Proceedings of the 6th Hellenic Data Management Symposium (HDMS2007), Athens, Greece, July 2007.

An Analysis of Database System Performance on Chip Multiprocessors

Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju G. Mancheril
Stavros Harizopoulos†, Anastasia Ailamaki and Babak Falsafi

Database Group and Computer Architecture Lab (CALCM) †MIT CSAIL
Carnegie Mellon University

http://www.cs.cmu.edu/~stageddb

ABSTRACT

Prior research shows that database system performance is dominated by off-chip data stalls, resulting in a concerted effort to bring data into on-chip caches. At the same time, high levels of integration have enabled the advent of chip multiprocessors and increasingly large (and slow) on-chip caches. These two trends pose the imminent technical and research challenge of adapting high-performance data management software to a shifting hardware landscape.

In this paper we characterize the performance of a commercial database server running on emerging chip multiprocessor technologies. We find that the major bottleneck of current software is data cache stalls, with L2 hit stalls rising from oblivion to become the dominant execution time component in some cases. We analyze the source of this shift and derive a list of features for future database designs to attain maximum performance. Towards this direction, we propose the adoption of staged database system designs to achieve high performance on chip multiprocessors. We present the basic principles of staged databases and an initial implementation of such a system, called Cordoba.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Relational databases. H. 2.6 [**Database Machines**]. H. 3.4 [**Systems and Software**] Performance evaluation. C. 1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]

General Terms

Performance, Design, Experimentation.

Keywords

Database Engine, Performance Characterization, Chip Multiprocessors, Staged Database Systems.

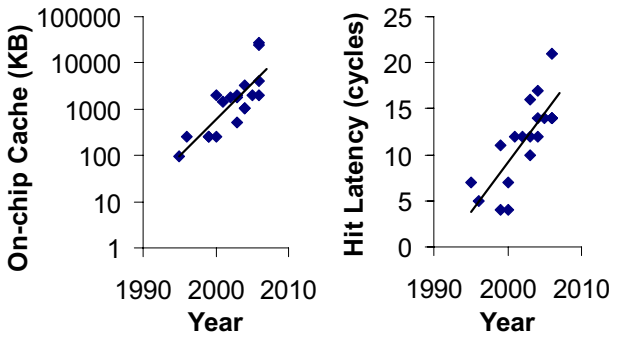

Figure 1. Historic trends of on-chip caches on (a) size, and (b) latency.

1. INTRODUCTION

Database management systems is a multibillion dollar industry with high-end database servers employing state-of-the-art processors to maximize performance. Unfortunately, recent studies show that processors are far from realizing their maximum performance. Prior research [23] indicates that adverse memory access patterns in database workloads result in poor cache locality and overall performance. Database systems are known to spend at least half of their execution time on stalls, implying that data placement should focus on the second-level (L2) cache [4], typically found on chip in modern processors.

Over the past decade, advancements in semiconductor technology have dramatically changed the landscape of on-chip caches. The increase in the number of transistors available on-chip has enabled on-chip cache sizes to increase exponentially across processor generations. The trend of increasing on-chip cache sizes is apparent in Figure 1 (a), which presents historic data on the on-chip cache sizes of several processors in the last two decades. The upward trend in cache sizes shows no signs of a slowdown. Industry advocates large caches as a microarchitectural technique that allows designers to exploit the available transistors efficiently to improve performance [8], leading to modern processors with mega-caches on chip (e.g., 16MB in Dual-Core Intel Xeon 7100 [28], and 24MB in Dual-Core Intel Itanium 2 [37]).

Large caches, however, come at the cost of high access latency. Figure 1 (b) presents historic data on the L2 cache

access latency, indicating that on-chip L2 latency has increased more than 3-fold during the past decade — e.g., from 4 cycles in Intel Pentium III (1995) to 14 cycles in IBM Power5 (2004). Caches enhance performance most when they capture fully the primary working set of the workload; otherwise, they provide only marginal improvements in the miss rate as size increases. Database workloads typically have a small primary working set which can be captured on chip, and a large secondary set which is beyond the reach of on-chip caches for modern processors. Conventional wisdom dictates that large on-chip caches provide significant performance benefits as they eliminate off-chip memory requests. In reality, a large cache may degrade the performance of database workloads because the cache's high latency slows the common case (cache hits) and introduces stalls in the execution, while the additional capacity fails to lower the miss rate enough to compensate.

Over the past decades, microprocessor designs focused primarily on tolerating stalls by exploiting instruction-level parallelism (ILP). The resulting wide-issue out-of-order (OoO) processors overlap both computation and memory accesses, but fall short of realizing their full potential when running database workloads. Database workloads exhibit large instruction footprints and tight data dependencies that reduce instruction-level parallelism and incur data and instruction transfer delays [4, 26]. Thus, increasingly aggressive OoO techniques yield diminishing returns in performance, while their power dissipation is reaching prohibitive levels [8]. The shortcomings of out-of-order processors, along with the continued increase in the number of transistors available on chip, have encouraged most vendors to integrate multiple processors on a single chip, instead of simply increasing the complexity of individual cores. The resulting chip multiprocessor (CMP) designs may affect data stalls through promoting on-chip data sharing across cores and increasing contention for shared hardware resources.

In this paper we investigate the performance of database workloads on modern CMPs and identify data cache stalls as a fundamental performance bottleneck. Recent work in the database community [3, 4, 26] attributes most of the data stalls to off-chip memory accesses. In contrast to prior work, our results indicate that the current trend of increasing L2 latency intensifies stalls due to L2 hits[1], shifting the bottleneck from off-chip accesses to on-chip L2 hits. Thus, merely bringing data on-chip is no longer enough to attain maximum performance and sustain high throughput. Database systems must also optimize for L1D locality.

In this study we recognize that chip multiprocessor designs follow two distinct schools of thought, and present a taxonomy of processor designs and DBMS workloads to distinguish the various combinations of workload and system configuration. We divide chip multiprocessors into two "camps." The *fat camp* employs wide-issue out-of-order processors and addresses data stalls by exploiting instruction-level parallelism (e.g., Intel Core Duo [1], IBM Power 5 [19]). The *lean camp* employs heavily multithreaded in-order processors to hide data stalls across threads by overlapping data access latencies with useful computation (e.g., Sun UltraSparc T1 [20]). Even though LC is heavily multithreaded, it is a much simpler hardware design than the complex out-of-order FC. We divide database applications into *saturated* workloads, in which idle processors always find an available thread to run, and *unsaturated* workloads, in which processors may not always find threads to run, thereby exposing data access latencies. We characterize the performance of each database workload and system configuration pair within the taxonomy through cycle-accurate full-system simulations using FLEXUS [14] of OLTP (TPC-C) and DSS (TPC-H) workloads on a commercial DBMS. Our results indicate that:

- High on-chip cache latencies shift the data stall component from off-chip data accesses to L2 hits, to the point where up to 35% of the execution time is spent on L2 hit stalls for our workload and CMP configurations.
- Increasing the L2 cache size from 4MB to 26MB reduces throughput by up to 30%. In comparison, increasing the cache size while keeping the L2 hit latency constant yields nearly 2x speedup due to lower miss rates.
- High levels of on-chip core integration increase L2 hit rates, improving performance by 12-15% and increasing the relative contribution of L2 hit stalls to 10% and 25% of execution time, respectively, for DSS and OLTP.
- The combined effects of high L2 latency and on-chip core integration increase the contribution of L2 hit stalls on execution time by a factor of 5 for DSS and a factor of 7 for OLTP over traditional symmetric multiprocessors, explaining the observed departure from prior research findings.
- Conventional DBMS hide stalls only in one out of four combinations of chip designs and workloads. Despite the significant performance enhancements that stem from chip-level parallelism, the fat camp still spends 46-64% of execution time on data stalls. The lean camp efficiently overlaps data stalls when executing saturated workloads, but exhibit up to 70% longer response times than the fat camp for unsaturated workloads.
- To hide stall time when executing DBMS across the entire spectrum of workloads and systems, the software must improve both L1D reuse/locality and exhibit high thread-level parallelism across and within queries and transactions. Data locality helps eliminate stalls independent of workload type. Increased parallelism helps exploit the abundance of on-chip thread and processor execution resources when the workload is not saturated.

We propose the adoption of staged database system designs as a flexible system architecture that has the potential to achieve high performance in the emerging chip multiprocessors. We present the basic principles of the staged database design and an initial implementation of such a system, called Cordoba.

The remainder of this document is structured as follows. Section 2 proposes a taxonomy of chip multiprocessor technologies and workloads. Section 3 presents our experimental methodology and Section 4 analyzes the behavior of a commercial database server on chip multiprocessors, as a function of hardware designs and workloads. Section 5 discusses the effects of hardware parameters on data stalls. Section 6 discusses software techniques to enhance parallelism and reduce the L2 hit stall component. Section 7 introduces the basic principles of staged database systems and Cordoba, our prototype staged execution engine. Finally, Section 8 presents related work, and Section 9 concludes.

[1] We refer to the time spent by the processor accessing a cache block that missed in L1D but was found in L2 as "L2 hit stalls".

Table 1. Chip multiprocessor camp characteristics.

Core Technology	Fat Camp (FC)	Lean Camp (LC)
Issue Width	Wide (4+)	Narrow (1 or 2)
Execution Order	Out-of-order	In-order
Pipeline Depth	Deep (14+ stages)	Shallow (5-6 stages)
Hardware Threads	Few (1-2)	Many (4+)
Core Size	Large (3 x LCsize)	Small (LC size)

2. CMP CAMPS AND WORKLOADS

In this section we propose a taxonomy of chip multiprocessor technologies and database workloads and analyze their characteristics. To our knowledge, this is the first study to provide an analytic taxonomy of the behavior of database workloads in such a diverse spectrum of current and future chip designs. A recent study [13] focuses on throughput as the primary performance metric to compare server workload performance across chip multiprocessors with varying processor granularity, but has stopped short of a detailed performance characterization and breakdown of where time is spent during execution. Through a series of simulations we find that the behavior of database systems varies as a function of hardware and workload type, and that conventional database systems fail to provide high performance across the entire spectrum. The taxonomy enables us to concentrate on each segment separately and derive a list of features a database system should support.

2.1 Fat Camp vs. Lean Camp

Hardware vendors adopt two distinct approaches to chip multiprocessor design. One approach uses cores that target maximum single-thread performance through sophisticated out-of-order execution and aggressive speculation (fat-camp or FC). Representative chip multiprocessors from this camp include Intel Core Duo [1] and IBM Power5 [19]. The second approach favors much simpler designs with cores that support many thread contexts[1] in hardware (lean-camp or LC). Such cores overlap stalls in a given thread with useful computation by other threads. Sun UltraSPARC T1 [20] and Compaq Piranha [5] fall into this camp. Table 1 summarizes the characteristics of each technology camp.

Integrating multiple cores on a chip multiprocessor exhibits similar effects within each camp (e.g., increase in shared resource contention). In this paper we study the increasing performance differences between fat and lean camps when running identical database workloads, assuming that both camps are supported by the same memory hierarchy. Thus, it suffices to analyze the characteristics of each camp by focusing on the characteristics of the different core technologies within each camp.

Because LC cores are heavily multithreaded, we expect them to hide stalls efficiently and provide high and scalable throughput when there is enough parallelism in the workload. However, when the workload consists of a few threads, the LC cores cannot find enough threads to overlap stalls, leaving long data access latencies exposed. On the other hand, the FC cores are optimized for single-thread performance through wide pipelines that issue/complete multiple instructions per cycle, and out-of-order speculative execution. These features exploit instruction-level parallelism within the workload to hide stalls.

[1] We refer to hardware threads as "hardware contexts" to distinguish them from software (operating system) threads.

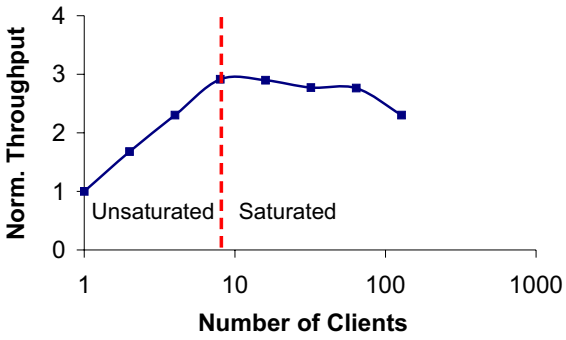

Figure 2. Unsaturated vs. saturated workloads.

Thus, we expect LC cores to outperform FC cores when there is enough parallelism in the workload, even with much lower single-thread performance than that of an FC core. However, when the workload consists of few threads, we expect the response time of the single-thread optimized FC cores to be significantly lower than the corresponding response time of their LC counterparts.

In addition to the performance differences when comparing single cores, an LC CMP can typically fit three times more cores in one chip than an FC CMP, resulting in roughly an order of magnitude more hardware contexts in the same space. In this paper we do not apply constraints on the chip area. Keeping a constant chip area would favor the LC camp because it would have a larger on-chip cache than the FC camp, allowing LC to attain even higher performance in heavily multithreaded workloads, because LC is able to hide L2 stalls through multithreading.

2.2 Unsaturated vs. Saturated Workloads

Database performance varies with the number of requests serviced. Our unsaturated workload highlights single-thread performance by assigning one worker thread per query (or transaction) it receives. A conventional DBMS can increase the parallelism through partitioning, but in the context of this paper we can treat this as having multiple clients (instead of threads). As explained in Section 6.1, the reader should also keep in mind that not all query plans are trivially parallelizable.

We observe that the performance of a database application falls within one of two regions, for a given hardware platform, and characterize the workload as unsaturated or saturated. A workload is unsaturated when processors do not always find threads to run. As the number of concurrent requests increases, performance improves by utilizing otherwise idle hardware contexts. Figure 2 illustrates throughput as a function of the number of concurrent requests in the system when running TPC-H queries on a commercial DBMS on a real 4-core IBM Power5 (FC) server. Increasing the number of concurrent requests eventually results in a saturated workload, where there are always available threads for idle processors to run. Peak performance occurs at the beginning of the saturated region; increasing the number of concurrent requests too far

overwhelms the hardware, reducing the amount of useful work performed by the system and lowering performance.

3. EXPERIMENTAL METHODOLOGY

We use FLEXUS [14] to provide accurate simulations of chip multiprocessors and symmetric multiprocessors running unmodified commercial database workloads. FLEXUS is a cycle-accurate full-system simulator that simulates both user-level and operating system code. We use the SimFlex statistical sampling methodology [35]. Our samples are drawn over an interval of 10 to 30 seconds of simulated time (as observed by the operating system in functional simulation) for OLTP, and over the complete workload execution for DSS. We show 95% confidence intervals on performance measurements using paired measurement sampling. We launch measurements from checkpoints with warmed caches and branch predictors, then run for 100,000 cycles to warm queue and interconnect state prior to collecting measurements of 50,000 cycles. We use the aggregate number of user instructions committed per cycle (i.e., committed user instructions summed over the simulated processors divided by total elapsed cycles) as our performance metric, which is proportional to overall system throughput [35].

We characterize the performance of database workloads on an LC CMP and an FC CMP with the UltraSPARC III instruction set architecture running the Solaris 8 operating system. The LC CMP employs four 2-issue superscalar in-order cores. The LC cores are 4-way multithreaded, for a total of 16 hardware contexts on the LC CMP. The hardware contexts are interleaved in round-robin fashion, issuing instructions from each runnable thread in turn. When a hardware context stalls on a miss it becomes non-runnable until the miss is serviced. In the meantime, the LC core executes instructions from the remaining contexts.

The FC CMP employs four aggressive out-of-order cores that can issue four instructions per cycle from a single hardware context. The two CMP designs have identical memory subsystems and clock frequencies and feature a shared on-chip L2 cache with size that ranges from 1MB to 26MB.

We estimate cache access latencies using Cacti 4.2 [36]. Cacti is an integrated cache access time, cycle time, area, leakage, and dynamic power model. By integrating all these models together, cache trade-offs are all based on the same assumptions and, hence, are mutually consistent. In some experiments we purposefully vary the latency of caches beyond the latency indicated by Cacti to explore the resulting impact on performance or to obtain conservative estimates.

Our workloads consist of OLTP (TPC-C) and DSS (TPC-H) benchmarks running on a commercial DBMS. The saturated OLTP workload consists of 64 clients submitting transactions on a 100-warehouse database. The saturated DSS workload consists of 16 concurrent clients running four queries from the TPC-H benchmark, each with random predicates. We select the queries as follows [29]: Queries 1, 6 are scan-dominated, Query 16 is join-dominated and Query 13 exhibits mixed behavior. To achieve practical simulation times we run the queries on a 1GB database. We corroborate recent research that shows that varying the database size does not incur any microarchitectural behavior changes [29]. Unsaturated workloads use the above methodology running only a single client, with intra-query parallelism disabled to highlight single-thread performance. We tune both the OLTP and DSS workloads to minimize I/O overhead and maximize CPU and memory system utilization.

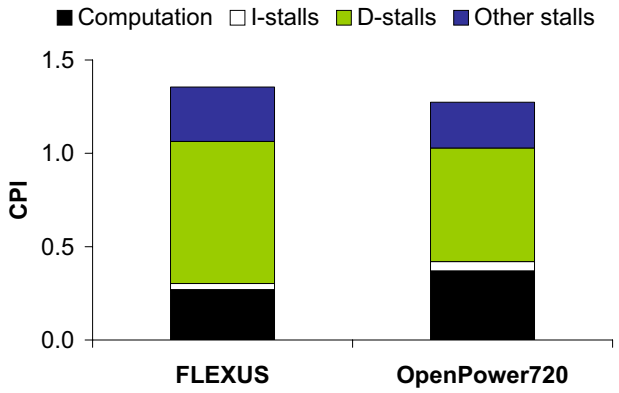

Figure 3. FLEXUS validation using the saturated DSS workload.

We validate FLEXUS by comparing against an IBM OpenPower720 server that runs the same workloads. We calculate the cycles per instruction (CPI) on OpenPower720 by extracting Power5's hardware counters through pmcount [2], post-processing the raw counters using scripts kindly provided by IBM, and comparing the results with a FLEXUS simulation that approximates the same IBM server. Figure 3 presents the absolute CPI values and their respective breakdowns. The overall simulated CPI is within 5% of the measured CPI for both OLTP and DSS workloads. The computation component for OpenPower720 is 10% higher, which we attribute to Power5's instruction grouping and cracking overhead. The data stall component is 15% higher for FLEXUS due to the absence of a hardware prefetcher mechanism.

While employing a stride prefetcher will not change the performance trends that are the focus of this paper, it is instructive to discuss its performance implications on our workload mix. Prior research [32] measures the impact of hardware prefetching on the performance of OLTP and DSS workloads and finds that even complex hardware prefetchers that subsume stride prefetchers yield less than 10% performance improvement for OLTP workloads and scan-dominated DSS queries. Join-dominated DSS queries do see as much as 50% improvement, but contribute relatively little to total execution time in our DSS query mix. Even if a stride prefetcher could match the performance improvements of [32], we estimate that the performance improvement due to a stride prefetcher on our OLTP workload will be less than 10%, while the performance improvement on our scan-dominated DSS workload will be less than 20%. However, the performance trends due to the increasing L2 latencies will remain the same.

4. DBMS PERFORMANCE ON CMPS

In this section we characterize the performance of both CMP camps on a commercial DBMS running unsaturated and saturated DSS and OLTP workloads. For unsaturated workloads the performance metric of interest is response time, while for saturated workloads the performance metric of interest is throughput. Figure 4 (a) presents the response time of the LC CMP normalized to the FC CMP when running unsaturated (single-thread) workloads. Figure 4 (b) presents

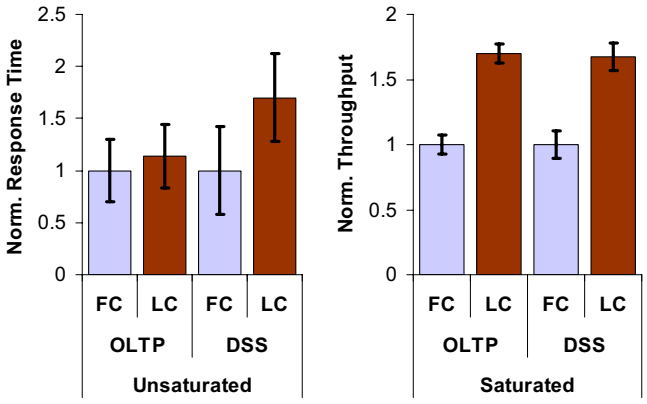

Figure 4. (a) Response time and (b) throughput of LC normalized to FC.

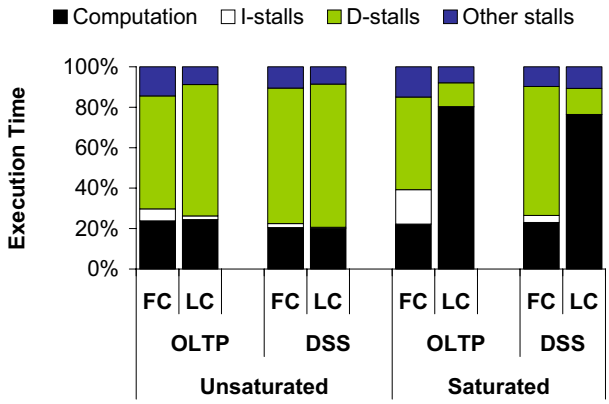

Figure 5. Breakdown of execution time.

the throughput of the LC CMP normalized to the throughput of the FC CMP when running saturated workloads.

The LC CMP suffers up to 70% higher response times than FC when running unsaturated (single-thread) DSS workloads and up to 12% higher when running unsaturated OLTP workloads, corroborating prior results [26]. The performance difference between FC and LC on unsaturated OLTP workloads is narrower due to limited ILP. Even though FC exhibits higher single-thread performance than LC, the LC CMP achieves 70% higher throughput than its FC counterpart when running saturated workloads (Figure 4 b).

Figure 5 shows the execution time breakdown for each camp and workload combination. Although we configure the CMPs with an unrealistically fast 26MB shared L2 cache, data stalls dominate execution time in three out of four cases. While FC spends 46% - 64% of execution time on data stalls, LC spends at most 13% of execution time on data stalls when running saturated workloads, while spending 76-80% of the time on useful computation. The multiple hardware contexts in LC efficiently overlap data stalls with useful computation, thereby allowing LC to outperform significantly its FC counterpart on saturated workloads.

Despite prior work [4] showing that instruction stalls often dominate memory stalls when running database workloads, our CMP experiments indicate that data stalls dominate the memory access component of the execution time for all workload/camp combinations. Both camps employ instruction stream buffers [18], a simple hardware mechanism that automatically initiates prefetches to successive instruction cache lines following a miss. Our results corroborate prior research [26] that demonstrates instruction stream buffers efficiently reduce instruction stalls. Because of their simplicity, instruction stream buffers can be employed easily by the majority of chip multiprocessors, thus we do not further analyze instruction cache performance.

We conclude that the abundance of threads in saturated workloads allows LC CMPs to hide data stalls efficiently. The multiple hardware contexts available on the LC CMP allow it to perform useful computation while some of the contexts are stalled on long latency data access operations, thereby improving overall throughput. In contrast, the FC CMP fails to utilize fully its hardware resources because database workloads exhibit limited ILP. FC processors would also benefit from multithreaded operation, but their complexity limits the number of hardware contexts they can employ. Our calculations show that each FC core would require more than 15 hardware contexts to fully overlap data stalls, which is infeasible due to the complexity and power implications it entails. Thus, FC CMPs cannot hide data stalls the way context-rich LC CMPs can.

However, we expect that in spite of less than ideal performance on database workloads, FC CMPs will still claim a significant market share due to their unparalleled single-thread performance and optimized execution on a variety of other workloads (e.g., desktop, scientific computing). Thus, database systems must be designed to perform well on both CMP camps, independent of workload type. To maximize performance across hardware and workload combinations, database systems must exhibit high thread-level parallelism across and within queries and transactions, and improve data locality/reuse. Increased parallelism helps exploit the abundance of on-chip thread and processor execution resources when the workload is not saturated. Data locality helps eliminate stalls independent of workload type.

Figure 5 shows that in six out of eight combinations of hardware and workloads, data stalls dominate execution time even with unrealistically fast and large caches. In Section 5 we analyze the data stall component of execution time to identify dominand subcomponents and trends, that will help guide the implementation and optimization of future database software. In the interest of brevity, we analyze data stalls by focusing on saturated database workloads running on FC CMPs, but the results of our analysis are applicable across all combinations of hardware and workloads that exhibit high data stall time.

5. ANALYSIS OF DATA STALLS

In this section we analyze the individual sub-components of data cache stalls and identify the emerging importance of L2 hit stalls, which account for up to 35% of execution time for our hardware configurations and workloads. This represents a a 7-fold increase as compared to traditional symmetric multiprocessors with small caches running the same workloads.

Section 5.1 explores the impact of increased on-chip cache sizes on the breakdown of data stalls, both for constant (low) hit latencies and for realisitic latencies provided by Cacti. In Section 5.2 we analyze the impact of integrating multiple cores into a single chip. Finally, in Section 5.3 we study the effects

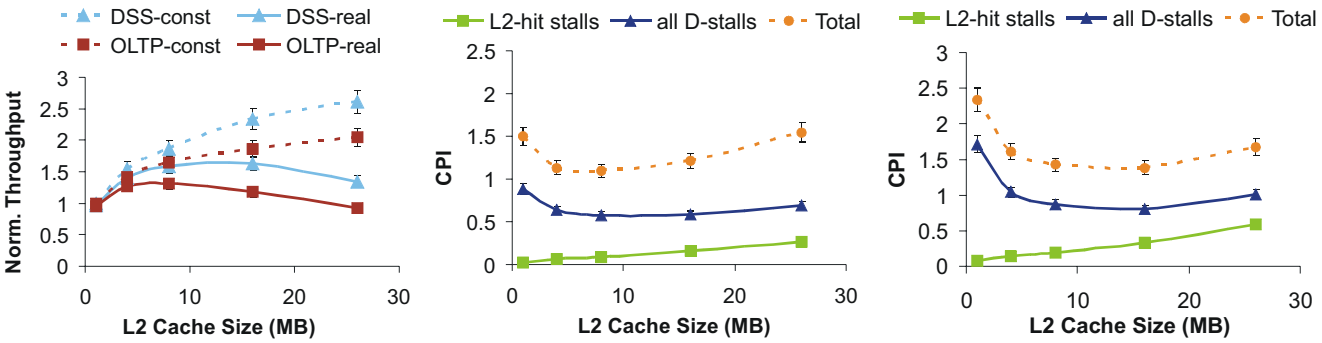

Figure 6. Effect of cache size and latency on (a) throughput, (b) CPI contributions for OLTP, and (c) CPI contributions for DSS.

of high levels of on-chip integration by increasing the number of available cores on chip.

5.1 Impact of On-Chip Cache Size

Large on-chip L2 caches shift the data stall bottleneck in two ways. First, large caches exhibit high hit rates. As more requests are serviced by the cache, data stalls shift from memory to L2 hits. Second, rising hit latencies penalize each hit and increase the number of stalls caused by L2 hits without changing the number of accesses to other parts of the memory hierarchy.

Figure 6 (a) presents the impact of increasing cache size on DBMS performance. We simulate both OLTP and DSS workloads on a FC CMP, with cache sizes ranging from 1MB to 26MB. To separate the effect of hit rates from that of hit latencies, we perform two sets of simulations. The upper (dotted) pair of lines shows the performance increase achieved when the hit latency remains fixed at an unrealistically low 4 cycles. The lower (solid) pair of lines shows the performance under the more reasonable hit latencies estimated using Cacti for each cache configuration. These estimates are conservative because hit latencies estimated by Cacti are typically lower than the ones achieved in commercial products.

In all cases, increasing the cache size significantly improves performance as more of the primary L2 working set fits in the cache. However, the realistic-latency and constant-latency performance curves quickly begin to diverge, even before the cache captures a significant fraction of the entire working set. Even though there is no cycle penalty for increasing L2 sizes in the constant-latency case, we see diminishing returns because even the biggest cache fails to capture the large secondary L2 working set. In contrast, realistic hit latencies further reduce the benefit of larger caches, and the added delay begins to outweigh the benefits of lower miss rates. The adverse effects of high L2 hit latency reduce the potential performance benefit of large L2 caches by up to 2.2x for OLTP and 2x for DSS.

Figure 6 (b) and (c) show the contributions of realistic L2 hit latencies to data stalls and overall CPI for OLTP and DSS respectively. In the constant-latency case (not shown) the stall component due to L2 hits quickly stabilizes at less than 5% of the total CPI. On the other hand, realistic latencies are responsible for a growing fraction of the overall CPI, especially in DSS, where they become the single largest component of execution time. The remainder of the CPI increase comes from instruction stalls due to L2 hits, again an artifact of larger (and slower) caches. Instruction stalls due to L2 are especially evident in the OLTP workload, where they account for roughly half of the overall CPI increase.

Increasing cache sizes and their commensurate increase in latency can have dramatic effects on the fraction of time spent on L2 hit data stalls. For our workloads running on a FC CMP we measure a 12-fold increase in time spent in L2 hit stalls when increasing the cache size from 1MB to 26MB; rising hit latencies are responsible for up to 78% of this increase.

5.2 Impact of Core Integration on Single Chip

In this section we study the outcome of integrating multiple processing cores on a single chip. We compare the performance of a commercial database server running OLTP and DSS workloads in two variants of our baseline system: (a) a 4-processor SMP with private 4MB L2 caches at each node, and (b) a 4-core CMP with a single shared 16MB L2.

Figure 7 presents normalized CPI breakdowns for the two systems, with labels indicating the actual CPI. We observe that the performance of the CMP systems is higher. The difference in the performance between the SMP and the CMP systems can be attributed to the elimination of coherence traffic. Data accesses that result in long-latency coherence misses in the SMP system are converted into L2 hits on the shared L2 cache of the CMP and fast L1-to-L1 on-chip data transfers. Thus, the L2 hit stall component of CPI increases by a factor of 7 over the corresponding SMP designs, explaining the disparity of our results as compared to prior research findings [29].

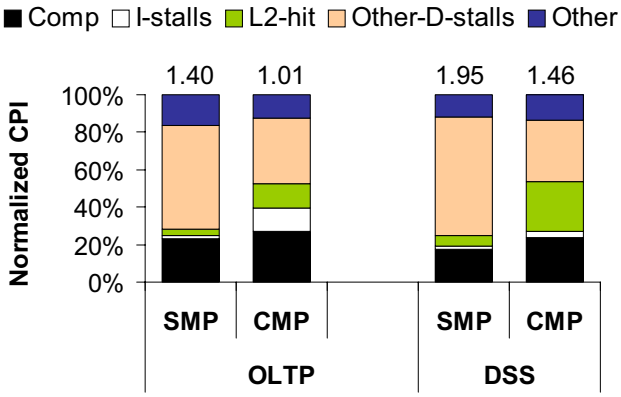

Figure 7. Impact of chip multiprocessing on CPI.

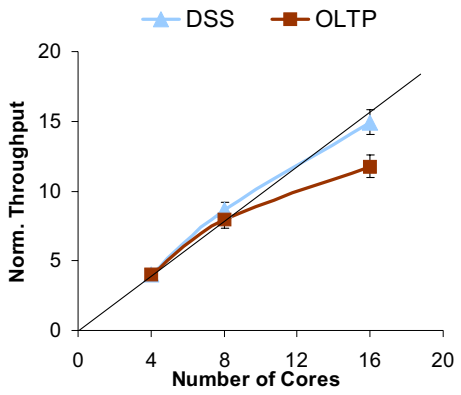

Figure 8. Impact of CMP core count on throughput.

5.3 Impact of On-Chip Core Count

Chip multiprocessors integrate multiple cores on a single chip, which promotes sharing of data through the common L2 cache. At the same time, contention for shared hardware resources my offset some of the benefits of fast on-chip data sharing. To study the impact of high levels of core integration on chip we simulate saturated OLTP and DSS workloads on a FC chip multiprocessor with a 16MB shared L2 as we increase the number of cores from 4 (the baseline) to 16.

Figure 8 presents the change in performance as the number of processing cores increases. The diagonal line shows linear speedup as a reference. We observe a 9% superlinear increase in throughput at 8 cores for DSS, due to an increase in sharing, after which pressure on the L2 cache adversely affects performance for both workloads. OLTP, in particular, realizes only 74% of its potential linear performance improvement. The pressure on the cache is not due to extra misses — in fact, the L2 miss rate continues to drop due to increased sharing as more cores are added. Rather, physical resources such as cache ports and status registers induce queueing delays during bursts of misses. These correlated misses are especially common in OLTP and are largely responsible for the sublinear speedup when adding more cores.

5.4 Ramifications

Our analysis of data stalls and the impact of microarchitectural parameters on L2 hit stalls have several ramifications. First, the shifting L2 hit bottleneck indicates that simply bringing data on chip no longer suffices to attain maximum performance. In the future it will become increasingly important to bring data beyond L2 and closer to L1. The shift arises primarily from the combination of increased L2 hit latencies and the effects of integration; lower miss rates from larger cache sizes contribute much less to this effect.

Second, incorporating large (slow) caches on chip can have detrimental effects in the performance of DBMSs. Optimized future CMP designs should incorporate caches large enough to capture the primary L2 working set, but not larger, so they can maintain low hit latencies. This observation runs counter to the conventional wisdom that larger caches are always a good way to use extra transistors [8].

Third, increasing the number of cores that share an on-chip L2 cache does not cause an inordinate number of additional cache misses for database workloads. In fact, these workloads exhibit significant sharing between cores. However, they do cause extra pressure that can reduce performance in spite of the lower miss rate. We expect that future CMP designs will feature specially-designed L2 caches to reduce this pressure, allowing workloads to benefit from the effects of sharing.

6. PARALLELISM AND LOCALITY

To maximize performance across the full spectrum of workloads and hardware configurations a DBMS must intelligently balance parallelism and locality. Under light load both fat and lean camp systems suffer from idle hardware contexts and exposed data stalls. The database system should try to improve response time by splitting requests into many software threads that exploit the available hardware resources.

6.1 Increasing Parallelism

Database applications are inherently parallel. A workload consisting of multiple queries or transactions causes the creation of a large number of worker threads to serve the requests, resulting in inter-transaction and -query parallelism. In many cases, individual requests can be decomposed into multiple threads to increase the level of parallelism even further. This is particularly useful when the workload is unsaturated, in which case dividing work among more threads can utilize the otherwise idle hardware contexts.

Pipelining and operator-level parallelism [17] divide a query into producer-consumer pairs, which can execute in parallel and reduce overall execution time. Besides increasing parallelism, pipelining and operator-level parallelism also enhance the temporal locality of intermediate data, so they are quickly reused and can be safely discarded afterwards. For example, they allow the DBMS to avoid materializing intermediate results, which consume buffer pool pages and may incur significant I/O overheads when their size grows.

Partitioning, which is orthogonal to operator-level parallelism, divides the input data set into subsets that can be processed concurrently. Database tables can be partitioned either vertically or horizontally, depending on the expected access patterns (i.e., queries). A query that accesses a partitioned table may then be divided into a number of sub-queries, each operating on one data partition, thus improving performance.

However, partitioning is static and complex to set up. Workloads where the types of requests are unknown in advance or fluctuate often gain little from any one partitioning scheme, and cannot be repartitioned easily. Moreover, there always exist queries that a chosen partitioning scheme cannot handle efficiently (e.g., vertical partitioning penalizes queries that access entire rows). Also, some queries and most transactions are not amenable to partitioning, or cannot be fully partitioned. For example, queries that require sorted outputs but read unsorted inputs must serialize at a sort stage.

6.2 Improving Data Locality

Saturated LC systems exhibit nearly ideal performance, as they are designed to overlap stalls with useful computation from a pool of eligible threads. On the other hand, saturated FC systems face a significant data stall bottleneck that must be addressed to improve performance. Any technique that reduces data stalls will also improve the performance of unsaturated workloads, independent of hardware configuration.

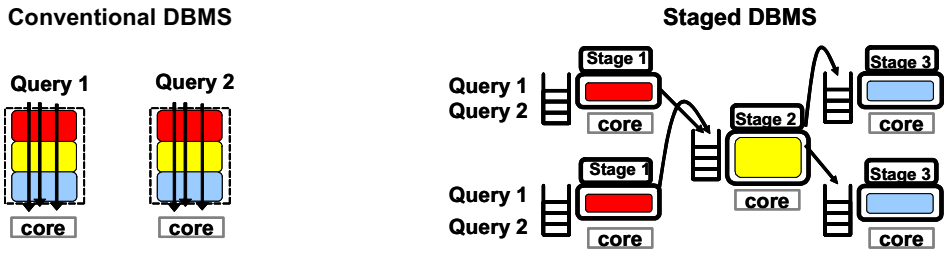

Figure 9. Example of query execution in (a) conventional DBMS, and (b) staged DBMS.

STEPS [16] explicitly schedules threads in transactional workloads to achieve high instruction locality. Similar techniques may improve data locality. For example, binding producer/consumer threads together on the same core can improve L1-D locality. Similar to STEPS, the producer could yield to the consumer whenever it produces enough data to fill the L1-D cache, allowing the consumer to use the data before it is pushed further down the memory hierarchy. The abundance of hardware contexts on LC cores also provides interesting scheduling opportunities, especially in cases where multiple threads access the same data over a short time window.

A growing number of research proposals are striving to provide cache-friendly access patterns and data structures. MonetDB/X100 [39] uses vectorized processing and column-wise storage to improve both spatial and temporal data locality in query processing. PAX [3] restructures the data layout in the disk and memory pages to reduce the number of cache misses. Similarly, [27] proposes a modified (read-only) B+ tree that improves cache behavior. Chilimbi [11] proposes the use of compilers to automatically produce cache-conscious structures. Chen et al. [10] utilize prefetching to hide the data access latencies in hash-joins, while [30] describe software optimization techniques for cache-conscious execution.

However, these approaches focus mainly on bringing data on chip, thereby improving L2 hit rates. While these techniques will improve L1 hit rates as well, this is merely a by-product of their main focus and they do not account for the small L1-D sizes. As L2 accesses are getting increasingly slow and data stalls shift from off-chip accesses to on-chip hits, we need to re-evaluate these techniques to improve also L1-D hit rates.

7. STAGED DATABASE SYSTEMS

Conventional DBMSs are optimized for outdated hardware architectures, assuming private hardware resources and coarse-grain OS-managed threads with no coordination of resource usage among them. However, modern chip multiprocessors employ shared hardware resources and facilitate fine-grain communication between software threads. The monolithic design of traditional DBMSs impedes their ability to overcome the performance challenges imposed by the shifting hardware landscape and utilize the hardware resources efficiently. On the other hand, unconventional database system designs (e.g., staged database systems) may provide efficient solutions. We believe that staged database systems hold great potential in addressing both the parallelism and locality requirements, so they are the focus of our discussion in this section.

7.1 Design and Opportunities

The recently proposed staged software servers [15, 17] have emerged as a potential successor to conventional software designs, because they naturally enhance the application's inherent parallelism, and exhibit properties that can be leveraged by the hardware to eliminate performance bottlenecks. To highlight the design philosophy behind staged software, Figure 9 on the left depicts how a conventional software server processes requests. As separate queries enter the system, a single thread is allocated for each one. The threads execute on different processors, and perform each operation (depicted by a box) in sequence.

On the other hand, staged software servers, shown in Figure 9 (right), decompose a traditional monolithic software server into self-contained stages connected to each other through queues. Each stage implements limited functionality (e.g., one relational operator), maintains private data and control mechanisms, and has its own pool of threads. Incoming requests are decomposed into "packets", one packet per operation, and routed to the appropriate stages where they wait on their respective queues. Each packet indicates the work requested on behalf of the incoming query, and can be scheduled individually, as its execution is decoupled from the others. Because the stages participating in each query execute in parallel on different cores, staging naturally enhances workload parallelism and can utilize otherwise idling computational resources.

Because communication between stages is explicit, the access and sharing patterns are known in advance and can be exposed to the execution system. Thus, no complicated hardware prediction mechanisms are required to re-discover them to hide the data access latency [12, 24, 34], overcoming a significant obstacle of conventional DBMS software [31]. Instead, a staged database system may enhance L1-D locality through explicit scheduling of packets and producer-consumer pairs.

At the same time, a staged DBMS's highly modular design easily incorporates dynamic work sharing policies. When a new packet arrives at a stage's queue, the stage thread searches the queue for other packets that request the same operation. If it finds work sharing opportunities, it can "merge" the packets and execute the operation only once, sending the intermediate results to all interested parties. In the example shown in Figure 9, the two queries scan the same two database tables in stage 1, join the results in stage 2, and then each query computes a different aggregate function. The staged DBMS detects the sharing opportunity and performs the table scans and the join only once by merging the stage 2 packets, and copies the results to the next stage for each query. Thus, staged database systems can facilitate sharing of arbitrary sub-queries, and can adaptively share work at run time based on the aggregate workload and the available on-chip parallelism.

The modular design of staged database systems allows them to exhibit high levels of inherent fine-grain parallelism, enhances locality through scheduling, and exposes work sharing

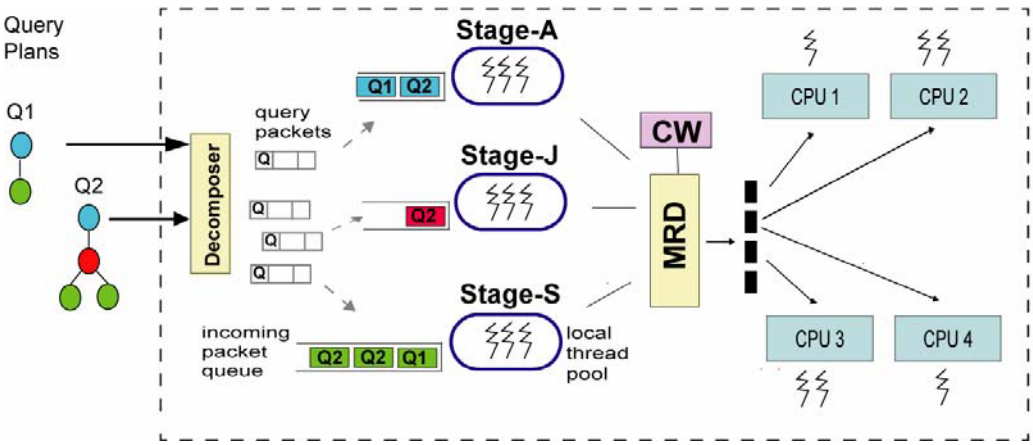

Figure 10. Design of Cordoba. The Micro-Request Dispatcher (MRD) schedules worker threads to cores. The core wizard (CW) stores the necessary run-time information for the dispatching decision.

opportunities that can be exploited adaptively at run time. These characteristics make staged database systems an ideal candidate to alleviate the constantly shifting performance bottlenecks without the overhead of completely redesigning the system. We believe that staged database systems hold great potential and we are eager to explore the design space. Toward this direction, we implemented a prototype staged execution engine, called Cordoba, which we describe in the next section.

7.2 Cordoba: an Adaptive Staged Database System for Chip Multiprocessors

In this section we outline the design and implementation of Cordoba, a prototype staged database engine for CMPs. Cordoba's design is geared towards addressing the parallelism and locality requirements identified earlier in this paper.

Figure 10 illustrates the design of Cordoba. Cordoba is a database engine that follows the principles of staged database systems. In its operator-centric architecture each conventional database operator becomes an independent component (*stage*) with its own pool of worker threads that serve requests from a stage-local queue. Figure 10 depicts three stages, one for each of the aggregate, join, and scan operators. Every incoming query is decomposed into a number of smaller requests (*packets*) that can execute in parallel. The packets are routed to the queue of their corresponding operator, and communicate with the other packets of the same query through intermediate buffers that reside in shared memory. Intermediate results are packed into pages (typically 4KB in size) instead of communicating them one tuple at a time as in the traditional demand-driven pipelined execution [17]. Communicating in page granularity has been shown to improve both instruction and data locality, as well as reducing the synchronization cost between the producer-consumer pairs [7, 25, 38].

Central to Cordoba's design are two components: the core wizard (CW) and the micro-request dispatcher (MRD). The core wizard maintains runtime information about the system. It uses a stage matrix to track the binding of packets to cores, and keeps statistics on system utilization (e.g., number of client connections, core load). The CW utilizes the available hardware performance counters to update its estimates of the running workload and hardware utilization. The MRD consults the stage matrix to decide the appropriate scheduling policy given the running workload and hardware configuration, and routes the incoming requests accordingly. The MRD has the flexibility to favor (a) locality, by routing all requests that operate on the same database resources to the same core and enabling thread throttling, or (b) pipelined parallelism, by routing all packets of a specific query or transaction to different cores and pipelining intermediate results.

The pipelining of intermediate results between operators enhances cache utilization and locality as data are quickly reused and can be safely discarded afterwards. Intelligent scheduling of worker threads can enhance data locality even further. For example, as mentioned in Section 6.2, binding producer/consumer threads together on the same core can improve L1-D locality. The producer can yield to the consumer whenever it produces enough data to fill the L1-D cache, allowing the consumer to use the data before it is pushed further down the memory hierarchy. The abundance of hardware contexts on lean camp cores also provides interesting opportunities for scheduling, especially in cases where multiple threads access the same data over a short time window. Thus, Cordoba's design has the potential to enhance both parallelism and locality, addressing the requirements imposed by the advent of chip multiprocessors.

8. RELATED WORK

Barroso et al. [6] conduct a performance study of OLTP and DSS workloads on a distributed shared memory multiprocessor and conclude that instruction and data locality on OLTP can be captured effectively by large caches. Our study shows that data stalls dominate execution on chip multiprocessors even with very large caches, because the dominant stall component then shifts to L2 hits. Barroso et al. [5] compare the performance of the Piranha chip multiprocessor against a single out-of-order processor with similar resources. However, they do not consider FC cores for the Piranha chip, and lack the key feature of fine-grain multi-threading to mask memory latency inside the processor core, which is critical to achieve high aggregate CMP performance.

Ranganathan et al. [26] examine the performance of database workloads on shared-memory multiprocessors and identify simple optimizations that improve performance when employed by aggressive out-of-order processors. However this study does not consider lean cores, which outperform their aggressive out-of-order counterparts on saturated workloads

despite their low single-thread performance. Lo et al. [22] study the performance of database workloads on simultaneous multithreaded (SMT) processors and show that aggressive wide-issue out-of-order SMT processors can outperform their single-threaded counterparts. However, wide-issue out-of-order processors with many hardware contexts are complex designs, and their area and power overhead render them unsuitable for CMPs that target database workloads. In our study we show that even simple in-order multithreaded processors can outperform aggressive out-of-order ones when the workload exhibits significant thread-level parallelism.

Several recent studies propose to re-design a software application for staged execution to improve performance. Chakraborty et al. [9] propose hardware migration to distribute a thread's dissimilar fragments of computation across the cores of a CMP. Their design automatically separates the application-level (user) computation from the OS calls it makes, and routes the system calls, page faults and interrupts to separate threads that execute on pre-specified cores. However, their proposal focuses on improving instruction locality and relies on clearly separable code paths (user-level vs. OS) with reasonably independent data footprints and *minimal* data communication between them, which are difficult to find in the user-level code of today's monolithic DBMSs.

Cohort Scheduling, proposed by Larus and Parkes [21], proposes a mechanism to assemble similar tasks into stages and schedule their execution together to reduce memory stalls. As we argue in this paper, staging can also reduce L2-hit stalls in addition to memory stalls, and can also provide higher levels of fine-grain parallelism. Welsh et al. [33] propose a staged event-driven architecture for implementing high-performance internet services. The primary focus in their work is to prevent resource over-commitment when the demand of the server exceeds the service capacity, and not the memory hierarchy performance, which is the dominant bottleneck in data-intensive applications like database servers. Our philosophy for a staged database system is derived from [15], which outlines the main characteristics of staged database servers.

9. CONCLUSIONS

High levels of integration have enabled the advent of chip multiprocessors and increasingly large (and slow) on-chip caches. These two trends pose new performance challenges to the database community, which is not ready for dramatic shifts in hardware design. This study presented a performance characterization of a commercial database server in a number of representative chip multiprocessor technologies. The simulation results indicate that data cache misses are the performance bottleneck in memory-resident databases, with L2 hit stalls rising from oblivion to become the dominant single execution time component in some cases. We discussed several techniques to reduce the data stall component and derived a list of features for future database designs. Finally, we presented Cordoba, a prototype staged database engine that targets chip multiprocessors.

10. ACKNOWLEDGEMENTS

We cordially thank Tom Wenisch for his feedback and technical support throughout this research effort, and the anonymous reviewers for their valuable feedback in early drafts of this paper. This work was partially supported by grants and equipment from Intel, two Sloan research fellowships, an IBM faculty partnership award, and NSF grants CCR-0205544, CCR-0509356, and IIS-0133686.

REFERENCES

[1] Intel Corporation. "Intel Core Duo Processor and Intel Core Solo Processor on 65nm Process Datasheet." 2006.

[2] IBM Corporation. "Pmcount for Linux on Power Architecture." 2006.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. "Weaving Relations for Cache Performance." In Proc. 27th International Conference on Very Large Data Bases, 2001.

[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 International Conference on Very Large Data Bases (VLDB), 1999.

[5] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing." In Proc. 27th Annual International Symposium on Computer Architecture, 2000.

[6] L. A. Barroso, K. Gharachorloo, and E. Bugnion. "Memory System Characterization of Commercial Workloads." In Proc. 25th Annual International Symposium on Computer Architecture (ISCA), 1998.

[7] P. A. Boncz, S. Manegold, and M. L. Kersten. "Database Architecture Optimized for the New Bottleneck: Memory Access." In Proc. 25th International Conference on Very Large Data Bases (VLDB), 1999.

[8] S. Borkar. "Microarchitecture and Design Challenges for Gigascale Integration". Keynote. In Proc. 37th International Symposium on Microarchitecture (MICRO), 2004.

[9] K. Chakraborty, P. M. Wells and G. S. Sohi. "Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly." In Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2006.

[10] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. "Improving Hash Join Performance Through Prefetching." In Proc. 20th International Conference on Data Engineering (ICDE), 2004.

[11] T. M. Chilimbi, M. Hill, and J. R. Larus. "Cache-Conscious Structure Layout." In Proc. SIGPLAN Conference on Programming Language Design and Implementation, 1999.

[12] T. M. Chilimbi and M. Hirzel. "Dynamic hot data stream prefetching for general-purpose programs." In Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2002.

[13] J. D. Davis, J. Laudon and K. Olukotun. "Maximizing CMP Throughput with Mediocre Cores." In Proc. 14th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2005.

[14] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. "SIMFLEX: A Fast, Accurate, Flexible Full-

System Simulation Framework for Performance Evaluation of Server Architectures." In ACM SIGMETRICS Performance Evaluation Review, 2004.

[15] S. Harizopoulos, and A. Ailamaki. "A Case for Staged Database Systems." In Proc. 1st International Conference on Innovative Data Systems Research (CIDR), 2003.

[16] S. Harizopoulos, and A. Ailamaki. "STEPS towards cache-resident transaction processing." In Proc. 30th International Conference on Very Large Data Bases (VLDB), 2004.

[17] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. "QPipe: A Simultaneously Pipelined Relational Query Engine." In Proc. 24th ACM SIGMOD International Conference on Management of Data, 2005.

[18] N. P. Jouppi. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers." In Proc. 17th Annual International Symposium on Computer Architecture (ISCA), 1998.

[19] R. Kalla, B. Sinharoy, and J. Tendler. "IBM Power5 chip: A dual-core multithreaded processor." In IEEE MICRO, 2004.

[20] P. Kongetira, K. Aingaran, and K. Olukotun. "Niagara: A 32-way Multithreaded SPARC Processor." In IEEE MICRO, 2005.

[21] J. R. Larus, and M. Parkes. "Using Cohort Scheduling to Enhance Server Performance." In Proc. General Track: 2002 USENIX Annual Technical Conference, 2002.

[22] J. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. "An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors." In Proc. 25th Annual International Symposium on Computer Architecture (ISCA), 1998.

[23] A. M. G. Maynard, C. M. Donelly, and B. R. Olszewski. "Contrasting Characteristics and Cache Performance of Technical and Multi-user Commercial Workloads." In Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.

[24] K. J. Nesbit and J. E. Smith. "Data cache prefetching using a global history buffer." In Proc. 10th IEEE Symposium on High-Performance Computer Architecture, 2004.

[25] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. "Block Oriented Processing of Relational Database Operations in Modern Computer Architectures." In Proc. 17th International Conference on Data Engineering (ICDE), 2001.

[26] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. "Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors." In Proc. 8th International Conference of Architectural Support for Programming Languages and Operating Systems, 1998.

[27] J. Rao, and K. A. Ross. "Making B+-Trees Cache Conscious in Main Memory." In Proc. 19th ACM SIGMOD International Conference on Management of Data, 2000.

[28] S. Rusu, S. Tam, H. Muljono, D. Ayers, and J. Chang. "A Dual-Core Multi-Threaded Xeon Processor with 16MB L3 Cache." In Proc. IEEE International Solid-State Circuits Conference, 2006.

[29] M. Shao, A. Ailamaki and B. Falsafi. "DBmbench: Fast and Accurate Database Workload Representation on Modern Microarchitecture." In Proc. IBM Center for Advanced Studies Conference, 2005.

[30] A. Shatdal, C. Kant, and J. F. Naughton. "Cache Conscious Algorithms for Relational Query Processing." In Proc. 20th International Conference on Very Large Data Bases, 1994.

[31] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. "Memory Coherence Activity Prediction in Commercial Workloads." In Proc. 3rd Workshop on Memory Performance Issues (WMPI), 2004.

[32] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. "Spatial Memory Streaming." In Proc. 33rd Annual International Symposium on Computer Architecture (ISCA), 2006.

[33] M. Welsh, D. Culler, and E. Brewer. "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services." In Proc. 18th ACM Symposium on Operating Systems Principles (SOSP), 2001.

[34] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. "Temporal Streaming of Shared Memory." In Proc. 32nd Annual International Symposium on Computer Architecture (ISCA), 2005.

[35] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. "SimFlex: Statistical Sampling of Computer System Simulation." IEEE MICRO Special Issue on Computer Architecture Simulation and Modeling, 2006.

[36] S. J. E. Wilton and N. P. Jouppi. "CACTI: An Enhanced Cache Access and Cycle Time Model." In IEEE Journal of Solid-State Circuits, 1996.

[37] J. Wuu, D. Weiss, C. Morganti, and M. Dreesen. "The Asynchronous 24MB On-Chip Level-3 Cache for a Dual-Core Itanium®-Family Processor." In Proc. IEEE International Solid-State Circuits Conference, 2005.

[38] J. Zhou, and K. A. Ross. "Buffering Database Operations for Enhanced Instruction Cache Performance." In Proc. 23rd ACM SIGMOD International Conference on Management of Data, 2004.

[39] M. Zukowski, P. Boncz, N. Nes, and S. Heman. "MonetDB/X100 - A DBMS in the CPU Cache." In IEEE Data Engineering Bulletin, 2005.