Accelerating Database Operators Using a Network Processor

Brian T. Gold* bgold@cmu.edu
Anastassia Ailamaki* natassa@cmu.edu
Larry Huston† larry.huston@intel.com
Babak Falsafi* babak@cmu.edu

*Carnegie Mellon University
Pittsburgh, PA

†Intel Research Pittsburgh
Pittsburgh, PA

ABSTRACT

Database management systems (DBMSs) do not take full advantage of modern microarchitectural resources, such as wide-issue out-of-order processor pipelines. Increases in processor clock rate and instruction-level parallelism have left memory accesses as the dominant bottleneck in DBMS execution. Prior research indicates that simultaneous multithreading (SMT) can hide memory access latency from a single thread and improve throughput by increasing the number of outstanding memory accesses. Rather than expend chip area and power on out-of-order execution, as in current SMT processors, we demonstrate the effectiveness of using many simple processor cores, each with hardware support for multiple thread contexts. This paper shows an existing hardware architecture—the *network processor*—already fits the model for multi-threaded, multi-core execution. Using an Intel IXP2400 network processor, we evaluate the performance of three key database operations and demonstrate improvements of 1.9X to 2.5X when compared to a general-purpose processor.

1. INTRODUCTION

Memory access stalls dominate execution time in modern database management systems (DBMSs) [3, 6, 15, 19]. Exponential increases in processor clock rates and the relatively slow improvement of DRAM access latency only exacerbate the memory access bottleneck. Moreover, expending chip area and power on wide-issue superscalar microarchitectures or larger out-of-order instruction windows has produced diminishing returns for database workloads [15, 20].

Research has shown that memory access *latency* is the key bottleneck in DBMS performance, and current architectures are unable to expose multiple pending accesses to the memory system [15, 20]. Unlike most scientific applications, database operations exhibit sequences of dependent memory accesses, which limit the opportunity for speculation and out-of-order execution to issue parallel memory accesses in a single-threaded microarchitecture. Instead of focusing on instruction-level parallelism in a single thread, recent proposals show thread-level parallelism can significantly improve database performance by increasing the aggregate number of pending misses.

For example, simultaneous multithreading (SMT) hides single-thread memory access latency by interleaving the execution of threads on an aggressive out-of-order pipeline. Lo et al. showed a 3X improvement in OLTP throughput with a simulated eight-context SMT architecture [17]. However, the overhead of supporting multiple contexts on an aggressive microarchitecture limits the expansion of SMT architectures beyond four or eight threads.

In this paper, we increase memory-level parallelism further by using many simple processing cores with basic hardware support for low-overhead context switching. Unlike SMT, each core in our model executes a single thread in program order and switches contexts only on off-chip memory accesses. The area and power that would be devoted to an aggressive microarchitecture are instead used to integrate more cores on a single chip, increasing the thread parallelism by a factor of eight over the most aggressive SMT proposals [17].

Our evaluation makes use of an existing hardware platform, the *network processor*, which we find well-suited for executing many relational operators. Using a real hardware prototype, we demonstrate a 1.9X to 2.5X performance improvement over a 2.8GHz Pentium 4 Xeon on sequential scan, index range scan, and hash join operators. In addition to reporting performance improvements, we give insight on how to map common database operators to the network processor architecture.

This paper is organized as follows. In Section 2, we review work related to query execution on current and future hardware. In Section 3, we give an overview of network processor architecture and the IXP2400 in particular. In Section 4, we show how to map common database operators onto a network processor. In Section 5, we present the results of this study and conclude in Section 6.

2. RELATED WORK

In addition to architectural approaches to improving memory-level parallelism, database researchers have proposed a number of software techniques designed to reduce memory access latency. Cache-conscious data placement attempts to restructure the layout and placement of data to improve spatial and/or temporal locality [2, 9]. However, for many applications where access patterns are effectively

Proceedings of the First International Workshop on Data Management on New Hardware (DaMoN 2005); June 12, 2005, Baltimore, Maryland, USA.

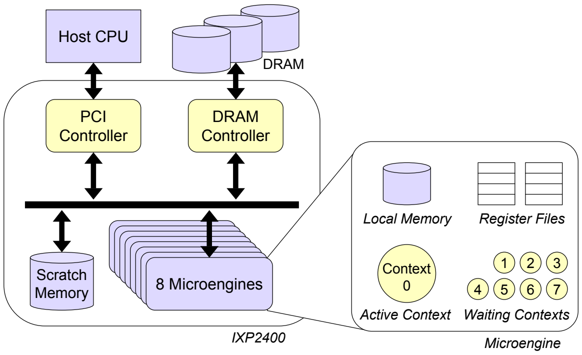

Figure 1: IXP2400 network processor features.

randomized (such as a hash join), memory access remains a performance bottleneck.

Software prefetching hides memory access latency by inserting timely prefetch instructions among existing data access references. Jump pointers [14, 18, 21] were proposed for linked data structures such as the hash table chains studied in this paper. The basic idea of a jump pointer is to automatically prefetch cache blocks from specially inserted pointers in existing data structures. In a hash table, for example, the 'next' pointer of each linked list element is prefetched as soon as a list element is fetched. Jump pointers may require extensive modifications to software data structures and may provide insufficient lookahead cache miss latency increases.

In [8], Chen et al. showed that with properly inserted prefetch instructions in a hash join algorithm, a performance increase of 2.0-2.9X could be achieved in simulation. This work studied two approaches to prefetching in a hash join and detailed how to partition the join code to insert timely prefetch instructions. Group prefetching exploits spatial parallelism to overlap multiple cache misses from independent probe tuples. Software-pipelined prefetching, by contrast, hides memory accesses in a pipeline of probe operations. Our hash join implementation is most similar to the group prefetching in [8], because we exploit inter-tuple parallelism across multiple hardware threads.

In addition to SMT, the architecture community has investigated several other multi-threading strategies. Similarly to the network processor model, switch-on-miss multi-threading switches contexts on infrequent events such as L2 cache misses [1, 10]. Prior evaluations used a limited number of thread contexts (four or fewer) and a single core, whereas current technology affords us 8 thread contexts per core and 8 or more processing cores per chip. Tera's MTA [4] implemented fine-grained multi-threading by storing 128 thread contexts in hardware and switching among ready threads every clock cycle. We are not aware of any evaluation of database workloads on the MTA architecture.

A number of recent papers have explored the use of graphics processors in the execution of database operators [5, 11, 22]. Modern graphics processors are exemplified by 8-16 parallel pipelines and very high-bandwidth memory interfaces, making them well-suited for some database operations. The principle disadvantage to graphics architectures remains their restrictive programming model, which limits control flow and prohibits random writes to memory.

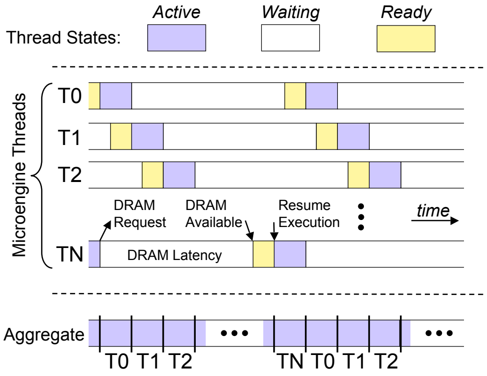

Figure 2: Thread execution on a microengine.

Consequently, a small subset of database operators have been mapped to current graphics architectures. In contrast, the network processor's general purpose programming model supports any relational operator that a conventional architecture can execute.

3. NETWORK PROCESSOR OVERVIEW

Many network applications process numerous independent streams of packets. The high bandwidth and parallelism required in these situations often exceeds what a general-purpose microprocessor can provide. The recent emergence of special-purpose architectures, termed network processors, addresses the need for high-throughput compute elements in such applications.

In this paper, we use the Intel IXP2400 network processor, shown in Figure 1 as a block diagram. Network processors contain a number of simple processing cores, termed microengines (MEs) in the Intel chips. The IXP2400 has 8 microengines clocked at 600 MHz, and each microengine has hardware support for 8 thread contexts. Switching thread contexts takes just 4 clock cycles. More information on the network processor hardware can be found in [12, 13].

The IXP2400 has several levels of memory, listed in Table 1. Unlike a general-purpose processor, the IXP2400 has no hardware-managed memory hierarchy. Data in local memory can only move to and from DRAM (or SRAM) with an explicit sequence of microengine instructions. This explicit control allows the programmer to control 'cached' data structures and know they will remain in low-latency storage. The software-managed memory makes programming more difficult, particularly for algorithms with complex control flow. Our experience shows that relatively few data structures need to move between levels of memory, and memory access is otherwise straightforward.

Table 1: IXP2400 Memory Hierarchy

Type	Size	Latency (cycles)
DRAM	1GB	> 120
SRAM	128MB	> 90
Scratchpad	16KB (on-chip)	> 60
Local Memory	2560B (per-ME)	3

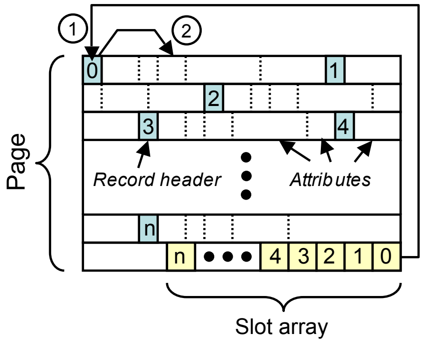

Figure 3: Record layout on a page.

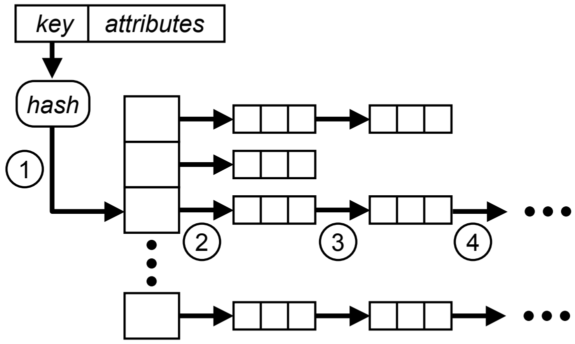

Figure 4: Probing a hash table.

3.1 Programming Model

Each microengine uses its low-overhead context switching to hide memory access latency from individual threads. Thread execution is non-preemptive—a thread explicitly releases control of the microengine, and the next thread is chosen round robin from the set of ready threads. Figure 2 illustrates this style of multi-threaded execution. Threads wait for specific hardware *signals* from the memory controller or other peripheral, indicating their operation has completed and the thread is ready to run.

Although the programming model is somewhat unusual, the basics of parallel programming still apply. When mapping database operators to threads in a network processor, we desire independent threads that do not require synchronization. Where synchronization is required, we place these threads on the same microengine, wherever possible. The non-preemptive programming model means no explicit synchronization is required between threads on a single microengine. Only one thread runs at any time, and the programmer controls when a context switch occurs. Intra-microengine coordination is accomplished through global registers, which can be accessed in a single cycle and incur no overhead.

In mapping common relational operators to the network processor, we find that data accesses can be statically mapped to one level in the memory hierarchy (list in Table 1). For example, we know that individual record attributes in a sequential scan will have no temporal locality and can therefore be kept in DRAM. Metadata, such as page headers or lock tables, should be kept close the processor in local memory or scratchpad space.

The next section gives an explanation of how to decompose common database operators onto network processor threads.

4. RUNNING DATABASE OPERATIONS

In this paper, we study three fundamental database operators: a sequential scan, a clustered or non-clustered index scan, and a hash join. Prior work [3] used a similar set of operations to evaluate DBMS execution behavior. The sequential scan models linear, contiguous access to one relation. The non-clustered index scan represents random, non-clustered access over a single table. Two-table access patterns are modeled through an equijoin, implemented using the most common and challenging operator—a hash join. The hash join models contiguous access to one table, and a dependent, random access to the second relation.

4.1 Sequential Scan

We model a sequential scan by iterating over all records on a set of pages, accessing one or more attributes in each record. We use slotted data pages with a layout derived from the Shore storage manager [7]. In general, records are variable length, and both a slot array and record header are used to find attribute data in the page. Figure 3 illustrates the sequence of operations in our sequential scan model. Reading an attribute requires at least accessing the slot array (labeled '1' in the figure), however accessing variable-length records requires a second indirection through the record header (labeled '2').

In the network processor implementation, each page is scanned by one microengine, with each thread fetching data from one record. By assigning pages to each microengine, we avoid costly synchronization overhead while tracking completed record indices. Pages are assigned round-robin to distribute work.

When a page is first accessed, the slot array is brought into the microengine's local memory to reduce subsequent access latency. Threads access tuple data in 64-byte chunks, which are stored in local registers and processed accordingly. The evaluations in this paper use attributes smaller than the maximum 64-byte DRAM transfer, so only one memory access is required.

4.2 Index Scan

Similarly to the sequential scan, we model an index scan by iterating over a set of pages, accessing one or more attributes in each record. In this case, however, we access a uniformly-distributed fraction of the records on each page. This model corresponds to a range scan where a list of pages is built from an index lookup, and then the pages are accessed sequentially. We assume the data access time dominates index lookup.

The network processor implementation of the index scan follows directly from the sequential scan discussed above. The key performance impact of the index scan is the overhead of accessing the slot array. When accessing all records on a page, this initial cost is amortized. For low selectivities, however, the slot array access approaches the tuple access in cost.

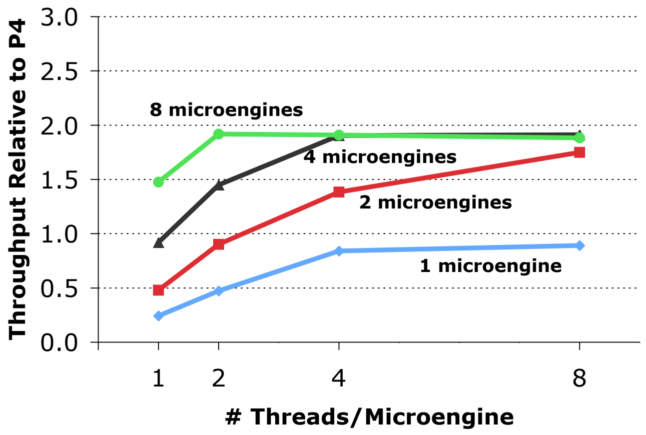

Figure 5: Relative throughput in sequential scan for varying amounts of parallelism, as compared to a 2.8GHz Pentium 4.

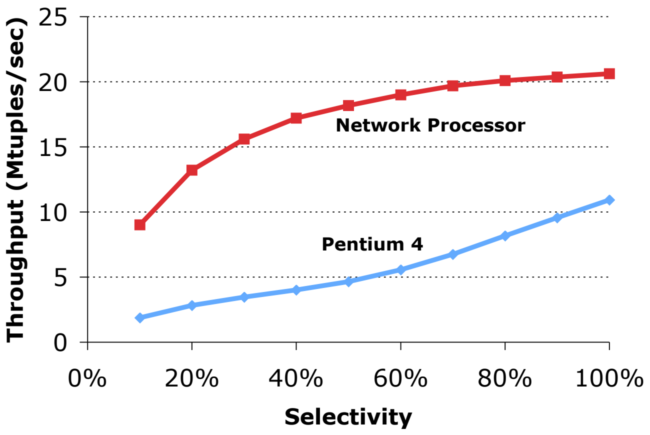

Figure 6: Throughput in index scan with varying selectivity. The network processor result uses 8 microengines with 8 threads per microengine.

4.3 Hash Join

We model the probe phase of a hash join, which for large relations will dominate time spent constructing the hash table itself. Figure 4 shows the sequence of data accesses when probing the hash table. Similar to the sequential scan, each page of the outer relation is assigned to one microengine in the network processor. Each thread in a microengine walks the hash table with a single record from the assigned page until a match is found.

To minimize hash bucket length, we place the hash table header in DRAM to allow it to grow beyond the network processor's limited on-chip storage. A general-purpose microprocessor with large on-chip cache has a distinct advantage here, as the frequently accessed bucket headers will likely be kept in cache. For modestly-sized outer relations, however, the network processor will offset the cost of accessing the header by overlapping the cost of pointer-chasing from many simultaneous probes.

5. EVALUATION

In this section, we describe the methodology used in this work and our experimental results. We study three fundamental operators used in nearly all DBMS queries: sequential scan, index scan, and hash join. We discuss the implications of these results on other, similar query operators.

5.1 Methodology

The experiments presented in this paper use a Radisys ENP-2611 development board, which places an Intel IXP2400 network processor and 256MB PC2100 DRAM on a PCI expansion card. The IXP2400 contains 8 microengines clocked at 600MHz, each with support for 8 thread contexts.

Network processor DRAM is separate from the main memory on the host computer. When a microengine thread accesses DRAM, it accesses the local DRAM on the expansion card. Integrating a network processor with a full DBMS on a host processor requires transferring data from system memory to DRAM on the network processor expansion card. We do not model these overheads in this paper.

We use a 2.8GHz Pentium 4 Xeon with 512KB L2 cache and 3GB of PC2100 DRAM as the general-purpose reference platform. We disable Hyper-Threading in our experiments, because these operators are bound by the number of miss handlers (four in the Pentium 4 used). Experiments using Hyper-Threading on this processor showed no performance improvements. For the Pentium 4, the database operators were coded in C and compiled using gcc-3.2 with `-O6` optimization. Experiments with the Intel C Compiler (icc) found no measurable difference. Where applicable, we report cache miss rates and branch predictor accuracy using the PAPI performance counter library.

Our evaluations use the TPC-H dbgen tool to generate realistic data tables using a scale factor of 0.25 (250MB ASCII dataset). The scan operators use the *orders* table, while the join operator uses *orders* as the inner relation and the *lineitem* relation as the outer. We chose a 250MB dataset so that all data would fit in memory on the network processor expansion card.

5.2 Sequential Scan

We sequentially scan the records of the TPC-H *orders* table and extract customer keys from each record. Figure 5 shows the relative throughput for varying levels of thread parallelism, as compared to the 2.8GHz Pentium 4 implementation. We observe that an aggregate total of four threads are required to (nearly) match the Pentium 4 throughput. More than four threads on a single microengine produces diminishing returns as the bottleneck becomes instruction execution in the microengine's simple pipeline. That is, threads remain in the *ready* state waiting for another thread to yield control of the pipeline.

The number of miss handlers constrains the Pentium 4 throughput, as only four L1D misses can be outstanding at any time. Because TPC-H relations consist of variable-length records, the hardware stride prefetcher reduces L2 misses by just 10%. Each record access incurs one off-chip miss; however, the sequential scan consumes just over one-third of the available memory bandwidth with hardware prefetching turned off. The Pentium 4 is unable to expose

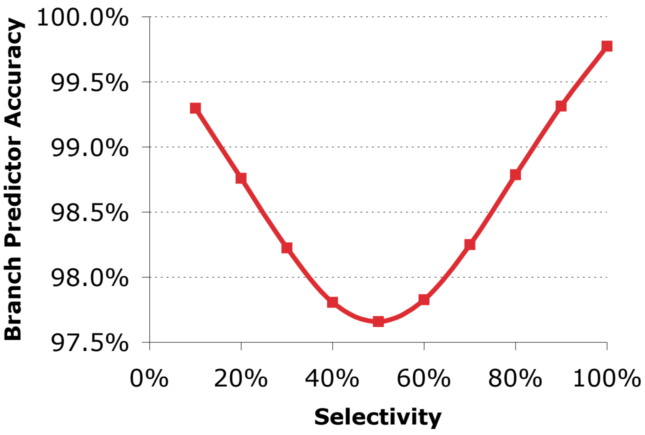

Figure 7: Branch predictor accuracy for index scan.

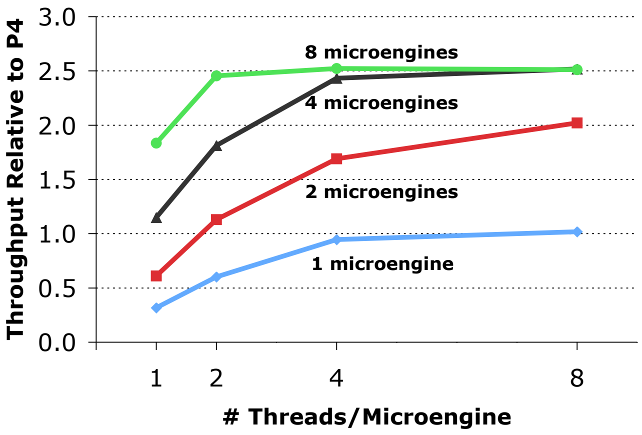

Figure 8: Relative throughput in probe of hash table for varying amounts of parallelism, as compared to a 2.8GHz Pentium 4.

sufficient memory-level parallelism to hide the long off-chip access latency.

The peak speedup of 1.9X is obtained with an aggregate of 16 threads on the network processor. Note, however, that the bottleneck with 16 threads is now the memory controller, which limits the number of simultaneous outstanding accesses to 16. Adding more than 16 threads produces no improvement, whether through more microengines or increasing the number of threads per microengine. Alleviating this bottleneck requires a more aggressive memory controller or increasing the number of memory controllers and distributing accesses across them. The Intel IXP2800 takes the latter approach, using three Rambus DRAM channels and distributing all DRAM requests across the three channels.

5.3 Index Scan

The index scan also iterates over the *orders* table, but now tuples are selected at random within each page. Figure 6 illustrates the results, where the network processor throughput is obtained from 8 microengines with 8 threads per microengine. Results were identical using any configuration providing at least 16 threads.

We see a sharp drop in throughput on the network processor as selectivity decreases, because of the relative increase in overhead of accessing the slot array. As fewer tuples are fetched from a page, the initial slot array access becomes more expensive. The shape of the Pentium 4 curve differs due to branch misprediction effects. As selectivity approaches 50%, branch mispredictions peak, as illustrated in Figure 7. The cost of instruction rollback on a misprediction is not the primary source of performance overhead. Rather, the lost opportunity for look-ahead and prefetching of future tuples is the major overhead in the non-clustered range scan. Prior work indicates a significant fraction of time in DBMS execution is due to branch mispredictions [3].

5.4 Hash Join

We model a ‘simple’ hash join where the inner relation fits in a single hash table in memory. The results here are applicable to partition-based hash joins, such as GRACE [16], which repetitively execute simple hash joins over each partition. In our model, elements in the hash table contain order keys from the TPC-H *orders* relation. We use 32K hash buckets to restrict the average bucket depth to 2.08. This configuration was intentionally biased towards the Pentium 4 to make our analysis conservative. Using fewer buckets increases the length of each bucket chain and improves the network processor's relative performance. Each cell in the hash bucket chain contains four $(key, pointer)$ pairs to minimize memory accesses in both the network processor and Pentium 4 implementations.

Figure 8 shows the relative join throughput as the number of microengines and threads varies. The data-dependent branches and cache misses on the Pentium 4 implementation have a significant impact on single-thread performance. Given a total of 4 threads, the network processor meets or outperforms the Pentium 4. As in the sequential and index scans, the limited parallelism in the memory controller of the IXP2400 limits speedup beyond 2.5X.

The thread-level parallelism and memory-controller bottleneck on the IXP2400 account for the similarity in Figures 5 and 8. In these operations, the fundamental performance improvement comes from issuing more parallel requests to memory on the network processor. The relative performance in the hash join is higher than the sequential scan because the data-dependent, random memory access patterns hamper the Pentium 4's ability to expose parallel memory accesses.

6. CONCLUSIONS

Memory access stalls dominate DBMS execution time, and the continued scaling of processor clock frequency only exacerbate the problem. We demonstrate that existing network processors, with their multi-core, multi-threaded architecture, can expose more parallel accesses to memory than a conventional, single-threaded processor. We study the implementation of three fundamental database operators: sequential scan, index scan, and hash join. We demonstrate speedups ranging from 1.9X for the sequential scan to 2.5X for the hash join. With selectivity less than 50%, the network processor outperforms a 2.8GHz Pentium 4 by more than 3X in an index scan.

7. ACKNOWLEDGEMENTS

The authors would like to thank Jared Smolens for help with early versions of this work, Minglong Shao for help with Shore, and members of the Carnegie Mellon Impetus group and the anonymous reviewers for their feedback on earlier drafts of this paper. This work was partially supported by grants and equipment donations from Intel corporation and a graduate fellowship from the Department of Defense.

8. REFERENCES

[1] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, pages 48–61, June 1993.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001*, pages 169–180, 2001.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999*, pages 266–277, 1999.

[4] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Proceedings of the 6th ACM International Conference on Supercomputing*, July 1992.

[5] N. Bandi, C. Sun, A. E. Abbadi, and D. Agrawal. Hardware acceleration in commercial databases: A case study of spatial operations. In *Proceedings of the 30th International Conference on Very Large Data Bases, August 31 - September 3 2004*, pages 1021–1032, 2004.

[6] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999*, pages 54–65, 1999.

[7] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, May 24-27, 1994.*, pages 383–394, 1994.

[8] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *Proceedings of the 20th International Conference on Data Engineering, March 30 - April 2, 2004*, pages 116–127, 2004.

[9] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Making pointer-based data structures cache conscious. *IEEE Computer*, 33(12):67–74, 2000.

[10] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu. Evaluation of multithreaded uniprocessors for commercial application environments. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, pages 203–212, 1996.

[11] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 13-18, 2004*, pages 215–226, 2004.

[12] Intel Corporation. *Intel IXP2400 Network Processor: Hardware Reference Manual*. Intel Press, May 2003.

[13] E. J. Johnson and A. R. Kunze. *IXP2400/2800 Programming: The Complete Microengine Coding Guide*. Intel Press, 2003.

[14] M. Karlsson, F. Dahlgren, and P. Stenström. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture, 8-12 January 2000*, pages 206–217, 2000.

[15] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance characterization of a quad pentium pro smp using oltp workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture, June 27 - July 1, 1998*, pages 15–26, 1998.

[16] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Comput.*, 1(1):63–74, 1983.

[17] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture, June 27 - July 1, 1998*, pages 39–50, 1998.

[18] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1-5, 1996*, pages 222–233, 1996.

[19] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a join? Dissecting cpu and memory optimization effects. In *Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000*, pages 339–350, 2000.

[20] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, October 3-7, 1998*, pages 307–318, 1998.

[21] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture, May 2-4, 1999*, pages 111–121, 1999.

[22] C. Sun, D. Agrawal, and A. E. Abbadi. Hardware acceleration for spatial selections and joins. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, June 9-12, 2003*, pages 455–466, 2003.