How Fast is '-fast'?
Performance Analysis of KDD Applications using Hardware Performance Counters on UltraSPARC-III

Adam Czezowski
CAP Research Group
Department of Computer Science
Australian National University
Canberra ACT 0200, Australia
Adam.Czezowski@anu.edu.au

Peter Christen
CAP Research Group /
ANU Data Mining Group
Department of Computer Science
Australian National University
Canberra ACT 0200, Australia
Peter.Christen@anu.edu.au

ABSTRACT

Modern processors and computer systems are designed to be efficient and achieve high performance with applications that have regular memory access patterns. For example, dense linear algebra routines can be implemented to achieve near peak performance. While such routines have traditionally formed the core of many scientific and engineering applications, commercial workloads like database and web servers, or decision support systems (data warehouses and data mining) are one of the fastest growing market segments on high-performance computing platforms. Many of these commercial applications are characterised by more complex codes and irregular memory access patterns, which often result in a decrease of performance that is achieved. Due to their complexity and the lack of source code, performance analysis of commercial applications is not an easy task. Hardware performance counters allow detailed analysis of program behaviour, like number of instructions of various types, memory and cache access, hit and miss rates, or branch mispredictions. In this paper we describe experiments and present results conducted with various KDD applications on an UltraSPARC-III platform, and we compare these applications with an optimised dense matrix-matrix multiplication. We focus on compiler optimisations using the `-fast` flag and discuss differences in un-optimised and optimised codes.

Keywords

Data mining, performance analysis, compiler optimisation, UltraSPARC-III, C4.5, Apriori.

1. INTRODUCTION

Commercial applications like database and web servers, or decision support systems (data warehouses and data mining) represent one of the most rapidly growing segments in the high-performance computing market. Modern processors and memory systems are designed to be efficient and achieve high performance with applications that have regular memory access patterns (like dense linear algebra software). They often perform purely when running commercial applications. Complex codes, irregular memory access patterns, dynamic data structures and dynamic memory allocation schemes characterise such applications. This paper aims to analyse the characteristics of such commercial applications, especially to give an insight into their cache and memory access behaviour.

A rapidly growing segment of commercial applications is data mining or KDD (Knowledge Discovery in Databases) [14], which deals with the analysis of large and complex data sets. KDD combines techniques from machine learning, statistics, databases and high-performance computing. Tasks involved are data cleaning and pre-processing, data exploration, clustering, predictive modelling, association rules generation, decision tree induction, and others. Three common characteristics of these applications are (1) they operate on large data sets, (2) they are compute and memory intensive, and (3) they involve irregular memory access patterns which is due to their dynamic and often recursive data structures (like hash tables and trees, index or linked lists). The first two characteristics make them attractive for implementation on high-performance platforms, specially for shared memory multiprocessors (SMPs), where all CPUs have access to the same memory system, while the last characteristic is an obstacle for efficient system utilisation and high performance. Traditional compiler optimisation techniques (based on arrays and data locality), which proved to be successful for many scientific and engineering applications, can only be applied with limited success to such data structures.

Hardware performance counters are an easy to use instrument and they can provide detailed analysis of application performance behaviour at instruction level. Such counters are available on most modern microprocessors, including *UltraSPARC*, *Pentium* and *Alpha*. They can count various events, including different types of instructions (loads, stores, branches or floating-point operations), cache hits and misses, TLB[1] misses, branch mispredictions, cycles and instructions completed, and others. Machine and operating-system dependent libraries (like the *Solaris* `libcpc` [12]) provide access to hardware counters on a specific platform and operating system. Platform independent counter libraries are currently under development in various research

[1] Translation Lookaside Buffer

projects. Two such libraries are the *Performance Counter Library (PCL)* [4] and the *Performance Application Programming Interface (PAPI)* [7]. Their aim is to provide a set of platform independent counters, that allow easy portability of programs instrumented with these libraries, and to allow inter-platform performance comparisons. As we currently restrict our research to a *Solaris/UltraSPARC* platform, we use the `libcpc` [12]. Although we have made initial experiments with *PAPI* we have turned to the `libcpc` as it was the only library available to support *UltraSPARC-III* counters at the time.

In the next section we present the four applications and the data sets chosen for our experiments, and in Section 3 we discuss our experimental setup, the *Solaris/UltraSPARC* platform used and give an example of how source code can be instrumented with calls to hardware performance counter libraries. Results are then presented in Section 4 and conclusions are given in Section 5. Related work in the area of performance analysis of KDD and other commercial applications is presented in Section 6, and finally we give an outlook on future plans in Section 7.

2. APPLICATIONS AND DATA SETS

We choose three KDD applications and one vendor optimised linear algebra code for hardware counter performance analysis using the `libcpc` [12] library on a *Solaris/UltraSPARC* platform. We only counted events in the core computation routines, i.e. without file in- or output. The KDD programs we analysed use mainly input from text files (which is generally slow). Commercial versions of such programs would either read data from binary files or access them directly from a database server. We now present the analysed programs and subsequently discuss the data set we used.

2.1 Decision Tree Induction – C4.5

The freely available popular decision tree induction program *C4.5*[2] [19] was chosen as a typical KDD application. This program has already been analysed in its memory access behaviour using a machine simulator [5; 6].

C4.5 is written in ANSI C, it reads data from text files and then builds a decision tree. Only the tree building routine (i.e. the function `BestTree()`) was analysed, with the complete primary data set (the table loaded from disk) stored in main memory. This data is stored in an array with pointers to vectors, with each vector (one data record) having a length corresponding to the number of attributes. Every element in this vector consists of a `short` and a `float` variable. In the case of a categorical type attribute, the category number is stored as a `short`, while a continuous attribute (a real number) is stored as a `float`. Thus one of the two variables is always unused. The decision tree is a complex recursive data structure that is built dynamically in the tree building routine.

2.2 Association Rule Induction – APRIORI

Mining association rules is a popular data mining algorithm. It is for example used to analyse market basket data to find frequent item sets and extract rules like *'if a customer buys milk then she will most likely also buy cheese.'* For our performance analysis we use a freely available implementation of the *APRIORI*[3] algorithm [2]. An older version of this program is incorporated in the data mining tool *Clementine 5.0*. The program is written in C, it reads a text file with transactional data and writes the resulting rules either into a text output file or displays them. The items in the input transactions are stored in a vector data structure as integer numbers. Once all data is loaded, the items are sorted with descending frequency, and a prefix tree is built, which is then modified and updated for item sets of increasing length. Besides pointers to parent and (variable number of) child nodes, the prefix tree contains a counter vector, stored as integer numbers. Once all frequent item sets are found, they are sorted and the extracted rules are displayed or saved. The code analysed includes the sorting and recoding of the items, the creation of the item set tree, the checking of the subset size, and finally the sorting of the transactions (in order to find the frequent item sets).

2.3 Additive Models – ADDFIT

The *ADDFIT* algorithm [9] was developed by the ANU Data Mining group and implemented sequentially and on distributed memory platforms (using C/MPI) [10]. This algorithm builds an additive model of the data by assembling a dense symmetric linear system in a first step, which is then solved in a second step using either a sequential or parallel solver [10; 21]. For the performance analysis we are only interested in the first step, as it involves reading the data from (binary) files once and assembling each data record into a matrix and vector at data dependent locations. The primary data structure (the table from disk) is loaded first and then the assembly is started. Only the assembly routine is analysed using hardware performance counters. The data dependent assembly results in irregular memory access patterns. For each continuous attribute in a data record four non-zero values are added into the matrix, while for a categorical attribute only one value is added. The locations where these values are added is data dependent and can be anywhere in the matrix. As the assembled linear system is symmetric, only a dense upper triangular matrix with a corresponding vector is allocated, whereby each entry is a `double` sized floating-point value. The size of this linear system is determined by the number of categories for categorical attributes, and the resolution of the model for continuous attributes, but it is completely independent from the size (i.e. number of records) in the primary input data set.

2.4 Dense Matrix-Matrix Multiplication – BLAS (SUNPERF)

To allow a comparison with a platform optimised application with regular memory access patterns we also instrumented a dense matrix-matrix multiplication (the *BLAS* routine `dgemm()` as implemented in Sun's *SUNPERF* library) with calls to the `libcpc` hardware counter library. The `dgemm()` routine is typically used in the core of various scientific and engineering applications. For the performance analysis two *Hilbert* matrices were created and dense matrix-matrix multiplications of two such matrices where performed and analysed.

[2] `http://www.cse.unsw.edu.au/~quinlan/`

[3] `http://fuzzy.cs.uni-magdeburg.de/~borgelt/`

Characteristic	Level-1 Instruction	Level-1 Data	Level-2 Unified	D-TLB	I-TLB
Size	32 KB	64 KB	8192 KB	512x8 KB	128x8 KB
Associativity	4-way	4-way	1-way	2-way	2-way
Line length	32 Bytes	32 Bytes	512 Bytes	–	–
Latency (cycles)	2	2	15	–	–
Miss cost (cycles)	15	15	75	–	–
Write policy	Write-invalidate	Write-through	Write-back	–	–

Table 1: UltraSPARC-III cache, D-TLB and I-TLB characteristics.

2.5 Data Sets and Data Structures

For *C4.5* and *ADDFIT* we used the *Census-Income* data set which is freely available from the *UCI KDD Archive*[4]. This data consists of a training file which contains 199,523 records and a test set with 99,762 records. For our purpose we concatenated both files into one to get a large enough test data. The *Census-Income* data set contains 5 continuous and 37 categorical attributes.

For *APRIORI* we created synthetic data sets of various size and complexity using a data set generator as described in [2]. For the tests we then choose a smaller data set with 10,000 records and a larger one with one million records.

The *primary* data structures used by the KDD applications hold the input data. They are mostly of arrays or vectors, and their size and dimension usually increases linearly with the size of the input data set. In the case of *ADDFIT*, this data is only used once (i.e. each data record is accessed once), but for *C4.5* and *APRIORI* usually several iterations are needed each accessing the primary data structure.

The size of the *secondary* data structures built by the KDD applications, i.e. the decision tree in *C4.5*, the prefix tree in *APRIORI*, and the dense matrix used by *ADDFIT*, are not directly proportional to the size of the input data. Rather, they are data dependent (e.g. a *C4.5* decision tree) or their size is determined by some parameters (e.g. model resolution in *ADDFIT*). The size of the prefix tree for *APRIORI* depends both on the data as well as on parameters like *support* and *confidence*, which have to be set by the user. It is therefore often very hard to specify the amount of memory some KDD applications will use.

3. EXPERIMENTAL SETUP AND PROCEDURES

A common way to obtain detailed, performance related data at the level of architectural units, i.e. at the level of cache, memory, integer and floating point units, is to use full machine simulators. Although indispensable to evaluate new alternative architectural designs, full machine simulators are slow and often provide a simplified view of a real architecture [11]. Another, more convenient way to gather such data is to use specialised registers called *hardware performance counters* or simply *hardware counters*. Today, nearly all general purpose microprocessors on the market have such counters and provide user level interfaces via specialised libraries. In the next few sections we describe our hardware platform and give some more insight into how to use hardware counters.

3.1 Hardware Platform

The presented experiments throughout this report were conducted on a *Sun-Blade-1000* workstation with two *UltraSPARC-III* (US-III) processors running at 750 MHz, and having in total 2 GBytes of main memory. The summary of characteristics for both Level-1 Data and Instruction caches as well as the unified Level-2 cache are presented in Table 1. Cache latencies where established experimentally using the *Calibrator*[5] tool and averaged as their exact value depends on the SDRAM model [22], prefetch cache operation and the instruction mix. We do not provide TLB latencies in Table 1 as in the US-III processor the D-TLB is an integral part of the data cache unit (DCU) and the I-TLB and the instruction cache are part of the instruction issue unit (IIU). Thus, both TLBs latencies are hidden by Level-1 cache latencies. Because of the complexity of *software-managed* TLBs only a range of the miss costs can be given. It lies somewhere between one hundred and a few hundred cycles. At this stage we will not go into the analysis of the *UltraSPARC-III* prefetch and write caches behaviour. However, we acknowledge that such an analysis will assist in formulating a more accurate representation of the quite sophisticated *UltraSPARC-III* on-chip memory system.

The test workstation is running *SunOS Version 5.8* with *Forte Developer 7 Compiler* and *Development Tools Collection*. We have chosen a *Sun-Blade-1000* workstation as test platform because of its modern *UltraSPARC-III* microprocessor and wide range of events that can be monitored using performance counters. In the future we would like to undertake similar study on *Intel Pentium* and other processors.

3.2 Performance Counters

All run time measurements of the various hardware events were obtained by using *UltraSPARC-III* hardware performance counters. Two specialised processor registers are used for this purpose. The *Performance Control Register (PCR)* controls which event and which mode (user, system or both) of operation is selected for a pair of 32-bit *Performance Instrumentation Counters (PICs)*. There are over seventy possible events that can be counted, logically gathered into the following groups: instruction execution rates, integer unit statistics and stall counts, pipeline R-stage stall counts, recirculate counts, memory access statistics, system interface, floating point operation and memory controller statistics. To name just a few typical events: both Level-1 and Level-2 cache references and misses, D-TLB (Data Translation Lookaside Buffer) and I-TLB (Instruction Translation Lookaside Buffer) misses, pipeline stalls, branch mispredictions, floating point addition and multiplication pipe com-

[4] `http://kdd.ics.uci.edu/`

[5] `http://www.cwi.nl/~manegold/Calibrator/`

```
/*
  The example code measures instruction and cycle counts while transposing a matrix.

  Compilation and execution:

  sun% cc -xO2 -o simple_cpc simple_cpc.c -lcpc
  sun% simple_cpc
  pic0=Cycle_cnt,pic1=Instr_cnt,sys,nouser: 6114336263, 3157421639 cpi=1.94
  sun%
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <inttypes.h>
#include <libcpc.h>

void transpose512(){
    static int  x[512][512];
    static int  y[512][512];

    ...
}

void main() {
    char*               eventnames = "pic0=Cycle_cnt,pic1=Instr_cnt,sys,nouser";
    int                 cpuver;
    cpc_event_t         event, before, after;
    unsigned long long  pic0, pic1;

    /* Get processor version and check if hardware counters are available */
    if ((cpuver = cpc_getcpuver()) == -1) exit(EXIT_FAILURE);

    /* Translate event string into data structure */
    if (cpc_strtoevent(cpuver, eventnames, &event) != 0) exit(EXIT_FAILURE);

    /* Bind events to process */
    if (cpc_bind_event(&event, 0) == -1) exit(EXIT_FAILURE);

    (void) cpc_count_usr_events(1);  /* Enable user mode counting */
    (void) cpc_count_sys_events(1);  /* Enable system mode counting */

    /* Sample counters to get start values */
    if (cpc_take_sample(&before) == -1) exit(EXIT_FAILURE);

    transpose512();  /* Do some computations... */

    /* Sample counters to get stop values */
    if (cpc_take_sample(&after) == -1) exit(EXIT_FAILURE);

    (void) cpc_count_usr_events(0);  /* Disable user mode counting */
    (void) cpc_count_sys_events(0);  /* Disable system mode counting */

    pic0 = after.ce_pic[0] - before.ce_pic[0];  /* Get user count value */
    pic1 = after.ce_pic[1] - before.ce_pic[1];  /* Get system count value */

    printf("%s: %lld, %lld cpi=%f\n", eventnames, pic0, pic1,(float)pic0/(float)pic1);
}
```

Figure 1: Simplified example code augmented with calls to Sun Performance Library `libcpc`.

Option	Comment
-fns	Disable gradual underflow for improved performance.
-fsimple=2	Enable non-IEEE 754 standard, aggressive floating-point operations that may lead to different numeric results.
-fsingle	Use single-precision arithmetic.
-ftrap=%none	Disable the floating-point traps.
-xalias_level=basic	Pointers to different C basic data types do not alias each other. References using char pointers may alias any other type.
-xarch=v8plusb	Generate 32-bit subset of SPARC-V9 ISA including the VIS 1.0 and US III extensions. Runs only on US III.
-xbuiltin=%all	Inline or substitute intrinsic functions for system functions.
-xcache=64/32/4:8192/512/1	Definition of cache properties passed to the compiler.
-xchip=ultra3	Specifies target processor to ultra3.
-xdepend	Loop dependency analysis and optimisation.
-xlibmil	Inline selected libm math routines for optimization.
-xmemalign=8s	Assume 8-byte data alignment.
-xO5	Set optimisation to the highest possible level.
-xprefetch=auto,explicit	Enable automatic generation of prefetch instructions.
-xvector=no	Disable vectorised mathematical library functions.
-depend	Perform dependency analysis to optimise loops.

Table 2: Expansion of `-fast` macro for *Forte Developer 7 C 5.4 Sun* compiler release.

pletions and many others. Events can be counted in user and/or system mode.

Access to these counters is obtained via function calls to the `libcpc` and `libpctx` [12] libraries which have to be embedded into the application's source code. The main feature of these libraries is the very low overhead. For example the counter start and stop calls have overheads of less than 3,000 cycles as measured on a *Sun-Blade-1000*. In parallel with performance counter libraries, *SunOS 5.8* provides two monitoring tools: *cputrack* and *cpustat* that display counter statistics. While *cputrack* gathers such statistics for a particular process, *cpustat* provides system-wide statistics hence requires super user privileges.

An example of a simple code which was augmented by calls to the performance counter library is presented in Figure 1. In this code there are the following function calls to the `libcpc` library: `cpc_getcpuver()` is used to determine if the processor provides any performance counters. It returns an abstract description of the processor which is used by every other function in `libcpc`. The function `cpc_strtoevent()` translates the event specification from a string representation into the appropriate set of control bits for the processor and stores them in a `cpc_event_t` structure. Calls to `cpc_bind_event()` bind the selected two events to the current *light-weight process (LWP)*. The counters can then be sampled at any time by using `cpc_take_sample()` and reading the `ce_pic[]` fields of the `cpc_event_t` structure.

3.3 Methods

Each experiment was performed on a dedicated CPU wholly reserved to the test application. This isolation of the code being measured from the activity of the other processes being run on the system ensured good reproducibility of the results particularly in the system mode.

The `-fast` option in the *C Forte Developer 7 Compiler Suite* is most likely to be used by many programmers as a starting point for compiler optimisations. We wanted to see and measure the effects of this option on the four applications to be evaluated. This option is in fact a macro which combines many different optimisations that benefit in a wide range of programs [12]. We can not explain in detail every option used but we have expanded the `-fast` macro in Table 2 with comments. It is worth to note that the content of the `-fast` macro depends on the release of the *Sun* compiler. The one shown here is for the *Forte Developer 7 C 5.4* release. In the un-optimised or default mode the compiler assumed a generic *UltraSPARC* architecture and 4-byte data alignment.

We need to stress that our goal is not to perform optimisations of the selected applications, merely to comment on their performance in two situations, i.e. when compiled with and without the `-fast` optimisation flag.

All tests were run over weekends or night times when the machine was otherwise idle. Table 3 shows the characteristics of the test programs and data sets we used. For the three KDD applications we choose a smaller and a larger data set each that resulted in a heap size (i.e. size of the data structure in main memory) between 4 and 19 MBytes and between 62 and 100 MBytes, respectively. For our comparison application *BLAS* we added a small problem size which resulted in a 1 MByte heap size. All results for a given program/data set pair in Table 3 are averaged over the number of iterations listed, and both run times and user code percentages are given for the codes compiled without optimisation.

4. RESULTS

We performed experiments with the four selected applications *ADDFIT*, *APRIORI*, *C4.5* and *BLAS* both with the `-fast` optimisation turned on and off, and we report them here as *optimised* and *un-optimised*. As can be seen from Table 3, all of the applications – with the exception of *C4.5* with the large data set – do spend more than 90% of their time in user space. We therefore only report results in user space. The measurements in the system (or kernel) space were performed and will be reported in the near future. The

Program	BLAS (SUNPERF)			ADDFIT	
	small	medium	large	small	large
Data	209 × 209 matrices	660 × 660 matrices	2090 × 2090 matrices	Census with 10,4858 records	Census with 209,715 records
Run time	0.03 sec	1.10 sec	44.03 sec	1.09 sec	5.89 sec
Iterations	100	10	1	10	10
Heap size	1 MB	10 MB	100 MB	10,024 KB	90,408 KB
User code	99.46%	97.09%	93.03%	99.64%	96.36%

Program	APRIORI		C4.5	
	small	large	small	large
Data	T5I4D10K with 10,000 records	T10I8D1000K with 1,000,000 records	Census with 8,322 records	Census with 266,305 records
Run time	3.36 sec	31.78 sec	2.35 sec	421.04 sec
Iterations	10	1	5	1
Heap size	19,776 KB	70,512 KB	3,960 KB	62,152 KB
User code	89.37%	94.30%	98.43%	75.93%

Table 3: Program and test characteristics.

Program		BLAS (SUNPERF)			ADDFIT		APRIORI		C4.5	
		small	medium	large	small	large	small	large	small	large
Loads	Un-optimised	23.1	23.1	23.2	38.2	38.7	28.1	27.0	34.5	33.7
	Optimised	23.1	23.1	23.2	30.2	31.9	21.5	21.3	30.0	24.0
	Change	**0**	**0**	**0**	**-21**	**-18**	**-24**	**-21**	**-13**	**-29**
Stores	Un-optimised	1.3	1.0	1.0	12.5	13.4	9.5	14.0	8.4	8.5
	Optimised	1.3	1.0	1.0	11.2	11.6	6.7	9.3	9.7	5.1
	Change	**0**	**0**	**0**	**-11**	**-13**	**-29**	**-34**	**+16**	**-39**
Branches	Un-optimised	4.2	4.1	4.1	6.7	5.6	15.2	13.1	7.2	10.2
	Optimised	4.2	4.1	4.1	9.8	8.4	19.7	22.8	11.6	20.7
	Change	**0**	**0**	**0**	**+46**	**+49**	**+30**	**+74**	**+60**	**+103**
FP ops	Un-optimised	58.7	59.1	59.1	3.5	3.8	0.02	0.0	8.4	7.7
	Optimised	58.7	59.1	59.1	5.6	7.7	0.04	0.0	14.7	16.1
	Change	**0**	**0**	**0**	**+62**	**+104**	**+93**	**+91**	**+76**	**+110**
Others	Un-optimised	12.7	12.7	12.6	39.1	38.5	47.2	45.9	41.5	39.9
	Optimised	12.7	12.7	12.6	43.2	40.4	52.1	46.6	34.0	34.1
	Change	**0**	**0**	**0**	**+10**	**+5**	**+10**	**+2**	**-18**	**-15**

Table 4: Percentile values of loads, stores, branches, floating point operations and other instructions present in optimised and unoptimised codes. The relative change of instruction mix between optimised and unoptimised codes, also expressed in percentages, is given in bold fonts.

use of the `-fast` optimisation moderately increased the percentage of time spent in system space.

4.1 Instruction Mix

Ideally, an execution time breakdown into computation, memory stalls, branch misprediction and resource stalls would be desirable. Due to interdependencies in the measured hardware events, a reliable and meaningful breakdown into these components is impossible using the data we have collected (even assuming average latencies per counted event would be grossly inaccurate). However, we were able to count the number of different instructions types in an application, and Table 4 shows the percentage values of loads, stores, branches, floating-point operations and all other (mainly integer operations) instructions in user mode. The numbers are given for both the un-optimised and the optimised compilations, and the corresponding percentage change (positive means an increase, negative a decrease).

No changes can be seen for the *BLAS* test runs because they are basically a simple call to a *SUNPERF* library routine (which is part of the operating system) and thus not affected by compilation options. For the three KDD applications we can clearly see that the ratio of loads and stores is generally reduced (one exception is *C4.5* with the small data set which has a noticeable increase in store instructions), while the percentages of all other instruction types are mostly increased. This can be explained due to compiler optimisations, like using register variables for loop counters and other frequently used variables, which reduces the number of loads and stores to access these variables.

The three main differences between the *BLAS* application and the KDD applications is the larger percentage of stores in KDD applications, their very small number of floating-point operations compared to the dense matrix-matrix multiplication, and finally the much larger percentage of all other instructions, which are mainly integer computations,

Program	BLAS (SUNPERF)			ADDFIT		APRIORI		C4.5	
	small	medium	large	small	large	small	large	small	large
Un-optimised	30.15	9.39	2.44	9.05	15.56	6.45	2.29	1.67	0.19
Optimised	30.13	9.38	2.44	19.03	35.09	10.26	3.76	2.51	0.22
Improvement	0%	0%	0%	53%	56%	37%	39%	33%	14%

Table 5: Ratio of allocated memory per execution time (MB/sec).

for KDD programs. This larger amount of input and output results in a higher load on the cache and memory system, but also in almost unused floating-point units. One possible improvement for KDD applications might be a re-design of algorithms to increase their use of floating-point operations.

4.2 Memory Allocation

As the memory imprint has a significant impact on application performance we have tested each of the applications with different data sets as indicated in Section 2.5.

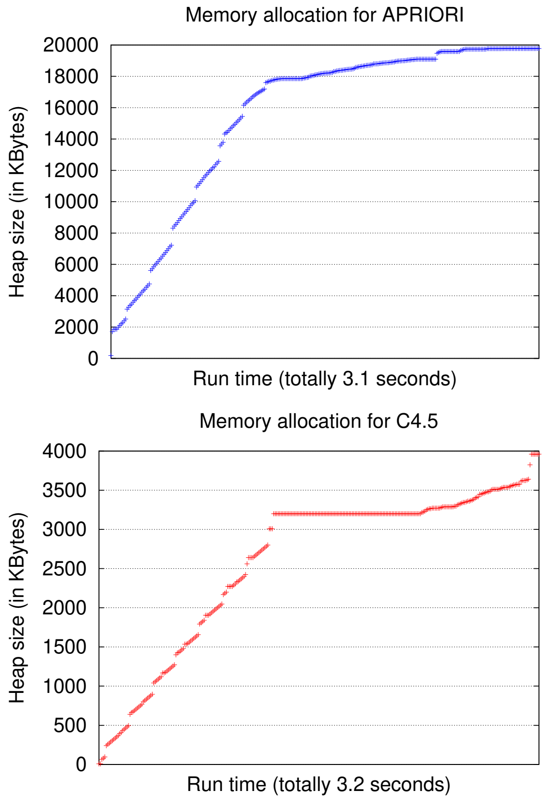

Figure 2: Dynamic memory allocation for APRIORI and C4.5

The maximal allocated memory for data (the heap size) is listed in Table 3. While *BLAS* and *ADDFIT* allocate all the memory they need (basically the dense matrices plus some buffers for input and temporary data) at the beginning, both *APRIORI* and *C4.5* dynamically allocate smaller memory blocks at run time. Figure 2 shows the heap sizes as measured with the Unix command `pmap` (and a small Python script filtering the output) with the smaller data sets for both *C4.5* and *APRIORI*. Two phases are clearly distinguishable, the first being the loading of the input data (steeper slope of the graphs), while the second being the computation of the decision tree or the frequent item sets (prefix tree), respectively. Such dynamic memory allocation and re-allocation results in many system calls and also prevents good data locality.

It is interesting to see the ratio of number of megabytes allocated by each application per execution time measured in seconds, as displayed in Table 5. This indicates the overall capacity of a particular application to process the given amount of data. Of course this "megabyte throughput" depends on the complexity of the operations performed on a given data set but it gives an idea on relative performance of the tested applications.

The improvement factors between the un-optimised and optimised codes are similar to those presented in Table 6, which were based on execution times only. We see rather poor performance of *C4.5* and most of the applications with larger data sets. The only exception from this is *ADDFIT*. For the large data set it achieves 35 MB/sec "megabyte throughput", larger than for the smaller data set which was 19 MB/sec. This is explained by the fact that the size of the linear system being assembled by *ADDFIT* is determined by the number of categories for categorical and the resolution of the model for continuous attributes. Since these attributes didn't change as we went from the small to the large data set the overall throughput has increased.

4.3 Overall Performance

The reduction in run times between un-optimised compilation and using the `-fast` compiler optimisation is listed in Table 6. As can be seen all KDD applications gain between around 10% and up to 50% with *ADDFIT*'s run time almost reduced to half. Optimised compilation does not affect the run time of the *BLAS* dense linear algebra code, because the library used for this application is already compiled and is unaffected by compiler optimisations.

MIPS and MFLOPS are popular measures to show the overall performance mainly for scientific and engineering codes. In Figure 3 one can clearly see that only the *BLAS* dense matrix-matrix multiplication is actually dominated by floating-point operations. The three KDD programs perform mainly integer and other operations (which is consistent with the results from Table 4), which is what one can expect from applications working on strings and integer numbers. It is also interesting to see that for all four applications the MIPS numbers decrease with larger data sizes (except *BLAS* which has a peak performance with a medium sized matrix).

For all tested applications, most of the time is spent in user mode, with around 90% (and more) in most cases. The only exception is *C4.5* with the large data set, in which case the user proportion is reduced to around three quarters of the total run time (see Table 3).

Program	BLAS (SUNPERF)			ADDFIT		APRIORI		C4.5	
	small	medium	large	small	large	small	large	small	large
Un-optimised	0.03	1.10	44.03	1.09	5.89	3.36	31.78	2.35	421.04
Optimised	0.03	1.10	44.03	0.52	2.79	2.23	19.93	1.56	375.50
Improvement	0%	0%	0%	52%	53%	34%	37%	34%	11%

Table 6: Run times (in seconds).

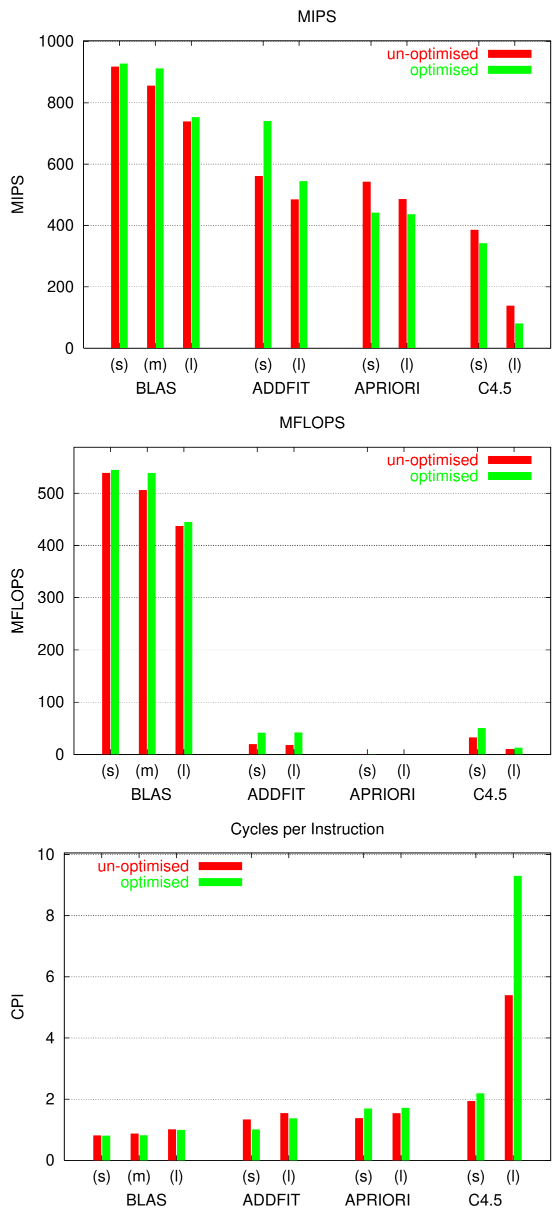

Figure 3: Overall performance indicators: MIPS, MFLOPS and CPI.

While compiler optimisations clearly help in reducing the run times of applications (the most important performance aspect from a user's point of view), for some applications this results in actually a poorer performance. The MIPS rate for both *APRIORI* and *C4.5* is smaller for the optimised code, and their CPI (cycles per instruction) rate is higher. This means that even though the program runs faster, the system utilisation is less and more time is wasted on resource stalls. The question now is where? Because the reduction in run time is larger than the reduction in e.g. load, store or branch instructions, the "density" of these instructions is higher (i.e. the ratio of loads, stores and branches per instruction is higher). This results in higher miss and stall rates in most cases, as can be seen in the figures in the following sections.

4.4 Cache Performance

The cache concept makes sense for applications which exhibit temporal and spatial locality. What about applications with irregular memory access patterns where traditional cache optimisation techniques simply will not work? In this section we describe performance of all four test applications at both the Level-1 and the Level-2 cache as well as the TLB.

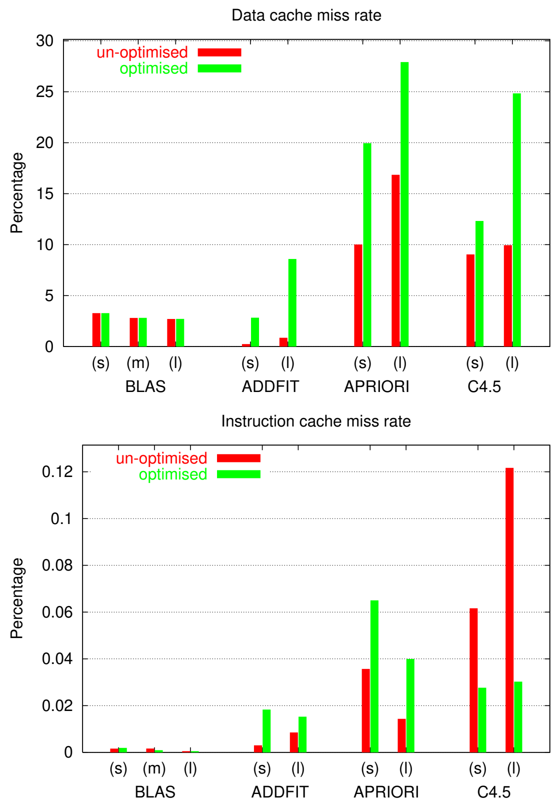

Figure 4: Data and instruction cache miss rates.

4.4.1 Level-1

Figure 4 shows Level-1 cache performance for the data and instruction cache. *BLAS* and *ADDFIT* exhibit small Level-1 cache misses of 3% and around 1% for the small data sets. Both *APRIORI* and *C4.5* have cache misses of 10% and more for the un-optimised versions. This behaviour is typical for irregular memory access patterns. Although, after optimisations, both applications' execution times were reduced the Level-1 cache misses were significantly increased. We are not surprised by good cache performance of *BLAS* for which the `dgemm()` matrix-matrix multiplication routine provided by the *SUNPERF* library is "cache friendly". However, it is unexpected that *ADDFIT* exhibits such good cache utilisation for un-optimised runs. The plausible explanation is that during the assembly stage (which performance we are measuring) the new data added to the dense symmetric matrix mostly are no further than a Level-1 cache size apart so the cache updates are rare. The instruction cache miss ratios are very small which reflect the fact of relative small number of branches and a lack of procedure calls. The increase in instruction cache miss ratios for *APRIORI* and *C4.5* is well correlated with their higher percentage of branch instructions as seen in Table 4.

We know that the main improvement in execution times after the introduction of the `-fast` compiler flag comes from the significant reduction in number of instructions. For *ADDFIT* this amounted to 39% reduction for the small data size (51% for large), for *APRIORI* 48% (46%) and for *C4.5* 42% (52%). On the same Figure 4 we see a dramatic increase in Level-1 cache misses for all applications except *BLAS* for which `-fast` option had no effect. Aggressive optimisations, and as such we should regard those introduced by using the `-fast` option, will lead to complete alteration of application characteristics such as data and instruction flow. It will be incorrect to try to relate new memory performance metrics to the old ones as the optimised (read new) application have nothing to do with the un-optimised (read old) except that both produce the same results.

Having said this, we still should try to understand why optimisations bring such drastic cache performance penalty. This is particularly visible in *ADDFIT* where Level-1 data cache miss ratio rose 8.5 fold for the small and 7.3 fold for the large data set. For *ADDFIT* we see particularly high number of counts for the *instruction queue being empty due to a refetch of a second branch within a fetch group* type of event and also very high number of counts for the *stalls in the store queue due to the store instruction being first in the group*. Although significant for overall performance, these type of events can not cause such Level-1 data cache miss increase. In order to bring light into this we had to resort to the inspection of the disassembled codes of both the un-optimised and optimised *ADDFIT*. Despite of the overwhelming complexity of the code a large number of consecutive double loads greater than the cache line size are prevailing in the optimised code and they are most likely to cause such high incidents of Level-1 cache misses. At this point immediately another question can be raised – why does the compiler introduce such *inefficient* loads distribution? We can only presume that the optimisations based on data flow analysis combined with interprocedural analysis yield higher performance gains than some "on the way inefficiencies" introduced by the compiler.

4.4.2 Level-2

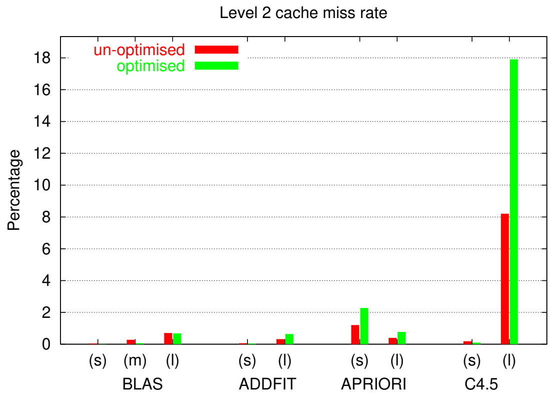

Figure 5: Level-2 cache miss rates.

The results for Level-2 (external) unified cache are presented in Figure 5. The Level-2 cache is large (8 MB), hence the number of misses is relatively low for most of the applications except *C4.5* with the large data set for which the miss rate is 8% for the un-optimised code and extraordinarily high 18% for the optimised version. Inspection of D-TLB misses in Figure 6 also reveal very high miss ratio for *C4.5* with the large data set. It is unlikely that such high miss ratio will be caused by inefficient instruction placement. Most likely this has to be due to the sorting of entire categorical attributes (using recursive quicksort) in *C4.5*, which results in almost no locality for data access. The problem propagates throughout the whole memory hierarchy. In this case we have observed high rate of stalls to memory banks which can possibly be reduced by exploring an alternative sorting method to deal with particularly large data sets.

One of the possible solution for large data sets will be to resign from quicksort and try another specifically designed sorting algorithm, or alternatively use quicksort on a smaller subsets of the categorical attributes. The guide to possible success of such an approach is the performance of *C4.5* for the small data set.

4.4.3 D-TLB

Overall D-TLB (Figure 6) and I-TLB (not shown) miss rates are very small. A problematic D-TLB miss rate only appears with *C4.5* executed with the large data set. Apart from earlier proposed remedy we could try to increase the page size from 8KB to 4MB and see how this single approach will work.

4.4.4 Data Stall

One of the more indicative metrics is data stall rate. It is a measure of the fraction of a CPU cycle which was wasted in waiting for data. This wait fraction is measured for both arriving and departing data from the CPU core. Most often the high percentages in this measure indicate cache misses or TLB misses but may also depend on instruction grouping and scheduling [12].

In Figure 7 we present data stall rates for all four test applications. Both *APRIORI* and *C4.5* show that over 50% of the CPU cycles were used for data coming or departing from the CPU core. This fact well correlates with the high Level-1 data cache misses for both applications.

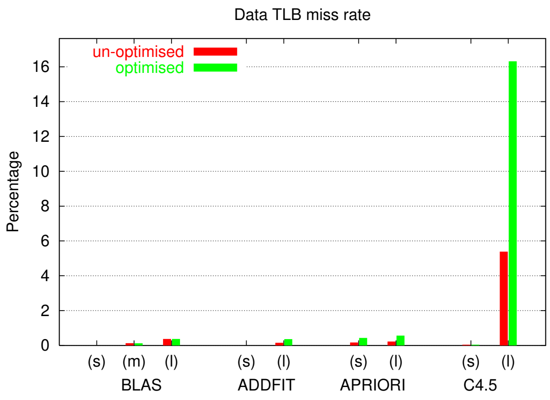

Figure 6: Data TLB miss rates.

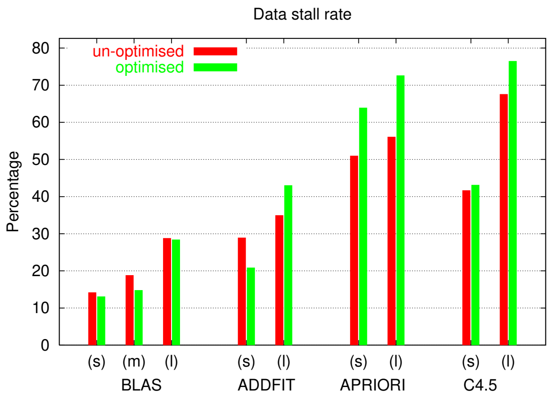

Figure 7: Data Stall rates.

It is interesting to point out that the biggest contribution to the data stall rates comes from the `Rstall_storeQ` event which means that the store queue is full and that the store instruction is the first instruction in a group. This is the case mostly for large data sets.

4.5 Branches

In Section 4.1 about instruction mix (Table 4) we saw that except for *BLAS* the optimised codes displayed increased number of branches as compared with un-optimised codes. The highest figures were for *APRIORI* and *C4.5* with large data sets. The increased frequency of branches can clearly be seen in the lower plot of Figure 8 (branch rate).

It is surprising or rather revealing how well branch prediction works in modern CPUs. Despite the relatively large increases in number of branches the branch miss rates remained almost unchanged for all tested applications but *APRIORI* for small data set. At this moment it is unclear to us why there is such a discrepancy in branch miss rate for *APRIORI* only.

5. CONCLUSIONS

In general we conclude that the performance counters proved to be a reliable and invaluable source of information about many aspects of applications performance. However, we see that the deep understanding of hardware and the measured code is essential to correctly interpret the results. We also see that smaller sections of code are better candidates for performance evaluation using hardware counters as their impact on hardware components can be easier understood and hence altered to remove performance bottlenecks.

Ways to improve the performance of KDD and similar applications is to try to use algorithms and data structures which follow as close as possible current computer architectural designs, and – if possible – redesign algorithms to better utilise the floating-point capabilities of modern processors. Designers of processors, computer and operating systems, and compilers on the other hand should be aware of the emerging class (and market) of commercial applications that do not have regular memory and I/O access patterns and that are not mainly performing floating-point operations. As such applications become more and more important in the future, there is a need for computing platforms that are optimised for them.

Another, quite separate issue, is the use of compiler flags. With the advancements of optimising compilers various compiler flags, in our opinion, often will have more bearing on performance than hand placed changes in the code. Is is not to say that we should not search for better algorithms. It merely points out the fact that todays compilers "are more aware" of underlaying hardware than the average user and are able to do optimisations on larger and larger sections of source code.

While many people simply compile their programs with a `-O`, `-O2`, `-fast` or a similar option, one question that arises is how well such compiler optimisations work on commercial applications that have dynamic and irregular data struc-

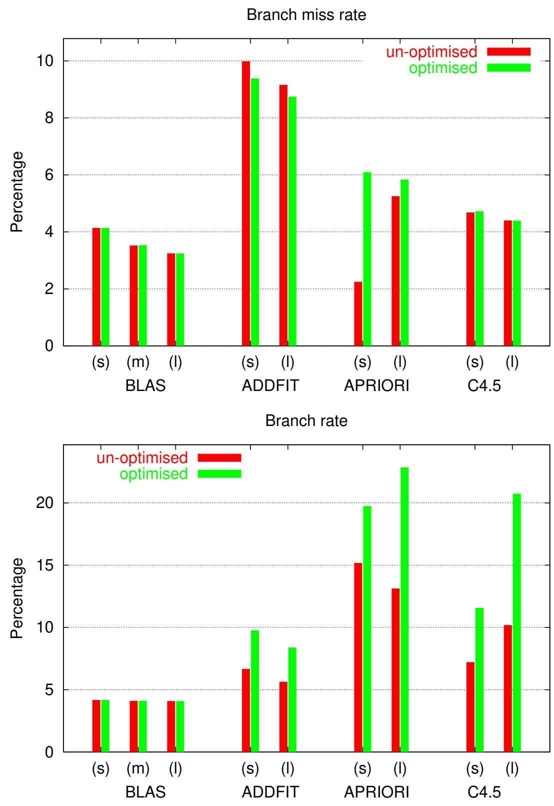

Figure 8: Branch miss rate and branch rate.

tures. Let us quote words from the ultimate reference on Sun compilers [12]:

> "The `-fast` option is a good starting point for compiler optimization of well-behaved programs."

Most compiler options were designed with regular applications in mind. Modern compiler have tens up to hundreds of different flags, that can change the run time of a given application tremendously. Many options are designed to improve floating-point intensive applications, but the question arises how good they help KDD and other commercial applications.

6. RELATED RESEARCH

There is much ongoing research in dedicated KDD and data mining algorithms and in their parallel implementations. In contrast, we are only aware of a small number of publications dealing with the performance analysis of KDD applications [5; 6; 16; 18]. More work has been done on analysing database servers and related commercial applications [1; 3; 15; 23]. To our knowledge, no performance analysis of KDD applications has previously been done using hardware counters.

The memory behaviour of a parallel association rule algorithm is discussed in [18]. The authors looked at custom memory placement scheme and found that simple schemes (like the different hash tree building blocks being allocated in a single memory region) can be quite efficient – improving the execution time for some data sets up to a factor of two. They state that the data structures used by association rules algorithms (hash trees and lists) exhibit poor locality, and the arbitrary allocation of memory makes it difficult to detect and eliminate false sharing. A run time memory allocation library based on the Unix `malloc()` library is presented, which allows customised memory allocation.

Memory characteristics of a parallel implementation of the self-organising map (SOM) neural network model is discussed in [16]. Four characteristics were examined and compared. First, the working set size (temporal locality), second the spatial locality and memory block utilisation, third the communication characteristics and scalability, and fourth the TLB performance. The authors use a simulation tool adapted from the *Augmint* toolkit. They conclude that the size of the working set is not sensitive to the number of input records.

In [5; 6] the popular decision tree induction algorithm *C4.5* is analysed in its memory and parallelisation characteristics. The authors are using *RSIM* [17] simulating three different instruction level parallelism (ILP) processors. One of their conclusions is that such an algorithm is limited by the memory latency and bandwidth, and cache size has a significant effect on performance as well. In [6] a parallel version of *C4.5* optimised for a *ccNUMA* is presented and analysed. This parallel version puts significantly less pressure on the memory hierarchy, and has a larger working set.

The memory system characteristics of some commercial workloads is studied in [3]. The authors present detailed performance studies of three different important classes of workloads: Online transaction processing (OLTP), decision support systems (DSS) and Web index search. They use monitoring experiments and *SimOS* [20] to study the effects of architectural variations. One of their findings is that operating system activity and I/O latencies do not dominate the behaviour of well-tuned database workloads. For OLTP a large off-chip cache is in favour, while DSS and the Web index search are primarily sensitive to the size and latency of on-chip caches.

A performance analysis of the *TPC-C* benchmark on a four-processor Pentium based SMP is presented in [23]. The authors analytically model the performance and then validate their results with simulations (using *SimOS*) and hardware counter experiments. They conclude that experimentally based evaluation of complex commercial applications is time consuming, and that analytical modelling is a feasible alternative.

The authors of [8] discuss methods to make pointer-based data structures cache conscious. They present three different methods. First, clustering (packing data structures that a program is likely to access at the same time into a cache block), secondly coloring (place elements in memory such that elements accessed at the same time map to non-conflicting cache regions) and thirdly compression (compressing data structures so that more elements fit into a cache block). They also propose structure splitting into hot and cold parts (with the hot parts being accessed more frequently than cold parts), and show the suitability of these approaches and the resulting performance improvements.

Four commercial database systems are compared in a study [1] on an *Intel Xeon* processor using hardware counters. The authors used memory resident databases and basic operations (simple queries) to identify common trends in the performance and memory access behaviour. One conclusion is that almost half of the time is spent on stalls. Detailed analysis presented show that 90% of the memory stalls are due to second-level cache data misses (while first-level data stalls are not important) and first-level instruction cache misses (while second-level instruction cache misses are not important). These results therefore suggest that database developers should pay more attention to the data placement (layout) in the second-level cache, and also focus on optimising the critical path for the instruction cache.

An earlier study [13] uses traces and simulations from an *IBM Power* architecture to contrast the differences between technical and commercial workloads. Six commercial applications (including TPC benchmarks, file servers, etc.) are compared to eight technical and scientific applications (which included computational chemistry codes, various simulations, etc.). The findings include that commercial applications are often multi-user and contain many processes, and thus have more operating system calls. The branch behaviour is much less predictable (no long loops, but more decision type branches), they also contain less floating-point operations and have different I/O characteristics. Commercial applications also have larger instruction foot-prints compared to technical applications, which means they can profit more from larger (instruction) cache sizes than technical applications. The authors also state that commercial data is not, and cannot be, very cache efficient because data is often private to processes. Process switching also increases the likelihood that cache contents are overwritten by the time a process is re-scheduled after context switching.

A simulator study using SimICS using two database engines is presented in [15]. The authors analyse and then model the size of working-sets for various database queries for decision support systems (thee queries from TPC-D). Their results show that the most performance critical working-sets are small even for large databases and they do not grow with the size of a database. These working-sets are caused by the instructions and private data that are needed to access a single tuple.

7. OUTLOOK

In this paper we presented experiments of analysing KDD applications with hardware counters on an *UltraSPARC-III* platform. To better understand the memory access characteristics of KDD applications further experiments with various data sets and other hardware event counters are needed. Running KDD applications on a machine simulator (e.g. *SPARC Sulima* [11]) will allow us to change machine parameters like cache size and access times, and thus help to find bottlenecks in such applications. We are also planning to port our experimental setup to different processors (e.g. *Fujitsu Primepower SPARC server* or *Intel Pentium*).
The vast number of options and switches present in optimising compilers is overwhelming. Most of them were designed to assist scientific and engineering applications but not KDD and similar commercial applications. Changing a particular option and tracing the effects of such change on the behaviour of the application (using performance counters) will lead to better understanding of the compiler options/application performance relation. It can also bring more light on which non-trivial changes in architecture could benefit KDD type applications. In future research we are therefore planning to conduct systematic tests with KDD applications using various possible compiler optimisations, to see which ones are favourable for such applications.

Acknowledgment

This research is funded by the ANU/Fujitsu CAP program. The authors would like to thank Peter Strazdins for helpful discussions on SPARC architecture and Solaris operating system issues, and Alistair Rendell for his support.

8. REFERENCES

[1] A.G. Ailamaki, D.J. DeWitt, M.D. Hill and D.A. Wood, *DBMSs on modern processors: Where does the time go?* Technical Report 1394, University of Wisconsin, Department of Computer Science, 1999.

[2] R. Agrawal and R. Srikant, *Fast Algorithms for Mining Association Rules*, in Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994.

[3] L. Barroso, K. Gharachorloo and F. Bugnion, *Memory System Characterization of Commercial Workloads*, Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98), 1998.

[4] R. Berrendorf and B. Mohr, *PCL – The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 2.0)*, Research Centre Juelich, Central Institute for Applied Mathematics, September 2000.
`http://www.kfa-juelich.de/zam/PCL/`

[5] J.P. Bradford and J. Fortes, *Performance and Memory-Access Characterization of Data Mining Applications*, Workshop on Workload Characterization, 1998. Workshop held in conjunction with the 31st Annual International Symposium on Microarchitecture.

[6] J.P. Bradford and J. Fortes, *Characterization and Parallelization of Decision Tree Induction*, School of Electrical and Computer Engineering, Purdue University, 1999.

[7] S. Browne, J. Dongarra, N. Garner, K. London and P. Mucci, *A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters*, Proceedings SC'2000, November 2000.
`http://icl.cs.utk.edu/projects/papi/`

[8] T.M. Chilimbi, M.D. Hill and J.R. Larus, *Making Pointer-Based Data Structures Cache Conscious*, IEEE Computer, December 2000.

[9] P. Christen, M. Hegland, O.M. Nielsen, S. Roberts, P.E. Strazdins and I. Altas, *Scalable Parallel Algorithms for Surface Fitting and Data Mining*, Elsevier Journal of Parallel Computing, special issue on Aspects of Parallel Computing for Linear Systems and Associated Problems, September 2001.

[10] P. Christen, O.M. Nielsen, M. Hegland and P.E. Strazdins, *Parallel Data Mining on a Beowulf Cluster*, Accepted by the HPC Asia 2001 Conference, Gold Coast, Queensland, Australia, September 2001.

[11] B. Clarke, A. Czezowski and P. Strazdins, *Implementation Aspects of Sparc V9 Complete Machine Simulator* , to appear in ACSAC-2002, the Australasian Computer Systems Architecture Conference, Melbourne, Australia, January 2002.

[12] R. Garg and I. Sharapov, *Techniques for Optimizing Applications*, SUN Blueprints, Sun Microsystems Press, 2002.

[13] A.M. Grizzaffi Maynard, C.M. Donnelly and B.R. Olszewski, *Contrasting characteristics and cache performance of technical and multi-user commercial workloads*, in Proceedings of the sixth international conference on Architectural support for programming languages and operating systems, San Jose, California, 1994.

[14] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, Morgan Kaufmann, 2000.

[15] M. Karlsson and P. Stenström, *An analytical model of the working-set sizes in decision-support systems*, in Proceedings of the international conference on Measurements and modeling of computer systems, Santa Clara, California, 2000.

[16] J.S. Kim, X. Qin and Y. Hsu, *Memory characterization of a parallel data mining workload*, in Workload Characterization: Methodology and Case Studies. Based on the First Workshop on Workload Characterization. IEEE Comput. Soc, Los Alamitos, CA, USA, 1999.

[17] V. Pai. P. Ranganathan and S. Adve, *RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors*, In Proceedings of the Third Workshop on Computer Architecture Education, February 1997.

[18] S. Parthasarathy, M.J. Zaki and W. Li, *Custom Memory Placement for Parallel Data Mining*, Technical Report 653, University of Rochester, Computer Science Department, 1997.

[19] J.R. Quinlan, *Programs for Machine Learning*, Morgan Kaufmann, 1993.

[20] M. Rosenblum, S.A. Herrod, E. Witchel and A. Gupta, *Complete Computer System Simulation: The SimOS Approach*, IEEE parallel and distributed technology: Systems and applications, vol. 3, no. 4, 1995.

[21] P.E. Strazdins and J.G. Lewis, *An Efficient and Stable Method for Parallel Factorization of Dense Symmetric Indefinite Matrices*, in Proceedings of the 5th International Conference and Exhibition on High-Performance Computing in the Asia-Pacific Region (HPC Asia 2001), Gold Coast, September 2001.

[22] *SPARC JPS1. Implementation Supplement:Sun UltraSPARC III*, May 2001.

[23] X. Zhang, Z. Zhu and X. Du, *Analysis of Commercial Workload on SMP Multiprocessors*, Proceedings of Performance'99, August, 1999.