Super Scalar Sample Sort

Peter Sanders[1] and Sebastian Winkel[2]

[1] Max Planck Institut für Informatik
Saarbrücken, Germany, `sanders@mpi-sb.mpg.de`
[2] Chair for Prog. Lang. and Compiler Construction
Saarland University, Saarbrücken, Germany, `sewi@cs.uni-sb.de`

Abstract. Sample sort, a generalization of quicksort that partitions the input into many pieces, is known as the best practical comparison based sorting algorithm for distributed memory parallel computers. We show that sample sort is also useful on a single processor. The main algorithmic insight is that element comparisons can be decoupled from expensive conditional branching using predicated instructions. This transformation facilitates optimizations like loop unrolling and software pipelining. The final implementation, albeit cache efficient, is limited by a linear number of memory accesses rather than the $\mathcal{O}(n \log n)$ comparisons. On an Itanium 2 machine, we obtain a speedup of up to 2 over `std::sort` from the GCC STL library, which is known as one of the fastest available quicksort implementations.

1 Introduction

Counting comparisons is the most common way to compare the complexity of comparison based sorting algorithms. Indeed, algorithms like quicksort with good sampling strategies [9, 14] or merge sort perform close to the lower bound of $\log n! \approx n \log n$ comparisons[3] for sorting n elements and are among the best algorithms in practice (e.g., [24, 20, 5]). At least for numerical keys it is a bit astonishing that comparison operations alone should be good predictors for execution time because comparing numbers is equivalent to subtraction — an operation of negligible cost compared to memory accesses.

Indeed, at least when sorting large elements, it is well known that memory hierarchy effects dominate performance [18]. However, for comparison based sorting of small elements on processors with high memory bandwidth and long pipelines like the Intel Pentium 4, there are almost no visible memory hierarchy effects [5]. Does that mean that comparisons dominate the execution time on these machines? Not quite because execution time divided by $n \log n$ gives about 8ns in these measurements. But in 8ns the processor can in principle execute around 72 subtractions (24 cycles $\times$ 3 instructions per cycle). To understand what the other 71 slots for instruction execution are (not) doing, we apparently need to have a closer look at processor architecture [8]:

[3] $\log x$ stands for $\log_2 x$ in this paper.

Data dependencies appear when an operand of an instruction B depends on the result of a previous instruction A. Such instructions have to be one or more pipeline stages apart.

Conditional branches disrupt the instruction stream and impede the extraction of instruction-level parallelism. When a conditional branch enters the first stage of the processor's execution pipeline, its direction must be *predicted* since the actual condition will not be known until the branch reaches one of the later pipeline stages. If a branch turns out to be *mispredicted*, the pipeline must be flushed as it has executed the wrong program path. The resulting penalty is proportional to the pipeline depth. Six cycles are lost on the Itanium 2 and 20 on the Pentium 4 (even 31 on the later 90nm generation). This penalty could be mitigated by speculative execution of both branches, but this causes additional costs and power consumption and quickly becomes hopeless when the next branches come into the way.

Of course, the latter problem is relevant for sorting. To avoid mispredicted branches, much ingenuity has been invested by hardware architects and compiler writers to accurately predict the direction of branches. Unfortunately, all these measures are futile for sorting. For fundamental information theoretic reasons, the branches associated with element comparisons have a close to 50 % chance of going either way if the total number of comparisons is close to $n \log n$.

So after discussing a lot of architectural features we are back on square one. Counting comparisons makes sense because comparisons in all known comparison based sorting algorithms are coupled to hard to predict branches. The main motivation for this paper was the question whether this coupling can be avoided, thus leading to more efficient comparison based sorting algorithms (and to a reopened discussion what makes a good practical sorting algorithm).

Section 2 presents super scalar sample sort (sss-sort) — an algorithm where comparisons are *not* coupled to branches. Sample sort [4] is a generalization of quicksort. It uses a random sample to find $k-1$ *splitters* that partition the input into *buckets* of about equal size. Buckets are then sorted recursively. Using binary search in the sorted array of splitters, each element is placed into the correct bucket. Sss-sort implements sample sort differently in order to address several aspects of modern architectures:

Conditional branches are avoided by placing the splitters into an implicit search tree analogous to the tree structure used for binary heaps. Element placement traverses this tree. It turns out that the only operation that depends on the result of a comparison is an index increment. Such a conditional increment can be compiled into a *predicated instruction*: an instruction that has a predicate register as an additional input and is executed if and only if the boolean value in this register is one. Their simplest form, *conditional moves*, are now available on all modern architectures. If a branch is replaced by these conditionally executed instructions, it can no longer cause mispredictions that disrupt the pipeline. Loop control overhead is largely eliminated by unrolling the innermost loop of $\log k$ iterations completely[4].

Data dependencies: Elements can traverse the search tree for placement into buckets independently. By interleaving the corresponding instructions for several elements, data dependencies cannot delay the computation. Using a two-pass approach, sss-sort makes it easy for the compiler to implement this optimization (almost) automatically using further loop unrolling or software pipelining. In the first pass, only an "oracle" — the correct bucket number — is remembered and the bucket sizes are counted. In the second pass, the oracles and the bucket sizes are used to efficiently place the elements into preallocated subarrays. This two-pass approach is well known from algorithms based on radix sort, but only the introduction of oracles make it feasible for comparison based sorting. The two-pass approach has the interesting side effect that the comparisons and the most memory intensive components of the algorithm are cleanly separated and we can easily find out what dominates execution time.

Memory Hierarchies: Sample sort is more cache efficient than quicksort because it moves the elements only $\log_k n$ times rather than $\log n$ times. The parameter k depends on several properties of the machine.

The term "super scalar" in the algorithm name, as in [1], says that this algorithm enables the use of instruction parallelism by getting rid of hard to predict branches and data dependencies. Were it not for the alliteration, "instruction parallel sample sort" might have been a better name since also superpipelined, VLIW, and explicitly parallel instruction architectures can profit from sss-sort.

Section 3 gives results of an implementation of sss-sort on Intel Itanium 2 and Pentium 4 processors. The discussion in Section 4 summarizes the results and outlines further refinements.

Related Work

So much work has been published on sorting and so much folklore knowledge is important that it is difficult to give an accurate history of what is considered the "best" practical sorting algorithm. But it is fair to say that refined versions of Hoare's quicksort [9, 17] are still the best general purpose algorithms: Being comparison based, quicksort makes little assumptions on the key type. Choosing splitters based on random samples of size $\Theta(\sqrt{n})$ [14] gives an algorithm using only $n \log n + \mathcal{O}(n)$ expected comparisons. A proper implementation takes only $\mathcal{O}(\log n)$ additional space, and a fallback to heapsort ensures worst case efficiency. In a recent study [5], `STL::sort` from the GCC STL library fares best among all quicksort competitors. This routine stems from the HP/SGI STL library and implements *introsort*, a quicksort variant described in [17]. Only for large n and only on some architectures this implementation is outperformed by more cache efficient algorithms. Since quicksort only scans an array from both ends, it has perfect spatial locality even for the smallest of caches. Its only disadvantage is

[4] Throughout this paper, we assume that k is a power of two.

Table 1. Different complexity measures for k-way distribution and k-way merging in comparison based sorting algorithms. All these algorithms need $n \log k$ element comparisons. Lower order terms are omitted. Branches: number of hard to predict branches; data dep.: number of instructions that depend on another close-by instruction; I/Os: number of cache faults assuming the I/O model with block size B; instructions: necessary size of instruction cache.

	mem. acc.	branches	data dep.	I/Os	registers	instructions
k-way distribution:						
sss-sort	$n \log k$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$3.5n/B$	$3\times$unroll	$\mathcal{O}(\log k)$
quicksort $\log k$ lvls.	$2n \log k$	$n \log k$	$\mathcal{O}(n \log k)$	$2\frac{n}{B} \log k$	4	$\mathcal{O}(1)$
k-way merging:						
memory [22, 12]	$n \log k$	$n \log k$	$\mathcal{O}(n \log k)$	$2n/B$	7	$\mathcal{O}(\log k)$
register [24, 20]	$2n$	$n \log k$	$\mathcal{O}(n \log k)$	$2n/B$	k	$\mathcal{O}(k)$
funnel $k'^{\log_{k'} k}$ [5]	$2n \log_{k'} k$	$n \log k$	$\mathcal{O}(n \log k)$	$2n/B$	$2k' + 2$	$\mathcal{O}(k')$

that it reads and writes the entire input $\log \frac{n}{M}$ times before the subproblems fit into a cache of size M.

Recent work on engineering sequential sorting algorithms has focused on more cache efficient algorithms based on divide-and-conquer that split the problem into subproblems of size about n/k and hence only need to read and write the input $\log_k \frac{n}{M}$ times. In the I/O model [2], with an omniscient cache of size M and accesses ("I/Os") of size B, we have $k = \mathcal{O}(M/B)$. In practical implementations, k has to be reduced by a factor $\Omega\left(B^{1/a}\right)$ for a-way associative hardware caches [15] (this restriction can in principle be circumvented [23]). On current machines, the size of the Translation Lookaside Buffer (TLB) is the more stringent restriction. This translation table keeps the physical address of the most recently accessed pages of virtual memory. Access to other pages causes *TLB misses*. TLBs are small (usually between 64 and 256) and for large n, TLB misses can get much more expensive than cache misses.

k-way Merging is a simple deterministic approach to k-way divide-and-conquer sorting and is the most popular approach to cache efficient comparison based sorting [18, 13, 22, 24, 20, 5]. Using a *tournament tree* data structure [12], the smallest remaining element from k sorted data streams can be selected using $\log k$ comparisons. Keeping this tournament tree in registers reduces the number of memory accesses by a factor $\log k$ at the cost of $2k$ registers and a cumbersome case distinction in the inner loop [22, 20, 24]. Although register based multi-way merging severely restricts the maximal k, this approach can be a good compromise for many architectures [20, 24]. This is another indication that most sorting algorithms are less restricted by the memory hierarchy than by data dependencies and delays for branches. Register based k'-way mergers can be arranged into a cache efficient k-way *funnel merger* by coupling $\log \frac{k}{k'}$ levels using buffer arrays [5]. By appropriately choosing the sizes of the buffer arrays, this algorithm becomes *cache-oblivious*, i.e., it works without knowing the I/O-model parameters M and B and is thus efficient on all levels of the memory hierarchy [7].

Function $sampleSort(e = \langle e_1, \ldots, e_n \rangle, k)$
  if n/k is "small" **then return** $smallSort(e)$ // base case, e.g. quicksort
  let $\langle S_1, \ldots, S_{ak-1} \rangle$ *denote a random sample of* e
  sort S // or at least locate the elements whose rank is a multiple of a
  $\langle s_0, s_1, s_2, \ldots, s_{k-1}, s_k \rangle := \langle -\infty, S_a, S_{2a}, \ldots, S_{(k-1)a}, \infty \rangle$ // determine splitters
  for $i := 1$ **to** n **do**
    find $j \in \{1, \ldots, k\}$ *such that* $s_{j-1} < e_i \leq s_j$
    place e_i *in bucket* b_j
  return $concatenate(sampleSort(b_1), \ldots, sampleSort(b_k))$

Fig. 1. Pseudocode for sample sort with k-way partitioning and oversampling factor a.

For integer keys, radix sort with $\log k$ bit digits starting with the most significant digit (MSD) can be very fast. This is probably folklore. We single out [1] because it contains the first mention of TLB as an important factor in choosing k and because it starts with the sentence "The compare and branch sequences required in a traditional sort algorithm cannot efficiently exploit multiple execution units present in currently available RISC processors." However, radix sort in general and this implementation in particular depend heavily on the length and distribution of input keys. In contrast, sss-sort does away with compare and branch sequences without assumptions on the input keys. For a recent overview of radix sort implementations refer to [19, 11]. Sss-sort owes a lot to MSD radix sort because it is also distribution based and since it adopts the two-pass approach of a counting phase followed by a distribution phase.

One goal of this paper is to give an example of how it can be algorithmically interesting and relevant for performance to consider complexity measures beyond the RAM model and memory hierarchies. Table 1 summarizes the complexity of k-way distribution and k-way merging for five different algorithms and five different complexity measures (six if we count comparisons). For the sake of this comparison, $\log k$ recursion levels of quicksort are viewed as a means of k-way distribution. We can see that sss-sort outperforms the other algorithms with respect to the conditional branches and data dependencies. It also fares very well regarding the other measures. The constant factor in the number of I/Os might be improvable. Refer to Section 4 for a short discussion.

2 Super Scalar Sample Sort

Our starting point is ordinary sample sort. Fig. 1 gives high level pseudocode. Small inputs are sorted using some other algorithm like quicksort. For larger inputs, we first take a sample of $s = ak$ randomly chosen elements. The *oversampling factor* a allows a flexible tradeoff between the overhead for handling the sample and the accuracy of splitting. In the full paper we give a heuristic derivation of a good oversampling factor. Our splitters are those elements whose rank in the sample is a multiple of a. Now each input element is located in the splitters and placed into the corresponding bucket. The buckets are sorted recursively and their concatenation is the sorted output.

Sss-sort is an implementation strategy for the basic sample sort algorithm. We describe one simple variant here and refer to Section 4 for a discussion of some possible refinements. All sequences are represented as arrays. More precisely, we need two arrays of size n. One for the original input and one for temporary storage. The flow of data between these two arrays alternates in different levels of recursion. If the number of recursion levels is odd, a final copy operation makes sure that the output is in the same place as the input. Using an array of size n to accommodate all buckets means that we need to know exactly how big each bucket is. In radix sort implementations this is done by locating each element twice. But this would be prohibitive in a comparison based algorithm. Therefore we use an additional auxiliary array, o, of n *oracles* – $o(i)$ stores the bucket index for e_i. A first pass computes the oracles and the bucket sizes. A second pass reads the elements again and places element e_i into bucket $b_{o(i)}$. This two pass approach incurs costs in space and time. However these costs are rather small since bytes suffice for the oracles and the additional memory accesses are sequential and thus can almost completely be hidden via software or hardware prefetching [8, 10]. In exchange we get simplified memory management, no need to test for bucket overflows. Perhaps more importantly, decoupling the expensive tasks of finding buckets and distributing elements to buckets facilitates software pipelining [3] by the compiler and prevents cache interferences of the two parts. This optimization is also known as *loop distribution* [16, 6].

Theoretically the most expensive and algorithmically the most interesting part is how to locate elements with respect to the splitters. Fig. 2 gives pseudocode and a picture for this part. Assume k is a power of two. The splitters are placed into an array t such that they form a complete binary search tree with root $t_1 = s_{k/2}$. The left successor of t_j is stored at t_{2j} and the right successor is stored at t_{2j+1}. This is the arrangement well known from binary heaps but used for representing a search tree here. To locate an element a_i, it suffices to travel down this tree, multiplying the index j by two in each level and adding one if the element is larger than the current splitter. This increment is the only instruction that depends on the outcome of the comparison. Some architectures like IA-64 have predicated arithmetic instructions that are only executed if the previously computed condition code in the instruction's predicate register is set. Others at least have a conditional move so that we can compute $j := 2j$ and then, speculatively, $j' := j + 1$. Then we conditionally move j' to j. The difference between such predicated instructions and ordinary branches is that they do not affect the instruction flow and hence cannot suffer from branch mispredictions.

When the search has traveled down to the bottom of the tree, the index j lies between k and $2k-1$ so that subtracting $k-1$ yields the bucket index $o(i)$. Note that each iteration of the inner loop needs only four or five machine instructions so that when unrolling it completely, even for $\log k = 8$ we still have only a small number of instructions. We also need only three registers for maintaining the state of the search (a_i, t_j, and j). Since modern processors have many more

$t := \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8}, \ldots \rangle$ //
for $i := 1$ **to** n **do** // locate each element
  $j := 1$ // current tree node := root
  repeat $\log k$ *times* // will be unrolled
    $j := 2j + (a_i > t_j)$ // left or right?
  $j := j - k + 1$ // bucket index
  $|b_j|$++ // count bucket size
  $o(i) := j$ // remember oracle

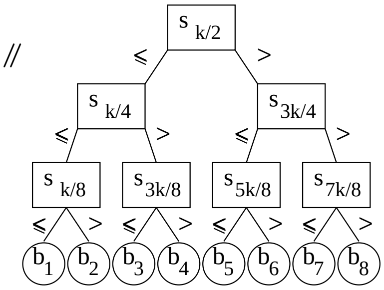

Fig. 2. Finding buckets using implicit search trees. The picture is for $k = 8$. We adopt the C convention that "$x > y$" is one if $x > y$ holds, and zero else.

physical renaming registers [5], we can afford to run several iterations of the outer loop in an interleaved fashion (via unrolling or *software pipelining*) in order to overlap their execution.

With the oracles it is very easy to distribute the input elements in array a to an output array a' that stores all the buckets consecutively and without any gaps. Assume that $B[j]$ is initialized to $\sum_{i<j} |b_i|$. Then the distribution is a one-liner:

for $i := 1$ **to** n **do** $a'_{B[o(i)]++} := a_i$ // (*)

This is quite fast in practice because a and o are read linearly, which enables prefetching. Writing to a' happens only on k distinct positions so that caching is effective if k is not too big and if all currently accessed pages fit into the TLB. Data dependencies only show up for the accesses to B. But this array is very small and hence fits in the first-level cache.

3 Experiments

We have implemented sss-sort on an Intel server running Red Hat Enterprise Linux AS 2.1 with four 1.4 GHz Itanium 2 processors and 1 GByte of RAM. We used Intel's C++ compiler v8.0 (Build 20031017) [6]. We have chosen this platform because the hardware and, more importantly, the compiler have good support for the two main architectural features we exploit: predicated instructions and software pipelining. We would like to stress, however, that the simple type of predicated instructions we need – the conditional moves – are supported by all modern processor architectures like the Intel Pentium II/III/4, the AMD Athlon and Opteron, Alpha, PowerPC and the IBM Power3/4/5 processors. Hence, with an appropriate compiler or manual coding, our algorithm should

[5] For instance, the Pentium 4 has 126 physical registers. The eight architected registers of IA-32 are internally mapped to these physical registers via renaming, allowing to resolve all false dependencies which are due to the limited number of architected registers [8]. In our case, this means that different iterations of the outer loop can be mapped to different registers, allowing them to be executed in parallel.

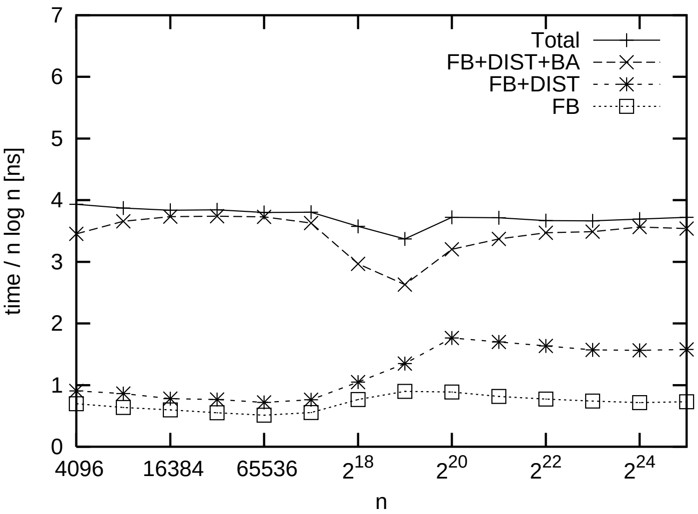

Fig. 3. Breakdown of the execution time of sss-sort (divided by $n \log n$) into phases. "FB" denotes the finding of buckets for the elements, "DIST" the distribution of the elements to the buckets, "BA" the base sorting routines. The remaining time is spent in finding the splitters etc.

work well on most machines (like on the Pentium 4, as demonstrated at the end of this section).

We have applied the `restrict` keyword from the ANSI/ISO C standard C99 several times to communicate to the compiler that our different arrays are not overlapped and not accessed through any other aliased pointer. This was necessary to enable the compiler to software pipeline the two major critical loops ("find the buckets" from Fig. 2, abbreviated "FB" in the following, and "distribute to buckets" ((*) in Sec. 2), abbr. "DIST").

Prefetch instructions and predication are utilized automatically by the compiler as intended in Section 2. The used compiler flags are "`-O3 -prof_use -restrict`", where "`-prof_use`" indicates that we have performed profiling runs (separately for each tested sorting algorithm). Profiling gives a 9% speedup for sss-sort. Our implementation uses $k = 256$ to allow byte-size oracles. sss-sort calls itself recursively to sort buckets of size larger than 1000 (cf. Fig. 1). It uses quicksort between 100 and 1000, insertion sort between 5 and 100, and customized straight-line code for $n \leq 5$.

Below we report on experiments for 32 bit random integers in the range $[0, 10^9]$. We have not varied the input distribution since sample sort using random sampling is largely independent of the input distribution.[6] The figures have the execution time divided by $n \log n$ for the y axis, i.e., we give something like "time

[6] Our current simple implementation would suffer in presence of many identical keys. However, this could be fixed without much overhead: If $s_{i-1} < s_i = s_{i+1} = \cdots = s_j$, $j > i$, change s_i to $s_i - 1$. Do not recurse on buckets $b_{i+1}, \ldots, b_j$ – they all contain identical keys.

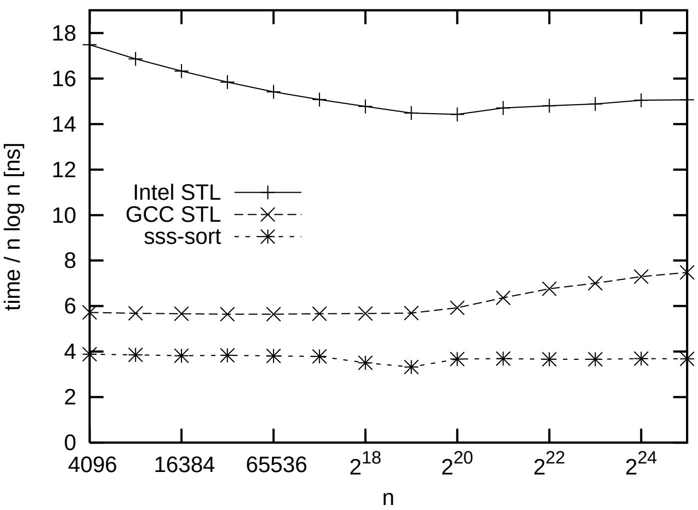

Fig. 4. Execution time on the Itanium, divided by $n \log n$, for the Intel implementation of `STL::sort`, the corresponding GCC implementation and for sss-sort. The measurements were repeated sufficiently often to require at least 10 seconds for all algorithms.

per comparison". For a $\Theta(n \log n)$ algorithm one would expect a flat line in the traditional RAM model of computation. Deviations from this expectations are thus signals for architectural effects.

Fig. 3 provides a breakdown of the execution time of sss-sort into different phases. It is remarkable that the central phases FB and DIST account only for a minor proportion of the total execution time. The reason is that software pipelining turns out to be highly effective here: It reduces the schedule lengths of the loops FB and DIST from 60 and 6 to 11 and 3 cycles, respectively (this can be seen from the assembly code on this statically scheduled architecture). Furthermore, we have measured that for relatively small inputs ($n \leq 2^{18}$), it takes on the average only 11.7 cycles to execute the 63 instructions in the loop body of FB, and 4 cycles to execute the 14 instructions in that of DIST. This yields dynamic IPC (instructions per clock) rates of 5.4 and 3.5, respectively; the former is not far from the maximum of 6 on this architecture.

The IPC rate of FB decreases only slightly as n grows and is still high with 4.5 at $n = 2^{25}$. In contrast, the IPC rate of DIST begins to decrease when the memory footprint of the algorithm (approx. 4n+4n+n=9n bytes) approaches the L3 cache size of 4 MB (this can also be seen in Fig. 3, starting at $n = 2^{17}$). Then the stores in DIST write through the caches to the main memory and experience a significant latency, but on the Itanium 2, not more than 54 store requests can be queued throughout the memory hierarchy to cover this latency [10, 21]: Consequently, the stall times during the execution of DIST increase (also those due to TLB misses). The IPC of its loop body drops to 0.8 at $n = 2^{25}$.

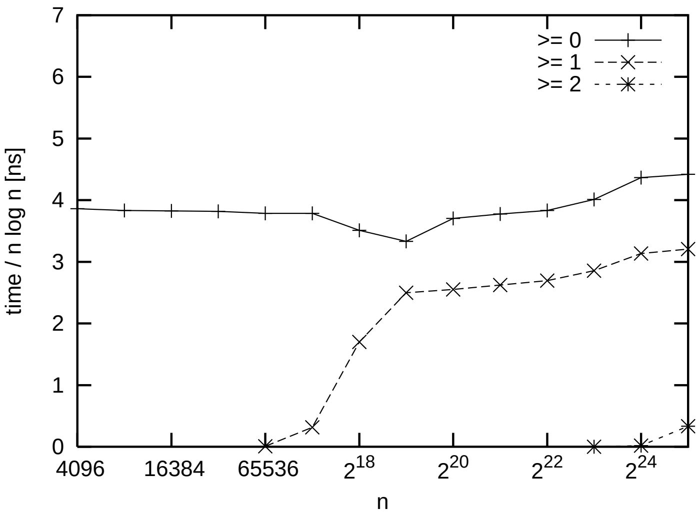

Fig. 5. The execution time spent in the different recursion levels of sss-sort. The total execution time (≥ 0) shows small distortions compared to Fig. 3, due to the measurement overhead.

However, Fig. 3 also shows — together with Fig. 5 — that at the same time the first recursion level sets in and reduces the time needed for the base cases, thus ameliorating this increase.

Fig. 4 compares the timing for our algorithm with two previous quicksort implementations. The first one is `std::sort` from the Dinkumware STL library delivered with Intel's C++ compiler v8.0. The second one is `std::sort` from the STL library included in GCC v3.3.2. The latter routine is much faster than the Dinkumware implementation and fared extremely well in a recent comparison of the best sorting algorithms [5]: There it remained unbeaten on the Pentium 4 and was outperformed on the Itanium 2 only by funnelsort for $n \geq 2^{24}$ (by a factor of up to 1.18).

As Fig. 4 shows, our sss-sort beats its closest competitor, GCC STL, by at least one third; the gain over GCC STL grows with n and reaches more than 100%. The flatness of the sss-sort curve in the figure (compared to GCC STL) demonstrates the high cache efficiency of our algorithm.

We have repeated these experiments on a 2.66 GHz Pentium 4 (512 KB L2 cache) machine running NetBSD 1.6.2, using the same compiler in the IA-32 version (Build 20031016). Since the latter does not perform software pipelining, we have unrolled each of the loops FB and DIST twice to expose the available parallelism. We have added the flag "`-xN`" to enable optimizations for the Pentium 4, especially the conditional moves. These were applied in FB as intended, except for four move instructions, which were still dependent on conditional branches. These moves were turned into conditional moves manually in order to obtain a branchless loop body.

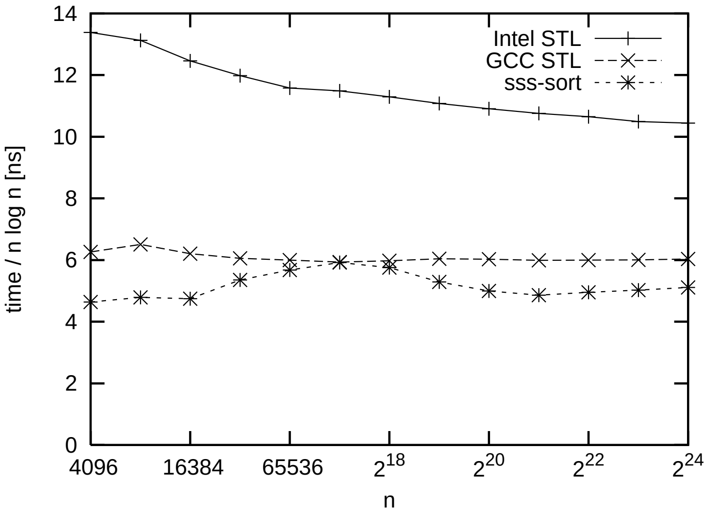

Fig. 6. The same comparison as in Figure 4 on a Pentium 4 processor.

The breakdown of the execution time, available in the full paper, shows that FB and DIST are here more expensive than on the Itanium. One reason for this is the lower parallelism of the Pentium 4 (the maximal sustainable IPC is about 2; we measure 1.1-1.8 IPC for the loop body of FB). Nevertheless, sss-sort manages to outperform GCC STL for most input sizes by 15-20% (Figure 6). The drop at $n = 2^{17}$ could probably be alleviated by adjusting the splitter number $k - 1$ and the base sorting routines.

4 Discussion

Sss-sort is a comparison based sorting algorithm whose $\mathcal{O}(n \log n)$ term contains only very simple operations without unnecessary conditional branches or short range data dependencies on the critical path. The result is that the cost for this part on modern processors is dwarfed by "linear" terms. This observation underlines that algorithm performance engineering should pay more attention to technological properties of machines like parallelism, data dependencies, and memory hierarchies. We have to take these aspects into account even if they show up in lower order terms that contribute a logarithmic factor less instructions than the "inner loop".

Apart from these theoretical considerations, sss-sort is a candidate for being the best comparison based sorting algorithm for large data sets. Having made this claim, it is natural to discuss remaining weaknesses: So far we have found only one architecture with a sufficiently sophisticated compiler to fully harvest the potential advantages of sss-sort. Perhaps sss-sort can serve as a motivating example for compiler writers to have a closer look at exploiting predication.

Another disadvantage compared to quicksort is that sss-sort is not inplace. One could make it almost inplace however. This is most easy to explain for the case that both input and output are a sequence of blocks, holding c elements each for some appropriate parameter c. The required pointers cost space $\mathcal{O}(n/c)$. Sampling takes sublinear space and time. Distribution needs at most $2k$ additional blocks and can otherwise recycle freed blocks of the input sequence. Although software pipelining may be more difficult for this distribution loop, the block representation facilitates a single pass implementation without the time and space overhead for oracles so that good performance may be possible. Since it is possible to convert inplace between block list representation and an array representation in linear time, one could actually attempt an almost inplace implementation of sss-sort.[7] If c is sufficiently big, the conversion should not be much more expensive than copying the entire input once, i.e., we are talking about conversion speeds of GBytes per second.

Algorithms that are conspicuously absent from Tab. 1 are distribution based counterparts of register based merging and funnel-merging. Register based distribution, i.e., keeping the search tree in registers would be possible but it would reintroduce expensive conditional branches and only gain some rather cheap in-cache memory accesses. On the other hand, cache-oblivious or multi-level cache-aware k-way distribution might be interesting (see also [7]). An interesting feature of sss-sort in this respect is that the two-phase approach allows us to precompute the distribution information for a rather large k (in particular larger than the TLB) and then implement the actual distribution exploiting several levels of the memory hierarchy. Let us consider an example. Assume we have 2^{23} bytes of L3 cache and want to distribute with $k = 2^{14}$. Then we distribute 2^{23} byte batches of data to a temporary array. This array will be in cache and its page addresses will be in the TLB. After a batch is distributed, each bucket is moved to its final destination in main memory. The cost for TLB misses will now be amortized over an average bucket size of 2^9 bytes. A useful side effect is that cache misses due to the limited associativity of hardware caches are eliminated [23, 15].

Acknowledgments. This work was initiated at the Schloss Dagstuhl Perspectives Workshop "Software Optimization" organized by Susan L. Graham and Reinhard Wilhelm. We would also like to thank the HP TestDrive team for providing access to Itanium systems.

References

1. R. Agarwal. A super scalar sort algorithm for RISC processors. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 240–246, 1996.
2. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[7] More details are provided in a manuscript in preparation.

3. V.H. Allan, R.B. Jones, R.M. Lee, and S.J. Allan. Software Pipelining. *Computing Surveys*, 27(3):367–432, September 1995.
4. G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 3–16, 1991.
5. G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. In *6th Workshop on Algorithm Engineering and Experiments*, 2004.
6. Carole Dulong, Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, John Ng, and David Sehr. An Overview of the Intel IA-64 Compiler. *Intel Technology Journal*, (Q4), 1999.
7. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Symposium on Foundations of Computer Science*, pages 285–298, 1999.
8. J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann, 3nd edition, 2002.
9. C. A. R. Hoare. Quicksort. *Communication of the ACM*, 4(7):321, 1961.
10. Intel. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, April 2003.
11. D. Jiminez-Gonzalez, J-L. Larriba-Pey, and J. J. Navarro. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, chapter Case Study: Memory Conscious Parallel Sorting, pages 171–192. Springer, 2003.
12. D. E. Knuth. *The Art of Computer Programming — Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.
13. A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *8th Symposium on Discrete Algorithm*, pages 370–379, 1997.
14. C. Martínez and S. Roura. Optimal sampling strategies in Quicksort and Quickselect. *SIAM Journal on Computing*, 31(3):683–705, June 2002.
15. K. Mehlhorn and P. Sanders. Scanning multiple sequences via cache memory. *Algorithmica*, 35(1):75–93, 2003.
16. Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, Kalifornien, 1997.
17. David R. Musser. Introspective sorting and selection algorithms. *Softw. Pract. Exper.*, 27(8):983–993, 1997.
18. C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC machine sort. In *SIGMOD*, pages 233–242, 1994.
19. N. Rahman. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, chapter Algorithms for Hardware Caches and TLB, pages 171–192. Springer, 2003.
20. A. Ranade, S. Kothari, and R. Udupa. Register efficient mergesorting. In *High Performance Computing — HiPC*, volume 1970 of *LNCS*, pages 96–103. Springer, 2000.
21. Reid Riedlinger and Tom Grutkowski. The High Bandwidth, 256KB 2nd Level Cache on an Itanium$^{\mathrm{TM}}$ Microprocessor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, San Francisco, February 2002.
22. Peter Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.
23. S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *11th ACM Symposium of Discrete Algorithms*, pages 829–838, 2000.
24. R. Wickremesinghe, L. Arge, J. S. Chase, and J. S. Vitter. Efficient sorting using registers and caches. *ACM Journal of Experimental Algorithmics*, 7(9), 2002.