Realizing Parallelism in Database Operations: Insights from a Massively Multithreaded Architecture

John Cieslewicz*
Columbia University
johnc@cs.columbia.edu

Jonathan Berry†
Sandia National Laboratories
jberry@sandia.gov

Bruce Hendrickson†
Sandia National Laboratories
bahendr@sandia.gov

Kenneth A. Ross‡
Columbia University
kar@cs.columbia.edu

ABSTRACT

A new trend in processor design is increased on-chip support for multithreading in the form of both chip multiprocessors and simultaneous multithreading. Recent research in database systems has begun to explore increased thread-level parallelism made possible by these new multicore and multithreaded processors. The question of how best to use this new technology remains open, particularly as the number of cores per chip and threads per core increase. In this paper we use an existing massively multithreaded architecture, the Cray MTA-2, to explore the implications that a larger degree of on-chip multithreading may have for parallelism in database operations. We find that parallelism in database operations is easy to achieve on the MTA-2 and that, with little effort, parallelism can be made to scale linearly with the number of available processor cores.

1. INTRODUCTION

Increased hardware support for multiple threads, in the form of chip multiprocessors and simultaneous multithreading, has emerged as a new trend in hardware design. The question for database researchers is this: How best can we use this increasing multithreading capability to improve database performance in a manner that scales well with machine size? Though the current state of the art in commodity processor design is only a handful of processor cores on a chip, each supporting a few thread contexts at most, this paper looks forward and targets an extreme case: database design on a system with hardware support for thousands of thread contexts. We conduct our experiments on existing hardware, the Cray MTA-2 [8]. We find that the large amount of multithreading offered by the MTA-2 alleviates the struggle that developers now face in overcoming the memory bottleneck present in most systems.

In recent years, the performance bottleneck challenging database researchers has shifted from disk access to memory access [2]. Steadily falling memory prices coupled with the arrival of 64-bit processors have resulted in commodity database systems where most operations can be done "in-memory," thus avoiding I/O bottlenecks, but increasing the relative cost of memory access. In this paper we will focus on these in-memory database operations with respect to decision support operations.

Database workloads are data intensive, rather than computationally intensive. Therefore, they do not benefit from faster chip clock rates as much as scientific workloads because the frequent memory accesses cause the processor to stall [2]. To combat this inefficiency, much recent research has focussed on cache-conscious data structures and algorithms for database operations (e.g., [16, 18, 19, 22]). This research has improved instruction level parallelism through intelligent access patterns that employ nonblocking memory accesses to overlap computation with memory operations without stalling the processor.

The hardware trend toward multithreading for improved performance, including multicore and multithreaded chip designs [1, 11, 13, 17, 21, 23, 24] may provide the means of mitigating the weaknesses of the memory hierarchy for memory intensive workloads. The ease with which multithreading techniques scale as the amount of on-chip threading increases will be of paramount importance to all applications, including databases. Today a dual-core chip is typical, but chips featuring many cores, some with simultaneous multithreading in each core, are entering the market [13, 24]. As the amount of on-chip, shared-memory multithreading hardware support increases, it is up to researchers in applications such as databases to develop effective techniques to take advantage of more thread contexts and realize the performance benefit that this trend has to offer.

We use the MTA-2 to gain insight into executing database operations on a system with many threads. Though the MTA-2 is not multicore, its high degree of shared-memory multithreading is a good proxy for a multicore architecture with many thread contexts per core. The MTA-2 has no cache. Instead, it uses parallelism to tolerate memory latency. The MTA-2 has been shown to be effective at mitigating the effects of high latency memory operations in several domains [3, 4]. Such a system design and its accompanying programming model also provide numerous advantages

*Funding provided by a U.S. Department of Homeland Security Fellowship administered by Oak Ridge Institute for Science and Education.

†Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for US DOE under contract DE-AC-94AL85000.

‡Supported by NSF Grant IIS-0534389.

Proceedings of the Second International Workshop on Data Management on New Hardware (DaMoN 2006) June 25, 2006, Chicago, Illinois, USA.

CPU speed	220 MHz
Maximum system size	256 processors
Threads per processor	128
Context switch cost	1 cycle
Maximum memory size	1 TB (4 GB/P)
Network topology	Modified Cayley
Network injection rate	220 MW/P

Table 1: MTA-2 System characteristics

to database management system implementers, including a familiar high-level programming model, and simpler memory and thread management. In the subsequent sections, we will describe the implementation of core database operators on the MTA-2, provide an experimental evaluation of these operators, and discuss design factors important to scalable multithreaded database operations.

2. THE CRAY MTA-2

This section provides a high level description of the MTA-2 system and some of the implications these system characteristics have on database operator design. For a more detailed treatment of the MTA-2 system see [7, 8].

The MTA-2 processor uses multithreading to overlap computation with high latency memory operations. Each processor has 128 thread contexts and the cost of switching between threads is one cycle. In fact, the 21-stage MTA-2 pipeline is designed so that a context switch must occur each cycle because only one instruction from a particular thread may be in the pipeline at a time. The obvious implication is that serial code will execute very slowly; therefore, multithreaded programming is required whenever possible. Every cycle, the processor examines the 128 thread contexts and chooses a thread to issue from in a fair manner. If no thread has an instruction that can begin execution, the processor must issue a "phantom" or "nop" instuction. The new Sun T1 processors use a similar multithreading technique in each core, switching among four threads on each clock cycle to hide the latency of L1 cache accesses [24]. Though the MTA-2's design targets higher latency main memory accesses, the emphasis on parallelism is similar. Each MTA-2 processor is capable of issuing a memory reference every clock cycle, and the memory system is similarly capable of satisfying a memory request from each processor in each cycle. A key trait of the MTA-2 is lightweight, fine-grained synchronization that enables parallelism by allowing the efficient locking of single words of memory as well as atomic, in-memory increment operations.

Table 1 shows characteristics of the MTA-2 [8]. Because there is no cache, all memory is distant and accessing that memory is a high latency operation. Rather than combatting the long latency, the MTA-2 memory system is designed to have a high bandwidth. Though each memory operation takes many clock cycles to complete, many of them can be in flight at the same time and the high degree of on-chip multithreading overlaps computation with this latency.

Because the Cray MTA-2 uses multithreading instead of a cache to overcome memory latency, programmers can focus on parallelism, rather than cache-conscious algorithms and data structures such as CSS-Trees, CSB$^+$-Trees, and column-wise or decomposed storage models [15, 16, 18, 19, 22]. Though the MTA-2 hashes the physical address space to distribute memory references among the memory units, the important characteristic is that its multithreaded design makes shared memory parallelism scale well. The explicit partitioning and message passing required by shared-nothing parallel machines is avoided.

Database operations are inherently parallel, that is, processing an individual tuple rarely directly involves other tuples from the relation. This observation is not novel and existing parallel database systems take advantage of parallelism in database operations at different levels of granularity. Parallelism on the MTA-2, however, is different. Whereas the aforementioned parallel systems are usually *shared-nothing*, which means that each processor has its own memory and data is explicitly partitioned among the nodes, the MTA-2 is an example of a *shared-everything* system since each processor can directly access all of the memory in the system. DeWitt and Gray [9] argue that shared-nothing parallel systems work best for database workloads. With the advent of the chip multiprocessor, it is important to revisit the shared-everything architecture and identify programming models and techniques which will result in the best performance when even a single chip supports a high degree of multithreading.

3. EXPERIMENTAL WORKLOAD

We conducted experiments involving a number of typical database operations to investigate their performance on the MTA-2. For the data layout we used a *Decomposed Storage Model* (DSM), also known as column-wise record storage [16, 22]. Record layout for the MTA-2 should be irrelevant since it has a uniform memory access cost and no cache. For completeness, we repeated a few experiments with row-based storage and found the results to be similar.

3.1 Select

We explored the scaling properties of a selection operation on the MTA-2. These results are briefly discussed in Section 4.1, but are also described in [7] as a primer on MTA-2 parallelism.

3.2 Join

The join operator combines two relations based on some comparison of attributes. There are many techniques for performing a join; the two algorithms we explored on the MTA-2 are the index join and the hash join.

The index join employs an index structure, such as a B$^+$-Tree, to perform the join. The "inner" relation is indexed and then probed by the "outer" relation.

The pointer following required to traverse an index structure such as a B$^+$-Tree performs badly on conventional processors because there is little spatial locality, assuming that either the index is larger than the cache or other work competes for cache resources. Also, loading the next node in an index traversal is dependent on the outcome of loading earlier tree nodes. Therefore, single-threaded execution stalls often during an index probe. We expect an index join using a B$^+$-Tree data structure to exhibit good processor utilization on the MTA-2 because the the MTA-2 will not stall during these data dependent loads.

The hash join builds a hash table on the inner relation as opposed to using a pre-existing index, as in the index join. The outer table then probes the hash table to compute the join. Examining a location in the hash table is a random access to memory that is unlikely to benefit from

```
SELECT(Tuple* in, Tuple* out,
    int n, int test, int k){
  int indexes[k];
  Tuple *outs[k];
  for(int i = 0; i < k; i++){
    indexes[i] = 0;
    outs[i] = out + i*n/k;
  }

  #pragma mta assert parallel
  for(int i = 0; i < n; i ++)
    if(predicate(in[i].value, test)){
      int outIndex = i % k;
      int tupIndex = int_fetch_add(&indexes[outIndex], 1);
      outs[outindex][tupIndex] = in[i];
    }
}
```

Figure 1: A simplified implementation of select with implicitly partitioned output. The first loop initializes the k output indexes, while the second loop performs the select.

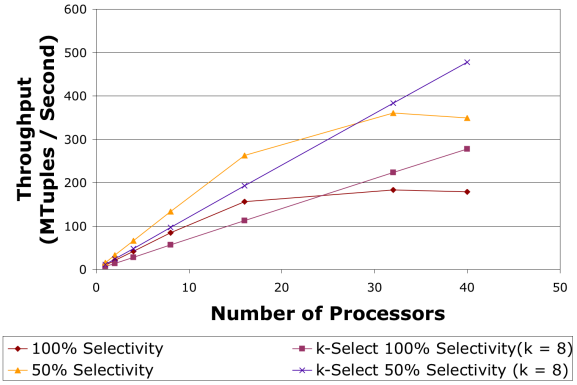


Figure 2: A comparison of selection throughput on the MTA-2 with (denoted *k-select*) and without implicit output partitioning. k is the number of output partitions. Note that the partitioning enables better scaling.

a cache. Following a chain of items that hash to the same location can require data dependent loads. Partitioned join algorithms improve the performance of the hash join, but with the MTA-2 we aim to avoid this partitioning step.

4. EXPERIMENTAL RESULTS

DeWitt and Gray identify two metrics by which to judge a parallel system: (1) *speedup* and (2) *scaleup* [9]. The goal of a parallel system is to speedup and scaleup *linearly*, that is, doubling the amount of hardware reduces the execution time by a factor of two, and doubling the hardware *and* problem size results in an unchanged execution time, respectively. We find that the MTA-2 exhibits both linear scaleup and speedup on database workloads.

For all of our experiments we used a 40 processor Cray MTA-2 with the same system specifications shown in Table 1. All experiments were implemented in C++ and compiled using Cray's MTA-2 compiler version 1.6.5.

4.1 Parallelizing Database Operations

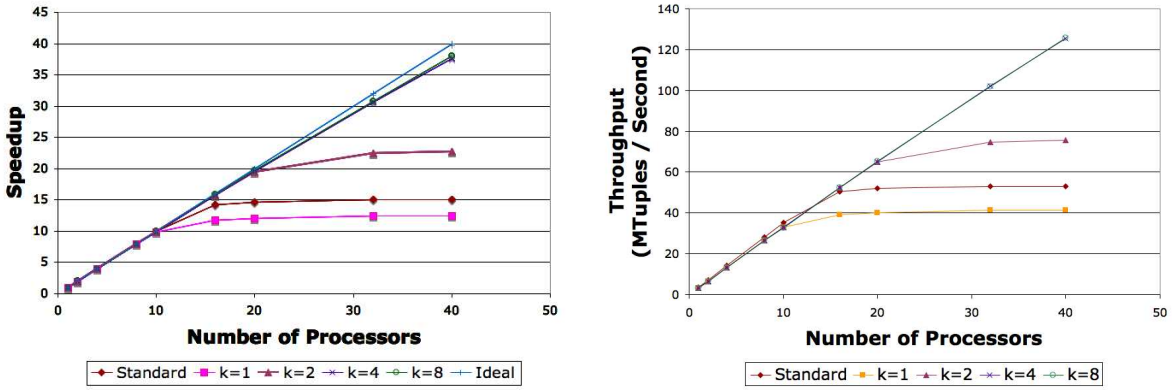
Here we present an overview of parallelism on the MTA-2 (for more details see [7]). With a few hints from the programmer, the MTA-2 compiler automates the creation of parallel code for loops with no loop-carried dependencies, first-order linear recurrences, and reductions. A loop-carried dependency is a variable assignment that depends upon other iterations of the loop. For example, if every iteration can be executed correctly in isolation, then the loop is easily made parallel. A recurrence is a special type of a loop-carried dependency that exists if values computed in one loop iteration are used by a subsequent iteration. If the recurrence is not complicated the compiler can still create a parallel loop. A reduction is similar to an aggregation and involves a loop-carried dependency that reduces many values to one value. For example, if a sum is being calculated, the dependency is in the variable accumulating the sum. If the compiler recognizes this pattern, it provides each thread with its own accumulation variable and then combines the results after the parallel processing is complete. To accomplish parallelism in database operations we will use implicit parallelism and eliminate loop-carried dependencies. An implicitly parallel loop must meet three criteria:

1. The loop must have only one entrance and one exit.
2. The loop must be controlled by a single variable that is incremented or decremented by an invariant amount on each iteration.
3. The loop exit condition must involve only the aforementioned variable and a loop invariant.

A *for* loop will generally meet all three criteria, provided there are no means of breaking out of the loop. Within the loop we apply the predicate in the case of a select or we probe the B$^+$-Tree or hash table in the case of an index or hash join, respectively. The for loop control variable indexes the probe table, whose tuples can be processed in parallel. At run time, the for loop iterations will be partitioned among many threads on the available processors, thus implicitly partitioning the input as well.

There are two common and related challenges facing the implementor: (1) coordinating the output of the operation and (2) avoiding memory hotspots. An index variable representing the next output location is a loop-carried dependency because the output location of the current result depends on the previous tuples processed. This dependency can be eliminated by using an `int_fetch_add` instruction, which is provided by the MTA-2 and performs the increment atomically in memory. We found that although loop-carried dependencies were eliminated and parallelism enabled, the scaling of the database operations was poor. This was because the output index created a *hotspot*, a location in memory that is accessed so frequently that it creates a bottleneck too severe for even a high degree of multithreading to overcome. Our solution to the hotspot problem was to implicitly partition the output into k partitions, each with its own output index that is incremented using `int_fetch_add`. The output partition and index to use is chosen by `input_index mod k`, thus distributing the memory accesses among the k output index variables.

Figure 1 shows a simplified version of the parallel select operator with implicit output partitioning and Figure 2 displays the throughput performance of the selection operation on the MTA-2. These figures show that the use of implicit partitioning enables good scaling, but also introduces some overhead, in the form of additional instructions per loop iteration. This overhead is very pronounced in the selection



(a) Speedup (b) Throughput

Figure 3: Hash Join on the MTA-2. k is the number of output partitions.

operation because selection is relatively simple and has few instructions per loop iteration, whereas the overhead is negligible for more complex operations such as the hash join (Figure 3b).

The output partitioning relieves the hotspot and allows for good scaling, but results in two new problems: (1) how to choose k and (2) what to do with partitioned output. Luckily the answers are easy. The value of k depends on the number of processors available in the system because more processors mean more threads potentially accessing the same output index variable. We found that overestimating k did not significantly affect performance, so a good rule would be to use a value of k assuming all processors in the system are available at runtime. In terms of partitioned output, operators can be designed to take partitioned input, or the output partitions can be merged together in parallel. We found that both options scale well.

The parallelism outlined above does not preserve tuple order. This is an important observation because the ordering characteristic is a valuable feature used by query optimizers. If the output of any operator needs to be ordered, the sorting of the intermediate result can also be done in parallel.

4.2 Hash Join

We implemented the Radix Hash Join algorithm used in MonetDB [16], which has been shown to have good cache performance on conventional processors. Because the MTA-2 does not use a cache, we eliminated the radix clustering portion of the algorithm that organizes the data in a more cache-conscious way. The execution time of this algorithm accounts for the hash table build phase as well as the probe phase of execution, both of which are implemented as parallel algorithms. We used the technique described in Section 4.1 to implement the probe phase of the hash join in parallel. Building the hash table in parallel is described below in Section 4.2.1.

The workload consists of two DSM relations with the schema: <id, value>, both integers. Each relation consists of eight million tuples. The relations were designed such that each tuple from the outer relation joined only once with the inner relation. The tuples were uniformally distributed.

Figure 3a shows speedup of the hash join as the number of available processors increases and Figure 3b shows the hash join throughput. The unoptimized implementation scales linearly through about ten processors. The barrier to scaling is that incrementing the output index creates a hotspot. Implicitly partitioning the output results in linear scaling when k is large enough to sufficiently distribute the increment operations.

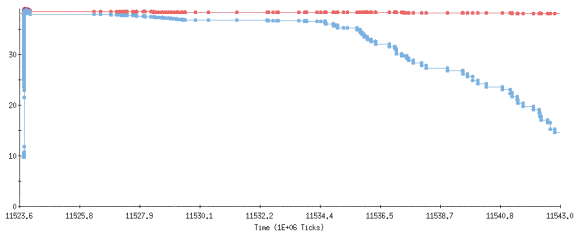


Figure 4: Hash join utilization of a system with 40 MTA-2 processors. The red points represent available system capacity and the blue points are the utilization of that capacity. This data was collected using Cray's *traceview* program.

Figure 4 shows the system utilization during the probe phase of the hash join with $k = 8$ on a 40 processor system. The overall processor utilization during the probe phase averages well over 80%. The drop off in utilization near the end of the hash join can be explained by the fact that the amount of work per loop iteration can vary slightly on each probe. Some of the implicitly created parallel partitions will complete sooner, which results in fewer instruction streams to fully occupy the system resources.

4.2.1 Building the hash table

As with the probe phase of the hash join, building the hash table in parallel also requires few modifications, but illustrates the importance of the lightweight, fine-grained synchronization.

Figure 5 shows the hash table build portion of the radix hash join without the use of the radix bits [16], since the MTA-2 will not require the radix clustering. The barrier to parallelism here is the need to synchronize the accesses to `bucket[idx]`. Unlike the case of coordinating the output, where we needed to synchronize a counter, the `int_fetch_add` instruction cannot be used in this case because the write is an assignment rather than an increment. Each word of memory on the MTA-2 has an auxiliary bit called the *full and empty bit*. This bit allows lightweight test and set type operations on individual words in memory. To accomplish

```
for(int i=0; i<inner_size; i++){
  int idx = HASH(inner_table[i].value) & M;
  next[i] = bucket[idx];
  bucket[idx] = i;
}
```

Figure 5: Building the hash table serially. Those familiar with the radix hash join will note that references to the number of radix bits has been omitted. M is a bit mask to transform the hash value into a valid bucket index. Overflow is handled by storing the index of the next tuple in the `next` array.

the necessary synchronization we read the `bucket[idx]` *full-empty* using the `readfe` function, which means that the read will not occur until the full and empty bit is set and the operation will leave the bit unset. When we write the variable with the new value, we write *empty-full* using the `writeef` function that will again set the full and empty bit. Figure 6 shows the changes required to parallellize the hash table build phase.

4.3 B$^+$-Tree Index Join

We also implemented a parallel index nested loop join using a B$^+$-Tree. The workload was identical to the hash join experiments. The index was pre-computed, so the join performance reflects only the cost of performing the join.

The scaling of the B$^+$-Tree index nested loop join is shown in Figure 7. For this experiment the minimum number of keys per index node was 64 and the maximum was 128. It was not necessary to use the implicitly partitioned output technique described in Section 4.1 in order to achieve good scaling, because the traversal of the index required more memory accesses and computation per probe tuple, thus relieving the hotspot related to coordinating the writing of output tuples. In the event more processors were available, output partitioning might be required.

We ran the same B$^+$-Tree index join experiments on an Intel Pentium 4 running at 2.8 GHz. As with the MTA-2 experiments, the data, including the B$^+$-Tree index, is memory resident when the experiments begin. We use the same B$^+$-Tree index structure with no modifications for better cache performance. Here we will briefly compare performance in terms of throughput and cycles per probe tuple processed between the MTA-2 and the Pentium 4. More extensive experiments can be found in [7]. Because each MTA-2 processor runs at only 220 MHz, a single MTA-2 processor can retire over ten times fewer instructions per second than the Pentium. Table 2 shows the throughput and cycles per tuple measures for three different MTA-2 system sizes and the Pentium 4. At 13 processors, the MTA-2 will retire close to the same number of instructions per second (2.86 billion) as the Pentium 4 (2.8 billion), but while the Pentium 4 has only one thread the MTA-2 has 1664 threads (128 per processor). These results show that the high degree to multithreading on the MTA-2 effectively overlaps high latency memory operations with computation, while the Pentium 4 stalls for memory accesses more often. This occurs because traversing the B$^+$-Tree index structure requires pointer following to access the next level of the tree. This pointer following represents a data dependency that performs poorly on single

```
#pragma mta assert parallel
for(int i=0; i<inner_size; i++){
  int idx = HASH(inner_table[i].value) & M;
  next[i] = readfe(&(bucket[idx]));
  writeef(&(bucket[idx]), i);
}
```

Figure 6: Changes needed to build the hash table in parallel are highlighted in blue.

threaded processors. [1]

A significant result is that the implementation of the parallel join operator required little effort and resulted in code that scales well regardless of the system's runtime capacity or load. In contrast, a cache-conscious design for a single-threaded processor would require a more elaborate implementation and careful attention to many parameters and may have difficulty adapting to runtime factors such as load, resulting in competition for resources including memory and the cache.

5. RELATED WORK

In addition to the cache sensitive algorithms and data structures as well as previous work on parallel database systems briefly mentioned in Section 2, other techniques have been developed to improve database performance on modern hardware. Please see [7] for more discussion of related work.

Simultaneous multithreading technology (SMT) is currently available in a number of commodity processors [17, 21, 23]. Using hardware simulations, SMT has been shown to be useful for database workloads [14]. Zhou et al. explore various techniques for using SMT to improve database performance and conduct experiments on real hardware [25]. This work shows that SMT benefits database operations when multithreading is used to hide high latency memory operations.

Like high latency memory operations, branch mispredictions also affect database performance. Ross [20] proposes techniques for improving branch prediction for selections, and Zhou and Ross [26] describe techniques for using SIMD instructions to reduce branch mispredictions. Since only one instruction per thread is in the MTA-2 pipeline at a time, branch conditions on the MTA-2 are evaluated before the branch is taken, thus eliminating the need to roll back instructions that should not have been issued. Any delay caused by a branch is absorbed by the system provided that there are a sufficient number of other threads ready to execute. This paragraph highlights an advantage of a multithreaded system that goes beyond mitigating the high cost associated with memory operations.

Gold et al. demonstrate the advantages of using a network processor for database workloads [10]. Their work also observes that using multithreading to hide the impact of high latency memory operations can lead to a large performance

[1]The authors acknowledge that a direct comparison is difficult because the amount of work performed per cycle is not equal. Also, a cache-conscious implementation of the B$^+$-Tree index will perform much better on the Pentium 4. Though a commodity system will beat the MTA-2 on hardware cost, this comparison shows that a high degree of multithreading allows good performance by focusing on parallelism as opposed to data layout.

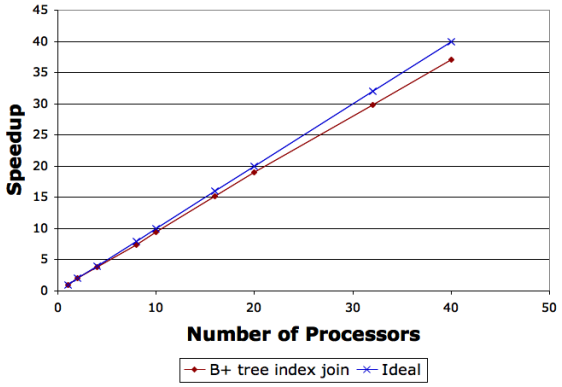


Figure 7: B$^+$-Tree Speedup vs. Ideal Speedup.

System	Throughput (MTuples/Sec)	Cycles/Tuple
MTA-2 (1P)	0.56	392
MTA-2 (13P)	6.78	421
MTA-2 (40P)	20.79	423
Pentium 4	0.80	3490

Table 2: B$^+$-Tree Index Join Throughput

improvement. Explicit cache management afforded by the network processor is useful because the programmer can explicitly control the cache and separate frequently accessed data from data that will be scanned just once. The network processor is nonpreemptive, thus requiring the programmer to explicitly release control of the processor. These features allow greater control over execution and data at the expense of more complex code.

Observing that high latency memory operations adversely affect many programs, modern processors often employ hardware prefetching [12, 21]. Hardware prefetching identifies sequential access patterns at runtime and then loads the next memory locations in the sequence into the cache before they are requested by the application. Many applications also exhibit random memory accesses that cannot benefit from hardware prefetching. Software prefetching allows the programmer to request that certain memory locations be loaded into the cache before they are actually required by the application. Database researchers have used software prefetching to improve the memory bottleneck in B$^+$-Trees [5] and hash join [6]. Software prefetch can be difficult to tune, and on some architectures prefetch instructions can be dropped [12]. The MTA's cacheless architecture does not rely on any type of prefetching because memory latency is incurred when the data is accessed, but that latency is hidden by overlapping computation from other threads.

6. DISCUSSION AND CONCLUSION

Hardware and software cannot be developed in isolation. In this section we reiterate the advantages and disadvantages of the MTA-2 with respect to databases. We explore the importance of this research given the recent advances in on-chip parallelism. This section provides a high level discussion of the challenges that exist in bridging the gap between the degree of shared-memory parallelism demonstrated in this paper, and that is currently available in commodity systems.

We found that the MTA-2's cache-less, massively multithreaded architecture provided many benefits for core database operations. The most important feature was the use of multithreading to efficiently hide the cost of high latency memory operations. Rather than requiring careful analysis of data layout and memory access patterns, as would typically be required of optimizing for cache performance, achieving good performance on the MTA-2 required identifying enough parallelism to ensure that some threads on the processor remained busy. Another important feature was the compiler which automatically generated the low level details necessary for parallelism and also provided feedback to the developer as to whether or not a loop could be made parallel. Not having to deal with the low level details of creating parallel code or data layout left the developer free to focus on high level aspects of efficiency and correctness. In terms of performance, we found that the MTA-2 performed well on database workloads, particularly operations involving significant pointer following or random memory accesses. Workloads such as a hash join or B$^+$-Tree index join exhibit much more efficient use of processor resources on the MTA-2 than on commodity cache-based hardware.

The MTA-2 architecture also has a number of drawbacks; chief among them is the very poor serial performance. Additionally, the MTA-2 is not a commodity architecture, so its benefits are not available for widespread use. That said, current trends toward increased on-chip parallelism make the lessons learned in this research about massive multithreading salient in guiding future developments of on-chip parallelism and the applications that will take advantage of that parallelism.

The trend in commodity architecture is toward more on-chip parallelism. How this parallelism should be provided and how applications should take advantage of it are open research questions. This research on the MTA-2 is useful because it allows insights into what a high degree of shared-memory parallelism, currently unimaginable in a commodity chip, has to offer. Perhaps our results could influence both hardware architects and database researchers in at least two ways. First, databases are an important workload for commodity chips and our work shows that databases are able to utilize a high degree of shared-memory parallelism. Therefore, architects could look for ways of incorporating some aspects of the MTA-2 system and programming model into commodity architectures. Second, this paper presents a new model of shared-memory parallelism to the database community. Though not directly applicable to commodity architectures, the intent is that this research will inspire new thinking about database operations from a shared-memory, parallel perspective, which will be important as the amount of shared-memory on-chip parallelism increases.

Because the MTA-2 differs greatly from conventional architectures, incorporating features from the MTA-2 into commodity processors will require practical compromises between apparently conflicting goals. Optimally, a future architecture would feature good serial performance, as well as high parallel performance that is easy to use. There are several key challenges. Foremost is the issue of the cache–a key feature of conventional hardware that enables good serial performance. Having a cache in a shared-memory system requires overhead to ensure cache coherency. The MTA-2 avoids this overhead by eliminating the cache and using multithreading to deal with the memory latency issue, at the expense of serial performance and a sequential physical address space.

In summary, commodity processors are evolving from a single core, single thread design to a multicore, multithreaded design. Previous research has demonstrated ways of improving the performance of certain aspects of database operations on commodity processors with, at most, a handful of threads. This paper used the MTA-2, an existing system, to demonstrate that database operations can be made to perform well on massively multithreaded, shared-memory architectures. It is likely that future architectures will feature more on-chip parallelism while retaining many serial optimizations. Therefore, a fusion of existing serial optimizations and new shared-memory parallel techniques will be necessary to achieve good database performance.

7. ACKNOWLEDGEMENTS

We thank Cray Inc., especially Simon Kahan, for assistance in learning about the MTA-2.

8. REFERENCES

[1] Advanced Micro Devices. AMD multi-core products. Available at http://multicore.amd.com/en/Products/.

[2] A. Ailamaki et al. DBMSs on a modern processor: Where does time go? In *VLDB*, 1999.

[3] W. Anderson et al. Early experience with scientific programs on the Cray MTA-2. In *ACM/IEEE Conf. on Supercomputing*, 2003.

[4] S. H. Bokhari and J. R. Sauer. Sequence alignment on the Cray MTA-2. *Concurrency and Computation: Practice and Experience*, 16(9), 2004.

[5] S. Chen et al. Improving index performance through prefetching. In *SIGMOD*, 2001.

[6] S. Chen et al. Improving hash join performance through prefetching. In *ICDE*, 2004.

[7] J. Cieslewicz et al. Unlocking parallelism in database operations: Insights from a massively multithreaded architecture. Technical Report SAND 2005-7065C, Sandia National Laboratories, 2005.

[8] Cray. Cray MTA-2 system. http://www.cray.com/products/programs/mta_2/.

[9] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6), 1992.

[10] B. T. Gold et al. Accelerating database operators using a network processor. In *DaMoN*, 2005.

[11] Intel. Intel multi-core platforms. Available at http://www.intel.com/technology/computing/multi-core/.

[12] Intel Corporation. IA-32 intel architecture optimization reference manual. Available via http://developer.intel.com.

[13] J. Kahle et al. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), 2005.

[14] J. L. Lo et al. An analysis of database workload performance on simultaneous multithreaded processors. In *ISCA*, 1998.

[15] R. MacNicol and B. French. Sybase IQ Multiplex - designed for analytics. In *VLDB*, 2004.

[16] S. Manegold et al. What happens during a join? Dissecting CPU and memory optimization effects. In *VLDB*, 2000.

[17] D. T. Marr et al. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, 2002.

[18] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB*, 1999.

[19] J. Rao and K. A. Ross. Making B+ trees cache conscious in main memory. In *SIGMOD*, 2000.

[20] K. A. Ross. Selection conditions in main memory. *ACM Transactions on Database Systems*, 29(1), 2004.

[21] B. Sinharoy et al. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5), 2005.

[22] M. Stonebraker et al. C-store: A column-oriented DBMS. In *VLDB*, 2005.

[23] Sun Microsystems, Inc. Ultrasparc processors. Available via http://www.sun.com/processors/index.html.

[24] Sun Microystems, Inc. UltraSPARC T1 supplement to the UltraSPARC architecture 2005. http://opensparc.sunsource.net/specs/UST1-UASuppl-current-draft-P-EXT.pdf.

[25] J. Zhou et al. Improving database performance on simultaneous multithreading processors. In *VLDB*, 2005.

[26] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, 2002.