RC22223 (W0110-086) October 31, 2001
Computer Science

IBM Research Report

Performance Characterization and Micro-Architecture Exploration of a Data Mining Application via Hardware Based Performance Monitoring and Simulation

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

Introduction

Internal complexity in modern microprocessors increasingly hides how programs actually execute. Increasing use of techniques that include the use of caches and out-of-order execution to increase performance results in non-deterministic execution. Trying to predict the actual performance of a program is no longer just adding up the execution times of the instructions and the memory references. This paper explores the use of two approaches to examine the execution of a program on a superscalar processor. The first approach utilizes hardware based performance monitoring to examine the execution of a widely used data mining program using a synthetic benchmark program for performance bottlenecks. The second approach explores the use of a micro-architecture simulator to explore the impact of micro-architecture changes on the performance of the same data mining application. Impact of the structure of the source code on the amount of instruction level parallelism is also explored in the micro-architecture simulator by studying the change in performance from a source level change. Results from both approaches are discussed and changes to the micro-architecture are proposed.

Environment

An IBM Power II [1] running IBM's version of Unix (AIX) was used for this work. Introduced in 1993, this is the first generation of IBM microprocessors to contain hardware performance monitoring [2]. The processor can monitor 20 programmable events comprised of 4 groups of five events from the Instruction Control Unit (ICU), Fixed-Point Unit (FXU), Floating Point Unit (FPU) and Storage Control Unit (SCU) and has a dedicated cycle counter. Events to be monitored in each unit are selected via system control registers. The mode of the processor during which data will be collected can be selected and are privileged, non-privileged or all. A bit is included in the processor status word (PSW), which can be set for a process and will restrict the monitoring to that process. Event counts are collected in 32 bit registers and there is no interrupt in case of overflow. The initial system that was used for this work was a node in an IBM RS/6000 SP located at the University of Massachusetts, which was a 66 MHZ IBM Power II node with 256 MB of memory. Due to hardware issues that are discussed later in the paper, we moved our work to an IBM RS/6000 SP, which contains the second generation of the IBM Power II. The second generation of the Power II is a single chip implementation and the node that we use is a 120 MHz processor with 512 MB of memory. Details of the performance monitoring hardware are similar to the earlier model except that the number of events that can be monitored simultaneously has been reduced to five and there are preset groups of events that can be sampled. An additional counter has been added that counts the number of instructions executed and, like the cycle counter, is dedicated and cannot sample other events. Figure 1 shows a block diagram of the hardware performance monitor.

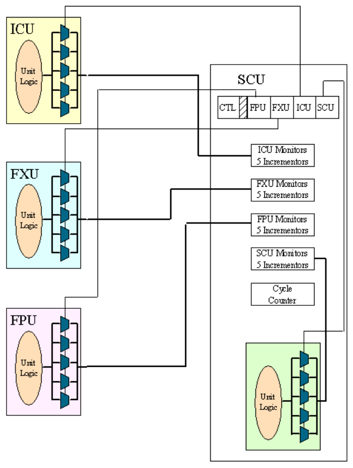
ICU
FXU
FPU
Unit Logic
SCU
CTL
FPU
FXU
ICU
SCU
ICU Monitors 5 Incrementors
FXU Monitors 5 Incrementors
FPU Monitors 5 Incrementors
SCU Monitors 5 Incrementors
Cycle Counter

Figure 1. PERFORMANCE MONITOR

The software environment utilizes the monitoring program RS2HPM [3], which provides an environment to access and utilize the performance monitoring hardware. Throughput of an IBM RS/6000 SP has been monitored with this package [4]. An AIX kernel extension is implemented to allow access to the performance counters and the configuration and control registers. A daemon is provided that implements an application interface to the kernel extension and provides a control interface for the performance monitor to the application via TCP/IP and provides commands to configure, stop/start and retrieve data from the performance monitoring hardware. The daemon is responsible for collecting the counters prior to counter overflow and for combining all of the partial counts to provide the summary counts. Reading and clearing the counters at a fixed frequency achieves this. Users are provided a utility that sets up the environment via the daemon, forks the user's program and collects the results from the daemon. Since a fixed number of events can be observed for each run of the program, the utility supports making multiple runs which are then formatted into a single report. Support was added for measuring user programs for the single-chip IBM Power II. A block diagram is shown in Figure 2.

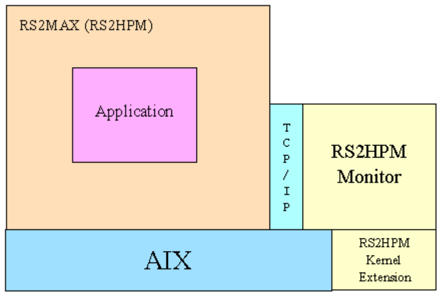

Figure 2. STRUCTURE OF RS2HPM

The commercial data mining application that was studied in the paper is the C4.5 [5] decision tree generator. C4.5 is widely used because of its availability and speed. Release 8.0 was used for this work.

Environment setup and debug

After the RS2HPM package was installed, the first step was to verify the results that the environment was producing. A simple program that was easy to verify was used. Source code and assembler code for the program body is:

```
Source:
        main(int argc, char **argv) {
          double a=0.1, b=1.001, c=0.0, d=1.0, e=0.5;
          int i,N;
          N=atoi(argv[1]);
          for (i=0; i<N; i++)
            c=c+a*b+d/(c+e);
        }

Assembler:
        __L48:
             fa      fp3,fp31,fp1        Floating Point Add
             fma     fp5,fp4,fp2,fp31    Floating Point Multiply and Add
             fd      fp3,fp0,fp3         Floating Point Divide
             fa      fp31,fp5,fp3        Floating Point Add
             bc    BO_dCTR_NZERO,CR0_LT,__L48 Decrement counter and branch if not zero
```

Figure 3. Source code and Assembler for Debug Program

Given the number of events that were monitored, 14 passes of the program are required to collect all of the different events. Only the non-privileged user processes were monitored. To compensate for performance variations in the environment between runs we run each group of 14 passes 30 times and average the results. Also the program was run three times prior to monitoring to warm up the cache. The initial results indicated high variability. This was attributed to the interference of other processes since the mode selection granularity is limited to privileged/non-privileged/all mode. Use of the bit in the PSW to restrict monitoring to only a single process was explored, but the current version of AIX does not preserve this bit in a context switch. Running AIX in "maintenance mode", which is equivalent to single user mode, was explored. Facilities that RS2HPM requires are present in single user mode. After rerunning the tests a variance was obtained that was deemed acceptable for our work.

Next the data that was being collected was examined. From the assembler fragment shown in Figure 3 there are two floating-point additions, one floating-point multiply/add, one floating-point divide and one branch-on-count per iteration. A loop count of 1,000,000 was used for our experiments. Distribution notes for RS2HPM documented a problem with floating point divides not being reported. From the experiments it was observed that the reporting of floating point divides is dependent on the code that is being executed. It was observed that the performance counters only reported 4 floating-point additions and 1 floating-point multiply/add per iteration. Branch statistics also reported incorrect information. For the run of 1,000,000 iterations there were 1,000,000 total branches. Detailed branch counts reported 1,000,000 conditional branches taken and 1,000,000 non-conditional count taken branches.

Given the two inconsistencies that were found in the performance monitor it was a concern as to whether there might be additional issues with this version of the processor. Moving our work to the second generation of the IBM Power II, which is a single chip implementation, was explored. It appeared that issues in the performance monitoring hardware had been fixed in this implementation. The issue in using the single chip version was that none of the nodes of our IBM RS/6000 SP contained the single chip processor. Access was obtained to a machine at IBM Research that contained nodes with the single chip processor. RS2HPM code required extensions to support detailed measuring on a single chip processor system. Given the smaller number of counters available and the restriction of preset groupings, the number of passes per run increased to twenty-six.

Rerunning the tests in single user mode produced the correct results. The only remaining inconsistency that was observed is that the totals for the floating-point operations were consistently slightly under the number of operations that should have occurred for the 1,000,000 loop iterations. Initially it was suspected that operations were being lost while the counters were being collected and reset. Upon rerunning the test program collecting data for the three modes (non-privileged, privileged, all) it was discovered that in the mode, where all system activity was collected, the correct count was obtained. If the counts from the privileged and non-privileged modes were added together, the correct number was obtained. It is speculated that this is occurring because the performance counter is enabled/disabled by the privileged/non-privileged state of the processor. The hypothesis is that the user operations that are in the processor pipeline are not flushed at the context switch and are allowed to complete. Performance counters are updated at the completion of the operation and since the processor is now in privileged mode the performance counters are disabled and the completed operations are not counted.

A simple program allowed the hardware performance monitoring to be installed and tested. It was possible to predict the expected results from the program and verify the results. This work allowed the identification of the incorrect behaviors and either resolution or understanding of them thus resulting in a verified environment to begin the application work.

Evaluation of c4.5

For the evaluation of C4.5 a synthetic benchmark called LED [6] was chosen, which generates test cases comprised of the state of the seven segments of a numeric LED display and the decimal value that they represent. States of the segments are binary values with 1 indicating that the segment is on. Decimal values have the range from zero to nine. Test case data is generated with a selectable probability of an error in each segment. We chose the default value of ten percent. Training sets of 1000, 5000, 10000, 50000, 100000 cases were generated. LED is not a particularly challenging problem for decision tree generation but it allowed the ability to scale the size of the problem and explore the data from the hardware performance monitor.

The first measurement that was looked at was the performance of the cache as the number of test cases increased. A node had a 256KB L1 data cache and no L2 cache. Figure 4 shows the miss rate per instance for the five test cases that were run. Results show a change in behavior at 10,000 cases, which is where it is believed that the size of the problem has exceeded the size of the L1 cache.

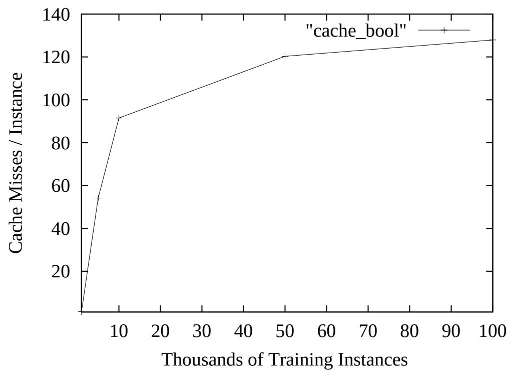

Figure 4. Data Cache Misses Per Instance vs. Number of Instances

A CPI plot shows a similar change in behavior that occurs at 10,000 cases. Figure 5 shows the CPI for the five test cases.

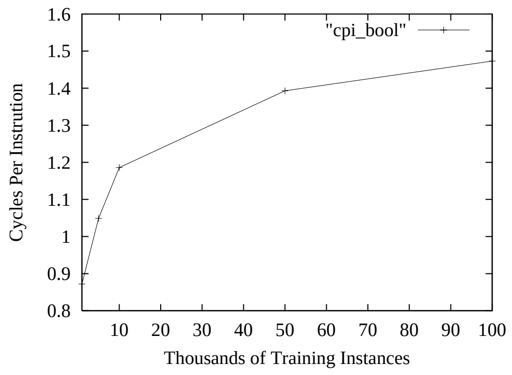

Figure 5. Cycles Per Instruction vs. Number of Instances

It appears that the behavior of C4.5 is very sensitive to the size of the cache. It could benefit from the additional cache capacity and bandwidth that would be available if the program was running on multiple nodes in parallel.

Table 1 shows the distribution of the number of total instructions that was executed by the three function units in the processor. These units are the fixed-point unit (FXU), floating point unit (FPU) and instruction control unit (ICU). Results are what we expected from C4.5, since the majority of the work in program involves the fixed-point unit for the counting of characteristics of the instances.

Number of Instances	%FXU	%FPU	%ICU
1000	75.2	5.8	18.9
5000	76.1	4.7	19.2
10000	76.2	4.4	19.3
50000	76.3	4.4	19.3
100000	76.4	4.3	19.3

Table 1. Percentage of Instructions Per Unit

Whether the bandwidth from the cache to the memory was sufficient for C4.5 was another question. Power II allows two outstanding requests from the cache to memory. Table 2 shows the percentage of the total number of cycles that either of the two fixed-point units where stalled due to a busy cache. A cache stall will only happen when there are two outstanding requests from the cache to the memory system. The memory bandwidth does not seem to be an issue for C4.5.

Number of Instances	% Total cycles FXU 0 held by cache	% Total cycles FXU 1 held by cache
1000	1.4	1.5
5000	1.2	1.5
10000	0.8	1.2
50000	0.6	1.0
100000	0.6	0.9

Table 2. Percentage of total cycles that each FXU Unit is stalled

The final question was how the available resources in the fixed-point unit were being utilized. Table 3 shows the number of instruction dispatched to the FXU when instructions were available. Two instructions per cycle can be dispatched to the FXU. For C4.5 the unit is issuing the maximum of two instruction sixty percent of the time.

Number of Instances	1+ / 0 Dispatched	1 / 1 Dispatched	2+ / 1 Dispatched	2+ / 2 Dispatched
1000	4.9	9.0	26.0	60.1
5000	5.3	8.2	26.5	60.0
10000	5.4	8.0	26.6	60.0
50000	5.5	7.7	26.8	60.0
100000	5.4	7.6	26.8	60.2

Table 3. Percentage of Dispatch Occurrence

From the performance data that was collected it appears that the IBM Power II is implemented with a well-balanced set of resources, which is indicated by the fact that the processor does not show any significant performance issues in running C4.5. As the number of functional units is increased in future processor generations, a similar balance should be preserved as opposed to adding more of one type of unit.

Study of Architectural Modifications

The question is what resources should be modified to increase the performance of C4.5 on a superscalar processor. In modifying the resources of the processor we need to insure that it is done in a balanced manner. If it is not then constraints from other elements in the processor may prevent the added resources from being efficiently utilized. A previous paper [reference to synthesis project] discusses potential modifications to the processor to increase the performance of C4.5. These resulted from work that modified a section of code to study if greater parallelism could be found by hardware as a result of restructuring the code. The conclusions of the paper indicate that the limits on the performance of the code could be the result of contention for resources for address calculation in the modified loop. Modifications that were suggested are: additional integer units, increased instruction fetch bandwidth and increased bandwidth and number of outstanding memory requests to memory.

Environment

To study the impact of modifying the micro-architecture a simulator that can simulate the micro-architecture is required. A functional simulator will work for software development but will not provide the hardware detail that is required to study the modification of processor resources. Since this work has been done on an IBM Power2 processor, the use of a version of SIMOS [7] that has been modified by the IBM Austin Research Laboratory [8] was explored. This version supports the PowerPC architecture. Given that the IBM Power2 heavily influences the PowerPC family, the architectures are similar. A copy of the simulator was installed on an IBM RS/6000 with a 32-bit PowerPC 604e processor running AIX. A benefit of SIMOS-PPC is that it can run a slightly modified version of the AIX operating system with reasonable performance and provides an environment that is equivalent to a real system.

The documentation for SIMOS-PPC indicates that there are three models available for the simulator. They are a fast, functional and detailed simulator. During bring up of the simulator on the 604e system, it was discovered that only the SIMPLE model, which is the functional simulator, was available. A startup message in the simulator stated that the BLOCK model was not debugged for a 32-bit environment and that the SIMPLE model would be used. The work was then transferred to an IBM RS/6000 with a 64-bit RS64 processor. The BLOCK model is supported on this system. With the simulator installed and operational, the source code was examined to determine what modifications could be made to the micro-architecture. It was discovered that only two of the three models have been implemented. The two models are BLOCK and SIMPLE. The BLOCK model implements a just-in-time (JIT) environment were the simulated instructions are mapped to instructions that are native to the processor on which the simulator is executing. This results in very fast functional simulation. The SIMPLE model is a serial functional simulator that simulates the instructions and is slower than the BLOCK model. The third model was indicated in the documentation to be a version of the Simple Scalar simulator modified to model the PowerPC architecture. After examination of the code it was determined that this model is currently not implemented in SIMOS-PPC. Given that no model in SIMOS-PPC can support the simulation of micro-architecture, SIMOS-PCC was not usable for this work.

Given that there are no available simulators that implement the Power/PowerPC micro-architecture, other simulators were explored. The Simple Scalar [9] simulator was chosen since its out-of-order model supports the modeling of micro-architecture. Although the basic micro-architecture is different from the Power/PowerPC architecture, the model is parameterized to allow some customization of the base micro-architecture. A set of parameters was chosen that closely models the Power2 architecture given the constraints of the base micro-architecture. Installation of Simple Scalar and the associated compiler environment was a challenge. After attempting to install the environment on IBM AIX and then a Linux system it was successfully installed on a Sun running Solaris 2.8. From reviewing various forums it appears that the problem is that Simple Scalar was developed with a bias towards BSD Unix and the first two operating systems are not BSD based.

With Simple Scalar installed and verified with diagnostics tests, the next step was to run C4.5. Since Simple Scalar is implementing a specific architecture, it is necessary to recompile C4.5 for that architecture. The GNU gcc environment that is shipped with Simple Scalar provides a cross-compiler that has the Simple Scalar instruction set as the target. The Simple Scalar environment has a number of simulators that implement different styles of simulation. The three of interest are FAST, SAFE and OUTORDER. FAST provides a fast functional simulator that implements none of the traditional hardware checks such as access permission on memory references. SAFE is a functional simulator that enforces all of the traditional hardware checks. OUTORDER is the micro-architecture simulator that will be used to study the impact of resource changes. Since OUTORDER is simulating the micro-architecture it is significantly slower than the other two simulators. C4.5 was first tried on the FAST simulator where it worked correctly. It was then tried on the SAFE simulator where a segmentation error occurred inside the simulator due to an attempted read of address 0. The OUTORDER simulator also encountered the same error. Simple Scalar provides a symbolic debugger for the environment that was not very useful in tracking down a source level problem. By adding print statements to the C4.5 source it was possible to determine that the problem was with a system call that was passed a pointer that was modified to point to address location 0. The call was to the time() system function and the results were used in the output generated by C4.5. Since it has no impact to the functionality of C4.5, the offending call was commented out. With this change C4.5 executed correctly on both the SAFE and the OUTORDER simulator.

Parameters For IBM Power2

The next step was to determine a set of parameters that models the Power2 micro-architecture within the constraints of the Simple Scalar micro-architecture. There are several aspects of the Power2 architecture that do not map well to the Simple Scalar micro-architecture. Power2 integrates the arithmetic/logic and multiplier/divide units into a single unit for both integer and floating point operations but Simple Scalar provides separate units for each. Power2 has two integer and two floating point units. It was decided to model the Power2 by using 2 integer ALUs, 1 integer mult/div unit, 2 floating point ALUs and 1 floating point unit. The width of the instruction decode and issue are limited to powers of two but the Power2 supports the dispatch of five instructions. A dispatch value of four was used. Table 4 lists the parameters that were used for the simulation of the Power2 architecture.

Parameter Description	**Flag**	**Value**
instruction fetch queue size (in insts)	-fetch:ifqsize	8
extra branch mis-prediction latency	-fetch:mplat	1
branch predictor type {nottaken|taken|perfect|bimod|2lev}	-bpred	nottaken
instruction decode B/W (insts/cycle)	-decode:width	4
instruction issue B/W (insts/cycle)	-issue:width	4
run pipeline with in-order issue	-issue:inorder	false
issue instructions down wrong execution paths	-issue:wrongpath	true
register update unit (RUU) size	-ruu:size	16
load/store queue (LSQ) size	-lsq:size	8
l1 data cache config, i.e., {<config>|none}	-cache:dl1	dl1:256:256:4:l
l1 data cache hit latency (in cycles)	-cache:dl1lat	1
l2 data cache config, i.e., {<config>|none}	-cache:dl2	none
l1 inst cache config, i.e., {<config>|dl1|dl2|none}	-cache:il1	il1:128:128:2:l
l1 instruction cache hit latency (in cycles)	-cache:il1lat	1
l2 instruction cache config, i.e., {<config>|dl2|none}	-cache:il2	none
flush caches on system calls	-cache:flush	false
convert 64-bit inst addresses to 32-bit inst equivalents	-cache:icompress	false
memory access latency (<first_chunk> <inter_chunk>)	-mem:lat	18 2
memory access bus width (in bytes)	-mem:width	64
instruction TLB config, i.e., {<config>|none}	-tlb:itlb	itlb:16:4096:4:l
data TLB config, i.e., {<config>|none}	-tlb:dtlb	dtlb:256:4096:2:l
inst/data TLB miss latency (in cycles)	-tlb:lat	30
total number of integer ALU's available	-res:ialu	2
total number of integer multiplier/dividers available	-res:imult	1
total number of memory system ports available (to CPU)	-res:memport	2
total number of floating point ALU's available	-res:fpalu	2
total number of floating point multiplier/dividers available	-res:fpmult	1
ratio of fetch to issue	-fetch:speed	2

Table 4. Simulator Parameters for Power2 Architecture

The parameters are a close approximation of the Power2 architecture. Any parameters that are not shown use their default values. The configuration format for the cache is <cache:sets:linesize:associativity:policy>. The configuration format for the TLB is <tlb:sets:pagesize:associativity:policy>. The caches and the TLBs were configured to match those in the Power2. The parameters were used to simulate the execution of C4.5 on the largest test case to establish a baseline for performance. The cumulative modifications that were studied are:

1) Increasing the number of integer ALUs from two to four.
2) Increasing the instruction fetches and decode to eight instructions per cycle from four and the instruction queue to sixteen from 8.
3) Increasing the number of memory ports from two to four.

We need to study the changes in a cumulative manner since each subsequent change provides increased resources for the previous change. The first change is to increase the number of integer ALUs. It was speculated in previous work [reference to synthesis paper] that the instruction level parallelism is limited due to the address calculations required to access the instance data. The choice was to increase the integer ALUs from two, the number in the Power2, to four. Next the flow of instructions was increased to explore if the performance bottleneck could be the lack of instruction for the functional units. To explore this we need the additional integer ALUs to consume the increased number of available instructions. Finally the number of memory ports is increased to determine if the memory is limiting the number of instructions that are available to the fetch unit.

Results of Simulation of Micro-Architecture Modifications

The modifications were implemented by modifying the configuration file and C4.5 was re-simulated with the new parameters. The first modification of increasing the number of integer ALUs improved the performance over the base case by 2%. Increasing the fetch and decode bandwidth in addition to the number of integer ALUs increased the improvement over the base case by 2.7%. Finally the increase of the number of memory ports with the other two modifications achieved an increase in performance over the base case of 3.3%. The addition of a significant amount of hardware resources did not provide a substantial increase in performance for C4.5. This may be due to a limited amount of instruction level parallelism in the code.

Given that we suspect that the performance is being limited by the code we are running, modifications to the source level code may result in additional parallelism being available for the hardware. In previous work [10], several modifications to a well-used routine in C4.5 were explored. The alternative implementations yielded no better results than the original code but since we have now modified the micro-architecture these changes may improve the performance. The piece of code from the subroutine ComputeFrequencies that was studied is:

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

This code walks through the instances totaling the number of classes for each of the possible values of the chosen attribute. It accounts for 7.8% of the total runtime of C4.5 on the largest test case. The modified version of the code is:

```
p = FP;
/* start up code for odd no. of elements */
if ( ((LP – p) % 2) != 1) 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];
}
```

The code was modified to step through the instance array and process two elements at a time. There is a potential resource conflict if both instances write to the same location in Freq but the hardware will insure that the operations are correctly executed. By modifying the main loop the hope is that the instruction level parallelism is increased. The goal is to reduce the execution time by increasing the amount of work that can be done each cycle. All of the micro-architecture modifications were rerun using the modified version of C4.5.

Simulation on the base configuration that closely matches the Power2 showed an increase in performance of .8% over the base case. Again we will be using the performance of the unmodified version of C4.5 on the Power2 configuration as the base case. Adding two additional integer ALUs for the first modification resulted in a 2.5% increase over the base case. Increasing the fetch and decode rate in the second modification produces a 3.3% increase in performance over the base case. Finally increasing the number of memory ports produces an increase in performance of 3.9%.

With all of the micro-architecture modifications and the modification of C4.5, a performance increase of 3.9% was achieved. If it is assumed that the time spent executing phases of C4.5 in the simulator is similar to the results obtained on the Power2 the increase in performance is much larger. The subroutine that was modified accounts for 7.8% of the execution time of the program on the Power2. For the case where the program is modified and all the micro-architecture are included, an increase in performance of 50% is achieved in the subroutine. Table 5 summarizes the percentage of performance increase for the modification to C4.5, the enhancements of the micro-architecture, and the loop for the modified version of C4.5.

Version	Original	Modified	Modified C4.5 Loop
Power2	Base	.8	10
Mod1	2	2.5	32
Mod2	2.7	3.3	42
Mod3	3.3	3.9	50

Table 5. Percentage of Performance Increase

For both versions of C4.5 the increase in the number of integer ALUs provided the largest increase in performance. Each incremental modification to the micro-architecture resulted in an incremental increase in performance. Modifications in this study were made to a set of interrelated resources. Each change resulted in additional resources being available to the previous modification. To propose modifications to the Power2 architecture, the implementation cost versus gain of each of these modifications would have to be studied. Also, additional programs would have to be studied to determine whether the performance increases due to the micro-architectural changes occur in a broader set of programs. From the results here, we would recommend that the number of ALUs be increased to four in the Power2.

Conclusions

This paper has explored the use of hardware-based performance monitoring to examine the execution characteristics of C4.5, which is a widely used data mining application. A methodology was developed that addressed controllable and uncontrollable variances. C4.5 was studied using a synthetic benchmark that allowed the scaling of the size of the test cases. The results indicate that the Power2 architecture is well balanced for C4.5. Several micro-architecture changes were explored to study the impact on the performance of C4.5. Simple Scalar, a micro-architecture simulator, was used to study three modifications of the micro-architecture and a modified version of C4.5. The single modification that resulted in the largest increase in performance of the simulated micro-architecture for both versions of C4.5 was the increase in the number of ALUs. Modifications to this resource, which is involved in address calculation, yielded higher performance but this must be viewed with the caveat that the entire program may have benefited from this change. Finally the results show the complex relationship between the program, compiler and micro-architecture.

References

1. 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.
2. E. H. Welbon, C. C. Chan-Nui, D. J. Shippy, and D. A. Hicks, "The POWER2 performance monitor", *IBM Journal of Research and Development*, 38(5), pp. 545-554, 1994.
3. Jussi Mäki. “POWER2 Hardware Performance Monitor Tools”, *http://www.csc.fi/~jmaki/rs2hpm-paper*, 1995.
4. Robert Bergeron, “Measurement of a Scientific Workload using the IBM Hardware Performance Monitor”, *Proceedings of Supercomputing ’98*, 1998.
5. J. Ross Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1992.

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. M. Rosenblum, S. Herrod, E. Witchel, A. Gupta, "Complete Computer Simulation: The SimOS Approach", *IEEE Parallel and Distributed Technology*, Winter 1995.
8. T. Keller, A. M. Maynard, R. Simpson and P. Bohrer, "SimOS-PPC full system simulator", *http://www.cs.umtexas.edu/users/cart/SimOS*.
9. D.C. Burger, T. M. Austin, and S. Bennett. "Evaluating Future Microprocessors-the SimpleScalar Tool Set", *UW Computer Sciences Technical Report #1308*, University of Wisconsin, July 1996.
10. M. Thoennes, "Interaction of a Superscalar Architecture and a Data Mining Application", *IBM Research Report RC22198*, IBM TJ Watson Research Center, Yorktown Heights, NY, September 2001.