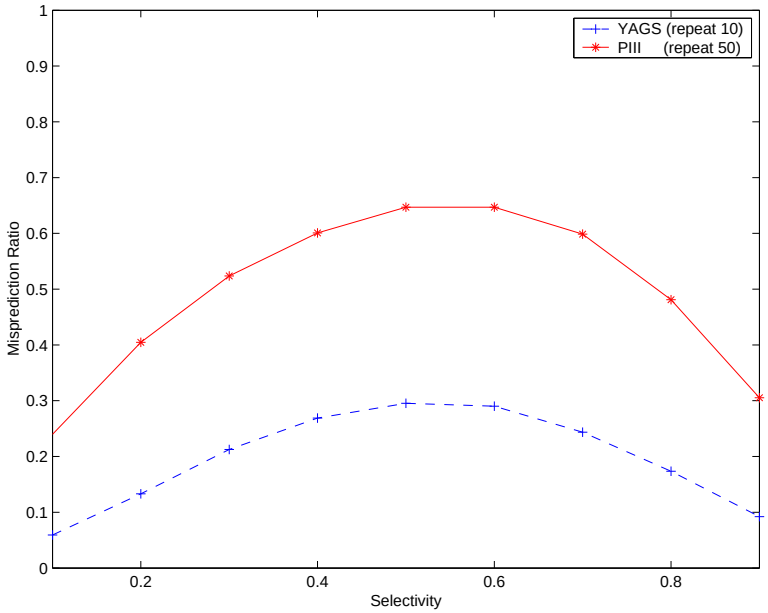

Fig. 1. Branch Predictor Comparison

B. Execution Results on a Real Machine

In this section, we discuss the experiments results collected on a real PIII machine. The machine has a 550 MHz Pentium III (Katmai) CPU, with a 512K L2 cache. As mentioned earlier, we use hardware performance counters to monitor the execution time and branch prediction accuracy. Three sets of experiments are conducted: the first and second sets try to reproduce the results shown in [1] (Figure 1 and 4 in [1] respectively); the third one shows how much the optimized plan can speed the execution up with varied selectivity values. Note that the CPU in our experiments has a lower frequency compared with that used in [1] (750 MHz). although they are of the same line of CPU model.

- Execution Time of Three Implementations:

  Figure 2 shows the execution time using three different evaluation plans: “&& branch” which stops evaluating the current row whenever the branching AND operation returns false, whereas the other two, “& branch” and

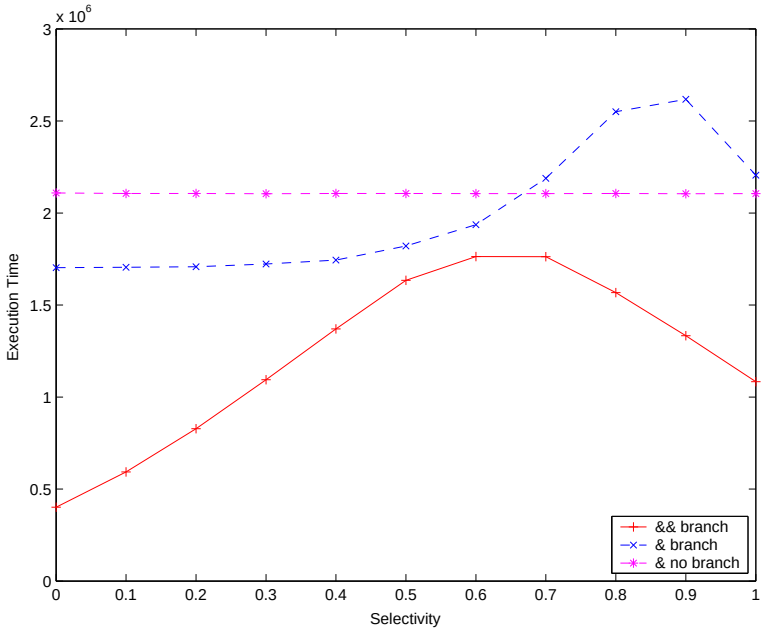

Fig. 2. Three Implementations: Pentium

"no branch", always compute the logical AND of all elements in a row. The table is filled with randomly generated 2000-row, 4-column boolean values. For each data point, all four columns have the same selectivity. Interestingly, although each curve has the similar shape as its counterpart plotted in [1], the relative position of the "&& branch" curve is much lower in our experiments. So "&& branch" always has the best performance for all selectivity ranges. This implies that not only CPU models (e.g. Pentium III vs. UltraSparc IIi) but also more detailed CPU configuration parameters (e.g. frequency) can affect the relative speed of different evaluation plans. Dynamic compilation or runtime specialization should be employed if such detailed information can not be determined at compilation time.

- Execution Time using Optimized Plans:

  Figure 3 evaluates the effectiveness of optimized plans for conjunctive selections. In this case, the selectivities of column 2, 3 and 4 are fixed, being 0.25, 0.5, 0.75 respectively. The selectivity of column 1 is varied from 0 to 1. Still the branching AND plan runs faster than branching with logical AND and no branch. We believe that the limited ALU bandwidth of our processor favors using branch to reduce ALU operations.

  It is also shown that when the first condition becomes less selective, certain optimized plans ("(1&2) && (3&4)" and "(2&3) && (1&4)") can further reduce the execution time. When the first logical AND term in such expressions has low selectivity, the && operator will eliminate the computation of the second logical AND, this can save execution time when ALU bandwidth is limited. In theory, the optimal speedup (against the "&&" plan) can be achieved by using the best plan according to the selectivity estimation.

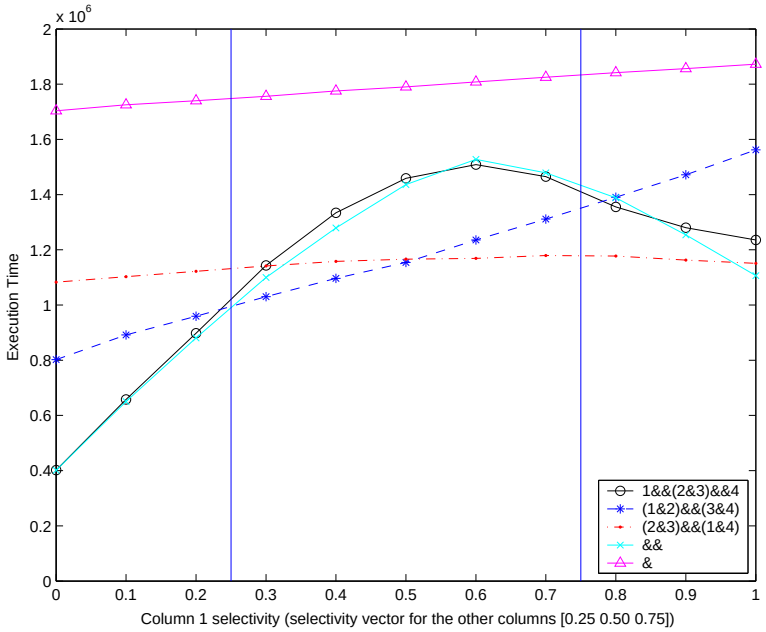

Fig. 3. Execution Time: Unequal Probabilities

- Optimal Speedup for Varied Selectivities:

  Based on Figure 3, we vary the selectivities of column 2, 3, and 4. By making the selectivities of these columns closer to 0.5, we want to compare the optimal speedup (among all plans in Figure 3) curves under three different selectivity vectors for column 2, 3, and 4: [0.25, 0.50, 0.75], [0.35, 0.50, 0.65], and [0.45, 0.50, 0.55]. We observe that the closer the selectivities are to 0.5, the smaller the optimal speedup. For example, the maximum speedup for [0.35, 0.50, 0.65] is 24%, while the maximum speedup for [0.45, 0.50, 0.55] is 16%.

To summarize, experiments on a Pentium III processor reveal that (1) contrary to [1], "&& branch" runs faster than "& branch" and "no branch" on our machine. Detailed CPU parameters are needed to explain this phenomena; (2) certain optimized plans proposed in [1] can further reduce the execution time compared with "&& branch;" (3) such optimizations have optimal speedups, which become smaller when the selectivities are closer to 50%.

C. Simulation Results

To understand the impact of future micro-processor design on conjunctive selection evaluation, we simulate different hardware configurations to observe the performance trend. We only study the effect of (1) ALU execution bandwidth (how many independent arithmetic operations can be finished per cycle), and (2) branch misprediction latency (how many cycles the CPU pays for each branch misprediction without doing useful computation), because these two factors are the most relevant to execution throughput. Branch predictor and its table size will determine branch (mis-)prediction ratio, which is also important to our study. However, we decide to use a state-of-art predictor (YAGS) with rather large history table, assuming that current research branch predictors are sufficiently good, even

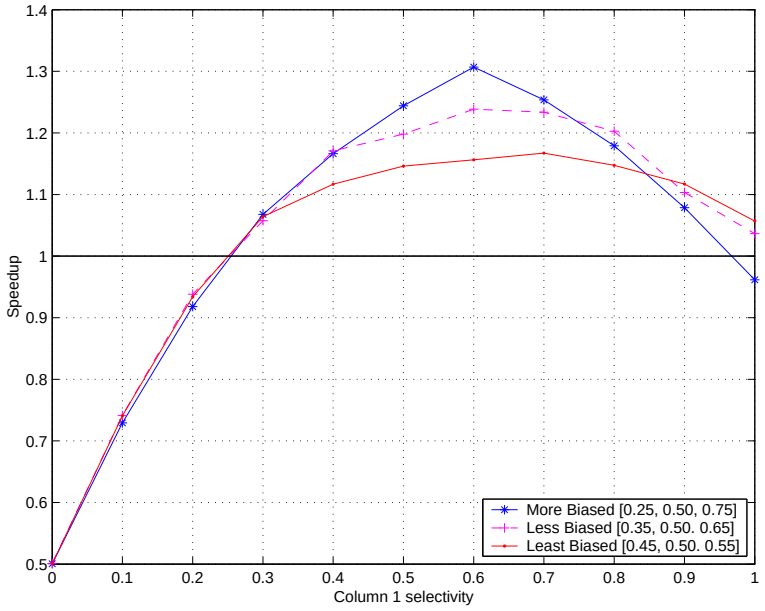

Fig. 4. Speedup Limit using Optimized Plans

for future processors. The size of the predictor is fixed, except in one experiment, where we try to study its impact on branch prediction accuracy for our interested workload.

Similar to Figure 2, we collect the execution time to evaluate a table of randomly generated boolean values. There are 12 sub-diagrams in Figure 5, all diagrams in the same row have the same ALU bandwidth, all diagrams in the same column have the same misprediction penalty. The x-axis is the selectivity, and the y-axis is the execution time in a relative unit. From top to bottom, 1, 2, 4, and 8 ALU units are configured for each row; from left to right, 2, 12, 20 cycles are wasted per misprediction for each column. Future processor design is moving to the right-bottom corner, while current high-end processors usually have 4 ALU units with misprediction penalty between 2 and 12 cycles. A YAGS branch predictor with 16K-entry table is used, the processor configuration is balanced accordingly.

Apparently, for future processors with more ALU bandwidth and larger penalty, branches degrade performance significantly, considering that the table is randomly filled and unsorted. Using arithmetic calculation instead of branches usually runs faster except when the table has extremely low or high selectivity, which in turn is not common for our interested workloads. Due to the simplicity and performance advantage of the “no branch” plan, optimization proposed in [1] would not be very useful in the future.

To understand how much a better branch predictor can help, we also run simulations with different YAGS table sizes. The branch misprediction ratio curves are shown on the left in Figure 6. The legend shows the value of `N` such that there are 2^N entries are in the history table. To our surprise, for our benchmark (repeat evaluation for 10 times), using larger history table can reduce the number of mispredictions even if the data is not sorted. By comparing the result with that of running on a 10X larger database table with 1/10 repeat times (shown on the right

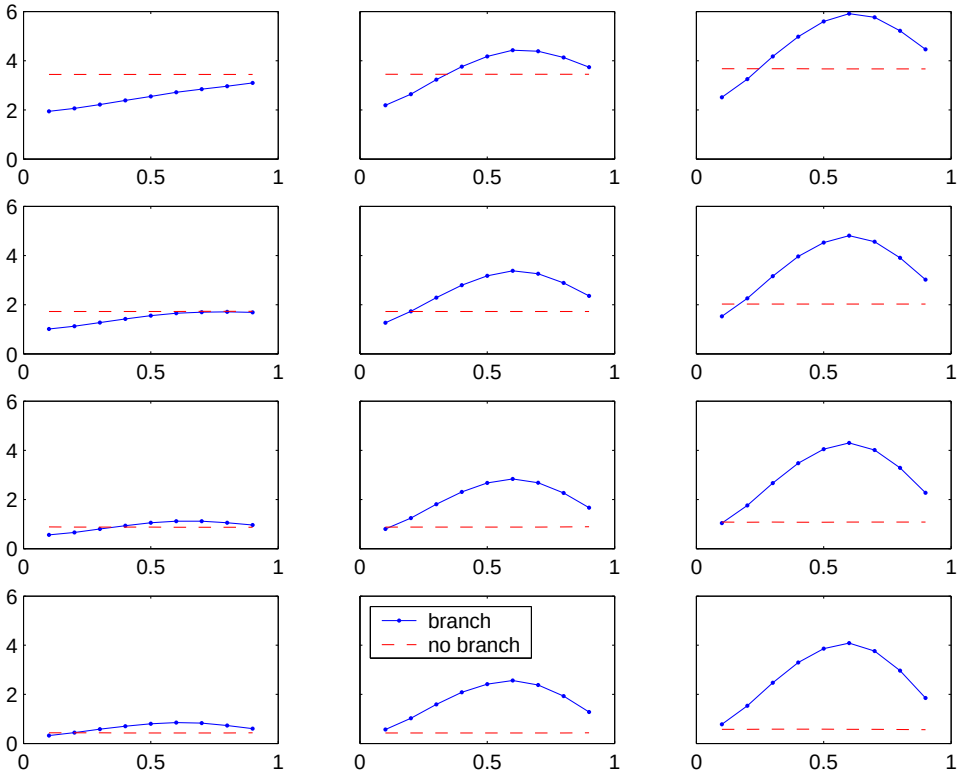

Fig. 5. Execution Time on Unsorted Table: Varied Hardware Configurations

in Figure 6), we realize that YAGS is very good at memorizing the history, but rather bad at predicting random numbers (larger tables may give less accurate guesses). The dramatic difference between the two diagrams raises questions on how should we conduct experiments: repeating selection many times on a small table will train the predictor and underestimate the branch misprediction ratio!

Lastly, we want to examine the effect of sorting the randomly generated data. By sorting the table used in the above experiments, we essentially make the branch prediction almost 100% accurate. In this extreme case, we compare the performance of "&& branch" with "no branch" under varied hardware configurations. According to Figure 7, branch now is faster (at most by 2X), while the difference between the two plans decreases as the selectivity increases. Although we feel it is not likely that every column in the table is sorted in reality, this diagram determines the optimal speedup that "&& branch" may have on any data set. A perfect predictor would thus reduce the number of ALU operations without paying for misprediction penalty.

By simulating different hardware configurations, for the workload we are interested, we conclude that branches will be more harmful in future processors, which usually have both higher computation bandwidth and larger misprediction penalty. Even using a large, sophisticated branch predictor, the branch prediction accuracy is low for the workload that researchers have previously studied (random, unsorted data), not to mention the underestimation caused by repeating the same selection. However, when the data is sorted, its regular pattern leads to almost perfect

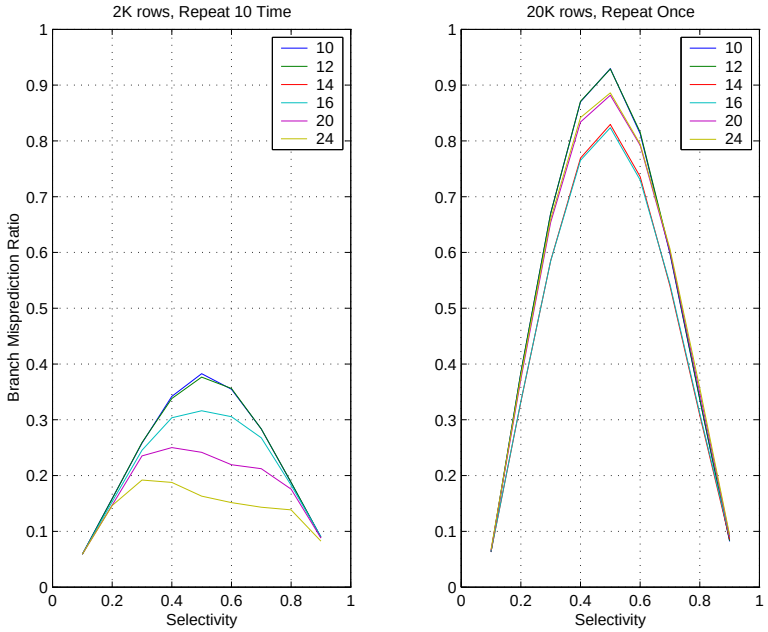

Fig. 6. Branch Misprediction Ratios for Varied YAGS Sizes, on Different Data Sets

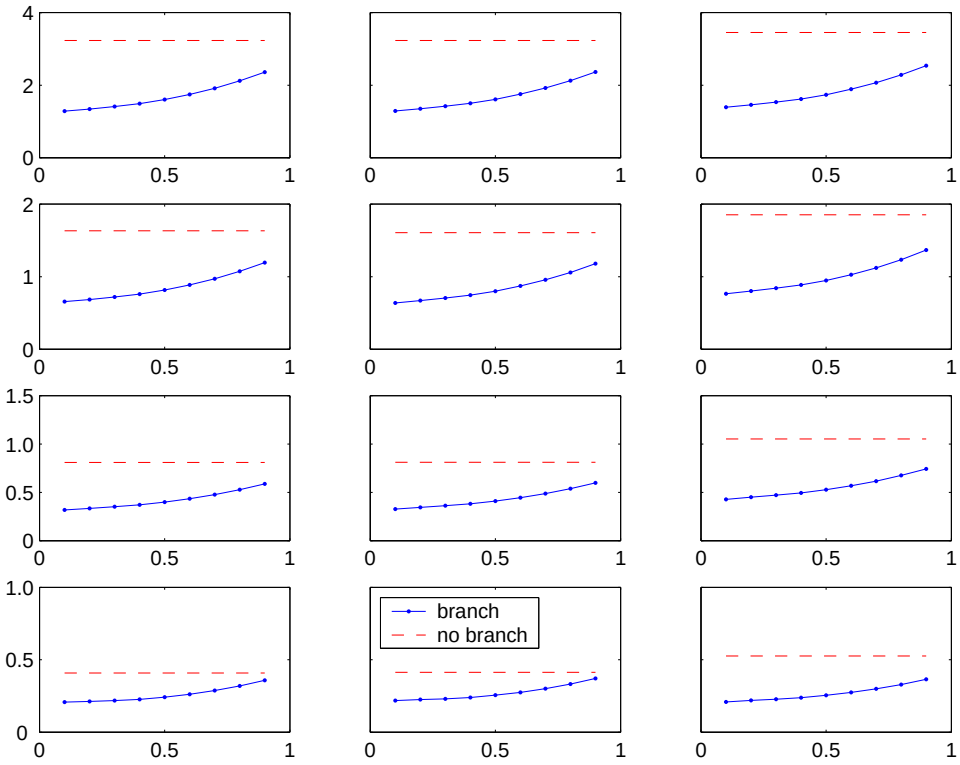

Fig. 7. Execution Time on Sorted Table

prediction thus less arithmetic computation, which is translated into smaller execution time. Whether the data layout is regular or random will be determined by the data structure used in conjunctive selections, these results can only help the designers to choose the fastest implementation according to their workloads.

VI. Conclusion

Researchers in database query optimizations have been aware of the cost of I/O and, more recently, memory accesses for a long time. Until recently, the impact of branch prediction has been mostly ignored. In this report, we re-examine the effect of branch predictions on conjunctive selection executions, for both current and future CPU designs. Our results show that branch prediction does affect the performance significantly. Besides the CPU model as used in previous study, more hardware parameters are needed to estimate the performance of selection plans. On one hand, when data is perfectly sorted, using branching AND will be faster. On the other hand, in the more common case, when the data is unsorted, using branching AND is always worse than the no branch plan for future processors which have higher arithmetic computation bandwidth and branch misprediction penalty. We expect the no-branch plan to always perform better in the future, so there will be no need for the optimizer to choose from alternative selection schemes.

References

[1] K. A. Ross, "Conjunctive selection conditions in main memory," in *ACM SIGMOD 2002*, 2002.

[2] S. Balakrishnan, "Conjunctive selection conditions and branch predictions," Tech. Rep., Spring 2002.

[3] K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor," 1996.

[4] L. Gwennap, "Digital Leads the Pack with the 21164," 1994.

[5] M. Slater, "AMDs K5 Designed to Outrun Pentium," 1994.

[6] J. Smith and G. Sohi, "The Microarchitecture of Superscalar Processors," 1995.

[7] J. E. Smith, "A study of branch prediction strategies," in *Proc. 8th Annual Symposium on Computer Architecture*, 1981.

[8] T. Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive training branch prediction," in *Proc. 19th Annual International Symposium on Computer Architecture*, 1992.

[9] Intel, "Using the rdtsc instruction for performance monitoring, http://cedar.intel.com/software/idap/media/pdf/rdtscpm1.pdf."

[10] "Pentium performance monitoring counters (pmc) device driver for linux, http://www.cs.cornell.edu/courses/cs612/2002sp/assignments/ps1/pmc/pmc.html."

[11] D. Burger, T. M. Austin, and S. Bennett, "Evaluating future microprocessors: The simplescalar tool set," Tech. Rep., July 1996.