Sequential In-core Sorting Performance for a SQL Data Service and for Parallel Sorting on Heterogeneous Clusters

Christophe Cérin [a], Michel Koskas [b], Hazem Fkaier [c],
Mohamed Jemni [c]

[a] *Université de Picardie Jules Verne, LaRIA, Bat Curi, 5 rue du moulin neuf, F-80039 Amiens cedex 1- France*

[b] *Université de Picardie Jules Verne, LaMFA/CNRS UMR 6140, 33 rue St Leu, F-80039 Amiens cedex 1- France*

[c] *Ecole Supérieure des Sciences et Techniques de Tunis, Unité de recherche UTIC, 5 Av. Taha Hussein, B.P 56 Bab Menara, 1008 Tunis - Tunisie*

Abstract

The aim of the paper is to introduce techniques in order to tune sequential in-core sorting algorithms in the frameworks of two applications. The first application is parallel sorting when the processor speeds are not identical in the parallel system. The second application is the Zeta-Data Project (Koskas , 2003) whose aim is to develop novel algorithms for databases issues. About 50% of the work done in building indexes is devoted to sorting sets of integers. We develop and compare algorithms built to sort with equal keys. Algorithms are variations of the 3way-Quicksort of Segdewick. In order to observe performances and to fully exploit functional units in processors and also in order to optimize the use of the memory system and the different functional units, we use hardware performance counters that are available on most modern microprocessors. We develop also analytical results for one of our algorithms and compare expected results with the measures. For the two applications, we show through fine experiments on an Athlon processor (a three-way superscalar x86 processor), that L1 data cache misses is not the central problem but a subtil proportion of independent retired instructions should be advised to get performance for in-core sorting.

Key words: hardware performance counters, in-core sorting algorithms with equal keys, two levels memory hierarchy, optimizing memory accesses, parallelism at the chip level, data structures for databases, parallel sorting.

1 Introduction

The story of sorting is long enough to deserve many books, among them the Knuth series (Knuth, 1998). One reason among others for the popularity of sorting is that sorted data are easier to manipulate than unordered data, for instance a sequential search is much less costly when the data are sorted. The quasi non predictable aspects of memory references in sorting algorithms make them good candidates to appreciate the performance of processors in real situations. Two supplementary elements contribute to the 'no-end' story.

First of all, the data are never accessed instantaneously with current processors. Computer architects (Hennessy and Patterson, 2002; Shriver and Smith, 1998) have imagined more and more complicated memory designs, called caches, in order to get the fastest memory accesses. Cache design implies time penalties and many parameters should be considered (cache line size, write and read policiess, replacement algorithms...).

Second, a majority of processors available on the market place (Athlon, Pentium processors for instance) are now equiped with special registers that record the activity of the processor at no cost, we mean with no overhead caused by the observation of events. So, we have now the possibility to observe finely the behavior of codes and analyse their performance without the use of a simulator. Many works have been accomplished in order to interface the observable events with codes. Among the current project we deal with `Perfctr`[1] which is used by many performance analyzer tools, for instance `Papi`[2].

Moreover, modern processors include parallelism in the chip. For instance, the Athlon processor is a three-way superscalar x86 processor. The chip has three integer pipelines, three floating point pipelines and three full x86 decoders. This means that the chip can potentially executes three instructions in parallel.

In this paper, we study how the number of L1 data cache misses and the number of retired instructions influence the "quality" of parallelism of a modern chip and for sorting with equal keys. This problem, studied for instance by Sedgewick[3] (Sedgewick, 1977; Bentley and Sedgewick, 1997), is of crucial importance in the Zeta-Data project. The problem of sequential sorting is also important for the implementation of parallel sorting when processor speeds are not identical (Cérin and Gaudiot, 2000a,b, 2002) because the parallel algorithms execute in parallel portions of sequential codes for sorting.

---

[1] See: http://user.it.uu.se/~mikpe/linux/perfctr/
[2] See: http://icl.cs.utk.edu/projects/papi/software/
[3] For more information, see also Robert Sedgewick home page at http://www.cs.princeton.edu/~rs

In this paper we investigate the performance of 7 sorting algorithms. All the algorithms are based on invariants that maintain (in different ways) equal keys in one position, keys lower than equal keys in another one and keys greater to equal keys in yet another position in the input array.

One goal is to sort 'in place' i.e. without any supplementary memory. The maintaining of the invariant implies memory references that can potentially create excessive cache misses or induces dependent instructions that will freeze the functional units. We show how to extract temporal and spacial locality in order to better exploit the different functional units (working in parallel) of the Athlon microprocessor. In this tuning, we still improve the sorting (with equal keys) stage of our first implementation by 20% and we beat again the 3way-Quicksort implementation of Sedgewick.

The organization of the paper is as follows. In Section 2 we summarize the advantage of working at the register level to collect information about the activity of a processor against the use of simulation techniques. Section 3 is about our motivations and related work on sorting. Section 4 introduces our selected algorithms incase of our applications. Section 5 introduces our experimental results. Section 6 concludes the paper.

2 The opportunities of working at the register level

Monitoring the collected processor events facilitates correlation between the structure of source/object code and the efficiency of the mapping of that code to the underlying architecture and done by the compiler. This correlation has a variety of uses in performance analysis including hand tuning, compiler optimization for better ressource usage, debugging, benchmarking, monitoring and performance modeling. In addition, it is hoped that this information will show itself useful in the discovery of commonly occurring bottlenecks in high performance computing codes.

We address the problem of cache effect on the performance of sorting with equal keys. We consider only the first level of the memory hierachy: register and L1 data cache. We do prefer to monitor the hardware events than to use a simulator. Despite the fact that a (software) simulator has the advantage to require no hardware, we guess that very fine architectural details are very difficult to simulate with a good precision.

For instance, modern processors are based on pipelined functional units, multiple functional units, speculative execution, several levels of cache memory and sometimes cache lines are shared between CPUs. Factors such as variations in the process scheduling and the operating system's virtual to physical page

mapping policy contribute to the difficulty to analyze cache misses output by a simulator.

To gain access to the hardware events on Linux/x86 platforms, we use `Perfctr`. With `Perfctr`, each Linux process has its own set of "virtual" counters: the counters appear to be private to a process and unrelated to the activities of other processes in the system. The virtual counters can be sampled in user-space without the overhead of a system call.

3 Motivations

Many works about sequential sorting have been done in the past to analyse the performances of RISC processors (Agarwal, 1996), (Nyberg et al., 1994), (Larriba-Pey et al., 1997) or to study processors with a low number of registers or with small caches (Arge et al., 2001), (Ranade et al., 2000).

In (Rahman and Raman, 2000), N. Rahman and R. Raman studied radix sort and more precisely the importance of reducing misses in the translation-look aside buffer (TLB). No experimental measures of the misses are accomplished. We do not use here radix-sort because LaMarca and Ladner in (LaMarca and Ladner, 1999) proved that radix trees do not perform well comparing to QuickSort based sorting algorithms. They showed that despite its lowest instruction count, radix sort has poor cache performance hence its overall performance is worse than the memory optimized versions of mergesort and quicksort.

Indeed, in our Zeta-Data application we sort couples of values (cell value, line index) and radix-sort based algorithms are in this case more complicated to implement efficiently.

3.1 Sorting and cache effects

One of the most valuable paper about the influence of cache and the impact of the memory hierarchy on sorting is the paper of LaMarca and Ladner (LaMarca and Ladner, 1999). However, the experiments are done with ATOM (Srivastava and Eustace, 1994) which is a simulator built in the beginning of nineties.

The paper of LaMarca and Ladner [4] (LaMarca and Ladner, 1999) explores the performance of four popular sorting algorithms: mergesort, quicksort, heap-

---

[4] A prior technical report is dated from 1994

sort and radix sort on DEC Alphastation 250 and trace driven simulation with ATOM. For each of the four sorting algorithms they choose "an implementation variant with potential for good overall performance and then heavily optimize this variant using traditional techniques to minimize the number of instructions executed". They concentrate on three performance measures: instruction count, cache misses and overall performance (time).

The two first competitors, regarding the execution time, are mergesort and quicksort. The main general lesson of the paper is that "Improving an algorithm's overall performance may require increasing the number of instructions executed while, at the same time, reducing the number of cache misses". We confirm only in this paper the first part of the sentence. We will show later that what is important (today with the current technology) is the number of instructions executed per processor cycle.

The authors apply memory optimizations in order to improve cache performance. Optimizations are based on *temporal locality* and *spacial locality* principles. A program exhibits *temporal locality* if there is a good chance that an accessed data will be accessed again in the near future. A program exhibits *spacial locality* if there is a good chance that subsequently accessed data are located near each other in memory.

Databases applications generate duplicates and Quicksort based algorithms can not provide performance in this case. It is well known for instance that Quicksort, even optimized for caches, has a complexity of $O(n^2)$ in case of duplicates i.e. in the case of an input "already" sorted.

We absolutely need new algorithms to capture performance for situations that are part of databases benchmarks such as the TPC benchmark. For instance, while building or modifying the table of indexes, one has to sort the fields of each column of the tables. This means for instance that a table with 5 lines (for instance the "region" table of the TPC) may be expanded in a table with 6 millions of lines (for instance the "lineitem" table of the TPC). Thus one has to sort the fields of each column of the table "region" expanded in the table "lineitem". This means that one has to sort arrays of 6 millions of items with only 5 different values.

4   Some algorithms for in-core parallel sorting

According to LaMarca and Ladner (LaMarca and Ladner, 1999) results, we keep Quicksort and Mergesort algorithms because they have produced the best performance in their simulations. So, we reduce here our study to comparisons based sorting algorithms.

Fastsort (Nilsson and Raman, 1995; Nilsson, 2000) is an $O(n \log \log n)$ sort that uses properties of the input (integers) to compress it in order to reduce the number of compare instructions to execute. No experimental feedback is known for Fastsort. Due to its potential, we also keep it.

We choose FAME (Ranade et al., 2000) which was drawn for the purpose of sorting on CPU with a few number of registers. FAME is a m-way merge-sort, thus, by essence it may reduce also cache misses since reducing register movements "implies" reducing data movement and L1 misses. The m-streams are merged by organizing comparaison tournament among the elements at the heads of the streams. The control structure is a finite state machine. Only time sorting results are known for FAME. Cache results is not reported in (Ranade et al., 2000). The implementation results done in (Ranade et al., 2000) are partly accomplished with $m = 4$ and thus FAME makes $\log_4 n$ passes over memory. We keep the same setting in this paper.

4.1   ZZZmerge and Zmerge

We introduce now our new cache concious algorithms ZZZmerge and Zmerge. ZZZmerge is simply an optimized version of Zmerge with less copies. So, we only introduce the principles of ZZZmerge. ZZZmerge is a z-way-mergesort with two steps. A virtual binary tree is built. We first sort all the leaves by packets of size $z * z$. We use insertion sort. Then a merge step occurs: at a certain level in the tree, we merge all values contained in pointers below nodes at that level, two by two. As with any mergesort algorithm, the merging step requires a supplementary buffer of size n but we do not make any copies: alternatively during the tree search we choose to work on the "original" buffer or on the "supplement" buffer. A swap on pointers implements the technique. The trick reduces significantly the cache misses. All of our experiments are done with $z = 4$ to facilitate the comparison with FAME.

We developed a single loop that performs $n/(z * z) - 1$ iterations or merge steps. A tedious calculus is required to compute the bounds of the portions to be merged. The last iteration merges the portions between indices 0 and $n/2 - 1$ and $n/2$ and $n - 1$, the two previous ones merge portions $[0 \cdots n/4 - 1]$ with $[n/4 \cdots n/2 - 1]$ and $[n/2 \cdots n/2 + n/4 - 1]$ with $[n/2 + n/4 \cdots n - 1]$ and so on! The first $n/z * z * 2$ iterations serve to merge the leaves, the following $n/z * z * 4$ serve to merge nodes at the first level in the virtual tree, and so on. Note that the input size should divide $z * z$ and be a power of 2. It is a limitation of our code at present time.

Theorem 1 *The number of cache misses of ZZZmerge is*

$$n/(z*z) + \frac{n}{z*z} \log n/(z*z)$$

Proof: we decompose the proof into two parts corresponding to the two steps of the algorithms. We assume that $z*z$ integers can fit in a line cache and that M stands for the cache size and it is a multiple of $z*z$ i.e. $M = k*z*z, k \geq 1$. Moreover, a cache miss will bring $z * z$ integers in the cache. Since we have $n/(z*z)$ insertion sorts to accomplish in the first step, each on vectors of size $z * z$, we get at most $n/(z * z)$ misses which is optimal.

For the second step now. Let $C(p)$ be the number of cache misses for merging two sorted sub-lists, each of size p. The cache complexity of this step is given by:

$$\sum_{i=1}^{\log n/(z*z)} 2^{i-1} \times C(n/2^i)$$

Clearly, $C(n/2^i) = 2 * (n/2^i)/(z * z) = n/(2^{i-1} * z * z)$, thus our sum reduces to:

$$\sum_{i=1}^{\log n/(z*z)} n/(z*z) = \frac{n}{z*z} \sum_{i=1}^{\log n/(z*z)} 1 = \frac{n}{z*z} \log n/(z*z) \tag{1}$$

The total number of cache misses $n/(z*z) + \dfrac{n}{z*z} \log n/(z*z)$ follows.

4.2   Selection of algorithms for the Zeta-Data project

Remind here that our goal is to efficiently sort integers with many duplicates. Our different strategies for sorting consider 4 bytes long integers. All the tested algorithms are a variation of the 3way-Quicksort algorithm and are linked to the 'dutch national flag problem'. So, we maintain here the same invariant than the invariant used for the dutch national flag problem in the sense that we maintain, during the execution, the following condition for the partitioning step:

<	=	>

The algorithm for the Dutch national flag gathers, in the middle of the array, elements equal to a partitioning element; on left, elements that are lower than the partitioning element; on right, elements that are greater than the partitioning element. After the partitioning step, it remains to sort recursively

the left and right portions. Our algorithms differ in the ways they produce the invariant above.

4.3   The different ways we maintain the invariant

Seven algorithms are under concern: TriEntiers,TriEntiers1, TriEntiers2, 3way-Quicksort, TriEntiers4, TriEntiers5 and TriEntiers6. 3way-Quicksort has been developped by Sedgewick, all the others by us. All the algorithms are Quick-Sort based algorithms and they differ on the partitioning step.

The different strategies in order to maintain the above invariant are presented in Figure 1. The picture describes the "intermediate invariant" that we use before rearranging data in order to produce the Duch national flag invariant. In practice, our algorithms start to check if the portion under concern is sorted (forward or backward) then, if not, we select a pivot (median element) in order to partition the input.
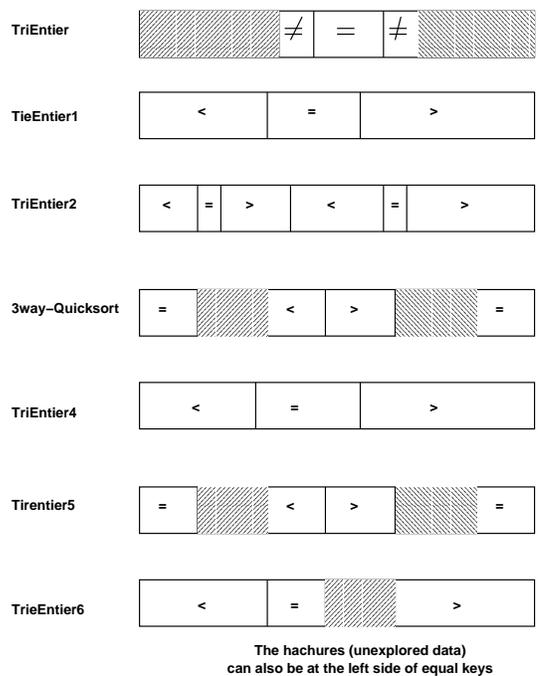

Fig. 1. "Intermediate invariants": the different strategies to maintain the invariant.

5   Experimental results

Our experiments are accomplished on an Athlon(tm) XP 1800+ processor equiped of 512MB of DDR PC2100. Our motherboard is a MS-6373 from MICRO-STAR INTERNATIONAL CO., LTD which requires 32MB of RAM

for video purpose. The front side bus speed is $2 * 137MHz$ (274MHz data rate). The Athlon processor is a three-way superscalar x86 processor. The L1 internal data cache on Athlon processors is a 64kB synchronous write-back (2-way, 64 byte line size). We focus on sorting an array of 32 bit integers chosen uniformly at random. For each input size, We run 30 trials and we measure the mean.

In the first part of the work, we employ gcc version 2.96-20000731 (Mandrake Linux 8.1 2.96-0.62mdk). All our codes are compiled with the following flags: `-O2 -fomit-frame-pointer -Wall`. With these settings, we avoid for instance that MMX registers and MOVNTQ assembler instructions to be (potentially) used: this instruction bypasses the on-chip cache and goes directly into a write combining buffer, effectively increasing the total write bandwidth. We denote by E1 the experiments according to such settings. In the second part of experiments, the `-march=athlon -O3` flags are used in conjuction with GCC 2.96 in order to optimize our codes for the Athlon processor. In the third part, we employ GCC 3.2 with the following flags: `-O3 -fprefetch-loop-arrays -mmmx -m3dnow -march=athlon -fomit-frame-pointer -Wall`. We do not use the `-msse -msse2 -march=athlon-xp` flags because we observed either a slowdown in the executions or core dumps, in particular with `Zmerge.c` file. We denoted by E2 the experiments according to these settings.

5.1   Results for E1 experiments

Figures 2 and 3 reveal the cache behaviours of our tested algorithms. Fastsort behaviour is very poor. When we examine its source code, we find bitwise AND and OR, and unrestricted bit shift, i.e., shift of an entire word (with zero filling) by a number of bit positions specified in a second word. We know that the cost of such operations is very high. FAME and 3-way-quicksort exhibit the best results. The result for FAME validates the approach. ZZZmerge is ranked third and it is about two times the values of 3-way-quicksort. It can be explained by the fact that it uses two buffers to manage the merge step.

Figures 4 and 5 show the mean execution times of our tested algorithms. ZZZmerge and 3-way-quicksort are the best two: Table 5.1 shows the details for the two best. We observe that ZZZmerge beats 3-way-quicksort by at least 5% despite two times more L1 misses (see Figure 2 and 3). The explanation is related to the number of instructions executed per cycle of the codes and we will comment this later on.

The number of retired instructions executed by our codes are given on Figures 6 and 7. We observe that 3-way-quicksort execute approximatively two times less instructions than ZZZmerge (ranked third). Fastsort is the second best
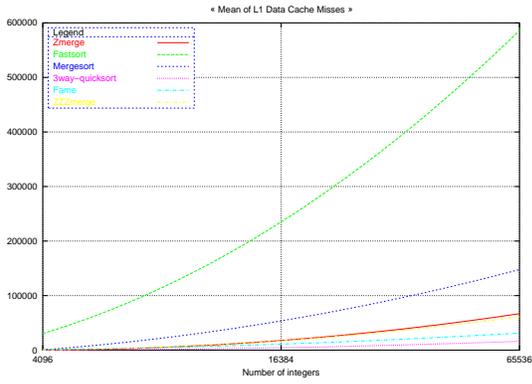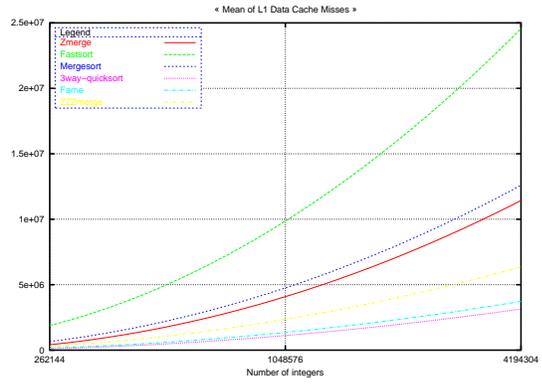
Fig. 2. Mean L1 data cache misses (Part 1)

Fig. 3. Mean L1 data cache misses (Part 2)

# int	Quicksort(ms)	ZZZmerge(ms)	Gain
4096	0.45203	0.33952	25%
16384	2.04220	1.55541	24%
65536	9.51033	7.51966	21%
262144	46.2616	43.9141	5%
1048576	219.138	196.967	11%
4194304	912.462	867.897	5%

Table 1
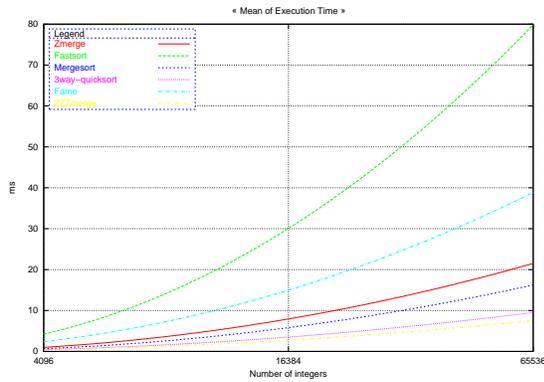Details of the two best execution times.
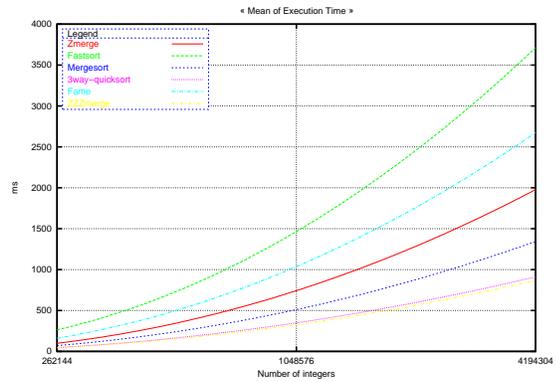

Fig. 4. Mean execution times (Part 1).

Fig. 5. Mean execution times (Part 2).

and this result was expected since the algorithm "compress" in some way the integers in order to reduce the number of comparisons.

Figure 8 shows the measured IPC (Instructions per Cycle). For a given input size, it is computed as the ratio of the mean value of the measured retired instructions over the mean value of the execution time (normalized by the
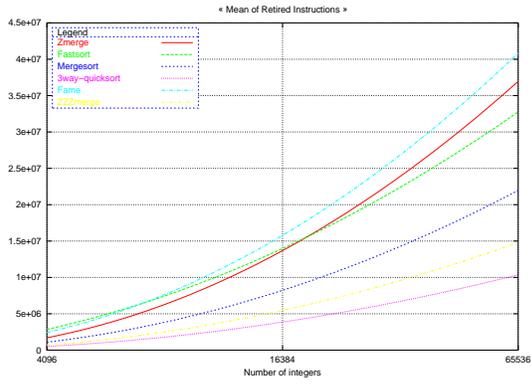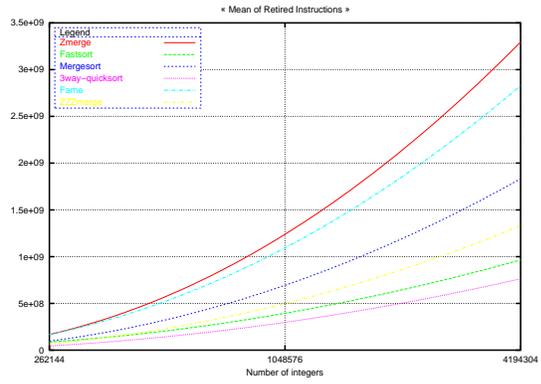
10

Fig. 6. Retired instructions (Part 1).

Fig. 7. Retired instructions (Part 2).

chip frequency). We note that in the best cases, the IPC varies between 1.3 and 1. We note that the IPC of ZZZmerge is 45% more important than the IPC of 3-way-quicksort: the number of independent instructions is probably more important with ZZZmerge than the others. Thus the execution units are better exploited. Since we cannot distinguish on Athlon the kind of instructions that the processor really execute we cannot analyze more in deep the observation.
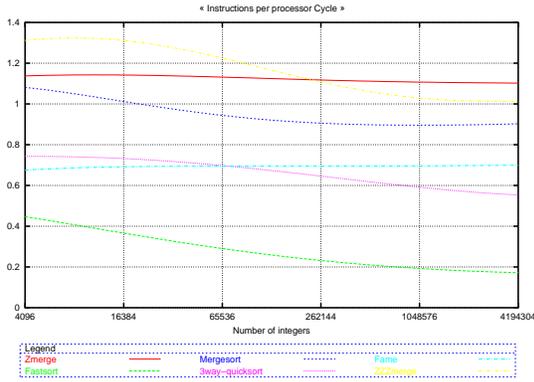

Fig. 8. Ratio of the mean value of the measured retired instructions over the mean value of the execution time (normalized by the chip frequency).
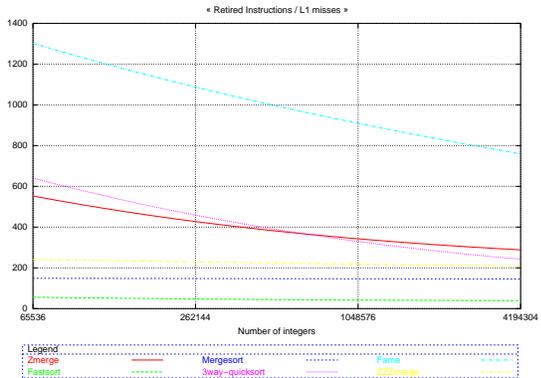

Fig. 9. Ratio of mean values of retired instructions over the mean values of L1 data cache misses.

The maximal value of the IPC on Athlon is 3, compared to 1.3 (or 1) in the best cases. There is a potential to improve the IPC, perhaps in increasing carefully a little bit more the number of instructions executed by the code.

Figure 9 shows the ratio of the mean values of retired instructions over the mean values of L1 data cache misses. We observe that fastsort results are very poor as expected (we have a miss each 50 instructions). We observe that the curve for ZZZmerge is very flat: we guess that a miss arrives in a regular way independently of the input size. This is a good property. In a converse way, the performance of 3-way-quicksort decreases as the input size increases. The

11

curves for ZZZmerge and 3-way-quicksort are very close for 4194304 integers. This observation explains why ZZZmerge finally beats 3-way quicksort in time. At least, we observe that FAME offers the best ratios but its IPC are not high enough to get a good execution time. On this point, FAME achieves the objective of reducing misses but at a price of a bad IPC: too many operations are needed to manage the FAME data structures.

Note: results of experiments with `-march=athlon -O3` flag settings and still for GCC 2.96 can be obtain upon request. Globally, they confirm the previous observations and in particular that ZZZmerge is the fastest algorithm.

5.2   Results for E2 experiments

All results of experiments with GCC 3.2 and `-O3 -fprefetch-loop-arrays -mmmx -m3dnow -march=athlon -fomit-frame-pointer -Wall` flag settings can also be obtained upon request. Again, they confirm the previous observations and in particular that ZZZmerge is the fastest algorithm. The use of GCC 3.2 confirms that more aggressive optimizations enforce the superiority of ZZZmerge. For instance, when we examine ZZZmerge assembler code, we find prefetch instructions and alignments.

5.3   Summarize

The experiments demonstrate that a subtle combination between the number of L1 data cache misses and the number of instructions executed by the Athlon processor leads, in practical cases, to the best algorithms when considering execution times. Despite the fact that the distinction between the types of instructions (load, store, add...) is not possible to observe on Athlon, we put forward the following conjecture: Any (sorting) program X with an IPC superior by a factor of at least two to a (sorting) program Y executing two times less instructions with two times less L1 data cache misses, is better in time than program Y.

Moreover, there is a potential for improving the IPC, perhaps in increasing carefully a little bit more the number of independent instructions executed by the codes. The next array summarizes the result of experiment for the case of duplicates. It gives the rank of the different algorithms for the execution time, retired instructions and cache misses metrics:

IPC	Exec Time	L1 data misses
1: TriEntier	1: TriEntier4, TriEntier5	1: TriEntier, TriEntier5
2: TriEntier4	2: TriEntier1	2: TriEntier4
7: TriEntier5		

Surprisingly, we observe that with two wins for the IPC and the number of misses, TriEntier is beaten in execution time. It is probably due to the fact that the instructions executed by TriEntier are much more dependent than those of TriEntier4 and TriEntier5. TriEntier does not execute enough "useful work", that is to say in putting elements at the final place.

We note that there is no absolute winner for the tests. To get performance in our application, a choice among algorithms should be made according to the number of duplicates. More work needs to be done to evaluate the most important problem and system parameters and make a quick decision for sorting in presence of duplicates.

6 Conclusion

In this paper we have investigated the cache effects between the first two level of memory hierarchy for sorting algorithms. We have in mind two applications where sequential sorting is important to get performance. The first application is parallel sorting for heterogeneous clusters and the second one is the development of a SQL service where sorting with equal keys is important. With the first application in mind, we devised a new mergesort algorithm that performs well in practice.

We will examine in the future if the organization of a tournament "à la FAME" could be reused more efficiently and could be added to ZZZmerge. Moreover, since ZZZmerge was invented to take into account two-levels memory problems, what is the amount of work to accomplish to provide an out-of-core version of ZZZmerge that could be a concurrent of existing out-of-core sorting algorithms?

In the case of sorting with equal keys, an important problem that occurs with the implementation of our SQL service in the Zeta-Data project, we devised new algorithms that performs well in practice, better than the known 3way-Quicksort when we have duplicates, typically 100-200 different values in an input vecteur of size four million. Our new algorithms improve by 20% the

performance of the initial sort implemented in the Zeta-Data project which is also better than 3way-Quicksort. We better exploit the functional units and we have relaxed the dependencies between instructions.

We plan also to experiment with our freely available codes on other processors than Athlon. Codes are simply dependent of `Perfctr` in the sense that AMD Athlon, Intel Pentium Series on Linux 2.2, 2.4 can be tested. The goal is to check if the assumptions done in a previous section is still true.

We are currently interfacing our sorting codes with the current release of Zeta-Data. We expect to improve significantly the performance since the internal data structure we are revisiting in Zeta-Data permit us to know the number of different values in the input and thus to adapt the choice of the best sorting algorithm. We are guessing here that the solution will be to build an hybrid algorithm.

References

Agarwal, R. C., june 1996. A super scalar sort algorithm for RISC processors. SIGMOD Record (ACM Special Interest Group on Management of Data) 25 (2), 240–246.

Akl, S., 1985. Parallel Sorting Algorithms. Academic Press.

Arge, L., Chase, J., Vitter, J. S., Wickremesinghe, R., 2001. Efficient sorting using registers and caches. Lecture Notes in Computer Science 1982, 51–61.

Bentley, Sedgewick, 1997. Fast algorithms for sorting and searching strings. In: SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms).

Cérin, C., Gaudiot, J.-L., 6May 2000a. Evaluation of two bsp libraries through parallel sorting on clusters. In: Proceedings of WCBC'00 (The Second International Workshop on Cluster-Based Computing) in conjuction with ICS'00 (International Conference on Supercomputing, sponsored by ACM/SIGARCH). Santa Fe, New Mexico, pp. pp 21–26.

Cérin, C., Gaudiot, J.-L., 28Nov.-2Dec. 2000b. An over-partitioning scheme for parallel sorting on clusters running at different speeds. In: Cluster 2000. IEEE International Conference on Cluster Computing. Technische Universität Chemnitz, Saxony, Germany. (Poster).

Cérin, C., Gaudiot, J.-L., 17-20Dec. 2000c. Parallel sorting algorithms with sampling techniques on clusters with processors running at different speeds. In: HiPC'2000. 7th International Conference on High Performance Computing. Bangalore, India. Lecture Notes in Computer Science. Springer-Verlag.

Cérin, C., Gaudiot, J.-L., 2002. On a scheme for parallel sorting on heterogeneous clusters. FGCS (Future Generation Computer Systems 18 (issue 4), the special issue is preliminary scheduled for publication in future vol.

Frigo, M., Leiserson, C. E., Prokop, H., Ramachandran, S., 1999. Cache-

oblivious algorithms. In: IEEE (Ed.), 40th Annual Symposium on Foundations of Computer Science: October 17–19, 1999, New York City, New York,. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, pp. 285–297.

Hennessy, J. L., Patterson, D., 2002. Computer Architecture, A Quantitative Approach (third edition). Morgan Kauffmann.

Knuth, D. E., 1998. Sorting and Searching, 2nd Edition. Vol. 3 of The Art of Computer Programming. Addison-Wesley, Reading, MA, USA.

Koskas, M., 2003. A hierarchical database management algorithm. Personal communication.

LaMarca, A., Ladner, R. E., Apr. 1999. The influence of caches on the performance of sorting. Journal of Algorithms 31 (1), 66–104.

Larriba-Pey, J.-L., Jimenez, D., Navarro, J., 1997. An analysis of superscalar sorting algorithms on an r8000 processor. In: Proceedings of the 17th International Conference of the Chilean Computer Science Society (SCCC '97), November 12-14, Valpariso, IEEE Computer Society.

Nilsson, A. A. T. H. S., Raman, R., 1995. Sorting in linear time. In: ACM (Ed.), In Proceedings of the 27th Annual ACM Symposium on the Theory of Computing (STOC). ACM Press, New York, NY 10036, USA, pp. 427–436.

Nilsson, S., Apr. 2000. The fastest sorting algorithm? Dr. Dobb's Journal of Software Tools 25 (4), 38, 40, 42, 44–45.

Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., Lomet, D. B., Jun. 1994. AlphaSort: A RISC machine sort. SIGMOD Record (ACM Special Interest Group on Management of Data) 23 (2), 233–242.

Rahman, N., Raman, R., October 2000. Adapting radix sort to the memory hierarchy. Tech. Rep. TR-00-02, Department of Computer Science, King's College London, a preliminary version of the paper appeared in the Proceedings of the 2nd Workshop on Algorithm Engineering and Experiments (ALENEX'00).

Ranade, A., Kothari, S., Udupa, R., 96-103Dec. 2000. Register efficient mergesorting. In: HiPC'2000. 7th International Conference on High Performance Computing. Bangalore, India. Lecture Notes in Computer Science. Springer-Verlag.

S.Chatterjee, S., 2000. Towards a theory of cache efficient algorithms. In: Proceedings of the Eleventh Annual ACM/SIAM Symposium on Discrete Algorithms (SODA'00). San Francisco, California, United States, pp. 829 – 838.

Sedgewick, R., 1977. The analysis of quicksort programs. Acta Informatica 7 (25), 327–355.

Shriver, B., Smith, B., 1998. The anatomy of a High-Performance Microprocessor - A Systems Perspective. IEEE Computer Societ.

Srivastava, A., Eustace, A., 1994. Atom: a system for building customized program analysis tools. In: In Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation, 1994. pp. 196–205.