

Evaluation of pipelined hash-join operations on uniform heterogeneous multithreaded architectures

Philip Garcia

Electrical and Computer Engr. Dept.
University of Wisconsin Madison
Madison, WI 53706 USA
pcgarcia@wisc.edu

Henry F. Korth

Dept. of Computer Science and Engr.
Lehigh University
Bethlehem, PA 18015 USA
hfk@lehigh.edu

March 16, 2006

Abstract

Multi-core and multithreaded processors have reached the mainstream market, and present both opportunities and challenges in the design of query processing algorithms. This paper studies the processing of a pipeline of hash-join operations, building upon prior work that focused on a single operation. It shows that it is necessary to have more threads available in the algorithm than the processor itself can actually run concurrently in order to achieve optimal performance. It also describes a buffer-management scheme that limits cache conflicts among threads. Experimental results show the relative merits of passing pointers to intermediate results versus full materialization. Finally, the paper discusses how to extrapolate from its results to future architectures with a larger number of cores and concurrent threads.

1 Introduction

Recently, multi-core and multithreaded processors have reached the mainstream market, and present both opportunities and challenges in the design of query processing algorithms. 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 [21], that can limit performance gains from multithreaded processors.

This study builds on the work in [12, 8, 28], but instead of focusing solely on optimizing a single hash-join operation, we examine a pipeline of hash-join operations on uniform heterogeneous multithreaded (UHM) processors, an architecture we describe in Section 2.1. The techniques we develop and evaluate are applicable beyond hash join 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 both more efficient and allow for more accurate runtime estimates that can be used by query optimizers.

In this paper, we make the following observations on current-generation processors:

- The assignment of threads to specific “processor thread slots” is necessary to achieve both high performance and high throughput.
- Single-die UHM architectures can share data among threads much more efficiently than traditional SMP architectures.

	P4	Xeon
Number Processors	1	2
Clock Speed	2.8GHz	3GHz
FSB speed	800MHz	533MHz
L1 Size	16KB	8KB
L2 Size	1MB	512KB
L3 Size	-	1MB

Table 1: Details of the processors used

- Hardware support for hardware and software prefetching can result in large performance gains within query pipelines.
- Multithreading as implemented on current SMT processors can improve runtimes of query pipelines significantly if the threads are scheduled properly.
- In order to exploit a multithreaded processor fully, a query pipeline may need to generate more threads than the architecture can execute concurrently.
- The penalty for passing fully materialized data versus pointers between pipeline operations is less on SMT processors than for single-threaded implementations.

We begin by describing the new computer architectures that are likely to take hold over the next few years. Then we discuss the implications of these new architectural features on database systems. In Section 4, we propose a threading model to help take advantage of these opportunities, and finally in Section 5, we discuss the results of our study of this threading model for a hash-join pipeline, and speculate how this model will perform on future 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. Architectural features include increased cache sizes, increased memory and cache latencies (when measured in CPU cycles rather than seconds), the ability to execute multiple threads on the same core simultaneously, and the packaging of multiple cores (processors) on the same die. The complexity of these features and their interaction led us to measure implementations on commercially available processors rather than rely on simulations. This provides a much more realistic view of both the processor and main-memory subsystem. We ran our tests on the Intel NetBurst architecture[16, 19, 5]. Table 1 shows the specifications of the processors we used. We focused on the Pentium 4 processor for most of this study, and unless otherwise noted, all results are for it. In this section, we discuss some of the details of multithreaded architectures and their impact on database query processing.

2.1 Future Multithreaded Architectures

Future multithreaded processor architectures will not only be designed to enable the highest performance per unit die area, but will also be designed to maximize performance per watt of power consumed[7, 6, 22, 3].

In order to achieve these goals, computer architects are switching from a focus on increasing instruction-level parallelism and clock frequencies, and instead are focusing on designing new architectures that can exploit thread-level parallelism (TLP). These new architectures manifest themselves in two ways: chip multi-processors (CMP) and multithreaded processors. Initial CMP systems were a logical extension of SMP systems, but with the multiple cores integrated on a single processor die. Many of the newer 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[26, 10, 27, 19].

Many of these new multithreaded or CMP processors belong to a class of processors called uniform heterogeneous multithreaded (UHM) processors[25]. This new architecture uses fine-grained threads and allows high-speed thread management and synchronization. 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 due to instruction mixes on multithreaded processors. 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 Those architectures require major innovations in application design to make full use of their capabilities.

Multithreaded processors are becoming the standard for high performance microcomputing. Intel has enabled a form of SMT(hyperthreading)[26, 10, 27, 19] into their Xeon and Pentium 4 processors for some time now, and they have recently started shipping dual-core processors (with SMT)[1]. Sun is shipping the Niagara chip, which currently integrates eight in-order SPARC processor cores (each capable of running 4 threads) on a single die[3]. AMD started shipping 2-way CMP processors in mid 2005[2]. IBM has also been shipping 2-way CMP processors for a couple of years and recently released a 2-way CMP processor with 2-way SMT on each core[17]. All of the major processor manufacturers plan on expanding upon their current offerings by integrating more cores into their processors.

Today's high-end servers (as commonly used for database systems) often contain 2-16 processors that are each capable of executing two threads. Within the next few years a single microprocessor will contain many more cores that are each capable of executing multiple threads (using fine-grained multithreading or SMT[26]). Many of these architectures (such as the current 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[7, 6, 3].

2.2 Impact on Database System Design

The architectural changes we have discussed force a re-examination of database system design. While databases have inherent parallelism within their workloads, these new architectural paradigms require designers to rethink query processing strategies. Concurrent transactions generate *inter*-query parallelism but that increased parallelism generates cache contention when threads or cores share cache. This puts a higher premium in *intra*-query parallelism, which current systems do not exploit to the same degree as inter-query parallelism.

The rapidly expanding number of concurrently executing threads in a UHM architecture[25] combined with increasing memory latency (in terms of cycles) means database systems must be capable of executing an ever-increasing number of threads at once to keep up with the ever-growing TLP 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

¹See [14] for an example of database processing on a graphics co-processor.

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 which thread context to schedule threads on, in order to make the greatest use of the resources available at the time.

While much work has been done on optimizing pipelines or hash-join operations, 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 [8, 12].

In this study, we examine intra-query parallelism within multiple hash-join operations. By breaking down each join using simple means, we have shown that such a query is not only executed faster, but overall system throughput can be improved.

2.3 Prior Work

The work we describe here differs from our earlier work [12] in several significant ways. In the earlier work, we addressed prefetching issues in a single hash-join operation, and compared our measurements on current architectures with prior work [8] on a simulated machine. Although multithreading was considered, it did not consider the threading model of the current paper in which a subset of a large number of available threads is chosen to run at a given time and pipelined through a variable number of buffers. Thus, here we consider a larger problem domain (pipelines) and a richer processing model aimed at UHM threading. Still earlier work [9] focused exclusively on cache issues in pipelining.

The prior work of Zhou, et al. [28], like [12] studied a single hash join on an SMT processor, however this work was done on the Northwood variant of the Pentium 4 which didn't fully support software prefetching, so a form of preloading data was used instead of prefetching.

The approach we take here is better targeted at the trends in processor hardware design and will fit well with the new architectures and processors being released concurrent with the writing of this paper.

3 Problem Description

In this study we focus on multi-way join operations using the widely used hash-join algorithm [12, 8, 18, 28, 24].

In order to simplify the problem domain, we limited our pipeline to a series of two joins, however the algorithm could easily be extended to support more general n -way joins that are commonly found within query pipelines. Our system was also designed to run entirely within the system's main memory. This execution model has been shown to be valid for systems with sufficient main-memory and sufficient disk I/O performance [4, 8].

One important concern when joining multiple relations within a query pipeline is whether or not to materialize the entire output tuple that the next operation will work upon, or to simply store pointers to the tuples that the next operation will use. While it has been shown in [9] that neither approach is optimal all of the time (it depends upon the size of the tuples in the relation involved among other factors), we reexamine the cost of materializing the relation fully at every stage of the pipeline in Section 5.1.

The issue of whether or not to materialize pointers becomes doubly important in a query pipeline consisting of operations O_1, O_2, \dots, O_m because the data must be brought into cache for the first join

(operation O_i) and is 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 footprint³, 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 [12, 11] 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.

Another important consideration in query-pipeline processing is buffer size and the number of buffers that should be allocated to facilitate inter-process communication. We show that shared buffer size has a major affect on overall algorithm performance as do prefetching attempts (done by both hardware and software). We then speculate on the impact that future architectures will have on the ideal values we obtained. We examine these scenarios on multithreaded computer systems for both the cases where we materialize the tuples at the conclusion of every stage in the pipeline and for the case when they are materialized only at the concluding stage of the pipeline.

3.1 Hash-join Algorithm Used

Our hash-join algorithm is modified from the Grace algorithm[18]. Our algorithm was designed under the assumption that the system performing the join has sufficient free main memory to hold the entire input relations, temporary structures (such as hash tables), as well as output relations. Focusing on only main memory and not disk greatly simplifies our analysis of the effects that both the main-memory/cache hierarchy and UHM have on query-pipeline performance.

Our system is an extension of the system developed in [12] and implements the form of software pipelining described in [8]. We chose to focus on this join algorithm as software pipelining was shown to outperform both group prefetching and cache-sized partitioning[12, 8, 23]. 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[12]. 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 as opposed to the standard hash-join algorithm that would overestimate the performance benefits of multithreading.

Our algorithm also differs from that in prior work[8, 12, 28] 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[12]. Because the data is no longer partitioned, we must use a different method of breaking up the workload among multiple threads than that in [12]. 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.

4 Threads and Buffers

Our threading model is based on using both control parallelization (through the pipelining of the query operation) and single-program data parallelism (SPDM)[15]. SPDM is a form of parallelism that allows the various threads to share the same code, and is used within the probe operation. In this section, we describe how we combined these forms of parallelism to create an efficient multithreaded series of joins.

²For our tests $i=1$ and $j=1$.

³This is assuming, of course, that the size of each tuple in the output relation is greater than the size of a pointer.

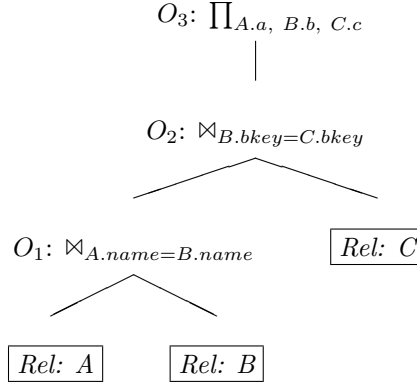


Figure 1: Example pipeline where O_1 is an index join, O_2 is a hash join, and O_3 is a projection.

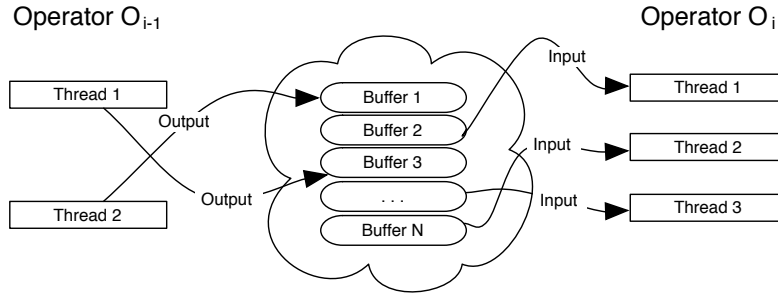


Figure 2: Example of two pipeline operations sharing a set of buffers.

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. It is important to choose both an adequate number of buffers, and an appropriate size (number of entries) in each buffer. 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 means each thread must acquire more buffers, resulting in slower performance. Conversely using buffers with more entries means the system will spend a smaller percentage of its time obtaining buffers, but the memory footprint of each buffer will be larger, and could result in poor sharing of the processor’s cache among the threads.

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 names⁴. This may likewise cause pipeline operation O_2 often to have little work to do, but occasionally, have more work than a single thread can effectively handle, resulting in an inefficient

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

utilization of processor resources.

Additionally, it is important that software designers for UHM processors take care when scheduling multiple threads. As Section 2.1 stated, the actions of one thread can adversely effect the performance of a second thread. In [21] it was noted that the SMT capabilities of Intel’s Netburst processor can sometimes cause the performance of current database systems to decrease. This is often due to a thread with “cache thrashing” behavior. For example, when one thread is running a large scan, it could cause a second thread to experience more cache misses than if the two operations were serialized. Such behavior must be accounted for in future database systems to squeeze the most performance out of the processor. Our threading system was designed with UHM processors in mind.

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 the majority of threads to sleep when they are not needed. This allows the operations that need the processing resources the most to utilize it while other operations wait until the input is ready.

We managed this by implementing a buffer system that handles unbalanced workloads by waking threads up upon availability of work, and putting them to sleep when their work is done. We implemented the buffer manager using a producer-consumer queue that shares buffers in common. We used the *pthreads* library[20] for the purposes of threading and inter-process communication.

The buffer manager, depicted in 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 the buffer or reading from it. For a more detailed description of the buffer management system see [13].

This buffer management system allows the system to allocate multiple threads for operation O_i and also for O_{i-1} . This enables the system to account for imbalances in workloads among operations as well as variances in the workload at a given time. The system can accomplish this by creating more threads for each operation than are needed at runtime, and only executing the threads that 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 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 this can be used to prevent pipeline threads from running alongside additional threads in the system that could cause the cache to thrash. This is because the pipeline operation executing will run out of buffers to produce into or consume from unless both threads are running concurrently.

An important consideration in the buffer management system is to choose buffer sizes that prevent cache-thrashing. The number of entries in the buffer is determined based on the need to keep information in cache between operations in the pipeline.

5 Experimental Results

We ran all our tests on both a dual 3.0 GHz Xeon Northwood processor as well as a 2.8 GHz Pentium 4 Prescott, described in Section 2 and Table 1. Due to the Northwood’s inability to properly handle prefetching[12, 5], most of our reported results are for the P4 Prescott.

Figure 3 shows the example query pipeline that we used for all of our tests. This pipeline differs from that of Figure 1 most notably in that the build and probe phases of hash join are separated into distinct operations. We also use key attributes for the joins and, to simplify the analysis of our

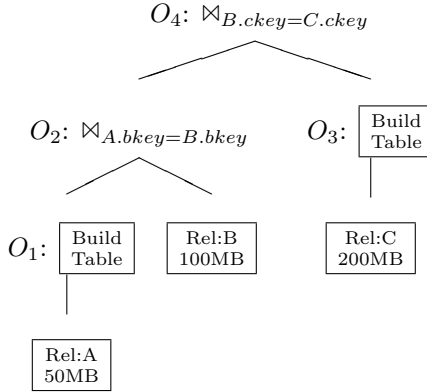


Figure 3: Pipeline used for our tests, where each entry in A matched exactly two entries in B , and each entry in B matched exactly two entries in C and tuple sizes were the same for all three relations. The size of each relation is shown.

results, specify the join selectivities. We used this pipeline because it is simple and has an imbalance in the workload between operations O_2 and O_4 . This allows us to test the effectiveness of the buffer-management system.

Our results show that allocating the appropriate number of buffers for the system can have a rather dramatic impact on overall runtime. In order to determine the ideal number of buffers to use, we ran a series of tests using multiple buffers of varying sizes. We concentrate on the results obtained when using the pointers method in this section for simplicity’s sake.

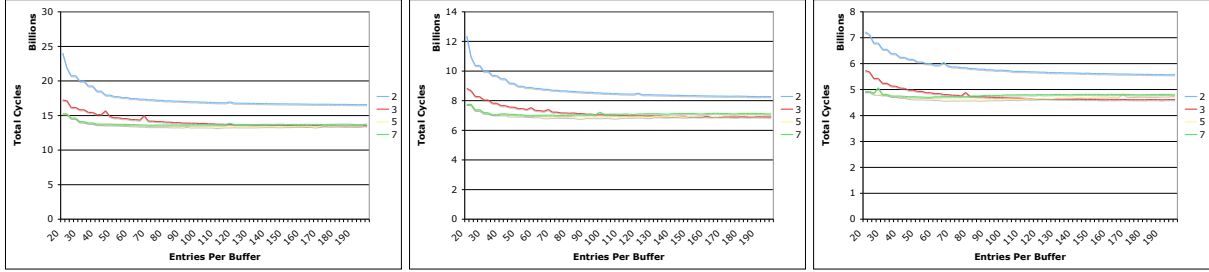
One of the first things we realized in designing the pipelined form of the algorithm was that running a single thread for each operation in the pipeline did not fully utilize processor resources. This is because operation O_2 processes approximately half as many tuples as O_4 . To help alleviate this 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 the hash-join operations (O_2 and O_4 in Figure 3) was 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 such that only one thread can write to a buffer at a time, ensuring correctness.⁵ While parallelizing the hash-join operation was rather straightforward, not every operation in the example pipeline can be parallelized easily (for example the build operations O_1 and O_3). The general problem of parallelizing database operations is worthy of further research in this context.

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.

Another trend that can be observed from these graphs is that as the number of entries in each buffer decreases, the runtime increases. This is logical because when the system is utilizing smaller buffers the overhead for inter-process communication becomes much greater. The effect of this is much

⁵While 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.



(a) Tuple Size=30 Bytes

(b) Tuple Size=60 Bytes

(c) Tuple Size=100 Bytes

Figure 4: Cycles to run the join pipeline on the Pentium 4 when using pointers to pass the intermediate relation. The numbers on the right represent how many buffers were in use by the system.

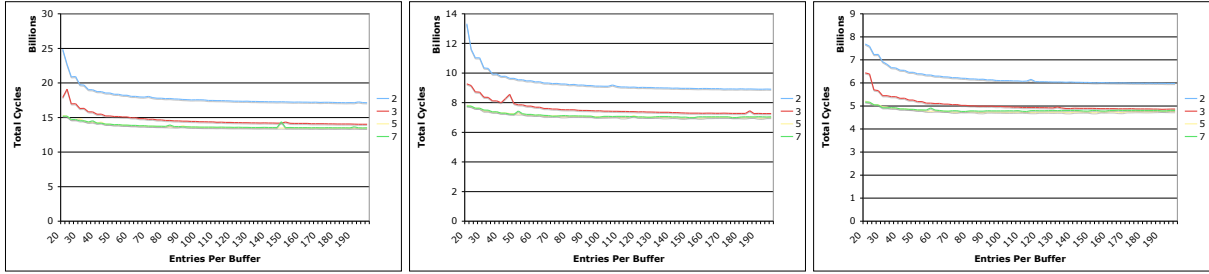
more pronounced when fewer buffers are available to the threads. To minimize cache usage, it is recommended to choose a buffering scheme such that the number and size of buffers is minimized while still obtaining runtimes on the plateau of the graphs shown in Figure 4.

5.1 Pointers Versus Materialization

Figure 5 shows data similar to those of Figure 4, but does so for the case of full materialization of intermediate results. Examining Figures 4 and 5, it is apparent that the runtimes are similar for the pointer version and the full version. This seemingly counters the previous belief that using pointers rather than copying the input tuples to a new buffer saves a significant portion of time that would otherwise be spent doing useful work.

However upon further examination we have found a combination of reasons for this discrepancy:

- operation O_2 (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 will 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 tuple 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 data, cache misses 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 memory copies 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.



(a) Tuple Size=30 Bytes

(b) Tuple Size=60 Bytes

(c) Tuple Size=100 Bytes

Figure 5: Cycles to run the 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.

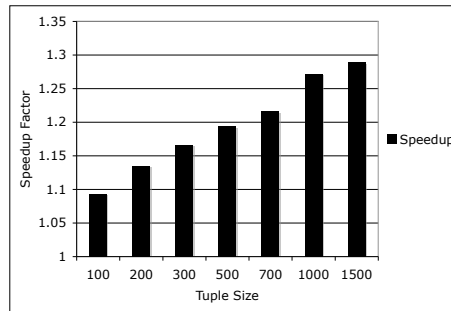


Figure 6: Speedups obtained by using pointers.

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 6, 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, even when using tuples as small as 20-bytes.

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 stalled⁶.

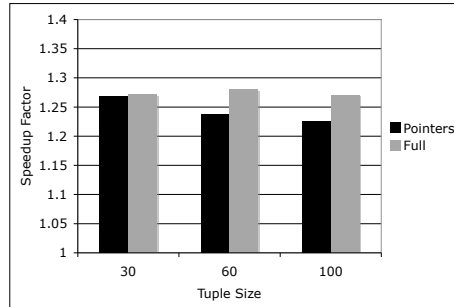


Figure 7: Multithreaded speedup factors over single-threaded for pointers and for full materialization.

5.2 Multithreaded Speedup

In order to understand the effects of SMT on the algorithm, we ran the same tests with SMT disabled. We did this to help quantify the speedup that SMT can bring to data-intensive algorithms. Figure 7 shows the speedups obtained when we ran the query pipeline with both hyperthreads enabled versus when we utilized only one hyperthread on the processor.

These numbers are similar to the speedups seen in [12] during the probe phase of the software-pipelining optimized hash join. It is important to note that these speedups were calculated using the minimum runtimes for both the single and multithreaded variants. These results likely underestimate the speedups that would be obtained on a production system, as the runtimes for the non-SMT algorithm show much greater fluctuation[13] due to greater buffer contention between the different operations executing.

We also ran our experiments on the multiprocessor SMT-enabled Xeon. Our experiments were not designed around the Xeon architecture, and the runtimes obtained are significantly larger than those on the Pentium 4. This is partially due to the faster memory subsystem of the Pentium 4, but more importantly it is due to the Northwood Xeon’s inability to properly handle prefetch instruction as explained in [12]. The comparison between these two architectures is given to illustrate how much of an effect minor architectural changes can have on overall system performance. Figure 8 shows the speedups obtained when splitting the threads up among the processors on this system⁷.

These tests allow us to compare the performance of bus-based SMP systems with single-chip SMT systems while executing query pipelines. 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. Additionally these results showed that straight SMP performed better than using a naïve combination of SMT and SMP.

These results suggest that SMP algorithms offers little advantage over SMT on the Northwood platform. One of the reasons for the poor multiprocessor performance is the Xeon’s bus-based architecture combined with the fact that our buffer structures did not take into consideration which processor produced the data it would be operating upon, resulting in excessive data being sent between the multiple Xeons. This bus-based communication (through the passing of data) is not reduced by using larger buffers, as it involves streaming data from one processor’s cache to the other. On a shared single-channel bus architecture (such as the Xeon) this means that one processor cannot access main memory while the other accesses the other processor’s cache.

⁶While 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.

⁷For more detailed performance see [13].

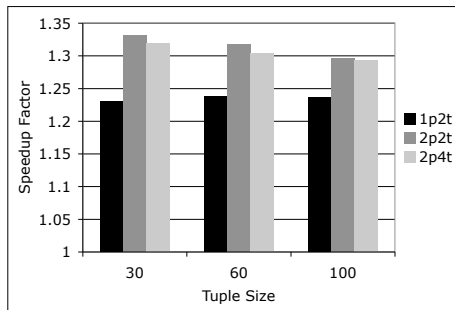


Figure 8: Dual processor results, where 1p2t represents the speedup when running two threads simultaneously on one processor, 2p2t two threads on two processors, and 2p4t four threads on two processors.

Issues such as these will not arise to the same degree in shared-cache architectures that support software prefetching, and the effects would likely be much more limited on systems utilizing a point to point communication architecture. These results are given instead to illustrate how powerful multithreaded processors can be, and show that the speedups achieved on them can be much greater than those on more traditional SMP systems. Additionally they help to illustrate why database system designers should take the architectural platform into consideration when designing future systems as algorithms that were inefficient when run on multiple bus-based processors may be optimal on multithread processors.

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 future processors. The need for highly parallel code to run on future multithreaded processors[3, 17, 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 are necessary to achieve the greatest throughput on newer processors, and additionally will allow the system to better control the execution of algorithms.

Much future work is still needed in expanding our threading model to support further operations (such as merge join, sort, selection, etc), and to examine performance on upcoming multithreaded processors[3, 17, 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 available to run on a given processor.

Future work that examines techniques to parallelize traditionally serial algorithms (such as building hash tables) is also necessary. These techniques must find novel way 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 on these 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.

References

- [1] Intel multi-core processor architecture development background. *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. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman. The microarchitecture of the Intel Pentium 4 processor on 90nm technology. *Intel Technology Journal*, (Q1):4–15, 2002.
- [6] D. Burger and J. R. Goodman. Billion-transistor architectures: There and back again. *IEEE Computer*, 37:22–28, Mar. 2004.
- [7] 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.
- [8] 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.
- [9] N. Dalvi, P. Narayan, P. Bohannon, and H. F. Korth. Optimization of query pipelines for improved cache performance. In *personal correspondence*.
- [10] 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.
- [11] P. Garcia and H. F. Korth. Multithreaded architectures and the sort benchmark. In *DaMoN 2005: First Int'l Workshop on the Data Management on New Hardware*, June 2005.
- [12] P. Garcia and H. F. Korth. Hash-join algorithms on modern multithreaded computer architectures. In *ACM Int'l Conf. on Computing Frontiers*, May 2006.
- [13] P. C. Garcia. Optimizing database algorithms for modern computer architectures. Master's thesis, August 2005. <http://www.cse.lehigh.edu/~pcg2/thesis.pdf>.
- [14] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPU TeraSort: High performance graphics co-processor sorting for large database management. In *Proc. ACM SIGMOD Int'l Conf. on the Management of Data*, June 2006.
- [15] W. D. Hillis and J. Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.
- [16] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, (Q1), 2001.

- [17] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. 2004.
- [18] 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.
- [19] 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.
- [20] F. Mueller. Pthreads library interface, 1993.
- [21] S. Oks. Be aware: To hyper or not to hyper. *Slava Oks Weblog* <http://blogs.msdn.com/slavao/archive/2005/11/12/492119.aspx>, November 2005.
- [22] P. S. Otellini. Multi-core enables performance without power penalties. In *Intel Developer Forum Keynote*, <http://www.embedded-controlseurope.com/pdf/ecedec05p26.pdf>, 2005.
- [23] A. Shatdal, C. Kant, and J. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of 20th Int'l Conf. on Very Large Data Bases*, pages 510–524, 1994.
- [24] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, 5th Edition*. McGraw Hill, 2006.
- [25] 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.
- [26] 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.
- [27] 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.
- [28] 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.