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

Peter Christen and Adam Czezowski

CAP Research Group

Department of Computer Science, Australian National University

Contact: **peter.christen@anu.edu.au**

Funded by the Fujitsu-ANU CAP project (see **http://cap.anu.edu.au**)

Outline

- Performance of modern computing platforms
- Characteristics of KDD / data mining applications
- Performance analysis
  - Hardware performance counters
- Selected data mining applications
  - Decision tree induction *C4.5*
  - Association rules *APRIORI*
  - Additive models *ADDFIT*
- Experimental results
- Conclusions

Performance of modern computing platforms

- There is an increasing gap between CPU and memory access speed (memory hierarchy)
  Registers → L1 caches → External cache → Main memory
- CPU caches are only useful (efficient) when many data items or instructions can be accessed (and re-used) directly from the cache (locality)
- Hardware and compilers assume regular memory access patterns
  - Regular data structures like matrices and vectors
  - Temporal and spacial locality
- High efficiency and high-performance for many scientific and engineering applications

Characteristics of data mining applications

- Operate on large and complex data sets
  (often access input data several times)
- Are compute and memory intensive
- Operate on dynamic and recursive data structures
  (hash tables, dynamic linked lists, trees, etc.)
- Data structure access is data dependent
  (often irregular and unpredictable)
- Size of data structures is data dependent
  (often not linear scalable with input data)
- Complex core routines
  (large instruction foot-prints)

Performance analysis

- Modern CPUs and computer systems are becoming more and more complex
  - Longer pipelines
  - Multiple functional units and multiple instruction issued
  - Speculative branch predictions
  - Several cache levels
  - Symmetric multiprocessing (SMP)
- Many of today's applications are very complex (multi-user, interactive, many functions and large data sizes)

Understanding program behaviour is important to achieve good efficiency and high performance

Performance analysis methods

- Profiling
  (information about where your program spent its time and which functions called which other functions while it was executing)
- Monitoring system utilisation
  (using commands like: `ps`, `iostat`, `top`, `kstat`, `vmstat`, `cputrack`, `cpustat`, `pmap`, `har`, etc.)
- Simulation
  (possibility to modify hardware parameters)
- Hardware performance counters
  (CPU registers that count hardware events)

Hardware performance counters

- Most modern CPUs have hardware event counter registers
- Possibility to count various hardware events
  (like MIPS, FLOPS, cycles per instructions, address bus utilisation, cache hit and miss rates, etc.)
- Control and access through library calls
  (e.g. `libcpc` on SPARC/Solaris, `PAPI`, `PCL`, etc.)
- Easy to instrument source code
  - Possible to analyse only parts of the code (like the computational core routines)
  - Possible to analyse programs with short run times

Example `libcpc` code on SPARC III

```
#include <libcpc.h>
int           cpc_cpuver;
cpc_event_t   cpc_event, start, stop;
char          *cpc_arg="pic0=cycle_cnt, pic1=instr_cnt";

cpc_cpuver = cpc_getcpuver();                    /* Get CPU version
cpc_strtoevent(cpc_cpuver, cpc_arg, &cpc_event);
cpc_bind_event(&cpc_event, 0);          /* Bind counter to process

cpc_take_sample(&start);

   /* ... add your code to analyse here ... */

cpc_take_sample(&stop);

printf("cycle_cnt: %lld, instr_cnt: %lld\n",
 (stop.ce_pic[0]-start.ce_pic[0]),(stop.ce_pic[1]-start.ce_pic[1])
```

Decision tree induction (C4.5)

- Given a data set with records (e.g. SQL table), where each record has the same attributes
- Build a classification model of the data (classify records into different classes)
- Tree is built using training data (labeled records)
- Primary (input) data structure
  - Array with pointers to vectors
  - Either a floating-point or an integer value
- Secondary data structure
  - Recursive tree
  - Not restricted to binary tree

Association rules (APRIORI)

- Freely available implementation by *C. Borgelt*
- Popular for market basket analysis
- Given a data set with transactions (which can have variable length)
- The task is to (1) find frequent large item sets and then (2) build rules from these item sets
- Primary (input) data structure
  - Vectors of item numbers (integers)
- Secondary data structure
  - Prefix tree and hash tables
  - Counter vector

Additive models (ADDFIT)

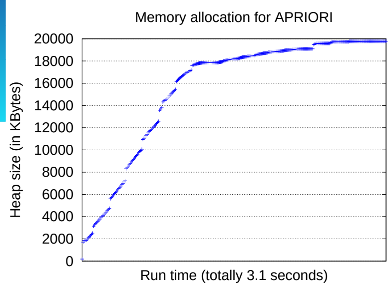
- Developed by the *ANU Data Mining Group* (2000)
- Build a predictive model of the data with additive functions $f(x_1, \ldots, x_d) = f_0 + f_1(x_1) + \ldots + f_d(x_d)$
- Two steps
  1. Assemble dense symmetric linear system from data
  2. Solve linear system sequential or in parallel
- Assembly is data dependent and results in irregular memory access patterns
- Primary data structure (input records) need to be accessed once only
- Secondary data structure is a dense linear system

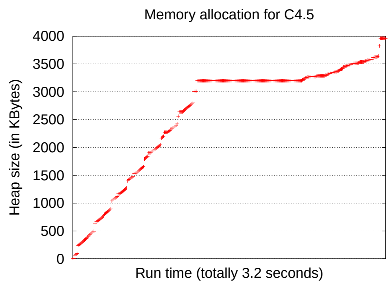
Characteristics of test applications

Program	BLAS (SUNPERF)			ADDFIT	
	small	medium	large	small	large
Data	209 × 209	660 × 660	2090 × 2090	104,858 rec	209,715 rec
Run time	0.003 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	10,000 rec	1,000,000 rec	8,322 rec	266,305 rec
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%

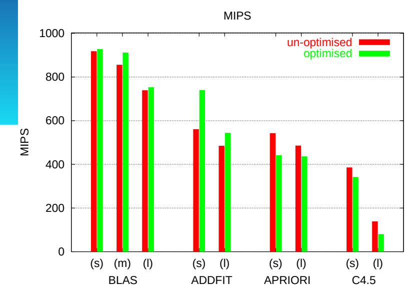
Dynamic memory allocation in APRIORI and C4.5

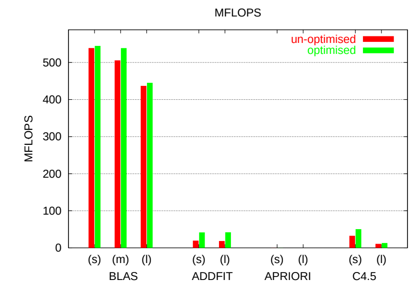




- First phase is loading data from file
- Second phase is computing frequent item sets or decision tree
- ADDFIT (like BLAS matrix-matrix multiplication) allocates all memory in one block at beginning

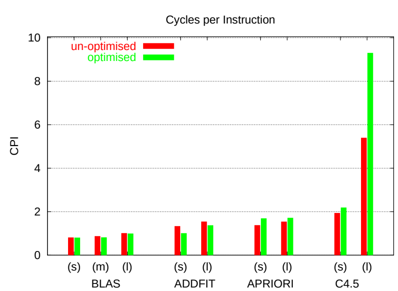
MIPS and MFLOPS





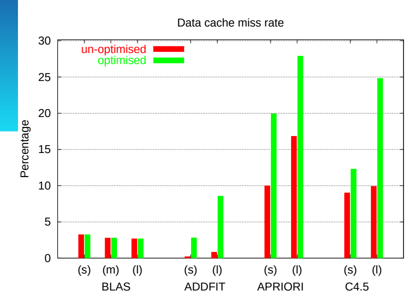
- Optimised compilation was done using the `-fast` option
- Data mining applications do not use floating-point units (instead mainly integer operations, plus more loads/ stores)
- MIPS rate generally decreases with larger data sizes

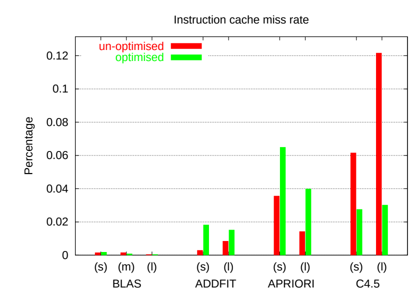
Cycles per instruction and run times



Program	ADDFIT		APRIORI		C4.5	
	small	large	small	large	small	large
Un-optimised	1.1 sec	5.9 sec	3.4 sec	31.8 sec	2.4 sec	421.0 sec
Optimised	0.5 sec	2.8 sec	2.2 sec	19.9 sec	1.6 sec	375.5 sec
Improvement	52%	53%	34%	37%	34%	11%

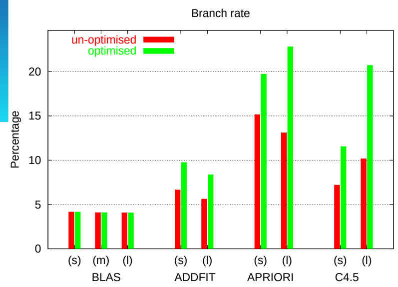
Cache miss rates

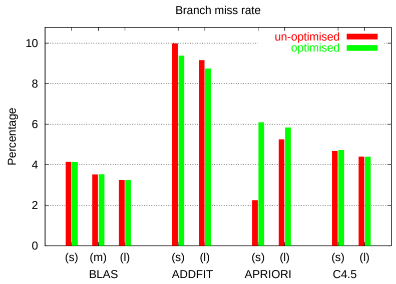




- Irregular memory access patterns result in higher data cache miss rates (less locality)
- Optimised compilation increases data cache miss rate
- Higher instruction cache miss rates due to more complex and longer *core* routines

Branch rates and branch miss rates





- Irregular data structures result in higher branch (miss) rates
- Data mining applications do not have long loops that are *'predictable'* (e.g. over vectors)
- Optimised compilation *'removes'* many loads and stores (e.g. for indices) from the code

Conclusions

- Performance analysis is important to
  - understand behaviour of modern complex applications
  - find bottlenecks both in software (applications as well as OS) and hardware (CPU and memory system)
  - improve efficiency and performance of modern computer systems
- Hardware counters are a good performance analysis tool (but it's easy to drown in numbers, and results can be hard to understand)
- Various improvements can be done on data mining applications (e.g. try to use floating-point operations)