Accessing Hardware Performance Counters in order to Measure the Influence of Cache on the Performance of Integer Sorting

Christophe Céerin
Université de Picardie Jules Verne
LaRIA, Bat Curi, 5 rue du moulin neuf
F-80000 Amiens - France
cerin@laria.u-picardie.fr

Hazem Fkaier, Mohamed Jemni
Université de Tunis EL MANAR, Faculté des Sciences de Tunis, Laboratoire LIP2
1060 Belvédère Tunis - Tunisie
hazem.fkaier@fst.rnu.tn, mohamed.jemni@fst.rnu.tn

Abstract

Hardware performance counters are available on most modern microprocessors. These counters are implemented as a small set of registers that count events related to the processor’s functions. The Perfctr toolkit is one of the most popular toolkits (for x86 processors) for monitoring these events. In this paper, it is used to discover the impact of L1 data cache misses on the overall performance of six integer sorting algorithms. Most of them are cache conscious algorithms recently introduced, or known to behave well according to previous simulations, or they are totally not explored. We demonstrate through experiments on an Athlon processor that a good balance between L1 data cache misses and retired instructions provides the fastest algorithm for sorting in practical cases. The fastest sorting algorithm is not obtained with the implementation that gives the smallest number of misses and the smallest number of instructions. The fastest algorithm in practice is thus a new flavour of mergesort that we have developed and it beats its rival.

Keywords: hardware performance counters, cache conscious and oblivious algorithms, in-core sorting algorithms, two levels memory hierarchy, parallelism at the chip level.

1 Introduction

It is often said that 25 to 50 percent of all the work performed by computers is being accomplished by sorting algorithms [Akl85]. 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.

Computer architects [HP02, SS98] have imagined more and more complicated memory designs (caches) in order to speedup memory accesses. Cache design implies time penalties and many parameters should be considered (cache line size, write and read politics, replacement algorithms...).

Moreover, 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 finelly the behavior of codes and analyse their performance without the use of a simulator. Many works are accomplished in order to interface the observable events with codes. Among the current projects [DHW97] we deal with Perfctr\(^1\) which is used by many performance analyzer tools, for instance Papi\(^2\).

The organization of the paper is as follows. In Section 2 we recall briefly 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 previous works that study cache effects for sorting. Section 4 introduces our selected algorithms. We also introduce a mergesort based algorithm called ZZZmerge. We investigate the performance of ZZZmerge both analytically and experimentally. Section 5 explains the results of experiments and lessons learn from them. Section 6 concludes the paper.

\(^1\)See: http://user.it.uu.se/~mikpe/linux/perfctr/

\(^2\)See: http://icl.cs.utk.edu/projects/papi/software/
2 The opportunities of working at the register level

There is two main techniques to evaluate the behavior of a code on a particular computer platform. Either you enter your code in a simulator of the target computer architecture or you run it on the platform. Both techniques have in common to provide a feedback about events and resources used that drive the performance of the code.

Monitoring the collected 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 resource usage, debugging, benchmarking, monitoring and performance modeling. In addition, it is hoped that this information will prove 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. We consider only the first level of the memory hierarchy: 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, RISC processors are based on pipelined functional units, multiple functional units, speculative execution, several levels of cache memory and some times 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.

3 Related work

Many works about sequential sorting have been done in the past to analyse the performances of RISC processors [Aga96], [NBC+94], [LPJN97] or to study processors with a low number of registers or with small caches [ACVW01], [RKU00]. In [RR00], 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 introduce now another study similar to our concerns.

One of the most valuable paper about the influence of cache on sorting is the paper of LaMarca and Ladner [LL99]. However, the experiments are done with ATOM [SE94] which is a simulator built in the beginning of nineties.

The paper of LaMarca and Ladner [LL99] explores the performance of four popular sorting algorithms: mergesort, quicksort, heapsort 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 first two winners when they consider 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 on that the important factor (with the current technology) is the number of instructions executed per processor cycle.

4 Selection of algorithms

According to LaMarca and Ladner [LL99] results, we keep Quicksort and Mergesort algorithms because they have produced the best performance in the simulations. So, we reduce here our study to comparisons based sorting algorithms... Fastsort excepted!

4.1 MergeSort

Mergesort is based on a global strategy: split the problem into two or more large pieces, solve the subproblems, then marry the solutions to solve the subproblems. We choose an implementation that is not recursive but merge iteratively pairs of 1-element blocks, then 2-element blocks, then 4-element blocks,... until the merge size is greater or equal to the size of the whole array. A double buffering technique is also used to reduce the amount of copies. We experiment with a code available on the Web3.

3See: http://www.csm.astate.edu/~rossa/datastruc/ and the file is merge.html
4.2 3-way Quicksort

We do prefer to use a version of Quicksort based on a 3-way partitioning scheme rather than the popular Quicksort. 3-way partitioning has been analyzed in [BM93] by Bentley and McIlroy and it has been introduced to sort with equal keys. One of the main property of 3-way partitioning is that it involves a linear number of comparisons if keys are all equal and no extra compares if there is no equal keys. Thus, 3-way quicksort extends the utility of the “traditional” Quicksort. We experiment with the code of Sedgewick available on the Web4.

4.3 Fastsort

The Fastsort algorithm [AHNR95, Nil00] is able to sort n integers in the range 0···2w − 1 in $O(n \log \log n)$ time, for arbitrary $w \geq \log n$ (a word length is $w$ bits). It 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.

The algorithm needs to know the value of $w$ itself, but uses no other precomputed constants. The key of success is a combination of two techniques namely packed sorting and range reduction. Packed sorting, introduced by Paul and Simon [PS80] and developed further in [KR81] and [AH92], packs several integers into a single word and operates simultaneously on all of them at unit cost (in the model). This is only possible, if several integers to be sorted fit in one word. Range reduction reduces the problem of sorting integers in a certain range to the problem of sorting on a smaller range. The combination of the two techniques is straightforward: first range reduction is applied to replace the original full-size integers by smaller integers of which several fit in one word, and then these are sorted by means of packed sorting.

We experiment with Stefan Nilsson code that is available on the Web5.

4.4 Fame

Ranade, Kothari and Udupa in [RKU00] introduce FAME (Finite Automaton MErgesort) which is a $m$-way-mergesort dedicated to optimize register use: the heads of the $m$ streams are maintained in the register file of the processor. At each step of the algorithm, a tournament is organized between heads of the streams and the winner is output to the final sequence. To minimize work and running time, the next tournament uses the results of the comparisons performed in the preceding tournaments.

The state of a tournament is encoded as a finite state machine and the states of the automaton code the relations between the values of the heads. For instance and for $m = 4$ (4 streams - the heads are denoted $s_0, s_1, s_2, s_3$), it is not difficult to see that we need 3 bits to code the smaller among $s_0, s_1, s_2, s_3$. One bit codes $s_0 < s_1$, another bit codes $s_2 < s_3$ and the third bit codes the smaller among the two previous smaller. Since we need 3-bits, we have $8 = 2^m - 1$ states in the automaton.

The keys at the heads and pointers to the stream heads are maintained in processor registers. We also need about $O(m)$ lines of code for each state. So, the code length is $O(m.2^m)$. This is a drawback for large $m$. So the code has two main properties. First of all, far small $m$ values, all the state addresses can fit in registers. Second, it completely eliminates register movements and in general unnecessary copies. These properties are in favour of reducing caches misses.

The implementation results done in [RKU00] are partly accomplished with $m = 4$ and thus FAME makes $\log_4 n$ passes over memory. We keep the same setting in our experiments.

4.5 ZZZmerge and Zmerge

We introduce now our new cache conscious algorithms ZZZmerge and Zmerge. ZZZmerge is simply an optimized version of Zmerge. 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 $z \times z$. 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 (see curves about Zmerge and ZZZmerge in the forthcoming sections).

We developed a single loop that performs $n/(z \times z) - 1$ 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

---

4See: http://www.cs.princeton.edu/~rs/
5See: http://www.nada.kth.se/~enlsson/
two previous ones merge portions \([0 \cdots n/4 - 1]\) with \([n/4 \cdots n/2 - 1]\) and \([n/2 \cdots n + n/4 - 1]\) with \([n/2 + n/4 \cdots n - 1]\) and so on! The first \(n/z \ast z \ast 2\) first iterations serve to merge the leaves, the following \(n/z \ast z \ast 4\) serve to merge nodes at the first level in the virtual tree, and so on.

Note that the input size should divide \(z \ast z\) and be a power of 2. It is a limitation of our code at present time.

**Theorem 1** The number of cache misses of \(ZZZ\text{merge}\) is

\[
\frac{n}{(z \ast z)} + \frac{n}{z \ast z} \log \frac{n}{(z \ast z)}
\]

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

Note that we are able to compute a theoretical bound and that a bound is not easy to obtain (if it is numerically feasible) for others algorithms. We also mention that our bound can also be compared to the bound for sorting given by Vitter in [VS94a] and [VS94b]. This is an advantage of our algorithm.

5 Experimental results

Our experiments are accomplished on an Athlon(tm) XP 1800+ processor equipped 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 \(2 \times 137MHz\) (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. The random generator is part of our codes and freely available with all the implementations. For each input size, we run 30 trials and we measure the mean. The standard deviations are also available as graphs plotted by Gnuplot.

Our machine is also characterized by the following benchmarking results. SiSoftware Sandra program (Version2002.1.8.59)\(^6\) gives: Dhrystone ALU: 4043 MIPS; Whetstone FPU: 2088 MFLOPS. For the memory bandwidth benchmark, we obtain: RAM Bandwidth Int Buffered aEMMX/aSSE: 1829MB/s; RAM Bandwidth Float Buffered aEMMX/aSSE: 1777MB/s.

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. 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.

We have also experiments with GCC 3.2 and the following compilation flags: -O3 -fprefetch-loop-arrays -mmmx -m3dnow -march=athlon -fomit-frame-pointer -Wall. All the results are confirmed and we obtain better time results for \(ZZZ\text{merge}\).

5.1 Results for i586 compilation settings

Figures 1 and 2 reveals the cache behaviours of our tested algorithms. Fastsort behaviour is very poor. When we examine its source code, we find bitwise and and or instructions, and unrestricted bit shift instructions. 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. \(ZZZ\text{merge}\) 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 3 and 4 show the mean execution times of our tested algorithms. ZZZmerge and 3-way-quicksort are the two best: 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 1 and 2). The explanation is related to the number of instructions executed per cycle of the codes and we will comment this fact below.

The standard deviations of the execution times are also introduced on Figures 5 and 6. We observe in particular on Figure 6 that 3-way-quicksort has the highest bias: it is expected since in the worst case, Quicksort has a theoretical lower bound of \(\Theta(n^2)\) whereas

\(^6\)See: http://www.sisoftware.demon.co.uk/sandra/
Figure 1. Mean L1 data cache misses (Part 1).

Figure 2. Mean L1 data cache misses (Part 2).

Figure 3. Mean execution times (Part 1).

Figure 4. Mean execution times (Part 2).

Figure 5. Standard deviation of execution times (Part 1).

<table>
<thead>
<tr>
<th># int</th>
<th>Quicksort(ms)</th>
<th>ZZZmerge(ms)</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>0.45203</td>
<td>0.33952</td>
<td>25%</td>
</tr>
<tr>
<td>16384</td>
<td>2.04220</td>
<td>1.55541</td>
<td>24%</td>
</tr>
<tr>
<td>65536</td>
<td>9.51033</td>
<td>7.51966</td>
<td>21%</td>
</tr>
<tr>
<td>262144</td>
<td>46.2616</td>
<td>43.9141</td>
<td>5%</td>
</tr>
<tr>
<td>1048576</td>
<td>219.138</td>
<td>196.967</td>
<td>11%</td>
</tr>
<tr>
<td>4194304</td>
<td>912.462</td>
<td>867.897</td>
<td>5%</td>
</tr>
</tbody>
</table>

Table 1. Details of the two best execution times.
mergesort algorithms are in $\Theta(n \log n)$. Mergesort implementations, in particular $ZZZ$merge have the lowest bias.

The number of retired instructions executed by our codes are given on Figures 7 and 8. We observe that 3-way-quicksort execute approximately two times less instructions than $ZZZ$merge (ranked third). Fastsort is the second best and this result was expected since the algorithm "compress" in some way the integers in order to reduce the number of comparisons.

The AMD Athlon processor is an aggressive, out-of-order, three-way superscalar x86 processor. It can decode, issue up to three x86 instructions per cycle with a centralized instruction control unit and two independent instruction schedulers - an integer scheduler and a floating point scheduler.

Figure 6. Standard deviation of execution times (Part 2).

Figure 7. Retired instructions (Part 1).

Figure 8. Retired instructions (Part 2).

Figure 9. Ratio of the mean value of the measured retired instructions over the mean value of the execution time (normalized by the chip frequency).

note that the IPC of $ZZZ$merge is 45% more important than the IPC of 3-way-quicksort: the number of independent instructions is probably more important with $ZZZ$merge 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 say more.

The maximal value of the IPC 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.

![Graph](image)

**Figure 10. Ratio of mean values of retired instructions over the mean values of L1 data cache misses.**

Figure 10 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-quick sort decreases as the input size increases. The curves for ZZZmerge and 3-way-quick sort are very close for 4194304 integers. This observation explains why ZZZmerge finally beats 3-way quick sort in time. At least, we observe that Fame offers the best ratios but its IPC are not enough high 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.

### 5.2 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. Of course, we also assume that programs X and Y satisfy the same specification.

### 6 Conclusion

In this paper we investigated the cache effects between the first two level of memory hierarchy in the case of six in-core integer sorting algorithms. We devised a new mergesort algorithm that performs well in practice. Actually, the implementation is limited to input sizes that must verify a property related to the arity of the virtual tree used in the model. We plan to bypass the limitation in a very near future.

At least, we will examine if the principle of data compression “à la Fastsort” or 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?

### References


