DBmbench: Fast and Accurate Database Workload Representation on Modern Microarchitecture

Minglong Shao* Anastassia Ailamaki* Babak Falsafi†
*Database Group
Carnegie Mellon University
{shaoml,natassa}@cs.cmu.edu
†Computer Architecture Laboratory
Carnegie Mellon University
babak@cmu.edu

Abstract

With the proliferation of database workloads on servers, much recent research on server architecture has focused on database system benchmarks. The TPC benchmarks for the two most common server workloads, OLTP and DSS, have been used extensively in the database community to evaluate the database system functionality and performance. Unfortunately, these benchmarks fall short of being effective in microarchitecture and memory system research due to several key shortcomings. First, setting up the experimental environment and tuning these benchmarks to match the workload behavior of interest involves extremely complex procedures. Second, the benchmarks themselves are complex and preclude accurate correlation of microarchitecture- and memory-level bottlenecks to dominant workload characteristics. Finally, industrial-grade configurations of such benchmarks are too large and preclude their use in detailed but slow microarchitectural simulation studies of future servers. In this paper, we first present an analysis of the dominant behavior in DSS and OLTP workloads, and highlight their key processor and memory performance characteristics. We then introduce a systematic scaling framework to scale down the TPC benchmarks. Finally, we propose the DBmbench, consisting of two substantially scaled-down benchmarks: μTPC-H and μTPC-C that accurately ($> 95\%$) capture the processor and memory performance behavior of DSS and OLTP workloads.

1 Introduction

Database workloads — such as Decision Support Systems (*DSS*) and Online Transaction Processing (*OLTP*) — are emerging as an important class of applications in the server computing market. Nevertheless, recent research [1, 3, 12] indicates that these workloads perform poorly on modern high-performance microprocessors. These studies show that database workloads have drastically different processor and memory performance characteristics as compared to conventional desktop and engineering workloads [18] that have been the primary focus of microarchitecture research in recent years. As a result, researchers from both the computer architecture and database communities are increasingly interested in careful performance evaluation of database workloads on modern hardware platforms [1, 2, 3, 4, 5, 7, 12, 13, 15, 19, 20].

To design microprocessors on which database workloads perform well, computer architects need benchmarks that accurately represent these workloads. There are a number of requirements that suitable benchmarks should satisfy. First, modern wide-issue out-of-order superscalar processors include a spectrum of mechanisms to extract parallelism and enhance instruction execution throughput. As such, the benchmarks must faithfully mimic the performance of the workloads at the microarchitecture-level to allow for designers to pinpoint the exact hardware bottlenecks. Second, microarchitecture simulation tools [17] are also typically five or more orders of magnitude slower than real hardware [16, 22]. To allow for practical experimentation turnaround, architects need benchmarks that are scaled down variations of the

workloads [9] and have minimal execution time. Third, the benchmark behavior should be deterministic when across scaled datasets and varying system configurations to allow for conclusive experimentation. Finally, the benchmark sources or executables should either be readily available [18] or at most require installation and setup skills characteristic of a typical computer system researcher and designer.

Unfortunately, conventional DSS and OLTP database benchmarks, *TPC-H* and *TPC-C* [8], fall far short of satisfying these requirements. The TPC benchmarks have been primarily designed to test functionality and evaluate overall performance of database systems on real hardware. These benchmarks have orders of magnitude larger execution times than needed for use in simulation. To allow for practical experimentation turnaround, most prior studies [1, 3, 4, 5, 13, 15, 20] employ ad hoc abbreviations of the benchmarks (scaled down datasets and/or a subset of the original queries) without justification. Many of these studies tacitly assume that microarchitecture-level performance behavior is preserved.

Moreover, the TPC benchmarks' behavior at the microarchitecture-level may be non-deterministic when scaled. The benchmarks include complex sequences of database operations that may be reordered by the database system depending the nature of the sequence, the database system configuration and the dataset size, thereby substantially varying the benchmark behavior. Recent research by Hankins et al. [9], rigorously analyzes microarchitecture-level performance metrics of scaled datasets for *TPC-C* workloads and concludes that performance metrics cease to match when the dataset is scaled below 12GB. Unfortunately, such dataset sizes are still too large to allow for practical simulation turnaround.

Finally, the TPC benchmark kits for most state-of-the-art database systems are not readily available. Modern database systems typically include over one hundred configuration and installation parameters. Writing and tuning the benchmarks according to the specifications [21] on a given database system to represent a workload of interest may require over six months of experimentation even by a trained database system manager [11] and requires skills beyond those at hand for a computer system designer.

In this paper, we present *DBmbench*, a benchmark suite representing DSS and OLTP workloads tailored to fit the requirements for microarchitecture research. The DBmbench is based on the key observation that the executions of database workloads are primarily dominated by a few intrinsic database system operations — e.g., a sequential scan or a join algorithm. By identifying these operations, microarchitecture-level behavior of the workloads can be mimicked by benchmarks that simply trigger the execution of these operations in the database system. We present the DBmbench benchmarks in the form of simple database queries, readily executable on database systems, and substantially reducing execution complexity as compared to the TPC benchmarks. Moreover, by isolating operation execution in stand-alone benchmarks, the datasets can be scaled down to only hundreds of megabytes while resulting in deterministic behavior precluding any optimizations in operation ordering by the database system.

Using hardware counters on an Intel Pentium III platform running IBM DB2, we show that the DBmbench benchmarks can match a key set of microarchitecture-level performance behavior, such as cycles-per-instruction (CPI), branch prediction accuracy, and miss rates in the cache hierarchy, of professionally tuned TPC benchmarks for DB2 to within 95% (for virtually all metrics). As compared to the TPC benchmarks, the DBmbench DSS and OLTP benchmarks: (1) reduce the number of queries from 22 and 5 to 2 and 1 simple queries respectively, (2) allow for scaling dataset sizes down to 100MB, and (3) reduce the overall number of instructions executed by orders of magnitude.

The remainder of this paper is organized as follows: section 2 introduces the basic database concepts used in this paper. Section 3 describes a framework to scale down database benchmarks and the design of DBmbench. Section 4 discusses the experimental setup and the metrics used to characterize behavior at the micro-architecture level. Sections 5 evaluates the scaling framework and the DBmbench. Section 6 presents a brief survey of recent database workload characterization studies and the research on microbenchmarks. Section 7 concludes the paper and outlines future work.

2 Background

Commercial database management systems (*DBMS*) organize data in *tables* according to the *relational model* [6]. Each table is defined by a set of *fields* and contains a set of *records*, whereas each record consists of values to the fields. To read, filter, or modify stored information, users submit *queries* to the DBMS using a query language such as *SQL*. Figure 1(a) shows an example set of relational tables, whereas Figure 1(b) illustrates an example query to extract the names and GPAs for all the students in the Computer Science department. The query comprises four clauses: SELECT chooses the fields to participate in the answer (name from Student and the average over score values from Course); FROM lists the tables to process (Student and Course); WHERE states the condition based on which to choose the records to participate in the answer (the student must have taken the course and the student should belong in the CS department); GROUP BY denotes that the average scores (GPAs) should be calculated per student name. The query essentially summarizes the *relational operators* to be used; the DBMS executes the query using a set of *physical operators*. Unless explicitly noted, "operator" means "physical operator."

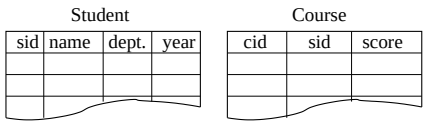
2.1 Basic Physical Operators

Operators are independent code pieces that consume one or two input streams of records and produce one output stream. To execute a query, the DBMS constructs a *query plan*, i.e., a cooperating tree of operators. A logical operator, may be executed using several physical operators; therefore, each query may be executed using one of many possible query plans. No matter how complex a SQL query is, however, it is executed using a finite set of basic operators. Most frequently used operators are those implementing a *scan*, a *join*, an *order-by*, a *group-by*, and an *aggregate*. For brevity, this section only discusses read-only operators; the implementation of update, deletion, and insertion operators is immaterial to this paper.

A *table scan* operator (corresponding to the "Scan" in Figure 2) reads through an entire table and generates a stream of records that satisfy a *predicate* (for example, "scan Student" in Figures 2(a), 2(b) only outputs Student records in CS). The *selectivity* of the predicate is calculated as the number of records that satisfy the predicate divided by the number of records in the input (in our example, in Student). An *index scan* operator ("IScan" in Figures 2(c)) provides the same set of results as a table scan by using an *index* (a B+ tree that uses field values as keys to point to records) to access the table. An index scan is different than a table scan in that it only accesses qualifying records through the index. The index Idept used in Figure 2(c), for instance, is built using Student.dept as key and can be used to quickly identify records of CS students. Assuming that Student is sorted on sid, the Idept is a *non-clustered* index, because the records are not sorted by the index key. This means that retrieving CS student records through Idept results in random record accesses in the Student table. Conversely, an index on sid is *clustered* as the records in the table are stored in the same order as the index. Retrieving a set of records within a range of sid values results in one index probe to locate the beginning value, and then a sequential scan through students to obtain all the remaining qualifying records.

Table joins match tuples from two tables based on an equality (*equijoin*) or other condition on common fields. Joins are typically implemented using variations of three algorithms: *nested-loop*, *sort-merge*, and *hash* join. The nested-loop join uses a two-level nested loop to compare each record of one table with all the records of the other, and can efficiently compute inequality joins. Sort-merge first sorts the input tables, and then merges the sorted runs using nested loops for duplicates; it is most efficient when one (or both) inputs are already sorted on the join field. The hash join creates a hash table on one of the inputs and then probes it with records in the other; it is often the most efficient for computing equijoins on unsorted inputs.

Order-by clauses are implemented using the *sort* operator, that sorts records in the input table based on a subset of fields. The sort operator is also used as part of a sort-merge join or to implement a group-by. The *group-by* operator classifies the input records into groups based on a subset of fields, and outputs the groups; it can be implemented using sorting or hashing. The *aggregate* operator applies a function (such as *max*, *sum*, etc.) on the input records and outputs a single value. Database textbooks describe operators in detail [14].

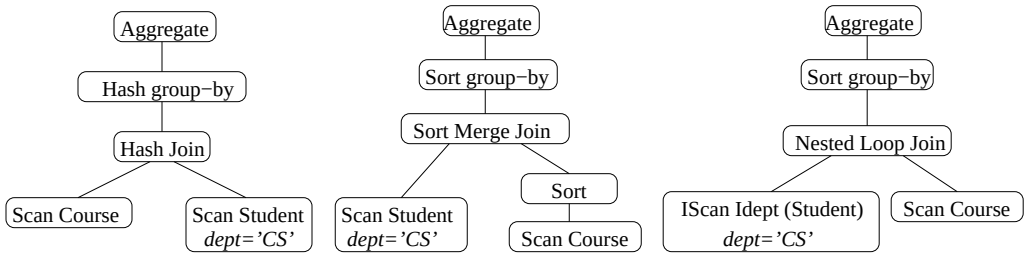


(a) **Tables.**

```
SELECT Student.name,
       avg(Course.score) AS GPA
FROM Student, Course
WHERE Student.dept='CS'                    (i)
      and Student.sid=Course.sid           (ii)
GROUP BY Student.name
```

(b) **An SQL query.**

Figure 1: **An example database workload consisting of two tables and an SQL query.**



(a) **Query plan 1.** (b) **Query plan 2.** (c) **Query plan 3.**

Figure 2: **Query plan examples for the query in Figure 1(b).**

2.2 Relational Query Optimization

As discussed above, there are multiple different physical operators a query execution engine can use to calculate the answer to a query. First, access to a table may be sequential or through an index; second, joins and group-by operations have various implementations; third, there are several valid permutations of inputs and operators in a query plan. Although all valid query plans calculate the same answer to the query, their relative performance may vary wildly depending on the data distributions, as well as the conditions used in the query. To evaluate, for instance, the subcondition (i) on Student we can either use a table scan or the non-clustered index Idept. If there are one million students in total, and half of them are in CS, we should just use a table scan to filter out the non-CS records. If, however, there are only a hundred CS students, it will be preferable to use the index, despite that it is non-clustered (and these will be one hundred random record accesses).

To make this decision, we evaluate the cost of each alternative based on the *selectivity* of the predicate in the subcondition. There may be other determining factors: for example, the input table size relatively to the available memory size (*buffer pool*) may determine the access method to be used, or the relative input sizes of the joined files may determine the algorithm and the order in which they are joined. The exact relative costs of each access method and each operation are calculated through a statistics-based cost model in the heart of the DBMS *query optimizer*.

Using the cost model, the optimizer evaluates all the possible plans, and orders the query execution engine to compute the answer based on the least expensive plan. Figure 2 shows three possible *query execution plans* for our example, whereas the total number of possible plans is exponential to the number of tables involved in the query. Query optimization in commercial DBMS is thus performed through dynamic programming, and different query plans execute different code. Therefore, the DBMS may use a different instruction mix to execute the same workload if any of the workload of system configuration parameters varies.

3 Scaling Down Benchmarks

This section outlines a framework to scale down benchmarks. We identify three dimensions along which we can abbreviate benchmarks and discuss the issues involved when scaling database benchmarks workload along the dimensions. Then, we present the design of DBmbench.

Decision-support system (DSS) workloads are typically characterized by long, complex queries (often 1MB of SQL code) running on large datasets at low concurrency levels. DSS queries are characterized from sequential access patterns (through

table scans or clustered index scans). By contrast, on-line transaction processing (OLTP) workloads consist of short read-write query statements grouped in atomic units called *transactions* [8]. OLTP workloads have high concurrency levels, and the users run many transactions at the same time. The queries in the transactions typically use non-clustered indexes and access few records, therefore OLTP workloads are characterized by concurrent random accesses.

The prevalent DSS benchmark is TPC-H [8]. TPC-H consists of eight tables, twenty-two read-only queries ($Q1$–$Q22$) and two batch update statements, which simulate the activities of a wholesale supplier. For OLTP, the TPC-C benchmark portrays a wholesale supplier and several geographically distributed sale districts and associated warehouses [21]. It is comprised of nine tables and five different types of transactions. TPC-H is usually executed in a single-query-at-a-time fashion while TPC-C models multiple clients running concurrently.

3.1 A Scaling Framework

A database benchmark is typically composed of a dataset and a workload (set of queries or transaction) to run on the dataset. Inspired by the differences between DSS and OLTP outlined in Section 3, we scale down a full benchmark along three orthogonal dimensions, shown in Figure 3: workload complexity, dataset size, and level of concurrency.

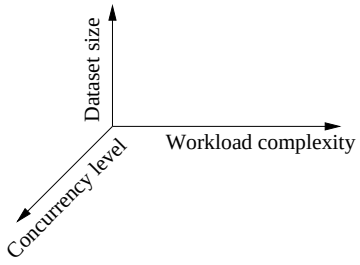


Figure 3: **Benchmark-scaling dimensions.**

In order to scale down a benchmark's workload complexity, one approach is to choose a subset of the original queries [3, 15, 20]. Another approach is to reduce the query complexity by removing parts of the query or reducing the number of items in the SELECT clause. Both methods effectively reduce query complexity at the cost of sacrificing representativeness; choosing a subset of queries may exclude important queries that significantly affect behavior, while the complexity reduction method may inadvertently result in dramatic changes to the query plans and thus modify the benchmark's behavior.

Scaling down along the dataset size dimension is fairly straightforward, because benchmark specifications typically provide rules or software to scale down datasets. The main concern when scaling down along this dimension is to preserve the performance characteristics of the workload, as reducing the database size is likely to alter the query plans (and consequently the instruction mix) and cause performance bottlenecks to shift. Similarly, scaling the level of concurrency is straightforward, because benchmarks include in their specifications the how many users should run per data unit. It is important to abide by the scaling rules in the specifications, to maintain the data and usage properties.

3.2 Framework Application to DSS and OLTP Benchmarks

From the perspective of benchmark evaluation, DSS queries are mostly read-only and usually access a large portion of the dataset. While there are also batch updates, read-only operations are the critical part in a DSS workload. Queries are executed one-at-a-time, and the execution process for each query is predictable and reproducible. Furthermore, while DSS queries vary enormously in functionality, they typically spend most of their time executing basic query operations such as sequential scan and/or join.

When examining the optimizer's suggested plans for TPC-H queries, we find that 50% queries are dominated by table scans (over 95% of their execution time is estimated to be due to table scans) whereas 25% of the queries spend more than 95% of the time executing nested-loop joins. The remaining 25% of the queries executed table scans for about 75% on average and nested-loop joins for about 25% on average. Therefore, we can represent scan-bound and join-bound queries by executing the two dominant operators.

Considering the complexity and depth of a TPC-H query plan, this result may seem counter-intuitive; however, the major part of the filtering is done at the lowest levels of the operator tree, and

the result size is reduced dramatically as execution continues to the upper levels of the tree. In conclusion, DSS workloads can be scaled down by (1) constructing representative queries that execute the dominant operators; (2) using small datasets that fit in the research testbed. The concurrency level is already low in DSS.

OLTP workloads are characterized by a large number of concurrent and continuous update-intensive transactions that generate random-like memory access patterns. Queries in OLTP workloads are simple and only touch a small fraction of the dataset. OLTP execution is quite different from that of DSS, in that it involves a stream of concurrent transactions including numerous simple queries and insert/update statements. Scaling down OLTP benchmarks involves decreasing the number of concurrent clients and reducing the dataset sizes. To accurately mimic the workload's scattered dataset access pattern, the concurrent clients should execute one or more queries with random access to memory.

3.3 DBmbench Design

DBmbench is a microbenchmark suite that can emulate DSS and OLTP workloads at the computer architectural level. DBmbench includes two tables and three simple queries. The design principles are (1) keeping table schemas and queries as simple as possible; (2) focusing on the dominant operations in DSS and OLTP.

DBmbench tables. DBmbench uses two tables, T1 and T2, as shown in Table 1. T1 and T2 have three fields each, $a1$, $a2$, and $a3$, which will be used by the DBmbench queries. "`padding`" stands for a group of fields that are not used by any of the queries. We use the values of these fields as "`padding`" to make records 100 Byte long, which approximates the average record length of TPC-H and TPC-C. The type of these fields makes no difference in the performance, and by varying its size we can experiment with different record sizes without affecting the benchmark's queries.

The values of field $a1$ are uniformly distributed between 1 and $150,000$, whereas $a2$ takes values randomly within the range of 1 to $20,000$ and $a3$ values are uniformly distributed between 1 and 50. The distributions and values in these tables are a properly scaled-down subset of the data distributions and values in the TPC-H tables.

DBmbench queries. Based on the discussion in Section 3.2, the design of the DSS microbenchmark mainly focuses on simplifying query complexity. Moreover, as discussed previously, scan and join operators typically dominate DSS query execution time. Therefore, we propose two queries for the DSS microbenchmark, referred to as μTPC-H, as follows: sequential scan query with sort (μSS) and join query (μNJ). The first two columns of Table 2 show the SQL statements for these two queries.

The μSS query is a sequential scan over table T1. We will use it to simulate the DSS queries whose dominant operators are sequential scans. The two parameters in the predicate, L_o and H_i, are used to obtain different selectivities. The order-by clause sorts the query results by the values inn the a3 field, and is added for two reasons. First, sort is an important operator in DSS queries, and the order-by clause increases the query complexity effectively to overcome common shortcomings in existing microbenchmarks [1, 11]. Second, the clause will not alter the sequential scan access method, which is instrumental in determining the basic performance characteristics.

Previous microbenchmarks use aggregation functions in the projection list to minimize the server/client communication overhead [1, 11]. To prevent the optimizer from omitting the sort operator, μSS uses "distinct" instead of the aggregate. "Distinct" eliminates duplicates from the answer and achieves the same methodological advantage as the aggregate, because the number of distinct values in $a3$ is small (less than or equal to 50), and does not interfere with the performance characteristics. Our experiment results corroborate these hypotheses.

Although previously proposed microbenchmark suites [11] often omit the join operator, it is actually an important component in DSS queries and has very different behavior from table scan [1]. To mimic the DSS workload behavior accurately, we consider the join operator and propose the μNJ query to simulate the DSS queries dominated by the join operator. The predicate "$L_o < T1.a2 < H_i$" adds an adjustable selectivity to the join query so that we can control the number of qualifying records by changing the values of L_o and H_i.

The OLTP microbenchmark, which we call μTPC-C, consists of one non-clustered index scan query (μIDX), shown in the third column of Ta-

Table T1	Table T2
CREATE TABLE T1 (a1 INTEGER NOT NULL, a2 INTEGER NOT NULL, a3 INTEGER NOT NULL, <padding>, FOREIGN KEY (a1) references T2);	CREATE TABLE T2 (a1 INTEGER NOT NULL PRIMARY KEY, a2 INTEGER NOT NULL, a3 INTEGER NOT NULL, <padding>);

Table 1: **DBmbench database: table definitions**

μSS query	μNJ query	μIDX query
SELECT distinct (a3) FROM T1 WHERE Lo < a2 < Hi ORDER BY a3	SELECT avg (T1.a3) FROM T1, T2 WHERE T1.a1=T2.a1 AND Lo < T1.a2 < Hi	SELECT avg (a3) FROM T1 WHERE Lo < a2 < Hi

Table 2: **DBmbench workload: queries**

ble 2. The μIDX query is similar to the μSS query in μTPC-H. The key difference is that, when evaluating the predicate in the "where" clause, the table scan through the non-clustered index generates a TPC-C-like random access pattern. The proposed μIDX query is a read-only query which only partly reflects the type of actions in TPC-C. The transactions also include a significant number of write statements (updates, insertions, and deletions). In our experiments, however, we found that adding updates to the DBmbench had no effect in the representativeness of the benchmark. The reason is that, like queries, updates use the same indexes to locate data, and the random accesses on the tables through index search is the dominant behavior in TPC-C. Therefore, the μIDX query is enough to represent the benchmark. We scale down the dataset to the equivalent of one warehouse (100MB) and the number of concurrent users to ten (as directed by the TPC-C specification).

4 Experimental Methodology

In this section, we present the experimental environment and methodology we use in the paper. Industrial-strength large-scale database servers are often configured with fully optimized high-performance storage devices so that the execution process is typically CPU- rather than I/O-bound. A query's processor execution and memory access characteristics in such settings dominate overall performance [3]. As such, we ignore I/O activity in this paper and focus on microarchitecture-level performance.

We conducted our experiments on a 4-way 733 MHz Intel Pentium III server. Pentium III is a 3-way out-of-order superscalar processor with 16 KB level-one instruction and data caches, and a unified 2 MB level-two cache. The server has 4 GB of main memory and four SCSI disks of 35 GB capacity. To measure microarchitecture-level performance, we use the hardware counters featured in the processors to count events or measure operation latencies. [1] We use Intel's EMON tool to operate the counters and perform measurements. The counted events include the total number of retired instructions, the number of cache misses at each level, mispredicted branch instructions, and CPU cycles, etc.

We use IBM DB2 UDB V.7.2 with Fix Package 11 [10] on Linux (kernel version 2.4.18) as the underlying database management system, and run TPC-H and TPC-C benchmarks. As in prior work [3, 4, 5, 11, 20], we focus on the read-only queries which are the major components of the TPC-H workload, but our results can easily be extended to include the batch updates. For our experiments, we used a slightly modified version of the TPC-C kit provided by IBM which has been opti-

[1] We have also verified that the microarchitecture-level event counts between the TPC benchmarks and DBmbench match on a Pentium 4 platform. However, we are not aware of an execution time breakdown model for the platform to match the stall time components, and therefore we omit these results in the interest of brevity.

mized for DB2. Prior work [1] suggests that commercial DBMS exhibit similar microarchitecture-level performance behavior when running database benchmarks. Therefore, expect the results in this paper to be applicable to other database servers.

For TPC-H, we record statistics for the entire execution of all the queries. We measure work units in order to minimize the effect of startup overhead. Each work unit consists of multiple queries of the same type but with different values of the substitute parameters (i.e., selectivity remains the same, but qualifying records vary). We run each work unit multiple times, and measure events per run. The measurement is repeated several times to eliminate the random factors during the measurement. The reported results have less than 5% discrepancy across different runs.

For TPC-C, we count a pair of events during a five-second fixed time interval. We measure events multiple times and in different order each time. For all experiments, we ensure that the standard deviation is always lower than 5% and compute an average over the per-event collected measurements.

When scaling dataset sizes, we also change the system configuration parameters to ensure the setup is valid. Database systems include a myriad of software-configured parameters. In the interest of brevity and to allow for practical experimental turnaround time, in this paper we focus on the buffer pool size as the key database system parameter to vary. As database applications are heavily memory-bound, the buffer pool size: (1) is expected to have the most fundamental effect on processor/memory performance, and (2) often determines the values of other memory-related database system parameters. For TPC-C, where the number of concurrent users is intuitively important for the system performance, we also vary the degree of concurrency. While we have studied other parameters (such as degree of parallelism), we did not find any insightful results based on them.

When measuring performance, we are primarily interested in the following characteristics: (1) query execution time breakdown, (2) memory stall time breakdown in terms of cycles lost at various cache levels and TLBs, (3) data and instruction cache misses per instruction at each level (4) branch misprediction per instruction.

To break down query execution time, we borrow the model proposed and validated in [1] for the the Pentium III family of processors. In this model, query execution time is divided into cycles devoted to useful computation and stall cycles due to various microarchitecture-level mechanisms. The stalls are further decomposed into different categories. Hence, the total execution time T_Q can be expressed by the following equation:

$$T_Q = T_C + T_M + T_B + T_R - T_{OVL}$$

T_C is the actual computation time; T_M is wasted cycles due to misses in the cache hierarchy; T_B refers to stalls due to the branch prediction unit including branch misprediction penalty and BTB miss penalty; T_R is the stalls due to structural hazards in the pipeline due to lack of functional units or physical rename registers; T_{OVL} indicates the cycles saved by the overlap of the stall time because of the out-of-order execution engine.

T_M is further broken down into six components:

$$T_M = T_{L1D} + T_{L1I} + T_{L2D} + T_{L2I} + T_{DTLB} + T_{ITLB}$$

These are stalls caused by L1 cache misses (data and instruction), L2 cache misses (data and instruction), and TLB misses respectively.

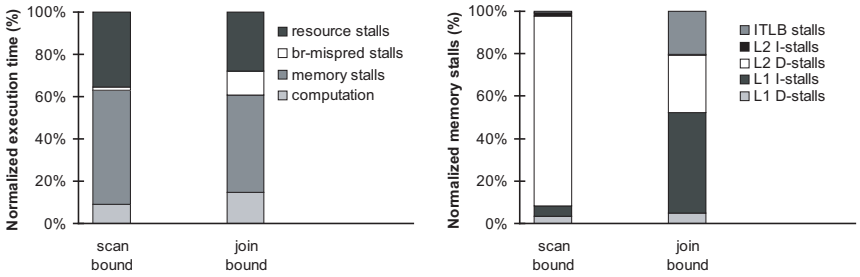
5 Evaluation

In this section, we compare and contrast the microarchitecture-level performance behavior of the TPC and DBmbench benchmarks. We first present results for the DSS benchmarks followed by results for the OLTP benchmarks.

5.1 Analyzing the DSS Benchmarks

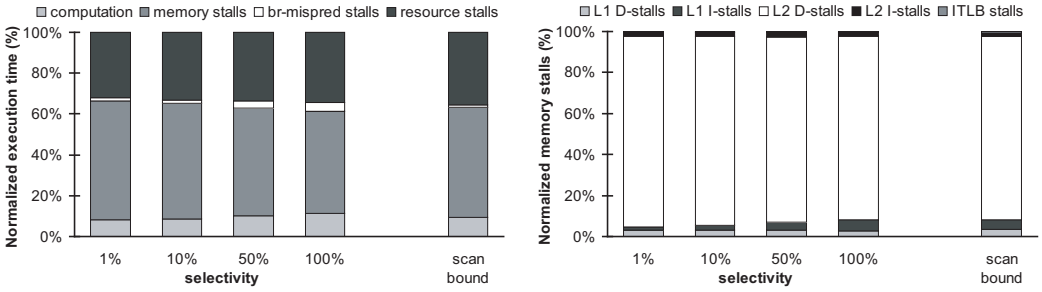
When taking a close look at the query plans provided by the optimizer, we corroborate our intuition from 3.2 that one of the two "scan" or "join" operators account for more than 95% of the total execution time in each of the TPC-H queries. We also find that these two operators remain dominant across database system configurations and dataset sizes. Therefore, we classify the TPC-H queries into two major groups: "scan bound" query and "join bound" query. We evaluate the microarchitecture-level performance these groups on a 10GB dataset.

Figure 4(a) shows the representative execution time breakdowns of the two groups. Each bar shows the contributions of the three primary microarchitectural stall components (memory stalls,



(a) **TPC-H execution time breakdown.** (b) **TPC-H memory stall breakdown.**

Figure 4: **TPC-H time breakdowns.** Representative time breakdowns for the "scan bound" and "join bound" groups, which spend their execution time mainly on sequential scan and join operators respectively.



(a) **Execution time breakdown comparison** (b) **Memory stall breakdown comparison**

Figure 5: **μSS vs. TPC-H "scan bound" query.** The graphs show the time breakdowns of μSS and TPC-H "scan bound" queries. For the μSS query, we vary its selectivity from 1% to 100% to show how selectivity affects the behavior.

branch stalls, and resource stalls) as a percentage of the total query execution time.

These results corroborate prior findings [1, 3] that on average the processor is idle more than 80% of the time when executing the TPC-H queries. In both groups, the performance bottlenecks are memory-related and resource-related stalls, each accounting for approximately 25% to 50% of the execution time. While we can not measure the exact cause of the resource-related stalls, our conjecture is that they are related to the load/store unit due to the high overall fraction of memory accesses in these queries.

Not surprisingly, the queries in the "join bound" group have a higher computation time component because joins are more computationally intensive than sequential scans. Furthermore, control-flow in joins are data-dependent and irregular, and as such the "join bound" group exhibits a higher branch misprediction stall (over 15%) component as compared to the "scan bound" group whose execution is dominated by loops exhibiting negligible branch misprediction stall time.

Figure 4(b) depicts a breakdown of memory stall time. The figure indicates that the "scan bound" group's memory stalls are dominated (over 90%) by L2 data misses. These queries simply thrash the L2 cache by marching over the entire dataset and as such have no other relatively significant memory stall component.

Unlike the "scan bound" queries, the "join bound" queries suffer from frequent L1 i-cache and i-TLB misses. These queries exhibit large and dynamic i-cache footprints that can not fit in a 2-way associative 16KB cache. The dynamic footprint nature of these queries is also consistent with their irregular control flow nature and their high branch misprediction stalls. Moreover, frequent branch misprediction also inadvertently pollutes the i-cache with the wrong-path instructions, thereby increasing the miss rate.

5.2 Comparison to μTPC-H

In this section, we compare the microarchitecture-level performance behavior of the "scan bound"

and "join bound" TPC-H queries against their μTPC-H counterparts. As before, TPC-H results assume a 10GB dataset while the μTPC-H results we present correspond to a significantly scaled down 100MB dataset.

Figure 5(a) compares the execution time breakdown of the μSS query and TPC-H queries in the "scan bound" group. The x-axis in the left graph reflects the selectivity of the predicate in the μSS query. These results indicate that the execution time breakdown of the TPC benchmark is closely mimicked by the DBmbench. Our measurements indicate that the absolute benchmark performances also match, averaging a CPI of approximately 4.1.

The μSS query with high selectivity sorts more records, thereby increasing the number of branches in the instruction stream. These branches do not exhibit any patterns and are difficult to predict, which unavoidably results in a higher branch misprediction rate. As shown in Figure 5(a), the μSS query successfully captures the representative characteristics of the TPC-H queries in the "scan bound" group: it exposes the same bottlenecks and has similar percentages of each component. Figure 5(b) compares the memory stall breakdowns of the μSS query and the "scan bound" queries. The μSS query exposes the same bottlenecks at the L2 (for data accesses) and L1 instruction caches.

To mimic the "join bound" queries, we focus on the nested loop join because it is the only join operator that appears to be dominant. To represent TPC-H's behavior accurately, we build an index on the join fields when evaluating μNJ. We do so because most join fields in the TPC-H workload have indexes, and the index decreases the query execution time significantly.

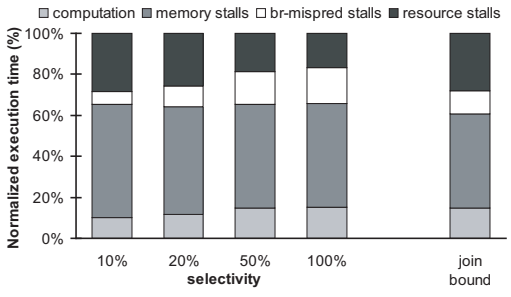


Figure 6: **μNJ vs. TPC-H "join bound" query.** The graph shows the time breakdowns of μNJ and TPC-H "join bound" queries. For the μNJ query, we vary its selectivity from 1% to 100% to show how selectivity affects the behavior.

Figure 6 examines the execution time breakdown of the μNJ query and the "join bound" queries. It shows that selectivity significantly affects the execution time breakdown of the μNJ query, and a 20% selectivity best represents the characteristics of a "join bound" query. We also verify that the absolute performance measured in CPI matches between the TPC queries and the scaled down DBmbench query with a 20% selectivity. The average CPI for these benchmarks are approximately 2.95.

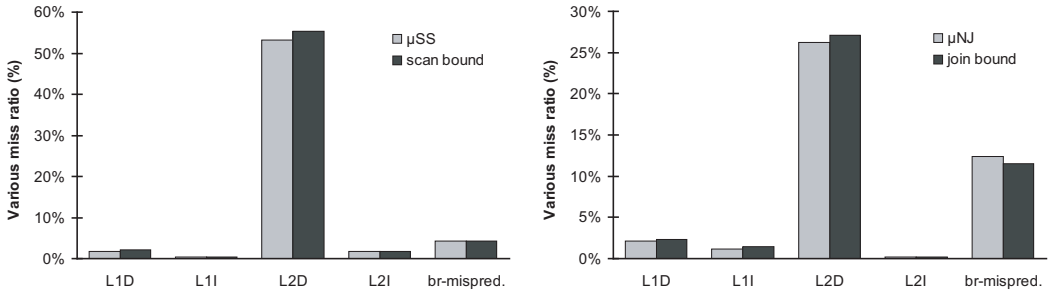
Figure 7(a) and Figure 7(b) compare the stall event frequencies across the benchmarks suites. Much like the "scan bound" queries, the execution of μSS is dominated by L2 cache misses. Similarly, besides the high fraction of L2 cache stalls, the execution of μNJ much like the "join bound" queries also incurs a high rate of L1 i-cache misses and branch mispredictions. The L1 d-cache misses are often overlapped Moreover, the actual differences in event counts between the benchmark suites are negligible.

In summary, the simple μSS and μNJ queries in μTPC-H closely capture the microarchitecture-level performance behavior of the "scan bound" and "join bound" queries in the TPC-H workload respectively. μTPC-H reduces the number of queries in TPC-H from 22 to 2. Moreover, μTPC-H allows for scaling down the dataset with predictable behavior from 10GB to 100MB. We measure a reduction in the total number of instructions executed from 1.8 trillion in TPC-H to 1.6 billion in μTPC-H, making μTPC-H a suitable benchmark suite for microarchitecture simulation and research.

5.3 Analyzing the OLTP Benchmarks

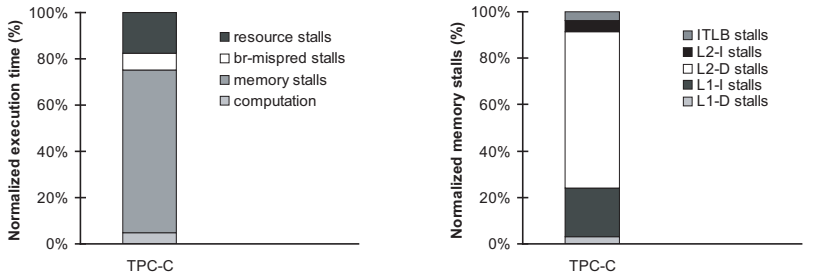
Figure 8 shows the execution time and memory stall breakdowns for a 150-warehouse, 100-client TPC-C workload corresponding to a 15GB dataset. Much like the TPC-H results, these results corroborate prior findings on microarchitecture-level performance behavior of TPC-C [1].

The effect of the high instruction cache miss rates result in an increased memory stall component, which is nevertheless dominated by L2 stall time due to data accesses. The reason is that, although the L2 data miss rate is not that high, in TPC-C each L2 data miss reflects I/O delays (TPC-C incurs I/O costs regardless of the dataset size, because it logs the transaction updates).



(a) **μSS and "scan bound" miss ratio comparison** (b) **μNJ and "join bound" miss ratio comparison**

Figure 7: **μTPC-H vs. TPC-H** The graphs compare the miss ratios of μTPC-H and TPC-H



(a) **TPC-C execution time breakdown.** (b) **TPC-C memory stall breakdown.**

Figure 8: **TPC-C time breakdowns.**

5.4 Comparison to μTPC-C

Figure 9(a) compares the execution time breakdown of the μIDX query and the TPC-C benchmark. It shows that the μIDX query with 0.01% and 0.1% selectivity mimics the execution time breakdown of TPC-C. Increasing the selectivity to 0.1%, however, also achieves the desired memory stall breakdown (shown in Figure 9(b)). The interesting result here is that selectivity is important to fine-tune memory stall breakdown.

The execution of a single μIDX query exhibits fewer stall cycles caused by L1 instruction cache misses. This is because the TPC-C workload has many concurrently running transactions which aggravate the localities in the instruction stream. We an improve the similarity by running multiple μIDX queries. We increase the L1 instruction cache miss rate from 0.017 to 0.032 with 10 currently running queries, which is similar to the L1 instruction cache miss rate of TPC-C ($\simeq$ 0.036).

Figure 10 shows the miss ratios of μIDX with a 0.1% selectivity and TPC-C. We can see from the graph that the branch misprediction rate of the μIDX query is the performance metric that is far from the real TPC-C workload. The simpler execution path of the μIDX query might be the reason for this discrepancy. The branch misprediction rate cannot be improved with a higher degree of concurrency. Fortunately, this discrepancy does not affect the performance bottleneck, as shown in Figure 9(a). This branch prediction mismatch, however, results in a small overall CPI difference of 8.4 for μIDX as compared to 8.1 for TPC-C.

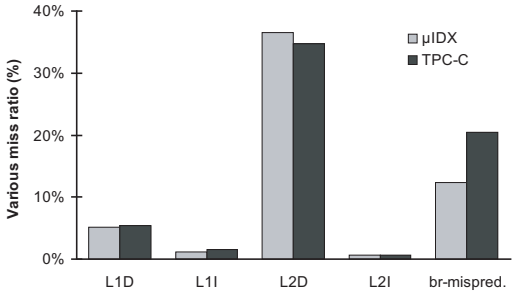
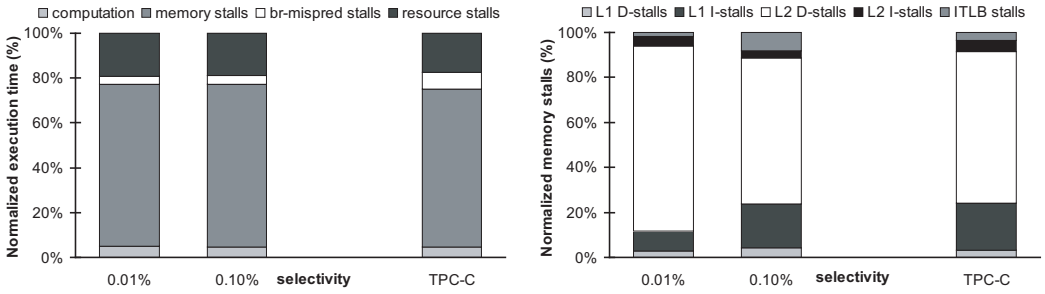


Figure 10: **μIDX vs. TPC-C:** The graph compare the miss ratios of μIDX and TPC-C



(a) **Execution time breakdown comparison** (b) **Memory stall breakdown comparison**

Figure 9: **μIDX vs. TPC-C.** The graphs show the time breakdowns of μIDX and TPC-C.

A single μIDX query is not enough to mimic the instruction-related performance behavior of TPC-C. We can achieve better approximation by running multiple μIDX queries. As for the data cache, the μIDX query represents TPC-C well on the L1 data cache. Less locality in our simple μIDX query's execution, however, causes a higher L2 data cache miss rate.

In summary, the μIDX query can expose the execution bottlenecks of TPC-C successfully. Running multiple μIDX queries (usually 10) can closely mimic the execution path of the TPC-C workload with a 7.06% relative error. The μIDX query fails to approximate the branch misprediction rate of the TPC-C workload. We should take this into account when predicting the branch behavior of the TPC-C benchmark. μTPC-C reduces the dataset size from 10GB to 100MB as compared to TPC-C. It also reduces five transactions containing approximately 50 queries to just a single query. The total number of instructions executed per transaction is reduced from 91.65 million in TPC-C to 2.75 million in μTPC-C.

6 Related Work

Database workloads evaluation at the architectural level is a prerequisite toward improving the suboptimal performance of database applications on today's processors. It identifies performance bottlenecks in software and hardware and points out the direction of future efforts. Several workloads characterization efforts [1, 2, 3, 4, 5, 13, 15, 20] explore the characteristics of OLTP and/or DSS on various hardware platforms using either a small-scale database or a subset of a standard workload or both. Three studies [3, 15, 20] emphasize the scale-down issues and demonstrate that the modified benchmarks they use do not affect the results. However, they still lack detailed analysis based on sufficient experiments on database systems with different scales. Most recently, Diep et al. [7] report how varying the configuration parameters affects the behavior of an OLTP workload. They propose a parameter vector consisting of number of processors, disks, warehouses, and concurrent clients to represent an OLTP configuration. They then formulate empirical relationships of the configurations and show how these configurations change the critical workload behavior. Hankins et al [9] continue this work by first proposing two metrics, average instructions per transaction (*IPX*) and average cycles per instruction (*CPI*) to characterize OLTP behavior. Then they conduct an extensive, empirical examination of an *Oracle* based commercial OLTP workload on a wide range of the proposed metrics. Their results show that the IPX and CPI behavior follows predictable trends which can be characterized by linear or piece-wise linear approximations.

There are a number of recent proposals for microbenchmarking database systems. The first processor/memory behavior comparison of sequential-scan and random-access patterns across four database systems [1] uses an in-memory TPC-like microbenchmark. The microbenchmark used consists of a sequential scan simulating a DSS workload and a non-clustered index scan approximating random memory accesses of an OLTP workload. Although the microbenchmark suite is sufficiently similar to the behavior of TPC benchmarks for the purposes of the study, a comprehensive analysis varying benchmark configuration parameters is beyond the scope of that paper. Another

study [11] evaluates the behavior of a similar microbenchmark. Their microbenchmark simulates two sequential scan queries (Q1 and Q6) from the TPC-H suite, whereas for TPC-C, it devises read-only queries that generate random memory/disk access to simulate the access pattern of OLTP applications. Computation complexity affects the representativeness of the proposed micro-DSS benchmark, while the degree of database multiprogramming affects the micro-OLTP benchmark.

In this paper, we build on the previous work as follows. First, we address the scaling problem from a database's point of view in addition to the traditional microarchitecture-approaches. We examine how query complexity, as one important dimension of the scaling framework, can be reduced while preserving their key hardware level characteristics. Second, we use a wealth of metrics that are important to obtain a complete picture of the workload behavior. Third, we build microbenchmarks for both DSS and OLTP workload.

7 Conclusions

Database applications and systems are emerging as the popular (if not dominant) commercial workloads. Computer architects are increasingly relying on database benchmarks to evaluate future server designs. Unfortunately, conventional database benchmarks are prohibitively complex to set up, and too large to experiment with and analyze when evaluating microarchitecture-level performance bottlenecks.

In this paper, we first presented a detailed performance study of the dominant DSS and OLTP benchmarks, TPC-H and TPC-C, and highlighted their key processor and memory performance characteristics. We then introduced a systematic scaling framework to scale down benchmarks for database workloads. We presented experiments on scaled-down TPC-H and TPC-C benchmarks to verify the viability of the framework.

Finally, we presented the DBmbench benchmarks, μTPC-H and μTPC-C, consisting of substantially scaled-down benchmarks for DSS and OLTP workloads. We showed that these benchmarks accurately mimic the microarchitecture-level execution time breakdown of TPC-H and TPC-C. The benchmarks allow computer architects to carry out detailed performance analysis of microarchitecture accurately with little complexity and reduced experimental turnaround time.

About the authors

Minglong Shao is a Ph.D. candidate in Carnegie Mellon University. She received her Bachelor's degree in Computer Science from Tsinghua University at Beijing, China. Her research interests are Database performance characterization and new data organization on modern CPUs and disks.

Anastassia Ailamaki is an Assistant Professor in Carnegie Mellon University. Her research interests are in the board area of database systems and applications, including building systems to strengthen the interaction between the database software and the underlying hardware and I/O devices, automated database design for scientific databases, storage device modeling and internet querying.

Babak Falsafi is an Associate Professor in the Electrical and Computer Engineering Department at Carnegie Mellon University. His research interests are primarily centered around computer architecture with emphasis on high-performance memory systems, multiprocessor architectures, and power-aware processor and memory architecture.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: where does time go? In *Proceedings of International Conference on Very Large Databases*, pages 266–277. Morgan Kaufmann Publishing, Inc., 1999.

[2] M. Annavaram, T. Diep, and J. Shen. Branch behavior of a commercial OLTP workload on Intel IA32 processors. In *Proceedings of International Conference on Computer Design*, September 2002.

[3] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of International Symposium on Computer Architecture*, pages 3–14, June 1998.

[4] L. A. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese. Impact of chip-level integration on performance of

OLTP workloads. In *Proceedings of International Symposium on High-Performance Computer Architecture*, January 2000.

[5] Q. Cao, P. Trancoso, J.-L. Larriba-Pey, J. Torrellas, R. Knighten, and Y. Won. Detailed characterization of a quad Pentium Pro server running TPC-D. In *Proceedings of International Conference on Computer Design*, October 1999.

[6] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), June 1970.

[7] T. Diep, M. Annavaram, B. Hirano, and J. P. Shen. Analyzing performance characteristics of OLTP cached workloads by linear interpolation. In *Workshop on Computer Architecture Evaluation using Commercial Workloads*, September 2002.

[8] J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, Inc., second edition, 1993.

[9] R. Hankins, T. Diep, M. Annavaram, B. Hirano, H. Eri, H. Nueckel, and J. P. Shen. Scaling and characterizing database workloads: bridging the gap between research and practice. In *Proceedings of International Symposium on Microarchitecture*, December 2003.

[10] IBM Corporation. *IBM DB2 Universal Database Administration Guide: Implementation*, 2000.

[11] K. Keeton and D. A. Patterson. *Towards a simplified database workload for computer architecture evaluations*, chapter 4. Kluwer Academic Publishers, 2000.

[12] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance characterization of a quad Pentium Pro SMP using OLTP workloads. In *Proceedings of International Symposium on Computer Architecture*, pages 15–26, June 1998.

[13] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of International Symposium on Computer Architecture*, June 1998.

[14] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2003.

[15] P. Ranganathan, S. V. Adve, K. Gharachorloo, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, volume 33, pages 307–318, November 1998.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[17] SimpleScalar tool set. SimpleScalar LLC. http://www.simplescalar.com.

[18] The Standard Performance Evaluation Corporation. *SPEC CPU Benchmark*. http://-www.specbench.org.

[19] S. S. Thakkar and M. Sweiger. Performance of an OLTP application on symmetry multiprocessor system. In *Proceedings of International Symposium on Computer Architecture*, 1990.

[20] P. Trancoso, J.-L. Larriba-Pey, Z. Zhang, and J. Torrellas. The memory performance of DSS commercial workloads in shared-memory multiprocessors. In *Proceedings of International Symposium on High-Performance Computer Architecture*, Feburary 1997.

[21] Transaction Processing Performance Countil. *TPC benchmarks*. http://www.tpc.org.

[22] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of International Symposium on Computer Architecture*, June 2003.