Multithreaded Architectures and The Sort Benchmark

Philip Garcia
Dept. of Computer Science and Engineering
Lehigh University
Bethlehem, PA 18015, USA
philipgar@lehigh.edu

Henry. F. Korth
Dept. of Computer Science and Engineering
Lehigh University
Bethlehem, PA 18015, USA
hfk@lehigh.edu

1. INTRODUCTION

New computer architectures present many challenges to database-system designers. As main memory has increased in size and its latency has increased (in terms of cycles), much research has been focused on improving database-system performance to optimize for these new bottlenecks [17, 16, 3, 1, 4, 12, 8].

In this paper, we consider how algorithms designed specifically for newer architectures (featuring simultaneous multithreading (SMT)[19, 13], symmetric multiprocessors (SMP), advanced memory units, chip multiprocessors (CMP), etc.) can help database systems address the cache/memory performance gap.

We chose a simple, yet important problem: the sort benchmark. Because the traditional sort benchmark [2] is based on a disk sort, we propose a variant of that problem for in-memory sorting in Section 3. As our benchmark platform, we chose the Intel NetBurst[9] architecture as implemented in the Xeon and Pentium 4 processors.

Our results show the following:

- Our thread-aware sort allows SMT (as implemented on Intel's NetBurst architecture) to achieve a significant (20%-60%) improvement over the performance of a single-threaded application depending on what the threads are doing.
- The gains offered by SMT on data-intensive applications will increase as architecture trends widen the processor/memory gap.
- Using our thread-aware sort, SMT can allow a second thread to run at nearly full speed when the first thread is busy servicing cache/translation look-aside buffer (TLB) misses.
- Adding semaphores to prevent multiple large memory accesses from occurring simultaneously can provide a significant speedup to bandwidth limited problems.
- A standard merge sort can be more efficient than using a selection tree to run the merge sort due to its superior cache-locality.

Copyright is held by the author/owner.
First International Workshop on the Data Management on New Hardware (DaMoN 2005) June 12, 2005, Baltimore, Maryland, USA

2. THE ARCHITECTURE

We chose to run our experiments on the Intel NetBurst architecture because it includes a superscalar design, a deep (20+ cycle) pipeline, advanced instruction-handling units, a large data cache, simultaneous multithreading, and an advanced data-prefetch unit.

Our test machines were a 3GHz dual Xeon with a 533MHz front-side bus, 512KB L2 cache, and 1 MB L3 cache[1] and a 2.8GHz Pentium 4 with an 800MHz front-side bus and 1 MB of L2 cache. Our applications were compiled using gcc 3.3 (with -O2 -march=pentium4 -msse2). Our machines were running Debian GNU/Linux with kernel version 2.6.6. All of our results were obtained using the PAPI [5] library.

3. BENCHMARK: IN-MEMORY SORT

To evaluate the ability of SMT to speed data-intensive operations, we adapted the sort benchmark of [2] to an in-memory benchmark specified as follows

- The input set consists of 2.5 million records of 100 bytes each.
- Records have a 10-byte key.
- The records are in random order.
- The records are contiguous in the process' address space.
- The output is a permutation of the input in key-ascending order.
- The output is stored in a contiguous memory address space that may be distinct from the input's location.

Our benchmark differs from the classic one in two ways:

- No disk I/O is considered.
- The number of records is increased from 1 million to 2.5 million.

These changes reflect our focus on access of memory by the processor as a serious system-performance issue and an increase in data size to one comensurate with the size of modern main memories and caches. We chose a 250 MB relation size because it is both significantly larger than our caches (including L3 on the Xeon) and still fits comfortably in our 2 GB main memory.

[1]The L3 cache is inclusive so all data in the L2 cache is also included in the L3 cache.

Number of tuples	$n = 2,500,000$
Set size (bytes)	α
Tuples per set	$\beta = \alpha/14$
Number of sets	$\gamma = n/\beta$
Size of in-memory quicksort relations	$\delta = n * 14$

Table 1: Sorting variables.

The original benchmark has led to significant work at setting records in sorting performance (based on speed and/or cost)[14, 15]. Our goal here is not to set absolute performance records, but rather to examine the impact of algorithmic choices on performance for the chosen architectures.

4. ALGORITHM DESIGN

The items that we sort are 14-byte key-pointer pairs consisting of a 10-byte key and a 4-byte pointer to the record itself. Table 1 describes the variables we use.

Our sort is based on the design of the Alphasort[14]. This sort works by first splitting the relation up into sets of α bytes. Each thread then reads in all β tuples of a set and quicksorts key-prefix and pointers (note: we chose not to use key prefixes for simplicity because our focus here is on the effects of the architecture on algorithm performance rather than aiming for a sort performance record). Once all γ sets are sorted the Alphasort[14] uses a replacement-selection tree to merge the output records.

4.1 Modification to the Alphasort

We found that the quicksort/selection-tree mergesort was not cache-optimal for the benchmark data set. Therefore, we designed our algorithm to perform a quicksort on subsets of the input data whose size was one parameter of our experiments.

The resulting sorted runs were then merged via a series of mergesort operations (merging 2 lists at a time). While we implemented a selection tree (there was no point in implementing a replacement-selection tree because our algorithm was based entirely in-memory), it turned out to have worse performance than the "standard" mergesort algorithm.

These results may seem counter-intuitive as the regular mergesort requires $\sim \log_2 \gamma$ passes through all n tuples and a selection-tree requires only reading the n tuples once. The reasons for this discrepancy can be explained by the following:

Because our best results were obtained when we used quicksort set sizes between 1KB and 4KB (see Section 5.1 for details) the selection tree has $\frac{2{,}500{,}000 tuples}{73 tuples/node} = 34,247$ nodes in the bottom layer of the tree[2]. Because each of the bottom nodes points to a full cache line containing the next tuple (128 bytes), we need over 4MB of cache to hold the bottom layer of the tree fully in cache.

Due to the size requirements of the bottom layer of the tree alone, most of the tree is not in cache due to excessive capacity cache misses. This results in up to $\log_2 \gamma$ cache misses to choose the next output tuple. Because of these misses, the selection tree's performance could not match that of the standard merge sort (which is bandwidth limited and not latency limited). While the selection tree's performance approached that of the mergesort when we sufficiently large quicksort set sizes ($\sim$1MB) that we could fit the tree entirely within cache, this resulted in a suboptimal overall solution.

[2]We chose the values of 73 nodes/set because $73*14 \approxeq 1KB$.

Quicksort		
Set Size	Comparisons	Writes
1022	14,703,518	27,076,473
262,136	42,073,884	68,124792
Merge Sort		
Set Size	Comparisons	Writes
1022	37,652,500	37,652,500
262,136	17,652,500	17,652,500

Table 2: Number of comparisons and writes required for running the quicksort and merge sort when running a single thread.

4.2 Multithreading

We designed the multithreaded algorithm to work whether the two threads were on separate processors (SMP) or running on the same processor (SMT). We experimented with variations in how the workload is assigned to threads, initially using a round-robin approach to quicksorting and obtaining two sets to merge. This method works well during the quicksort phase, however during the merge phase, the final (and most expensive) merge has to be handled by a single thread. Thus, this algorithm wound up having poor performance.

To overcome this problem, we modified our algorithm so that a fixed key value is chosen to partition the data sets between the two threads. Under this method, each thread scans the input dataset adding a tuple to its set only if the key value is within the bounds supplied to the thread. In the final merge stage, the threads place their respective results in memory so as to create a single sorted result. Each thread can work independent of the other (with the SMT version getting a double benefit because the thread that is "behind" has the relevant portion of the dataset waiting for it in the processor's cache). While this new method slowed down the single-threaded version slightly, it gave a considerable speedup to the multithreaded version, as we shall see in Section 5.3.

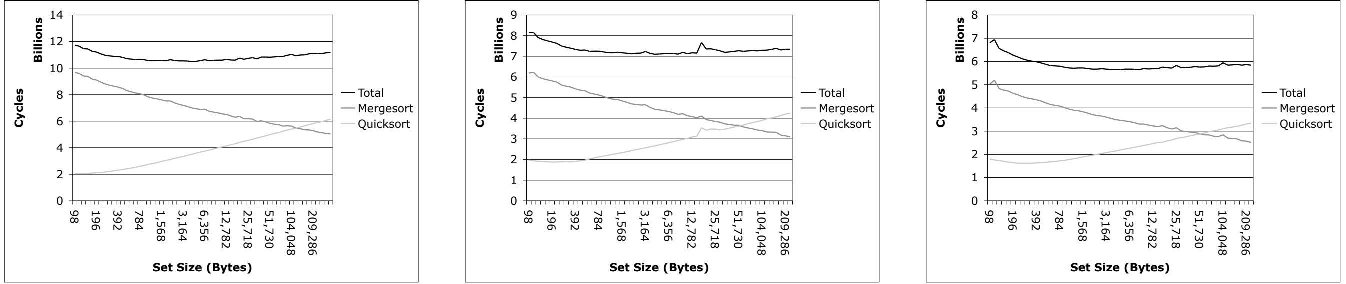
Ideally, the median value of the input dataset would be used to set the upper bound for one thread and the lower bound for the other. However determining the exact median requires a scan of the entire data set, so we used an approximation instead. If we assume that a good approximation of the median exists, each thread gets approximately half of the workload. A poor approximation not only imbalances the workload but, in the extreme may allow one thread to sort sufficiently far ahead of the other that we lose the cache-locality that is normally exhibited in accessing the data to be sorted.

5. ALGORITHM PERFORMANCE

In this section, we discuss the influence of the architecture on the performance of our algorithms.

5.1 Optimal Set Size for One Thread

Unlike the case of the Alphasort[14], the fastest quicksort runtime was not obtained when running with the set size equal to that of L2 (or L3) cache. This was due to the CPU-intensive aspects of quicksort. As Figure 1(a) shows, when we ran quicksort with a set size of 209,286 bytes, we



(a) 1 processor running 1 thread (b) 1 processor running 2 threads (SMT) (c) 2 processors running 2 threads (SMP)

Figure 1: Cycles to run the sort on the Xeon including the final merge.

spend almost 6 billion cycles to run all of the quicksorts. However, the graph also shows that the total time to sort and merge did not increase significantly, with the larger set sizes taking less than 10% longer than the "ideal" set size of between 1 and 4 KB.

This suggests that the speed of quicksort is approximately that of the merge sort despite the fact that the merge sort must read and write the entire relation of δ bytes $\lceil \log_2 \gamma \rceil$ times for the single-threaded version. When running the two-threaded version (see Figure 1(b)), the algorithm makes $\lceil \log_2(\gamma) \rceil - 1$ passes over the data because the quicksort initially splits the relation in half.

The fact that the graph is relatively level for set sizes between 1KB and 100KB suggests that memory bandwidth is not the primary bottleneck for the mergesort. If it were, the merge sort would not be very effective due to its large memory footprint. From Table 2, we can see that for a set size of 256KB the algorithm performs 7,370,366 extra comparisons and 21,048,319 more writes compared to a set size of 1KB. These extra comparisons and writes combined with mergesort's larger memory footprint lead to an interesting combination of the 2 sorts to get the best overall runtime.

5.2 The Final Merge

As we noted earlier the final merge consumed a large portion of total runtime. The reason this merge takes significantly longer than prior merges is due to its poor cache performance. While the other merges simply read two streams from memory and merge into a third stream of memory (which the NetBurst architecture's built-in hardware prefetch logic can easily detect[10]), the final merge must read from two streams of memory, then it must materialize the complete record by dereferencing the data pointer to the input tuple. It next writes the full 100-byte tuple to main memory.

While the three memory streams are all effectively prefetched in the final merge, materializing the data from the pointer results in poor cache performance because the initial relation is much larger than the size of our systems' caches and the hardware prefetch unit is incapable of determining the memory location to prefetch in advance.

We attempted to add simple software-prefetch logic to the final merge, however our results showed these to be inadequate at hiding the cache-miss latency. Upon examination, we realized that the Intel NetBurst architecture as implemented in the Xeon Northwood core is incapable of issuing software prefetch instructions if the prefetch results in a miss from its 64-entry[3] TLB. Because the Northwood's TLB could not address the full 250MB input relation our prefetch requests were ignored.

Although the newer Pentium 4 Prescott microprocessor has fixed this problem [10, 7], it is debatable how much prefetching could gain us due to the minimal amount of non-latent work that occurs in each iteration of the mergesort.

Due to the complexities in this final stage of the algorithm, we decided to report numbers for runs with the full final merge as well as runs where the final output is a listing (in sorted order) of key-pointer pairs. If this sort were just one operation in a pipeline of operations evaluating a database query, it could be beneficial to keep the key-pointer pairs and do further operations on the pair before dereferencing the pointer. Not materializing the relation would not only initially save time in sorting, but could potentially save time in the future by allowing future operations to have a much smaller memory footprint.

5.3 Multithreaded Execution

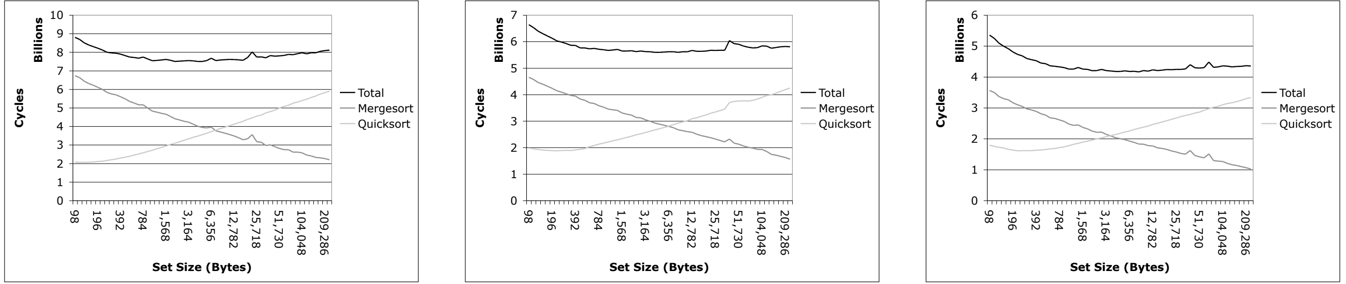
As Figures 1 and 3(b) show, we obtain a significant speedup by using multithreading. For 1-processor and 2-threads (1p2t), the algorithm sorts the relation about 48% faster than the single-threaded version with a speedup of 31% during the quicksort and 58% during the mergesort. The dual-processor (2p2t) version provides an even faster total speedup of 86% over the single-threaded version with a speedup of 60% during the quicksort and 100% during the merge sort. The mergesort is capable of a super-speedup (> 1) because the algorithm now requires one less pass over the data (due to initially splitting the relation in half).

When we chose not to materialize the final relation (but rather leave key-pointer pairs in sorted order), the SMT speedup was less, but the SMP speedup was approximately the same (See Figures 2 and 3(a)). For a more thorough explanation of these results see Section 5.4.

Table 3 shows the speedups obtained as a function of the quicksort set sizes. These values show that for small set sizes the multithreaded quicksort speedup is minimal. It also shows that as the set size increases so does the speedup achieved.[4] This effect is due to removing the contention

[3]Under Linux's default configuration each entry in the Xeon's TLB is capable of addressing 4KB of data.

[4]A greater speedup obtained for a given set size does not necessarily mean a lower overall runtime. See Figure 1.



(a) 1 processor running 1 thread (b) 1 processor running 2 threads (SMT) (c) 2 processors running 2 threads (SMP)

Figure 2: Cycles to run the sort without the final merge

Xeon without materializing final relation						
	Quick Sort		Merge Sort		Total	
Set Size	SMT	SMP	SMT	SMP	SMT	SMP
504	1.22	1.45	1.41	1.93	1.34	1.75
896	1.22	1.52	1.42	1.95	1.34	1.77
1,568	1.26	1.56	1.41	1.97	1.35	1.79
2,744	1.29	1.60	1.37	1.93	1.33	1.78
4,802	1.31	1.63	1.37	1.97	1.34	1.80
8,400	1.33	1.66	1.38	2.02	1.36	1.82
14,700	1.34	1.68	1.35	1.99	1.35	1.80
25,718	1.36	1.70	1.37	2.02	1.36	1.82
44,982	1.30	1.72	1.34	2.00	1.32	1.81
78,680	1.39	1.74	1.32	2.02	1.36	1.82
137,606	1.39	1.76	1.38	2.08	1.39	1.84
240,688	1.39	1.77	1.41	2.15	1.40	1.86

Table 3: Speedup over single threaded.

placed on the memory subsystem. When the quicksort set sizes are small, the time required to run the quicksort is minimal in comparison to the time required to retrieve the sets from main-memory. However, with larger sets this is no longer the case (even if the size of the quicksort set is greater than available cache quicksort has been shown to experience superior cache performance[14]).

While the SMT speedup reaches its maximum with a set size of about 70KB, the SMP algorithm reaches it at ~240KB.[5] The reason SMT reaches its peak with smaller set sizes is due to the fact that each thread is running slower than a thread running by itself on a standard SMP system. At these larger set sizes, the performance of the SMP system approaches that which we would expect from a multiprocessor system.

The performance of quicksort on the Pentium 4 was similar to that on the Xeon, however the speedup factor achieved for SMT was slightly less (1.24 vs 1.35 on the Xeon). This is due to the Pentium 4 having a faster memory speed and slower core clock speed. Because both threads spend less time accessing main memory on the Pentium 4, there is less speedup obtained by allowing one thread to read the set from main-memory while the other thread runs the quicksort. This correlates well with Tullsen, et al.'s results on concurrent integer and memory workloads [18].

[5]We do not show set sizes greater than 240KB in Table 3 because they are all lower than the maximum values.

Set Size	SMT Speedup	SMP Speedup
249,998	1.693	2.064
312,494	1.708	2.072
390,614	1.709	2.066
488,278	1.747	2.088
610,344	1.740	2.088
762,930	1.773	2.098
953,666	1.785	2.103
1,192,086	1.796	2.114
1,490,104	1.834	2.134
1,862,630	1.832	2.124

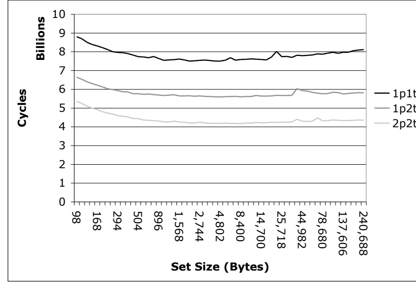
Table 4: Speedup for full merge on Xeon.

5.4 Hiding Cache stalls with SMT

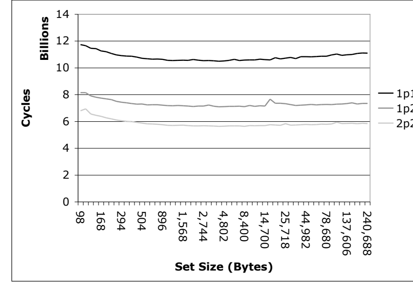
SMT has been shown to help alleviate both horizontal (limited instructional-level parallelism) and vertical (stalls) waste in modern superscalar processors[6, 19]. In the sort benchmark, SMT's primary benefit is to hide the vertical waste inherent in the workload by allowing another instruction stream to execute while the first stream is waiting on a cache miss. Previous study has also shown that all forms of multithreading are effective at handling vertical waste [11].

As Section 5.2 explained, the final merge in the mergesort is extremely expensive due to the poor cache locality found when dereferencing the tuple pointer. Because the memory accesses to obtain these data are too complex for the hardware prefetch mechanism to handle, these dereferences result in cache misses (or vertical waste). This causes the final merge to be extremely slow and taxing on the memory subsystem, as it must read in 90 bytes from "random" locations in main memory and then write them back into a known location (which should be effectively prefetched by the hardware prefetch units).

Our tests correlate well with previous research and showed that Intel's implementation of SMT (Hyper-Threading) was adept at hiding this latency [6, 19, 11]. Table 4 shows that by having two threads access memory at the same time, performance improved over 80% when compared to the single-threaded version. We used larger data sets so that we could compare primarily the run times for the final merge and not the previous merges. This is an extreme case of SMT's potential speedup, however it shows that SMT can have an enormous impact on runtimes when the majority of the processor's cycles are wasted waiting on cache misses.

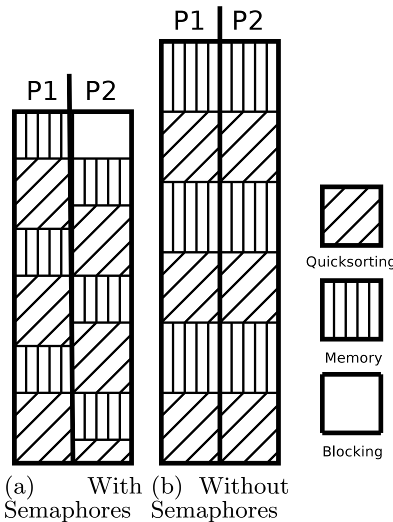


(a) Without final merge



(b) With final merge

Figure 3: Total number of cycles to run the full sort on the Xeon.



(a) With Semaphores (b) Without Semaphores

Figure 4: Two threads competing for compute and memory resources with and without semaphores.

6. SEMAPHORES FOR MEMORY ACCESS

We experimented with different options in the quicksort implementation to take maximal advantage of the underlying architectures. One interesting improvement that seems counter-intuitive is that the code ran faster when we inserted semaphores to control access to the main relation during retrieval of data for the quicksort. While this was not necessary for correctness (as the the threads access no tuple in common), when added, it forced main-memory access to occur from only one thread at a time.

As Table 5 shows, using semaphores to control access to main-memory speeds up the SMP quicksort significantly. This is because quicksort (which accesses only cache) cannot begin until the entire set has been selected from the relation in main memory. A quicksort can thus be forced to wait while the other processor consumes the entire main-memory bandwidth as illustrated in Figure 4.

Table 5 also shows that even SMT can take advantage of using semaphores to control the flow of the code. The SMT speedup is nowhere near as dramatic as the multiprocessor version ($\sim$ 3-4% on the Xeon and $\sim$ 3% on the Pentium 4).[6] In the SMT version, only one thread needs to access main-memory at a time because both threads in a processor share their data caches. Therefore the data should already reside in cache for the thread that is "further behind" in the relation, provided it is not too far behind.

Set Size	Single Threaded	SMT	SMP
1,022	0.978	1.041	1.266
2,044	0.985	1.044	1.216
4,088	0.991	1.041	1.189
8,190	0.996	1.037	1.161
16,380	0.998	1.033	1.135
32,760	0.995	1.031	1.113
65,534	0.997	1.032	1.104
131,068	0.998	1.027	1.093
262,136	0.995	1.014	1.058
524,286	0.999	1.017	1.082

Table 5: Quicksort speedup when using semaphores to control memory access on Xeon.

The reasoning behind the speedup when using semaphores on an SMT processor is therefore not directly due to the threads not accessing main-memory at the same time. The speedup is instead due to the sharing of functional units of the processor. By having one thread running the quicksort while the other is accessing main-memory, the thread in the quicksort can do more useful work because it has more functional units free for it to use. The difference in speeds here is minimal (as Table 5 shows), but that is to be expected since our savings are at the CPU level rather than the main-memory level. The Xeon's speedup is slightly larger than that of the Pentium 4. This makes sense because of the Pentium 4's faster memory subsystem. While the advantage of using semaphores is not as pronounced on SMT processors as it is with SMP, it does help show that thread synchronization on this level is beneficial.

This usage of semaphores shows that on modern multiprocessor systems an application should be aware of the memory-access patterns of its threads. If the timing of memory access' are known (such as in our case) the program can obtain a significant speedup by scheduling threads at the right time, or using synchronization constructs such as

[6]However, we shall explain later that architectural trends suggest larger gains in upcoming architectures that support more than just 2 threads.

semaphores. The necessity of these constructs grows as a system allows more concurrent threads to execute, because while total memory bandwidth remains relatively constant (with respect to the number of processors) the demand on the memory sub-system increases linearly with the number of processors.

7. CONCLUSION AND FUTURE WORK

While we took special consideration in the design of our application to take advantage of modern multithreading architectures, we left some optimizations for future work. In our sort, we are physically sorting the entire 10-byte keys. By initially sorting on only a small portion of the key (such as the word size on the architecture), we could speed up execution. We have also not experimented with other sorting routines that may experience superior performance over quicksort and merge sort. Our purpose in this paper was studying issues of algorithms and architectures, rather than aiming for an absolute speed record. Tuning for top speed remains future work.

We designed a system to take advantage of modern computing systems when using quicksort and merge sort to generate a final sorted relation. Our results show that special care should be considered in writing applications for modern multithreaded architectures. These new architectures offer many new ways to help conceal the main-memory bottleneck.

Our results have shown that NetBurst's implementation of SMT offers an effective way to hide some of the latency inherent in modern data-intensive applications. SMT does more to help close the memory/processor gap than hide cache misses, as it also allows for more efficient usage of the CPU's functional units. Our results also show that using a standard multithreaded programming approach can yield better performance than using a technique such as speculative slices to increase performance[20]. Comparing the gains on the P4 processor to those on the Xeon show that the P4 gains less from hyper-threading than the Xeon does due to waiting less time for the memory subsystem. This suggest that as the memory gap continues to widen, hyper-threading will give an even greater performance boost to programs, and become a necessity to maximize performance on faster processors. The potential is even greater for future architectures that will support more than 2 concurrent threads.

8. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277, 1999.

[2] Anonymous et al. A measure of transaction processing power. *Datamation*, 31(7):112–118, 1985.

[3] P. Bohannon, P. McIlroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *Proc. ACM SIGMOD International Conference on the Management of Data*, 2001.

[4] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *Proc. International Conference on Data Engineering*, 2004.

[5] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI for hardware performance monitoring on Linux systems. In *Conference on Linux Clusters: The HPC Revolution*, June 2001.

[6] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–18, September 1997.

[7] P. C. Garcia and H. F. Korth. Hash-join algorithms on modern multithreaded computer architectures. Technical Report LU-CSE-05-001, Lehigh University, 2005.

[8] G. Graefe and P.-A. Larson. B-tree indexes and CPU caches. In *Proc. IEEE International Confernece on Data Engineering*, pages 349–358, 2001.

[9] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, (Q1), 2001.

[10] Intel. *Intel Pentium 4 Processor Optimization*, 2001.

[11] H. Kwak, B. Lee, A. R. Hurson, S. Yoon, and W. Hahn. Effects of multithreading on cache performance. *IEEE Transactions on Computers*, 48(2):176–184, February 1999.

[12] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709–730, 2002.

[13] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, (Q1):4–15, 2002.

[14] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: A RISC machine sort. In *SIGMOD*, pages 233–242. ACM, May 1994.

[15] C. Nyberg, J. Gray, and C. Koester. A minute with Nsort on a 32P NEC Windows Itanium2 server. Technical report, Microsoft Research, April 2004.

[16] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *Proc. ACM SIGMOD International Conference on the Management of Data*, pages 475–486, 2000.

[17] A. Shatdal, C. Kant, and J. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of 20th International Conference on Very Lage Data Bases*, pages 510–524, 1994.

[18] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[19] D. M. Tullsen, S. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. ACM IEEE International Symposium on Computer Architecture*, pages 191–202, 1996.

[20] H. Wang, P. H. Wang, R. D. Weldon, S. M. Ettinger, H. Saito, M. Girkar, S. S. Liao, and J. P. Shen. Speculative precomputation: Exploring the use of multithreading for latency. *Intel Technology Journal*, (Q1):22–35, 2002.