Chapter 3

TOWARDS A SIMPLIFIED DATABASE WORKLOAD FOR COMPUTER ARCHITECTURE EVALUATIONS

Kimberly Keeton[1] and David A. Patterson
University of California at Berkeley
Computer Science Division
384 Soda Hall #1776
Berkeley, CA 94720-1776
{kkeeton,patterson}@cs.berkeley.edu

Abstract We propose and evaluate a simplified technique for studying the architectural behavior of database workloads. This "microbenchmark" technique poses simple queries of the database to generate the same dominant I/O patterns exhibited in more complex, fully-scaled workloads. The potential benefits from this microbenchmark approach include smaller hardware requirements, less extensive workload parameter tuning, and simpler database parameter tuning. We demonstrate that the microbenchmark workload exhibits processor and memory system behavior relatively similar to that of the more complex standardized benchmarks. We also enumerate several factors that impact the representativeness of these microbenchmark workloads.

Keywords: Database, transaction processing, decision support, microbenchmark, and performance evaluation.

1. INTRODUCTION

In the last five to ten years, several studies have explored the architectural characteristics of online transaction processing (OLTP) database workloads [3] [7] [8] [9] [16] [17] [18] [19] [22] [23] [24] [26] [27]

[1] This work was performed as part of the author's dissertation research. The author's present address is: Storage Systems Program, Hewlett-Packard Laboratories, 1501 Page Mill Road, M/S 1U-13, Palo Alto, CA 94304-1126. Her current email address is kkeeton@hpl.hp.com.

[33] and decision support (DSS) database workloads [3] [5] [15] [17] [18] [23] [32]. These studies used standard workloads defined by the Transaction Processing Performance Council (TPC), namely TPC-B and TPC-C for OLTP [10] and TPC-D, TPC-H and TPC-R for DSS [10] [30] [31]. Although these benchmarks specify well-defined workloads, they pose several challenges for the computer architects who attempt to use them in performance evaluations.

First, studying full-scale TPC database performance requires large hardware configurations, including at least tens or hundreds of disk drives and gigabytes of main memory [28] [29]. Researchers hoping to measure the performance of a real system must construct such a system, costing hundreds of thousands to millions of dollars. If, instead, the researcher wishes to simulate a full-scale system, he or she must simulate a large and complex I/O subsystem, requiring considerable computational resources and time.

Second, the complexity of the TPC benchmarks requires that researchers make many configuration choices, including whether data is accessed through the file system or through the raw disk device interface, the layout of data on disk to avoid access hot spots, and the choice of index creation to improve performance. Running the benchmark workloads requires tuning additional configuration parameters, such as database size (e.g., TPC-C's number of warehouses or TPC-D's scale factor) and the number of simulated OLTP clients.

Third, the database servers themselves pose both hardware and software configuration challenges. Large hardware systems force the researcher to decide on the number and speed of disks, I/O controllers, and I/O buses, and the amount and configuration of physical memory. Commercially available database servers have 75 to 200 initialization parameters to control runtime management issues such as the buffer pool size and management strategy, the degree of multithreading/processing, logging, and disk read-ahead. The operating system also presents numerous configuration alternatives. Although default values are often provided for these configuration parameters, they do not necessarily match the requirements of the intended workloads.

Few researchers have the resources to construct the large hardware systems required for fully scaled TPC benchmark workloads, nor the expertise to tune workload-specific and database-specific parameters so that the workloads run efficiently. Although some computer architecture researchers have found methods for working around many of the difficulties, the barriers to studying database workloads still exist. Essentially, one must have close industrial ties to study interesting configurations. As a result, database workloads have been used in fewer than ten percent of the performance evaluations reported in ISCA and ASPLOS over the last five

years. Computer system designers would benefit from simpler benchmarks with more modest hardware requirements, to decrease the start-up cost for studying database workload performance.

In this study, we evaluate a database microbenchmark workload that poses simple queries to the database to generate the same I/O patterns that dominate the complex workloads. OLTP behavior is approximated by a random microbenchmark, which is based on an index scan. DSS characteristics are approximated by a sequential microbenchmark, which is based on a sequential table scan. The potential benefits from this microbenchmark approach include smaller hardware requirements, less extensive workload parameter tuning, and simpler database parameter tuning. We demonstrate that the microbenchmark workload exhibits processor and memory system behavior relatively similar to that of the more complex standardized benchmarks. We also enumerate several factors that impact the representativeness of these microbenchmark workloads.

This chapter is organized as follows. Section 2 discusses our initial proposal for a microbenchmark workload, and reviews our experimental methodology. Section 3 and Section 4 present the characterization of our microbenchmarks, comparing their behavior with the behavior of the TPC workloads. We outline related work in Section 5 and present conclusions and future work in Section 6.

2. APPROACH: MICROBENCHMARKS TO APPROXIMATE COMPLEX WORKLOADS

Our high-level goal is to simplify the task of studying fully scaled OLTP and DSS workloads, by designing a simpler microbenchmark suite that possesses characteristics similar to those of the more complex workloads. We evaluate representativeness by comparing the processor and memory system characteristics of the microbenchmarks with that of fully scaled workloads running on similar hardware. We choose these metrics since most fully scaled database servers are configured with enough disks to be CPU-bound; hence processor and memory behavior are important factors in determining database performance [3].

Folklore proposes that the behavior of common OLTP and DSS database workloads can be approximated by their dominant I/O patterns. OLTP workloads are dominated by small random I/O operations - both reads and writes. DSS, on the other hand, is dominated by large sequential operations, which are mostly reads. In subsequent sections, we will show that this folklore is generally accurate.

2.1 Microbenchmark Design

Our approach is to pose simple queries against a database table whose size is larger than the size of main memory, ensuring that the queries will generate I/O requests. Sequential I/O patterns are performed by requesting a sequential scan of the table (without using any indices). For the random I/Os, we create a *non-clustered* index on the table. The term non-clustered means that the order of the index data is not the same as the order of the table data as it is stored on disk. Thus, reading the index in order will result in a series of random read operations to retrieve the data pages from disk.

To ensure that output data processing (for example, the communication of results between the server and client, and the formatting and presentation of the data by the client program) does not dominate the I/O processing of the queries, we compute aggregate functions, such as a count of the eligible rows, on the output data. Thus, only a single number is passed between the server and client, and formatted and displayed by the client. The use of aggregates in the microbenchmark is representative of the more complex workloads. DSS queries are designed to summarize data trends, and as a result, most of the TPC DSS queries compute one or more aggregates instead of returning large amounts of data to the user. In addition, OLTP workloads are designed to retrieve or modify a small portion of the database, resulting in a small amount of data returned to the client.

Figure 1 presents the database schema for our microbenchmark workload. The table `ubench` has 100 byte records (rows), which are fashioned after the AS^3AP benchmark schema [10]. To ensure that the table is too big to fit into the 256 MB main memory of our machine, we generate data for 10 million rows, for a total of 1 GB of data. For the random experiment, we also create the non-clustered index `ubench_ix.IR0` on the `IntRand0` column.

```
CREATE TABLE ubench {
  IntOrd0 INTEGER(4) NOT NULL, /* unique; sequential */
  IntRand0 INTEGER(4) NOT NULL, /* unique; permutation */
  IntConst0 INTEGER(4) NOT NULL, /* constant value = 0 */
  FloatOrd SMALLFLOAT NOT NULL,
  DoubleRand0 FLOAT NOT NULL, /* unique; permutation */
  DateOrd DATE YEAR TO SECOND NOT NULL,
  DecimOrd DECIMAL(18,2) NOT NULL,
  IntOrd1 INTEGER(4) NOT NULL, /* same as IntOrd0 */
  IntRand1 INTEGER(4) NOT NULL, /* same as IntRand0 */
  IntConst1 INTEGER(4) NOT NULL, /* same as IntConst0 */
  DoubleRand1 FLOAT NOT NULL, /* same as DoubleRand0 */
  Char2Const CHARACTER(2) NOT NULL,
  DecimRand DECIMAL(18,2) NOT NULL,
  IntOrd2 INTEGER(4) NOT NULL, /* same as IntOrd0 */
  IntRand2 INTEGER(4) NOT NULL, /* same as IntRand0 */
  IntConst2 INTEGER(4) NOT NULL, /* same as IntConst0 */
  DoubleRand2 FLOAT NOT NULL, /* same as DoubleRand0 */
  Char6Const CHARACTER(6) NOT NULL,
}

CREATE INDEX ubench_ix.IR0 ubench(IntRand0) BTREE
```

Figure 1. Microbenchmark database schema.

The most important column attributes for the majority of our experiments are the `IntOrd0`, `IntRand0`, and `IntConst0` attributes; the remaining column attributes are added for follow-on experiments, including manipulation of other data types. The table is organized on disk according to the `IntOrd0` attribute, which is a unique ordinal integer attribute with values assigned sequentially from 0 to 9,999,999; it can be considered a row identifier. We note that there is no primary key index on `IntOrd0`. `IntRand0` is a unique integer attribute whose values are assigned pseudo-randomly, according to a permutation to ensure that index accesses require a new random read for each row retrieved [25]. We use the following permutation to assign each successive `IntRand0` value to a new data page:

$$\texttt{IntRand0}_i = (\texttt{IntOrd0}_i \ \%\ \texttt{rowsPerPage}) * \texttt{numPages} + (\texttt{IntOrd0}_i \ /\ \texttt{rowsPerPage})$$

Here `rowsPerPage` corresponds to the number of rows that can fit in the page size dictated by the database. `numPages` is the number of pages

required to store the table. The operator `%` corresponds to a modulus operation, and the operator `/` corresponds to integer division. Finally, the `IntConst0` attribute is assigned a constant value for all rows. We chose 0 as the constant value.

To generate sequential read accesses, we use the following query, which will be called the "basic" sequential query in subsequent sections:

```
SELECT COUNT(*) FROM ubench WHERE IntOrd0 < const;
```

In the absence of a primary index on the `IntOrd0` column, the database optimizer doesn't know that the data is sorted by the `IntOrd0` attribute. As a result, this query requires that the `IntOrd0` attribute for all of the rows in the `ubench` table be examined; the database optimizer expresses this query as a sequential table scan. We chose a `const` value of 1000. Other research indicates that varying this `const` value, which varies the selectivity of the query, may impact the observed architectural behavior [1]. We leave this experiment as future work.

We also present results for a follow-on query that performs more computation per row than the basic query:

```
SELECT COUNT(*), SUM(DecimOrd), MAX(DecimOrd)
FROM ubench WHERE DecimOrd0 < const;
```

This query is dubbed the "compute"-intensive sequential query in subsequent sections. To ensure that the same amount of computation is performed for each row, we set `const` to 10,000,000, which has the effect of selecting all of the rows for the aggregate computations.

The random read accesses are generated using the following "basic" query:

```
SELECT COUNT(*) FROM ubench WHERE IntRand0 >=
minimum AND IntRand0 < maximum AND IntConst0 =
0;
```

Here the `IntRand0` attribute is examined to prompt the database to use the `ubench_ix.IR0` index. We include the check on the value of `IntConst0` to ensure that the database retrieves the data page for the row - thus generating a random read operation - rather than simply answering the query from the index alone. On each successive run, we change the values of `minimum` and `maximum` to ensure that new data is accessed, requiring new random I/O operations. The [`minimum...maximum`] range impacts the optimizer's choice of minimum-cost query plan: for small ranges, the optimizer chooses the index scan, which will generate a small number of

random I/O operations. For larger ranges, the optimizer determines that the cost of many random I/Os will be too high, and instead chooses a sequential table scan. Thus, we must strike a balance between minimizing the range for the optimizer to choose the desired plan, and maximizing the range to increase the query execution (and measurement) time. For our experiments, we chose a [`minimum...maximum`] range of 150,000, or 1.5% of the total rows.

Finally, we also present results for a random follow-on query that includes a higher degree of multiprogramming and access to more cache lines:

```
SELECT COUNT(*) FROM ubench WHERE IntRand0 >=
minimum AND IntRand0 < maximum AND IntConst0 = 0
AND IntConst1 = 0 AND IntConst2 = 0;
```

This query is called the multiprogrammed, or "multi," random microbenchmark in subsequent sections. As with the basic random microbenchmark query, we examine the `IntRand0` attribute to prompt the database to use the `ubench_ix.IR0` index, and check the value of `IntConst0` to ensure that the database accesses the data page for each eligible row. We also access several other fields (`IntConst1` and `IntConst2`) to ensure that all cache lines for an eligible row are examined. To increase the instruction cache footprint, we increase the degree of multiprogramming: for each experiment we simultaneously invoke five copies of the follow-on query. To ensure that data is not reused in the cache, we use different [`minimum...maximum`] ranges for the five simultaneous queries, and change the values between successive experiments.

2.2 Software Configuration

The hardware and software configurations used for our microbenchmark, OLTP, and DSS experiments are shown in Table 1. We measured Informix' parallel shared-memory database server [12] running on Windows NT 4.0. For the microbenchmarks, we attempted to use the default database settings as much as possible, changing only a minimal number of run-time parameters, such as the amount of memory allocated to the database buffer pool and the degree of I/O read-ahead. We allocated half of the physical memory (128 MB) to the database buffer pool. Read-ahead was set to maximize the number of pages read at a single time for sequential I/O operations.

Table 1. Summary of system configurations.

Characteristic	Microbenchmarks	OLTP (TPC-C)	DSS (TPC-D)
Processor	200 MHz Pentium Pro	200 MHz Pentium Pro	200 MHz Pentium Pro
No. of processors	1	1	4
Caches: L1 I/L1 D/L2 (unified)	8 KB/8 KB/256 KB	8 KB/8 KB/256 KB	8 KB/8 KB/256 KB
System chipset	82440 FX	82450 KX/GX (Orion)	82450 KX/GX (Orion)
System bus speed	66 MHz	66 MHz	66 MHz
Memory size	256 MB	512 MB	4 GB (3 GB per process)
Mem. organization	Non-interleaved	4-way interleaved	4-way interleaved
Memory read B/W	99 MB/s	213 MB/s	213 MB/s
Mem. read latency (dependent loads)	315 ns (63 cycles)	190 ns (38 cycles)	190 ns (38 cycles)
I/O system: data disks	6 Seagate 4.2 GB	90 Quantum 4.55 GB	56 Quantum 4.55 GB
I/O system: controllers	1 Adaptec U/UW SCSI	3 Adaptec UW SCSI	4 Adaptec UW SCSI
Operating system	NT 4.0 Server, SP 4	NT 4.0 Enterprise Server, SP 3	NT 4.0 Enterprise Server, SP 3
Database server	Informix IDS	Informix ODS	Informix IDS
Additional results in:	[14]	[14], [16]	[14], [15]

We experiment with performing I/O through the Windows NT file system (NTFS) and through the raw disk interface. We examine the raw disk interface to perform an apples-to-apples comparison with the full-scale OLTP and DSS workloads, which use the raw device interface to perform I/O. We examine NTFS data management to understand the sensitivity of results to the choice of I/O interface. This comparison is performed only for the "basic" microbenchmark queries described above. All follow-on queries use the raw disk interface.

For both sets of experiments, we attempt to distribute data as evenly as possible over six data disks. For the NTFS experiments, the data disks are managed as an NTFS stripe set, and the microbenchmark database resides in an NTFS file created on the stripe set. For the raw disk experiments, our microbenchmark database was hash partitioned over database-managed "fragments" on the six disks. We used the following hash function: `IntOrd0 % numDisks`.

The microbenchmark sequential and random queries were created as stored procedures on the database, to ensure that the query would be statically compiled (optimized) before being run, thus eliminating the run-time overhead of compiling the query. In the case of the "multi" random microbenchmark, we created five different stored procedures; each invokes the query once. The stored procedures were invoked using the command-line client interface provided with the database.

2.3 Hardware Configuration and Measurement Methodology

As shown in Table 1, the microbenchmark hardware system is considerably simpler than that used for the full-scale OLTP and DSS workloads: a uniprocessor Pentium Pro server with only 256 MB of main memory, and about an order of magnitude fewer disks.

Since our evaluation of microbenchmark representativeness focuses on processor and memory behavior, we next highlight several features of the Pentium Pro. The Pentium Pro implements dynamic execution using an out-of-order, speculative execution engine, which employs register renaming and non-blocking caches. Intel x86 instructions (macro-instructions) begin and end execution in program order. They are translated into a sequence of simpler RISC-like micro-operations (μops), which are permitted to execute out-of-order. Once decoded, μops are register renamed and placed into an out-of-order speculative pool of pending operations (the reservation station). Once their data arguments and the necessary computational resources are available, these μops are issued for execution in the out-of-order section of the processor. After execution has completed, an instruction's μops are held in a reorder buffer until they can be retired, which may occur only after all previous instructions have been retired, and all of the instruction's constituent μops have completed. The Pentium Pro retires up to three μops per clock cycle, yielding a theoretical minimum cycles per μop (μCPI) of 0.33. More information on the Pentium Pro can be found in [6] [11] [13] [21] [34].

Measurements were performed using the Pentium Pro hardware counters [13]. We present aggregate (user+operating system) activity, factoring out the idle loop. On the uniprocessor, this technique is possible because NT implements the idle loop using the HALT instruction. The event counters are inactive during this idle loop, ensuring that we can reliably separate system mode counter observations for the operating system and the idle loop. Idle time is negligible for the full-scale TPC workloads. We ran each microbenchmark query multiple times, measuring a single pair of events during each run. Each run was broken into two-second fixed duration intervals, and the number of events occurring during that two-second window was recorded. We found that the standard deviation for a given event was less than 10% of the mean for nearly all event types measured.

3. SEQUENTIAL I/O APPROXIMATIONS FOR DSS

This section examines how closely the architectural characteristics of the sequential microbenchmark resemble those of a DSS workload based on TPC-D [14] [15]. We focus on two of the scan-intensive queries, Q1 and Q6, since they are the operations most comparable to the sequential microbenchmark. We note that these DSS queries compute complex aggregate functions (for example, sums and averages of non-native decimal data types), which require more computation per row than the basic sequential microbenchmark. Q6 computes one such aggregate, while Q1 computes eight aggregates (predominantly of `decimal` types, with a few `integer` aggregates). Can the basic microbenchmark approximate the behavior of these more computationally-intensive queries?

3.1 Sequential Microbenchmark CPI Analysis

We first examine the breakdown of cycles and CPIs of the various workloads, shown in Table 2. We observe that the microbenchmark system exhibits more kernel time than the DSS queries, regardless of which I/O interface is employed. We assert that this is due to the small amount of computation that the database performs per row for the basic query. As a result, the kernel I/O processing becomes a non-trivial portion of the total processing per row. This assertion is supported by the compute-intensive query results, which exhibit roughly the same user-kernel time breakdown as the DSS queries. Like the full-scale DSS queries, the microbenchmark queries exhibit negligible idle time.

Table 2. Breakdown of time, measured cycles per micro-operation (µCPI) and measured cycles per macro-instruction (CPI) for the sequential microbenchmark and the DSS queries

Characteristic	NTFS seq. µbench	Raw seq. µbench	Compute seq. µbench	DSS Q1 (complex)	DSS Q6 (simple)
% user time	88%	86%	95%	99%	93%
% kernel time	12%	14%	5%	1%	4%
% idle time	0%	0%	0%	0%	3%
µCPI	1.06	1.11	1.08	0.86	0.94
CPI	2.00	1.86	1.91	1.39	1.65
µops/instr.	1.88	1.70	1.77	1.62	1.75

The microbenchmark system's overall µCPI differs from the DSS queries' µCPIs by 13% to 29%. CPI differences are even greater (13% to 44%), due to the somewhat higher ratio of µops to instructions for the microbenchmark. To understand these discrepancies, we decompose µCPI into its computation and stall components.

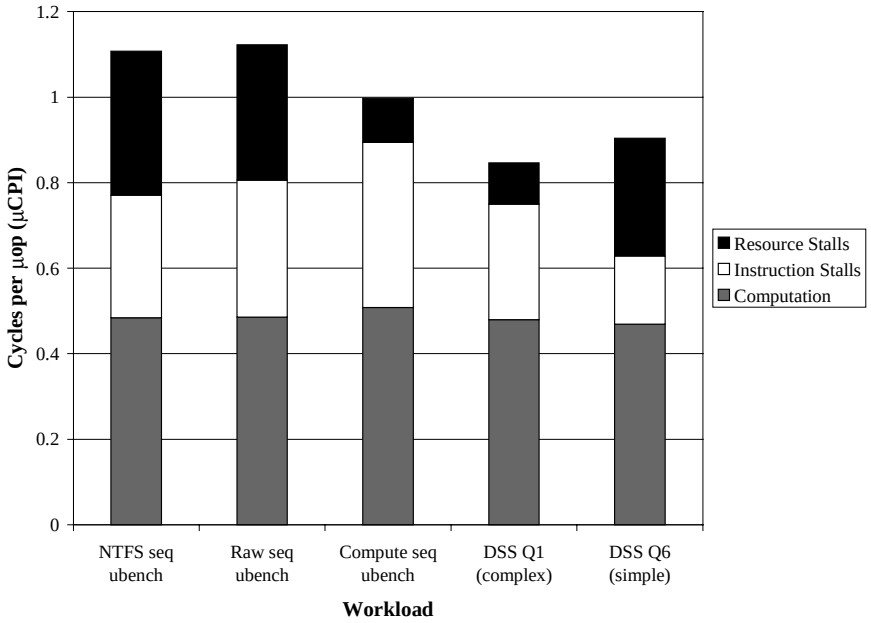

Figure 2. Breakdown of cycles per micro-operation (µCPI) for the sequential microbenchmark and the DSS queries

Figure 2 shows the detailed breakdown of the µCPI components, including computation, instruction stalls, and resource stalls. Computation µCPI is based on the µop retire profile described in Section 3.3; we assume µops retired in triple-retire cycles require 0.33 cycles in the steady state, double-retire cycle µops take 0.5 cycles, and single-retire µops need one cycle. (Note that this model implicitly assumes that µop execution takes one cycle.) This calculation determines the number of cycles per µop. Stall cycles are measured by two Pentium Pro counters as follows. Instruction-related stalls count the number of cycles instruction fetch is stalled for any reason, including L1 instruction cache misses, ITLB misses, ITLB faults, and other minor stalls. Resource stalls account for cycles in which the decoder gets ahead of execution. Examples of resource stalls are the conditions where execution units, reorder buffer entries, register renaming buffer entries, or memory buffer entries are full. In addition, serializing instructions (for example, `CPUID`), interrupts, and privilege level changes may spend considerable cycles in execution, forcing the decoder to wait and incrementing the resource stalls counter. Stalls due to data cache misses are not explicitly included in resource stalls; however, if some other resource becomes oversubscribed due to a long data cache miss, the resource stalls counter will be incremented.

From Figure 2, we observe that stall cycles comprise roughly half (45% to 55%) of the µCPI for all workloads. Stalls are somewhat more prevalent for the microbenchmark experiments, due in part to the slower memory speeds on the microbenchmark system. We see that the behavior of the basic query is nearly the same whether NTFS or the raw disk interface is used for I/O. The computation component is nearly identical for all workloads. The basic microbenchmark's instruction stall component is comparable to that of DSS Q1. Its resource stall component is roughly 1.2X that of Q6. We defer discussion of the compute-intensive query until the next section, where the cache behavior of these configurations is described more fully.

3.2 Sequential Microbenchmark Cache Behavior

Table 3 presents the number of cache misses per 1000 instructions retired and the miss ratios for both instruction and data caches for the sequential microbenchmark and the DSS queries. We begin our discussion with data cache behavior.

Table 3. Cache misses per 1000 instructions retired and cache miss ratios for the sequential microbenchmark and the DSS queries

Characteristic	NTFS seq. µbench	Raw seq. µbench	Compute seq. µbench	DSS Q1 (complex)	DSS Q6 (simple)
L1 I-cache misses	42 (3%)	38 (3%)	63 (6%)	40 (4%)	18 (1%)
L1 D-cache misses	23 (3%)	23 (3%)	20 (3%)	23 (3%)	29 (4%)
ITLB misses	0	0	1	1	0
L2 Inst. misses	1 (3%)	2 (4%)	1 (2%)	0 (1%)	1 (4%)
L2 Data misses	7 (28%)	6 (22%)	3 (14%)	1 (6%)	4 (15%)
Overall L2 misses	8 (12%)	8 (11%)	4 (5%)	1 (3%)	5 (11%)

We observe that the basic microbenchmark's L1 D-cache behavior is comparable to that of Q1, but its data-related L2 misses are at least 2X the L2 data misses of the DSS queries. This increased miss rate occurs because the microbenchmark performs less computation per row than the DSS queries do. This assertion is supported by the lower data-related L2 miss count of the compute-intensive microbenchmark query. Further evidence is provided by the trend observed in the DSS queries: Q6, which performs a single aggregate operation, experiences more L2 data misses than Q1, which performs more complex aggregates.

We hypothesize that this higher L2 data-related miss behavior contributes indirectly to the increased resource stalls experienced by the basic microbenchmark. The combination of higher L2 miss counts and a higher microbenchmark system memory latency result in an increase in the time the

processor must wait for the L2 miss to be satisfied by memory. As a result, conflicts for other resources, such as memory reorder buffer or register renaming entries, may be more likely to occur, resulting in the higher resource stall components experienced by the basic microbenchmark, as shown in Figure 2.

The basic microbenchmark's instruction cache miss behavior closely mirrors the behavior of Q1, leading to the similar instruction-related stall components shown in Figure 2. The majority of these stall cycles comes from L1 cache misses that hit in the L2 cache. The compute-intensive microbenchmark query, which includes several `decimal` aggregates and a comparison, has a larger instruction footprint, resulting in the increase in L1 I-cache misses. This high instruction miss count is responsible for the large instruction stall component shown for the compute-intensive microbenchmark query in Figure 2. We also experimented with decreasing the number of `decimal` aggregate operations for the compute-intensive microbenchmark to decrease the instruction miss count. The observed behavior was nearly identical to the compute-intensive behavior shown in Table 3.

We note that differences in L1 instruction cache miss behavior also exist between the two DSS queries. Q1 experiences more L1 I-cache misses due to the increased footprint required for the different `decimal` aggregate computations it must perform.

3.3 Sequential Microbenchmark ILP Behavior

Figure 3 shows the micro-operation retirement profile for the workloads, broken down by how many μops are retired in each type of retirement cycle. We observe that the majority of μops (60% to 65%) are retired in triple-retire cycles; single-retire and double-retire cycles account nearly equally for the remaining μops.

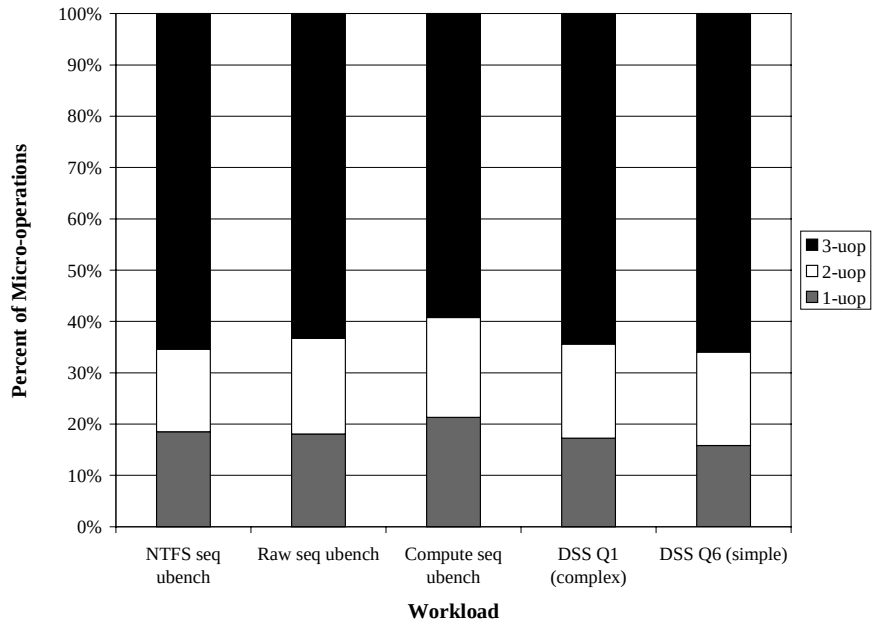

Figure 3. Micro-operation retirement profile, broken down by μops, for the sequential microbenchmark and the DSS queries

3.4 Discussion

These results suggest that the sequential microbenchmark is a promising technique for simplifying the evaluation of DSS workloads. Although the naive "basic" microbenchmark query is a good starting point for approximating the more complex workloads, it does not perform sufficient computation per row to achieve representative L2 data cache behavior. We demonstrated that increasing the amount and complexity of computation per row provides more representative behavior.

4. RANDOM I/O APPROXIMATIONS FOR OLTP

In this section, we determine how closely the random database microbenchmark approximates the processor and memory system behavior of a full-scale OLTP workload based on TPC-C [14] [16]. We note that the OLTP workload typically possesses a high degree of multiprogramming, and contains both read accesses and updates to the data, which generate logging activity. Thus, OLTP includes both read and write traffic to the data. Can a

simple index scan (read-only) operation approximate the behavior of the OLTP workload?

4.1 Random Microbenchmark CPI Analysis

We begin our analysis by quantifying the breakdown of cycles and CPIs of the systems, shown in Table 4. We observe that the breakdown of user, kernel, and idle time is roughly consistent between the random microbenchmark and the OLTP workload. The basic microbenchmark spends 5% to 10% more time in the kernel (and correspondingly less time at user level) than the fully-scaled OLTP workload. As in the sequential microbenchmark, we suspect that the small amount of database computation per row leads to the kernel I/O path being a non-trivial part of the total computation performed for each row. This hypothesis is strengthened by the "multi" random microbenchmark time breakdown. In this more highly multiprogrammed microbenchmark, the database spends more time switching between the different queries in the system, resulting in a somewhat higher percentage of user time and decreasing the relative importance of the kernel I/O path.

Table 4. Breakdown of time, measured cycles per micro-operation (µCPI), and measured cycles per macro-instruction (CPI) for the random microbenchmark

Characteristic	NTFS random µbench	Raw random µbench	Multi random µbench	OLTP
% user time	78%	72%	89%	83%
% kernel time	22%	28%	11%	17%
% idle time	0%	0%	0%	0%
µCPI	1.38	1.08	1.45	1.58
CPI	2.85	2.14	2.65	3.02
µops/instr.	2.06	2.00	1.83	1.91

We also see from Table 4 that the µCPI and CPI of the NTFS basic microbenchmark and the multiprogrammed microbenchmark closely mirror (within 15%) those of the full OLTP workload. In contrast, the µCPI and CPI of the basic microbenchmark running on the raw disk interface are considerably lower. To better understand the differences, we decompose the µCPI components further.

Figure 4 shows this breakdown of the µCPI components, including computation, instruction-related stalls and resource stalls. Stall cycles, rather than computation cycles, dominate the µCPI for all workloads; they comprise about 65% of the µCPI for the NTFS basic microbenchmark, the multiprogrammed microbenchmark, and the OLTP workload, and 55% of the µCPI for the raw disk basic microbenchmark. The computation components are very similar across the workloads, but the stall components vary more widely. Both basic microbenchmark configurations experience

about half as many instruction-related stalls as the full OLTP workload. This effect is due to the better instruction cache behavior exhibited by the microbenchmark. We describe this effect in more detail in the next section.

The NTFS basic microbenchmark experiences nearly 2X the resource stalls of the OLTP workload, whereas the raw disk basic microbenchmark experiences only about 75% of the OLTP total. We hypothesize that these relative amounts are due indirectly to the data cache behavior exhibited by the workloads, which is described in the next section.

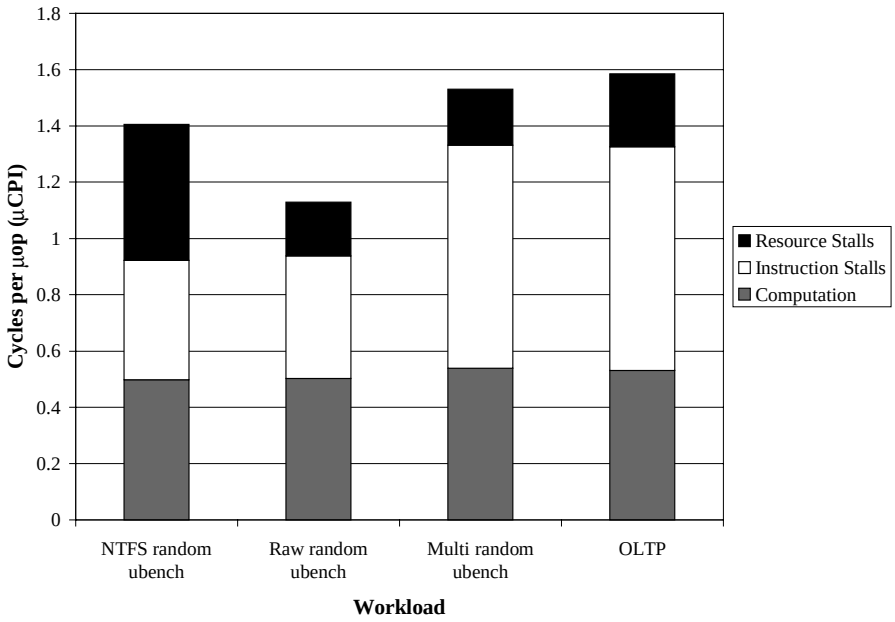

Figure 4. Breakdown of cycles per micro-operation (µCPI) for the random microbenchmark and the OLTP workload

4.2 Random Microbenchmark Cache Behavior

Table 5 presents the number of instruction and data cache misses and miss ratios for our microbenchmark and the OLTP workload. We focus first on instruction cache behavior. We observe that the basic microbenchmark has similar L1 and L2 instruction miss counts for the NTFS and raw disk interfaces. The number of microbenchmark instruction cache misses is noticeably smaller than the number of misses for the OLTP workload. This decrease is due to two factors: first, the decreased instruction working set that comes from the limited computation of the microbenchmark, and second, the decreased cache interference that comes with the microbenchmark's lower degree of multiprogramming. The OLTP workload possesses a larger instruction footprint because it uses a mix of five different

transactions; some of these transactions include update operations, which generate logging activity, including the kernel operations for performing disk writes. As described earlier, the basic microbenchmark does not include update operations. These lower instruction cache miss rates lead to the decreased instruction stalls experienced by the microbenchmark, relative to the OLTP system, as shown in Figure 4.

When we introduce a larger instruction working set and a higher level of multiprogramming into the microbenchmark by executing five queries simultaneously, we see that the instruction cache miss counts (especially L1 I-cache miss counts) increase considerably. This behavior results in the more representative instruction stall component observed for the multi random microbenchmark in Figure 4. The bulk of these instruction-related stalls are due to L1 misses that hit in the L2 cache.

Table 5. Cache misses per 1000 instructions retired and cache miss ratios of the random microbenchmark and the OLTP workload

Quantity	NTFS random µbench	Raw random µbench	Multi random µbench	OLTP
L1 I-cache misses	61 (3%)	70 (6%)	114 (10%)	99 (7%)
L1 D-cache misses	32 (4%)	23 (3%)	41 (6%)	51 (7%)
ITLB misses	1	2	5	4
L2 Inst. misses	3 (4%)	2 (3%)	3 (2%)	10 (11%)
L2 Data misses	10 (27%)	3 (11%)	5 (12%)	13 (26%)
Overall L2 misses	13 (13%)	5 (5%)	8 (5%)	23 (16%)

Focusing on the data cache behavior, we note that the basic microbenchmark's L1 data-cache miss count is 50% to 60% of the miss count for the OLTP workload. Again, the lower degree of multiprogramming leads to better cache behavior for the microbenchmark. Increasing the multiprogramming level results in more L1 data-cache misses, as shown in Table 5. We also investigated the possibility that a single query that touched all of the cache lines for a given database row would yield increased L1 data-cache misses, but the observed behavior was the same as the basic microbenchmark behavior.

Although NTFS microbenchmark L2 data cache behavior is similar to the OLTP L2 data behavior, the raw disk experiments (both basic and multi) experience much lower data-related L2 miss counts. This reduced count likely contributes to the reduced resource stall components for these microbenchmarks, as shown in Figure 4.

4.3 Random Microbenchmark ILP and Branch Behavior

Figure 5 illustrates the micro-operation retirement profile for the workloads, decomposed by how many µops are retired in each type of retirement cycle. We observe that the majority of µops (54% to 62%) are retired in triple-retire cycles, followed by single-retire cycles and finally double-retire cycles.

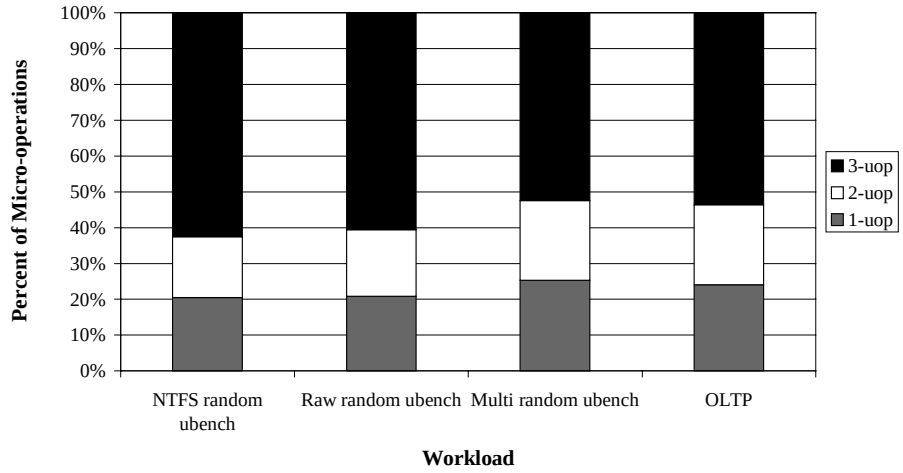

Figure 5. Micro-operation retirement profile, broken down by µops, for the random microbenchmark and the OLTP workload.

We present the branch behavior of the workloads in Table 6. We see that all workloads have similar branch frequencies of about 20%. The basic microbenchmarks' branch misprediction rates are about half the misprediction rate experienced by the OLTP system. This effect is due to the tighter instruction loop used for the microbenchmark, which minimizes jumps to other transactions, logging routines, and kernel write routines; this tighter loop leads to better branch prediction behavior. The greater multiprogramming level of the multi random microbenchmark produces more non-looping branches, resulting in worse branch prediction behavior that is more representative of OLTP misprediction behavior.

Table 6. Branch behavior for the random microbenchmark and the OLTP workload.

Quantity	NTFS random µbench	Raw random µbench	Multi random µbench	OLTP
Branch frequency	18.7%	17.4%	19.1%	20.2%
Branch misprediction ratio	8.6%	9.5%	16.4%	16.7%

4.4 Discussion

Our results indicate that a single random microbenchmark query is insufficient to approximate the more complex OLTP workload. The characteristics that differ the most between this basic microbenchmark and the OLTP workload are the instruction cache behavior and the branch misprediction behavior. We demonstrated that both of these characteristics could be made more representative by introducing multiple read-only queries to increase the microbenchmark multiprogramming level. The first-level data cache behavior, which differs somewhat between the workloads, also benefits from these proposals.

An alternate proposal for increasing the microbenchmark instruction cache footprint and branch misprediction rate is to introduce update operations. Our initial attempts at introducing updates have proven unsuccessful in generating a repeatable experiment. The deferred nature of database data page writes implies that the number of disk writes may vary drastically from run to run (resulting in query execution time variation), even though the number of rows updated is the same in successive runs. Attempts to decrease this variability by increasing the query duration have failed, due to the optimizer's estimation that the query will be unnecessarily long, resulting in the query being aborted. Thus, we believe that the most promising approach for producing a representative random microbenchmark lies in posing multiple read-only queries.

5. RELATED WORK

Many of the studies that use database workloads to evaluate computer architecture innovations have employed the complex OLTP [3] [7] [8] [9] [16] [17] [18] [19] [22] [23] [24] [26] [27] [33] and DSS [3] [5] [15] [17] [18] [23] [31] workloads defined by the TPC. These studies vary in their usage of full-scale data sets versus in-memory data sets. One study provides rules of thumb for using an in-memory version of the TPC-B OLTP benchmark to approximate the processor and memory behavior of TPC-C

workload running on a full-scale system [3]. Although these guidelines are quite useful, the authors do not present a head-to-head comparison of the resulting architectural behavior for the two systems.

Very few researchers have examined the use of database microbenchmarks as a means of simplifying the hardware and software requirements of database performance evaluation. The most closely related study, by Ailamaki, et al., uses in-memory microbenchmarks similar to the ones described above to evaluate commercial databases running on next-generation Intel hardware (a 400 MHz Pentium II Xeon uniprocessor, running Windows NT 4.0) [1]. A primary goal is to determine how database designers can modify their code to execute more efficiently on modern architectures. A secondary goal is to compare the differences in behavior between commercial databases. They find that half of the execution time is spent in stalls, which corroborates our findings. The majority of the stalls are due to second-level data cache misses and first-level instruction cache misses. While there are differences in the magnitudes of stall components between different databases, the relative importance of stall components is roughly consistent between the databases. They find that the microbenchmark behavior is similar to an in-memory version of TPC-D on the same database. A scaled-back (in-memory) version of TPC-C incurs more second-level cache and resource stalls than the microbenchmarks.

The biggest differences between this study and our study are that: 1) our microbenchmarks include I/O operations, which are a critical component of database workloads and 2) they measure a larger collection of commercial databases and an additional relational operator (join). Their in-memory tests represent the extreme case of simplifying hardware requirements for microbenchmarking. However, the representativeness of in-memory microbenchmarks for fully-scaled complex workloads is not clear, as the differences between in-memory experiments and experiments that perform I/O are not yet well-understood.

6. CONCLUSIONS AND FUTURE WORK

Although the TPC benchmarks provide standard database workloads for use in computer architecture evaluations, they pose several challenges to systems researchers, including large hardware requirements for full scale and complex hardware and software configuration choices. To remove these methodological hurdles, we propose and evaluate a simpler database workload that possesses more modest hardware requirements, yet still approximates the behavior of the more complex OLTP and DSS workloads. This “microbenchmark” approach is based on posing queries to the database

that generate the same dominant I/O patterns as the full workloads. The sequential microbenchmark, based on a sequential table scan, approximates DSS behavior, and the random microbenchmark, based on an index scan, approximates OLTP behavior. The key factor affecting the representativeness of the basic sequential microbenchmark is the complexity of the computation performed per database row. The key factor for the random microbenchmark is the degree of database multiprogramming. We demonstrate how these observations can be used to modify the basic microbenchmarks to produce processor and memory system behavior representative of the complex OLTP and DSS workloads. The initial results from our study show that this approach is a promising method for reducing the complexity of database performance evaluation. Additional evaluation is necessary, however, to refine this technique.

Towards that end, we note several additional areas for follow-on work. First, this study has focused on the most basic database primitive operations, sequential table scan and index scan; what is the behavior of other common database operations, such as joins and sorts? Second, related work indicates that some of these algorithms may experience multi-phased behavior if the data working set does not fit into memory, with separate phases for in-memory behavior, writes of temporary data to disk, and subsequent reads of the temporary data [14] [15]. (How) can a microbenchmark be designed to generate similar multi-phased behavior? Third, our microbenchmarks were evaluated on a uniprocessor, yet several studies suggest that the amount of L2 cache coherence traffic increases as the number of processors grows OLTP [3] [16]. (How) can a microbenchmark be designed to approximate multiprocessor cache coherence behavior? Fourth, how generally representative are our database microbenchmarks for different commercial databases? for other hardware platforms? Finally, as described in Section 5, in-memory microbenchmarks represent the extreme for minimizing system hardware requirements; do in-memory workloads possess behavior representative of workloads that perform I/O?

ACKNOWLEDGMENTS

This study has benefited greatly from methodological discussions with Jim Gray and Don Slutz from Microsoft's Bay Area Research Center and Eric Anderson from UC Berkeley. We thank Nisha Talagala for lending us the hardware for the microbenchmark system. John He at Informix actively supported the earlier work that led to the OLTP and DSS measurements.

REFERENCES

[1] A. G. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. “DBMSs on modern processors: Where does time go?” to appear in *Proc. of the 25th International Conference on Very Large Databases (VLDB ‘99)*, September 1999.

[2] R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. “The architectural costs of streaming I/O: a comparison of workstations, clusters, and SMPs,” Proc. 4th Symposium on High-Performance Computer Architecture (HPCA-4), pages 90 - 101, February 1998.

[3] L. Barroso, K. Gharachorloo and E. Bugnion. “Memory system characterization of commercial workloads,” In Proc. of the 25th Intl. Symposium on Computer Architecture (ISCA), June 1998.

[4] D. Bhandarkar and J. Ding. “Performance characterization of the Pentium Pro processor.” In *Proc. of HPCA-3*, February, 1997.

[5] Q. Cao, P. Trancoso, J.-L. Larriba, J. Torrellas, B. Knighten, and Y. Won. “Detailed characterization of a quad Pentium Pro server running TPC-D,” *Proc. of the Intl. Conference on Computer Design (ICCD)*, October 1999.

[6] R. P. Colwell and R. L. Steck. “A 0.6um BiCMOS processor with dynamic execution.” In *International Solid State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 176-177, February 1995.

[7] Z. Cvetanovic and D. Bhandarkar. “Performance characterization of the alpha 21164 microprocessor using tp and spec workloads.” In *Proc. of HPCA-2*, pages 270-280, February 1996.

[8] Z. Cvetanovic and D. D. Donaldson. “AlphaServer 4100 performance characterization.” *Digital Technical Journal*. 8(4):3-20, 1996,

[9] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu. “Evaluation of multithreaded uniprocessors for commercial application environments.” In Proc. of the 21st ISCA, June 1996, pp. 203 - 212.

[10] J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, Inc., 2nd edition, 1993. `http://www.benchmarkresources.com/handbook/index.html`.

[11] L. Gwennap. “Intel's P6 uses decoupled superscalar design.” *Microprocessor Report*, 9(2):9-15, 1995.

[12] *Informix Dynamic Server Administrator’s Guide, Vol. 1 and Vol. 2.*, Informix Corporation.

[13] Intel Corporation. *Pentium Pro family developer's manual, volume 3: Operating system writer's manual*. Intel Corporation, 1996, Order number 242692.

[14] K. Keeton. “Computer architecture support for database applications,” PhD dissertation, Univ. of California at Berkeley, July 1999.

[15] K. Keeton, Y. Q. He, and D. A. Patterson. “Performance characterization of decision support database workloads on a commodity SMP,” submitted for publication.

[16] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. “Performance characterization of the quad Pentium Pro SMP using OLTP workloads.” In Proc. of the 25th ISCA, June 1998. An extended version of this paper is available as University of California Computer Science Division Technical Report UCB/CSD-98-1001.

[17] 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 Proc. of the 25th ISCA, June 1998.

[18] T. Lovett and R. Clapp. “STiNG: A CC-NUMA computer system for the commercial marketplace.” In Proc. of the 23rd ISCA, pp. 308-317, May 1996.

[19]A. Maynard, et al. “Contrasting characteristics and cache performance of technical and multi-user commercial workloads.” In *Proc. of the 6th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 145-156, October 1994.

[20]L. McVoy and C. Staelin. “lmbench: Portable tools for performance analysis.” In *Proc. of the USENIX 1996 Annual Technical Conference*, January 1996.

[21]D. Papworth. “Tuning the Pentium Pro microarchitecture.” *IEEE Micro*, pages 8-15, April, 1996.

[22]S. E. Perl and R. L. Sites. “Studies of windows NT performance using dynamic execution traces,” In *Proc. of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 169-184, 1996.

[23]P. Ranganathan, et al. “Performance of database workloads on shared-memory systems with out-of-order processors,” In *Proc. of ASPLOS-VIII*, October 1998.

[24]M. Rosenblum, et al. “The impact of architectural trends on operating system performance.” In *Proc. of the 15th ACM SOSP*, pages 285Ð298, December 1995.

[25]D. Slutz. Personal communication, February 1999.

[26]S. S. Thakkar and M. Sweiger. “Performance of an OLTP application on Symmetry multiprocessor system.” In Proc. of the 17th ISCA, June 1990, pp. 228-238.

[27]J. Torrellas, et al. “Characterizing the cache performance and synchronization behavior of a multiprocessing operating system.” In *Proc. of ASPLOS-V*, pages 162-174, October 1992.

[28]TPC-C audited benchmark executive summaries, available from `http://www.tpc.org/.`

[29]TPC-H and TPC-R audited benchmark executive summaries, available from `http://www.tpc.org/.`

[30]Transaction Processing Performance Council. TPC Benchmark H (Decision Support) Standard Specification, Revision 1.1.0, 1998, http://www.tpc.org.

[31]Transaction Processing Performance Council. TPC Benchmark R (Decision Support) Standard Specification, Revision 1.0.1, 1998, http://www.tpc.org.

[32]P. Trancoso, J.-L. Larriba-Pey, Z. Zhang and J. Torrellas. “The memory performance of DSS commercial workloads in shared-memory multiprocessors.” In Proc. of HPCA-3, February 1997.

[33]B. Verghese, S. Devine, A. Gupta and M. Rosenblum. “Operating system support for improving data locality on CC-NUMA computer servers.” In *Proc. of ASPLOS-VI*, pages 279-289, October 1996.

[34]T. Yeh and Y. Patt. “Two-level adaptive training branch prediction.” In *Proc. IEEE Micro-24*, pages 51-61, November 1991.