Pipelined Hash-Join on Multithreaded Architectures

Philip Garcia
University of Wisconsin-Madison
Madison, WI 53706 USA
pcgarcia@wisc.edu

Henry F. Korth
Lehigh University
Bethlehem, PA 18015 USA
hfk@lehigh.edu

ABSTRACT

Multi-core and multithreaded processors present both opportunities and challenges in the design of database query processing algorithms. Previous work has shown the potential for performance gains, but also that, in adverse circumstances, multithreading can actually reduce performance. This paper examines the performance of a pipeline of hash-join operations when executing on multithreaded and multi-core processors. We examine the optimal number of threads to execute and the partitioning of the workload across those threads. We then describe a buffer-management scheme that minimizes cache conflicts among the threads. Additionally we compare the performance of full materialization of the output at each stage in the pipeline versus passing pointers between stages.

1. INTRODUCTION

Recently, multi-core and multithreaded processors have reached the mainstream market. Unfortunately, software designs must be restructured to exploit the new architectures fully. Doing so presents both opportunities and challenges in the design of query-processing algorithms. In this paper, we describe some of the challenges presented to database system designers by modern computer architectures. We then propose parallelization techniques that speed up individual database operations, and improve overall throughput, while avoiding some of the problems such as those described in [18], that can limit performance gains on multithreaded processors.

This study builds on the work in [9, 7, 24], but instead of focusing solely on optimizing a single join operation, we examine a pipeline of join operations on uniform heterogeneous multithreaded (UHM) processors, an architectural model that we describe in Section 2.1. The techniques we develop and evaluate are applicable beyond join, and relate to other data-intensive operations. By accounting for the heterogeneous threading model of modern processors and the efficient sharing of data offered by them, we develop query processing algorithms that are more efficient and allow for more accurate runtime estimates, which then can be used by query optimizers.

In this paper, we make the following observations:

- Assigning threads to specific "processor thread slots" allows for high performance and throughput.
- Single-die UHM architectures can share data among threads more efficiently than SMP architectures.
- Writing pointers to a buffer instead of writing the full tuple does not save as much work as previously thought.
- Hardware and software prefetching can result in large performance gains within query pipelines.
- Properly scheduling threads on an SMT processor can significantly improve query pipeline runtimes.
- To exploit a multithreaded processor fully, a query pipeline should generate more threads than the architecture can execute concurrently.
- A large memory bandwidth is required to keep all of the processing units busy in multi-core systems.

In Section 2, we describe the changes in computer architectures that motivate this work. Then, we discuss the implications of these new architectures on database systems and describe the specific database query-processing issues on which we focus. In Section 4, we propose a threading model to help take advantage of these processors, and finally in Section 5, we discuss the results of our study, and speculate how this model will perform on future UHM processors.

2. PROCESSOR ARCHITECTURE

Computer architectures are continuously evolving to take advantage of the rapidly increasing number of transistors that can fit on a single processor die. These new architectures include larger caches, increased memory and cache latencies (in terms of CPU cycles), the ability to execute multiple threads on the same core simultaneously, and the packaging of multiple cores (processors) on the same die. These new features interact in complex ways that make traditional simulations difficult. We have therefore chosen to run our tests on real hardware. This provides a more realistic view of both the processor and main-memory subystem.

We ran our tests on both a dual 3.0 GHz Xeon Northwood processor, a 2.0 GHz Core Duo (Yonah) processor, as well

Proceedings of the Third International Workshop on Data Management on New Hardware (DaMoN 2007), June 15, 2007, Beijing, China
Copyright 2007 ACM 978-1-59593-772-8 ...$5.00.

	P4 Prescott	Xeon Northwood	Core Duo
Number of cores	1	2	2
Clock Speed	2.8GHz	3GHz	2GHz
FSB speed	800MHz	533MHZ	667MHz
L1 Size	16KB	8KB	32KB
L2 Size	1MB	512KB	2MB (shared)
L3 Size	-	1MB	-

Table 1: Details of the processors used

as a 2.8 GHz Pentium 4 Prescott as shown in Table 1. All of the machines ran Debian GNU/Linux with kernel version 2.6. We focused on the results obtained on the Pentium 4 processor, and unless otherwise noted, all results given are for it.

In this section, we discuss some of the details of multithreaded architectures and their impact on database query processing.

2.1 Multithreaded Architectures

Multithreaded processor architectures are being designed not only to enable the highest performance per unit die area, but also to obtain the highest performance per watt of power consumed[6, 5, 19, 3]. To achieve these goals, computer architects are no longer focusing on increasing instruction-level parallelism and clock frequencies, and instead are designing new architectures that can exploit thread-level parallelism (TLP). These architectures manifest themselves in two ways: chip multiprocessors (CMP) and multithreaded processors. CMP systems are a logical extension of SMP systems, but with the multiple cores integrated on a single processor die. However, many of the CMP systems differ from traditional SMP systems in that the cores share one or more levels of cache. Multithreaded processors, on the other hand, allow the system to execute multiple threads simultaneously on the same processor core. One of the more popular forms of multithreading is simultaneous multithreading (SMT), however other methods are possible[8, 23, 16, 22].

Many of these new multithreaded and CMP processors belong to a class of processors called uniform heterogeneous multithreaded (UHM) processors[21]. This class of architectures allows multiple threads (of the same instruction set) to share limited resources in order to maximize utilization. In this model, not all hardware-thread contexts are equivalent, and the behavior of one thread can adversely effect the behavior of another. This effect is generally due to shared caches, but it could also be caused by poor instruction mixes. UHM architectures should not be confused with heterogeneous multiprocessors in which the processor units themselves vary significantly or have differing instruction sets, such as a graphics coprocessor.[1]

Multithreaded processors have become the standard for high-performance microcomputing. The major vendors of high-performance processors are currently focusing on dual and multi-core designs[2, 1, 14, 3], and many are either shipping processors using multithreaded and/or SMT technology [16, 14, 3] to accelerate their processors.

Today's high-end database servers often contain 2-16 processors that are each capable of executing two threads. Within the next few years it is likely that a single microprocessor will contain many more cores that are each capable of executing multiple threads (using fine-grained multithreading or SMT)[6]. Many of these architectures (such as the Sun Niagara processor[3]) will implement multiple simple cores that sacrifice single-thread performance but yield substantially more throughput per watt and/or die area[6, 5, 3].

[1]See [11] for an example of database processing on a graphics co-processor.

2.2 Impact on Database System Design

The architectural changes that we have discussed force a re-examination of database system design. Concurrent database transactions generate *inter*-query parallelism but that increased parallelism can result in cache contention when threads or cores share one or more levels of the processor's cache. This puts a higher premium in *intra*-query parallelism (see, e.g., [12]), which current database systems do not exploit to the same degree as inter-query parallelism.

The rapidly expanding number of concurrently executing threads in a UHM architecture[21] combined with increasing memory latency (in terms of cycles) means database systems must be capable of executing an increasing number of threads at once to keep up with the growing thread-level parallelism offered by modern computer architectures.

We propose a threading model that breaks down a query into not just a series of pipeline operations (where each stage executes a thread), but into a series of operations that themselves can be broken down and executed by multiple threads. This allows the system to choose a level of threading that is appropriate for both the workload presented to it and the architectural features of the machine on which it is running. Additionally, on UHM systems, the system can choose the thread context on which to schedule a thread, in order to make the greatest use of the resources available at the time.

While much work has been done on optimizing query pipelines, much of this work has focused on either uniprocessor or SMP systems that assume a homogeneous threading model. New designs with UHM processors must first decide on which physical processor to execute the thread, and separately decide both on which core within the processor, and on which thread within the core to run. New schedulers must take into account how many threads are currently executing on the core, as well as what each thread on the core is doing. Much of the work on query pipeline optimization has also not taken into account the effects of using software prefetch instructions within the pipeline to improve performance further, with exceptions being [7, 9].

In this study, we examine intra-query parallelism within multiple hash-join operations. By breaking down each join into parallelizable threads, we have shown that both response time and throughput can be improved.

2.3 Prior Work

The work we describe here differs from earlier work [9, 7, 24] in several significant ways. In earlier work, software prefetching was examined in a single-threaded simulation[7], and was later extended to run on real machines[24, 9]. The prior work of Zhou, et al. [24], examined a single hash-join operation on an SMT processor, however this work was done on the Northwood variant of the Pentium 4, which doesn't fully support software prefetching, so a form of preloading data was used instead of prefetching. [9] further built upon the model in [7, 24] and was designed such that multiple threads could perform a single hash join. That work, however, did not consider a pipeline of operations, and additionally required an initial partitioning that can result in suboptimal performance.

In this paper, we consider a larger problem domain (pipelines) and a richer processing model aimed at UHM processors. This work differentiates itself by studying not the algorithms involved, but rather the impact of architecture on the end result. Through executing an example database pipeline, we can observe the interaction of program structures with the system architecture. By doing this we gain valuable insight into how to best design query pipeline execution strategies, and how to best choose an appropriate platform for query processing systems.

3. PROBLEM DESCRIPTION

We chose to examine a pipeline of two joins; however our algorithm can easily be extended to support more general n-way joins. For this study, we examine the performance of the query pipeline when running on various computer architectures. We also examine the performance of our threading model as a function of the number, size, and type of data stored in the buffers used to share data among the threads.

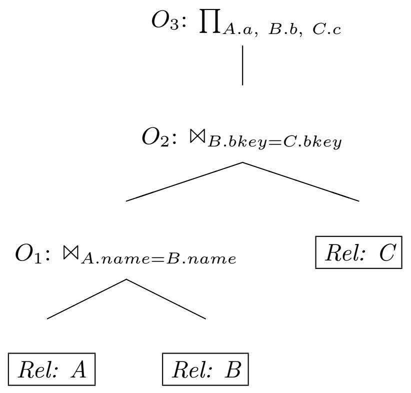
An important consideration in query-pipeline processing is the buffer size used and the number of buffers that are allocated to facilitate inter-process communication. We show that the buffer size has a major affect on overall algorithm performance as do prefetching attempts (done by both hardware and software).

Another important consideration is the issue of whether or not to materialize pointers. This becomes doubly important in a query pipeline consisting of operations $O_1, O_2, \ldots, O_m$ because the data must be brought into cache for the first join (operation O_i) and are possibly reused in the next join (operation O_{i+j}).[2] Because of this, materializing the output requires memory to store both the input relation and the output relation. This results in a larger overall cache footprint[3], although there is no deterministic way to tell how much larger this is on current computer architectures (due to streaming prefetch-buffers, memory access patterns, prefetch instructions etc). Recent research [9] has also shown that the time required to copy small amounts of data (<100 bytes) that is already loaded in cache can be prohibitively costly, and should therefore be avoided when possible. We examine the cost of materializing the relation fully at every stage of the pipeline in the Appendix.

Our hash-join algorithm is modified from the Grace algorithm[15]. Our algorithm was designed under the assumption that the system performing the join has sufficient free main memory to hold the entire set of input relations, temporary structures (such as hash tables), as well as output relations. This execution model has been shown to be valid for systems with sufficient main-memory and sufficient disk I/O performance[4, 7]. By doing an in-memory join, we are able to focus our analysis on the effects that both the main-memory/cache hierarchy and UHM processors have on query-pipeline performance. Disk accesses would not only distort those results, but make those results less applicable to modern systems with large-main-memories.

Our system implements the form of software pipelining described in [7]. We chose to focus on software pipelining, as it was shown to outperform both group prefetching and cache-sized partitioning[9, 7, 20]. Using the software-prefetch optimized code results in faster runtimes; however on multithreaded processors (running multithreaded algorithms), it has been shown that the speedup from multithreading is less when using the prefetch-optimized code due to there being fewer stall cycles to overlap execution[9]. Software prefetching still results in the best overall performance (even on multithreaded architectures), and it is therefore important that our measurements run with this algorithm rather than the standard hash-join algorithm, as that would overestimate the performance benefits of multithreading.

Our algorithm differs from prior work[7, 9, 24] in that we do not first partition the relations. Our previous results have shown that the size of the partition does not effect the throughput of the probe phase of the algorithm when prefetching is used[9]. Because the data are no longer partitioned, we must use a different method of breaking up the workload among multiple threads than that in [9]. We modified the system to use a series of buffers for both input and output so that multiple threads can cooperate to execute a single join concurrently.



O_3: $\prod_{A.a,\ B.b,\ C.c}$
O_2: $\bowtie_{B.bkey=C.bkey}$
O_1: $\bowtie_{A.name=B.name}$
Rel: C
Rel: A
Rel: B

Figure 1: Example pipeline where O_1 is an index join, O_2 is a hash join, and O_3 is a projection.

4. THREADS AND BUFFERS

Our threading model is based on using both control parallelization (through the pipelining of the query operation) and single-program multiple data parallelism (SPMD)[13]. This allows our model to allocate multiple threads for each operation in the query pipeline.

4.1 Threading Considerations

We use a buffer-management scheme to allocate data from the input relations to the various threads and also allow for forwarding output data to operations further in the pipeline. We found that the number of buffers used and their size can affect system performance. Using buffers with fewer entries allows the working set to be smaller and better able to fit within the processor's data cache, however this requires each thread to acquire more buffers, potentially resulting in slower performance. Conversely using buffers with more entries means the system spends a smaller percentage of its time obtaining buffers, but the memory footprint of each buffer is larger, and could result in poor sharing of the processor's cache among the threads.

[2]For our tests i=1 and j=1.
[3]This is assuming, of course, that the size of each tuple in the output relation is greater than the size of a pointer.

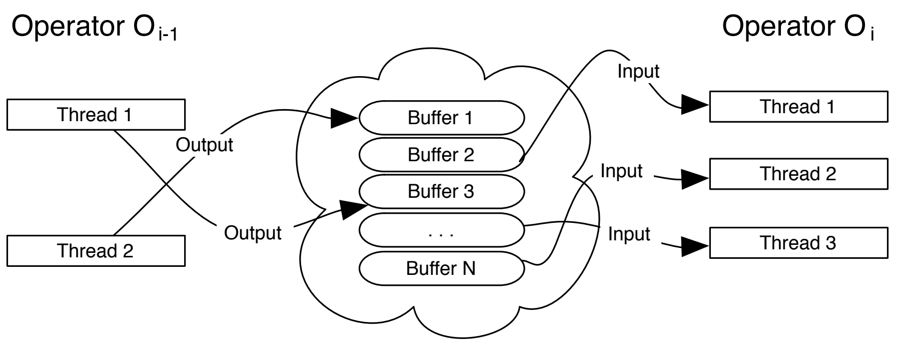


Figure 2: Example of two pipeline operations, O_{i-1} and O_i, sharing a set of buffers.

Another important issue to consider is the number of threads that are allocated for each operation in the pipeline. Even when we just concern ourselves with join operations, earlier joins can often take significantly longer than later ones depending on the selectivity of the earlier joins. This effect is coupled with the fact that pipelined workloads are not always evenly distributed.

Figure 1 shows an example pipeline in which pipeline operation O_1 may generate output tuples at a varying rate because there may be many tuples generated for common names, but far fewer for the less common names[4]. This would cause operation O_2 to have a varying workload and to alternate between periods of idling (due to lack of input data) and busy periods where it has sufficient work to allow it to take advantage of multiple threads or processors.

Additionally, it is important that databases running on UHM processors schedule multiple threads carefully to avoid one thread adversely effecting the performance of another. In [18], it was noted that enabling SMT on Intel's Netburst processor can be detrimental to database performance. This is often caused by "cache thrashing" behavior of a single thread. For example, when one thread is running a large scan, it could cause a concurrent thread to experience more cache misses than if the two operations were serialized.

4.2 The Buffer-Management System

Our threading model helps to solve these issues by letting each join operation in the pipeline be handled by multiple threads, while allowing many of threads to sleep when they are not needed. This allows the operations that need the processing resources the most to utilize them, while other operations wait until the input is ready.

We implemented a buffer system designed for unbalanced workloads. This was accomplished by waking threads up upon availability of work, and putting them to sleep when no new work is available. The buffer manager uses a producer-consumer queue that shares buffers in common. We used the *pthreads* library[17] for the purposes of threading and inter-process communication.

The buffer manager (Figure 2), contains a finite number of buffers that it allocates to the producer and consumer threads. A buffer consists of a collection of tuples or pointers, and we choose both the number of buffers to allocate and the number of tuples or pointers that each buffer contains. Each buffer can be used by one thread at a time regardless of whether the thread is writing to or reading from the buffer. The assignment of a buffer to a thread is made by the buffer system. This avoids any need for concurrency control (e.g., locking) while a thread is executing.

[4]While our system does not currently support index join or projection operations used in Figure 1, our threading model could easily apply there as well.

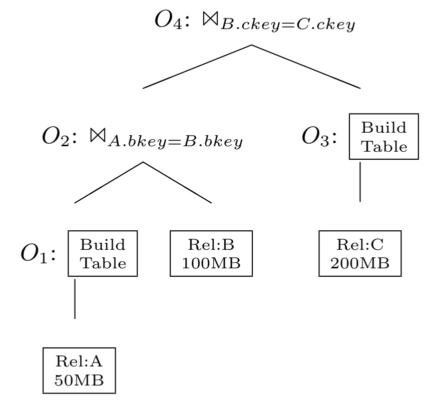


Figure 3: Pipeline used for our tests, where each entry in A matched exactly two entries in B, each entry in B matched exactly two entries in C and tuple sizes were the same for all three relations.

This buffer management system allows the system to allocate multiple threads for each operation in the pipeline (O_i and O_{i-1}, shown in the figure). Because each thread is constrained to execute on a single thread-context, the system accounts for imbalances in workloads among operations as well as variances in the workload, allowing the processor's resources to be utilized more effectively. The system accomplishes this by creating more threads for each operation than there are execution slots available on the processor. The system executes only those threads that currently have a buffer allocated to them. Limiting the size and number of buffers prevents any particular operation O_{i-1} from getting too far ahead of dependent operation: O_i. By limiting the number of buffers and their size, we can ensure that the output data produced by operation O_{i-1} is still in the processor's cache when it is consumed by operation O_i. Additionally, limiting the number and size of buffers can be used to prevent pipeline threads from running alongside concurrent threads in the system that could cause the cache to thrash.

5. EXPERIMENTAL RESULTS

Figure 3 shows the example query pipeline that we used for all of our tests. We used this pipeline because it is simple

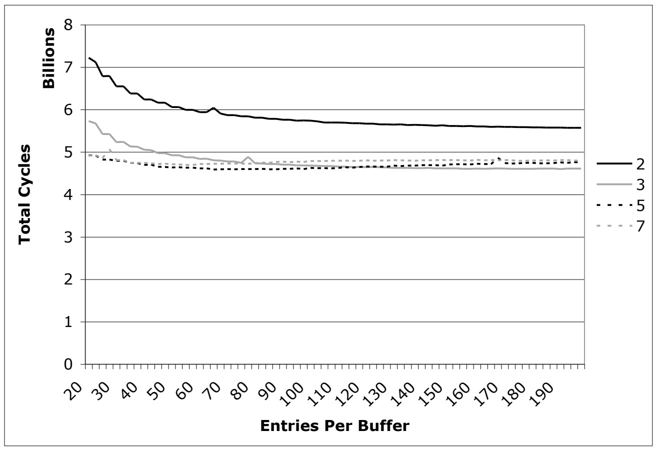


Figure 4: Pentium 4 results. The numbers on the right represent how many buffers were in use.

and has an imbalance in the workload between operations O_2 and O_4. This allows us to examine the effectiveness of the buffer-management system.

5.1 Number and Size of Buffers

To determine the ideal number of buffers to use, we ran a series of tests using multiple buffers of varying sizes. Running a single thread for each operation in the pipeline did not fully utilize processor resources. This is because operation O_2 processes approximately half as many tuples as O_4. To help alleviate this problem, we allocated two threads to O_4 and only a single thread to O_2. By doing this, we allowed the more expensive operation to utilize the majority of the processor's resources.

Parallelizing a hash-join operation (O_2 and O_4 in Figure 3) is fairly straightforward because multiple threads can share the hash table (as it is read-only). The input/output buffer system described in Section 4 protects the input and output data so that only one thread can write to a buffer at a time, ensuring correctness.[5] However, not all stages of the pipeline can be parallelized easily(for example the build operations O_1 and O_3). The general problem of parallelizing highly dependent database operations is left for future work.

Our tests were run for both the cases where the intermediate buffers contain the full output tuple, and when they contain only pointers to the tuple. We found that passing pointers was marginally faster (but, except for very large tuples, only marginally faster) than passing the full resultant tuples. Therefore, in this section we focus on the case of passing pointers between the pipeline stages. For a more detailed comparison of the performance when using pointers versus full materialization see the Appendix.

We found that the we needed at least one more buffer than the number of readers and writers concurrently executing. Figure 4 shows that at least three buffers are needed to utilize the processor effectively. This is logical as we executed three threads (of which only two executed simultaneously) so only having two buffers causes extra contention for shared objects between threads. Extrapolating these results to future architectures, we should have at least one more buffer than we have threads concurrently executing. These figures also show that enabling further additional buffers seems to do little to help or hinder performance.

[5]While this ensures correctness, it is important to note that when multiple threads execute a hash-join operation, the order of the input tuples is not preserved in the output, however this is rarely a problem, and an additional thread could be used to piece the buffers back together in order if necessary.

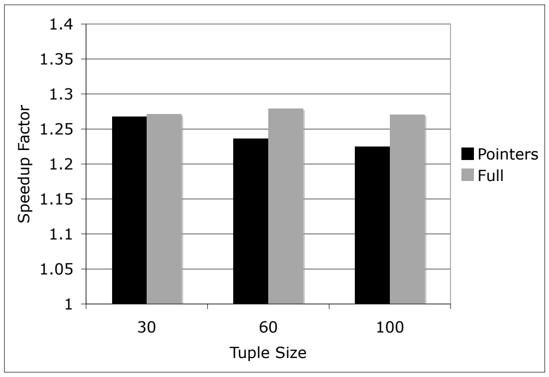


Figure 5: Multithreaded speedup factors for tuple-pointers and full materialization.

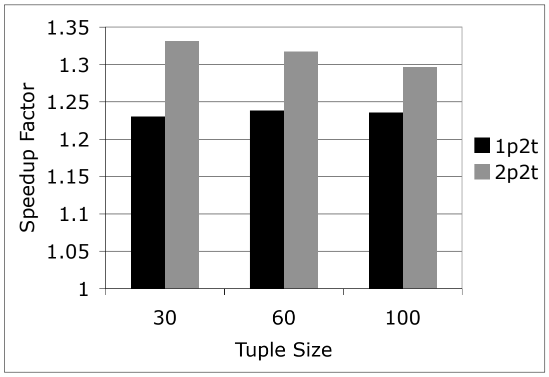


Figure 6: Xeon results: 1p2t means two threads on one processor, and 2p2t two threads, each on its own processor

5.2 Multithreaded Speedup

To quantify SMT's speedup on data-intensive algorithms, we ran our tests with and without SMT support. Figure 5 shows the speedup when we ran the query pipeline with both threads enabled versus when the case of only one thread enabled. This graph also illustrates SMT's greater benefit when copying the larger amount of data needed when fully materializing the output, rather than passing only pointers.

These numbers are similar to the speedups seen in [9] during the probe phase of the software-pipelining optimized hash join. However our results managed this speedup across the entire hash-join operation (including the build phase), and additionally accounts for a much finer-grain level of parallelism than that used in [9].

Performance was poor on the SMP/SMT Xeon. This is due partially to the Xeon's slower memory subsystem, but more importantly it is due to the Xeon's inability to properly

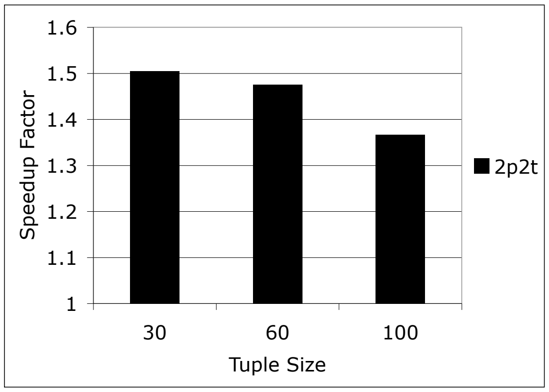


Figure 7: Multithreaded speedup factors obtained on the Core Duo processor (two processors, one thread per processor).

Architecture	30	60	100
Pentium 4 1p1t	6.32	3.17	2.12
Pentium 4 1p2t	4.98	2.56	1.73
Xeon 1p1t	11.95	6.13	4.17
Xeon 1p2t	9.71	5.06	3.38
Xeon 2p2t	8.98	4.75	3.22
Core Duo 1p1t	4.98	2.57	1.68
Core Duo 2p2t	3.33	1.75	1.22

Table 2: Wall-clock runtimes (in seconds) for tuple sizes of 30, 60, and 100 bytes. 1p2t means two threads on one processor, and 2p2t two threads, each on its own processor

handle prefetch instruction as explained in [9]. The comparison between these two architectures illustrates how much of an effect minor architectural changes can have on overall system performance. Figure 6 shows the speedups obtained when splitting the threads up among the different contexts available on this system. It is important to note when comparing these results to the other architectures, that the lack of software prefetching causes excessive data-cache misses, which SMT processors can effectively overlap with processing from the additional thread[22, 8, 23, 16, 9].

A rather surprising result of these experiments was that the dual-processor algorithm was only marginally faster than the SMT algorithm, running about 10% faster when using 30-byte tuples, and only 6% faster when using 100-byte tuples. One of the reasons for the Xeon's poor multiprocessor performance is due to its bus-based architecture, which quickly became overloaded as buffers were moved between the two procesors. Performance could be improved somewhat if two separate queues were used, with each processor producing distinct output. For example, if processor P_1 obtained data D_1, it would perform both operation O_1 and O_2 on it while P_2 would obtain D_2 and likewise perform the necessary operations upon it. This would prevent D_i from having to be sent across the bus for further processing on a different processing node. Such considerations are not necessary when executing upon many SMT or CMP processors, and aren't as important on processors that utilize point-to-point interconnects.

Figure 7 shows the speedups obtained on the Core Duo processor when executing the example query pipeline. Table 2 reveals that this platform outperformed the others, despite the fact that this platform had the slowest clock speed. The performance on this machine is due to the Core Duo's more efficient microarchitecture combined with its larger and faster cache. The multi-core speedup on the Core Duo was between 1.35 and 1.5. This is likely due to the limited memory bandwidth available on this mobile platform. This poor speedup also suggests that query optimizers should balance the workload among multiple physical processors to prevent any single core from being memory-bound.

6. CONCLUSION

In this paper, we have examined the impact of UHM processors on pipeline operations. Specifically, we studied the running of the hash-join algorithm in a pipelined fashion on a UHM processor. This overview of the effects of pipeline operations shows that a simple naïve approach to threading will not yield optimal performance on new processors. The need for highly parallel code to run on multithreaded processors[3, 14, 1, 2], combined with the increasing processor/memory gap and heterogeneous threading abilities of modern and future processors, have fueled the need for further research into query pipelines.

By examining the effect of multithreading, we have seen impressive gains in the performance of hash-join operations within queries and have shown the importance of combining SPMD techniques with traditional "unstructured" threading techniques (such as running a separate thread for each operation) when executing query pipelines. These techniques allow the system better control of the execution of algorithms and are necessary to achieve the greatest throughput on processors.

The issues of thread allocation and buffer management are of reduced complexity in our work due to the relative simplicity of the multicore and multithread architectures we tested and our focus on a relatively short pipeline of two joins. A higher number of cores with a higher number of concurrent threads per core opens the possibility of

1. Devoting more threads (and buffers) to each operation (greater horizontal parallelism).

2. Deepening the pipeline to include more joins as well as other operations (greater vertical parallelism).

These considerations add to the complexity of the buffer system.

Future work is still needed in expanding our threading model to support other operations (such as merge join, sort, selection, etc), and to examine performance on other multithreaded processors[3, 14, 1, 2]. This work should also focus on more parallel architectures than the two-threaded Pentium 4, taking into account the fast inter-core communication presented by these new UHM processors. By examining the performance of such operations, we can both stress the ability of our threading model to distribute evenly the workloads of multiple pipeline operations as well as determine more accurate estimates for ideal buffer sizes based upon processor cache size, and the number of threads that can run on a given processor.

Future work is needed to parallelize traditionally serial algorithms, including techniques to allow multiple threads to write to linked data structures simultaneously and efficiently. As the number of threads executing the probe phase increases, the percentage of time spent waiting in these serial algorithms will become excessively large. These stalls in the pipelines will become increasingly important as current techniques don't allow multiple threads to execute simultaneously. Finally, it will be necessary to develop good cost predictors for these parallel algorithms so that future database query optimizers have appropriate cost estimates on which to base their choice of overall execution strategy for the entire query.

7. REFERENCES

[1] Intel multi-core processor architecture development backgrounder. *Intel White Paper*, 2005.

[2] Multi-core processors– the next evolution in computing. *AMD White Paper*, 2005.

[3] Throughput computing: Changing the economics and ecology of the data center with innovative SPARC® technology. *Sun Microsystems White Paper*, November 2005.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proc. 25th Int'l Conf. on Very Large Data Bases*, pages 266–277, 1999.

[5] D. Burger and J. R. Goodman. Billion-transistor architectures: There and back again. *IEEE Computer*, 37:22–28, Mar. 2004.

[6] D. Carmean. Data management challenges on new computer architectures. In *First Int'l Workshop on Data Management on New Hardware (DaMoN)*, June 2005. Oral Presentation.

[7] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *Proc. Int'l Conf. on Data Engineering*, 2004.

[8] 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.

[9] P. Garcia and H. F. Korth. Hash-join algorithms on modern multithreaded computer architectures. In *ACM Int'l Conf. on Computing Frontiers*, May 2006.

[10] P. C. Garcia. Optimizing database algorithms for modern computer architectures. Master's thesis, August 2005. `http://www.cse.lehigh.edu/~pcg2/thesis.pdf`.

[11] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High performance graphics co-processor sorting for large database management. In *Proc. ACM SIGMOD Int'l Conf. on the Management of Data*, June 2006.

[12] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 102–111, 1990.

[13] W. D. Hillis and J. Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[14] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. 2004.

[15] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. In *New Generation Computing*, volume 1, pages 63–74, 1983.

[16] 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.

[17] F. Mueller. Pthreads library interface, 1993.

[18] S. Oks. Be aware: To hyper or not to hyper. *Slava Oks Weblog* `http://blogs.msdn.com/slavao/archive/-2005/11/12/492119.aspx`,Nov2005.

[19] P. S. Otellini. Multi-core enables performance without power penalties. In *Intel Developer Forum Keynote,* `http://www.embedded-controleurope.com/pdf/-ecedec05p26.pdf`,2005.

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

[21] D. Towner and D. May. The 'uniform heterogeneous multi-threaded' processor architecture. In A. Chalmers, M. Mirmehdi, and H. Muller, editors, *Communicating Process Architectures – 2001*, pages 103–116. IOS Press, September 2001.

[22] 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 Int'l Symposium on Computer Architecture*, pages 191–202, 1996.

[23] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual Int'l Symposium on Computer Architecture*, June 1995.

[24] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. Improving database performance on simultaneous multithreading processors. In *VLDB '05: Proceedings of the 31st Int'l Conf. on Very Large Data Bases*, pages 49–60, 2005.

APPENDIX

Pointers Versus Tuple Materialization

We discuss here in more detail our comparison between passing full output tuples between pipeline stages versus passing only pointers to those tuples.

Our results showed that the performance of the system was similar for the case when we passed pointers in the pipeline versus using full materialization[10]. This counters the previous belief that using pointers rather than copying the full tuples to a new buffer saves a significant amount of time that would otherwise be spent doing useful work. There are several reasons for this somewhat surprising result:

- Operation O_2 in Figure 3 (the only one copying data to the buffers) operates on half as many tuples as O_4, therefore the maximum possible speedup (where the time to run O_2 is zero) is 33%. When more operations utilize the buffers, the speedup may be greater.

- The time spent processing a single tuple in O_2 (discounting main-memory latency, which is hidden by the software prefetching and split evenly across the two processors) is less than the time it takes to process a tu-

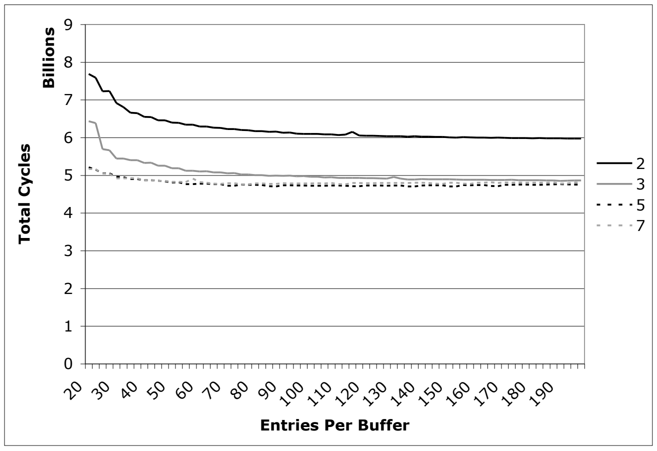


Figure 8: Cycles to run the multithreaded join pipeline on the Pentium 4 when fully materializing the intermediate relation. The numbers on the right represent how many buffers were in use by the system.

ple in O_4 because O_4 must copy data from three input relations.

- When the buffers fully materialize the data, the data are read back "in-order" during operation O_4, resulting in superior cache performance and eliminating excessive pointer chasing.
- Due to the latency-hiding nature of software-prefetch instructions, when pointers are used to pass values, data stalls are likely to occur as there is less useful work available in the rest of the hash-join algorithm to hide all of the cache-miss latency with prefetching.
- Hyperthreading allows multiple in-cache memory copy operations to occur simultaneously, hiding some of the extra time required to materialize the full tuple. Therefore the speedup of using pointers would be greater for the single-threaded algorithm.

These reasons help explain why using pointers to pass data between the operations does not result in as significant a speedup as initially expected. We also compared our results when using larger tuples. In Figure 9, we see that the speedups obtained due to using pointers are much greater, approaching the theoretical maximum of 33%. Thus, for large tuple sizes using pointers is a much more effective way to handle inter-process communication.

As UHM processors become more common, it will become even more important to use pointers to pass data between pipeline stages. On future architectures, it is likely that we will have more threads running on each processor core simultaneously. Under this model, context switches will occur on data cache misses. Because of this, memory latency can be hidden better than on current systems. This will enable non-latent threads to run while another thread is stalled[6].

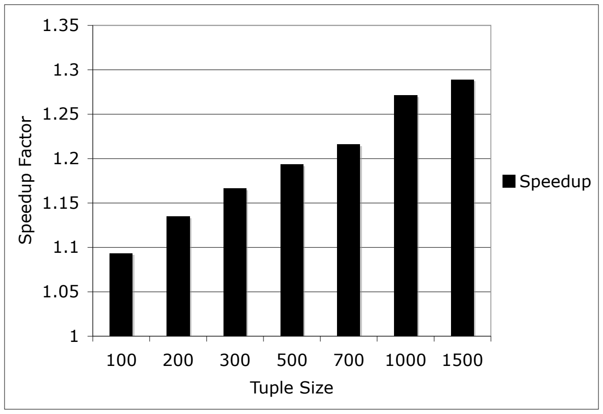


Figure 9: Speedups obtained by using pointers.

Thus, while our intuition about the merits of pointer passing do not hold true for our experiments, our data indicate a need to re-examine this issue in the context of future architectures.

[6]While this is also true on the Pentium 4 as two threads share the CPU, as the number of simultaneous threads that a processor can execute increases the overall system throughput will increase.