ADDICT: Advanced Instruction Chasing for Transactions

Pınar Tözün
EPFL
pinar.tozun@epfl.ch

Islam Atta
University of Toronto
iatta@eecg.toronto.edu

Anastasia Ailamaki
EPFL
natassa@epfl.ch

Andreas Moshovos
University of Toronto
moshovos@eecg.toronto.edu

ABSTRACT

Recent studies highlight that traditional transaction processing systems utilize the micro-architectural features of modern processors very poorly. L1 instruction cache and long-latency data misses dominate execution time. As a result, more than half of the execution cycles are wasted on memory stalls. Previous works on reducing stall time aim at improving locality through either hardware or software techniques. However, exploiting hardware resources based on the hints given by the software-side has not been widely studied for data management systems.

In this paper, we observe that, independently of their high-level functionality, transactions running in parallel on a multicore system execute actions chosen from a limited subset of predefined database operations. Therefore, we initially perform a memory characterization study of modern transaction processing systems using standardized benchmarks. The analysis demonstrates that same-type transactions exhibit at most 6% overlap in their data footprints whereas there is up to 98% overlap in instructions.

Based on the findings, we design ADDICT, a transaction scheduling mechanism that aims at maximizing the instruction cache locality. ADDICT determines the most frequent actions of database operations, whose instruction footprint can fit in an L1 instruction cache, and assigns a core to execute each of these actions. Then, it schedules each action on its corresponding core. Our prototype implementation of ADDICT reduces L1 instruction misses by 85% and the long latency data misses by 20%. As a result, ADDICT leads up to a 50% reduction in the total execution time for the evaluated workloads.

1. INTRODUCTION

Online transaction processing (OLTP) is one of the most demanding database applications and a multibillion-$/year industry. It is for this reason that OLTP has been one of the main applications that drive advancements in the data management ecosystem. Despite innovations in the database and computer architecture communities, recent workload characterization studies show that micro-architectural resources are severely underutilized when running OLTP [5, 22, 24]. Up to 80% of the execution cycles go to memory stalls [5]. As a result, on modern processors, OLTP barely achieves one instruction per cycle (IPC), far below the processors peak capability of four IPC [24].

Previous works on reducing memory stall time for data management systems aimed at reducing cache miss rates, focusing primarily on improving locality and cache utilization for data rather than for instructions. Proposals range from cache-conscious data structures and algorithms [4, 7] to sophisticated data partitioning and thread scheduling [17] at the software-side, whereas hardware techniques mainly target data prefetching [21].

Recent studies [5, 22, 23, 24] reveal that, for traditional transaction processing systems, the stall time due to L1 instruction misses is at least as problematic as long-latency data misses from the last-level cache. Improving code layout by writing better code or by compilation optimizations [18] does improve instruction cache utilization but does so by mainly reducing conflict misses. However, it is capacity misses that dominate L1 instruction misses on today's most commonly used server hardware [23]; the instruction footprint of a transaction is too big to fit in the L1, thus thrashing the L1 and leading to very lengthy stalls.

There are proposals that address capacity instruction misses in OLTP. These proposals are motivated on the observation that threads executing transactions in parallel on a multicore server *execute significant amount of common code*. To be able to reuse the common instructions already brought into L1, STEPS [8] and STREX [2] time-multiplex a batch of threads on the same core, whereas SLICC [1] spreads the computation of a transaction to several cores to localize the common instructions to specific caches. Nevertheless, STREX and SLICC are completely oblivious to software and miss the opportunity to more precisely improve instruction locality through software guidance. STEPS, on the other hand, is a pure software technique designed to run only on a single-core and requires significant manually-aided instrumentation. Furthermore, all three techniques increase average transaction latency and STREX and STEPS increase the potential of deadlocks due to extensive batching and context-switching.

The goal of this work is to better exploit the L1 caches when running transactions based solely on hints from the software-side. The traditional way of scheduling transactions considers each as one big, monolithic task. Therefore,

Proceedings of the VLDB Endowment, Vol. 7, No. 14

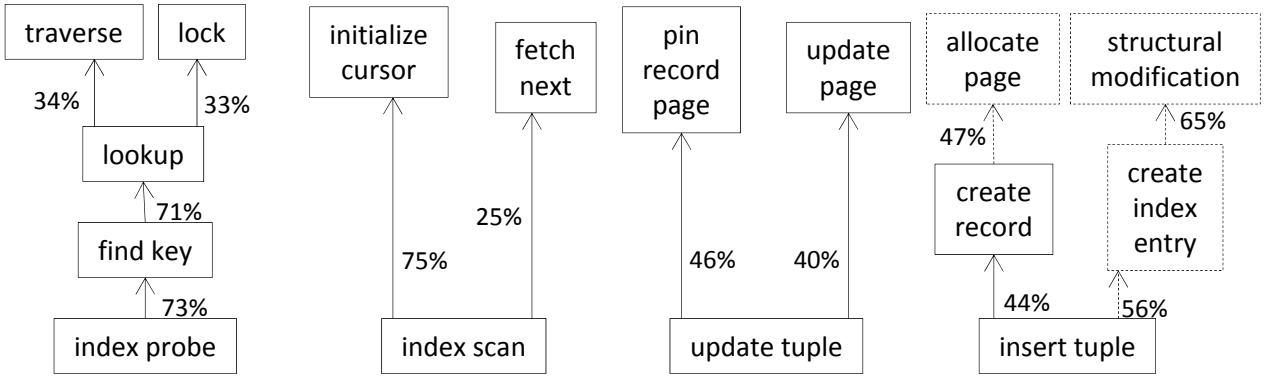

Figure 1: The flow graph of common database operations from the TPC-C transaction mix with the percentage of instruction footprints corresponding to each significant code part in these operations. An arrow from A to B with label $X\%$ means that $X\%$ of A's instruction footprint comes from executing B. The dashed lines indicate the code paths that are not always taken.

the granularity of tasks assigned to run on a core is too coarse, which leads to cache thrashing due to the large instruction footprint of the scheduled task. This work proposes to reduce the granularity of task-to-core assignment by scheduling the actions of common database operations. This approach bridges the gap between a transaction's instruction footprint and the L1 capacity.

To assign finer-grained tasks to cores while running transactions, we design ADDICT, a transaction scheduling mechanism that chases instruction cache locality. ADDICT first segments a database operation into smaller actions, where the instruction footprint of each action fits in a single L1 instruction cache. Then, it assigns specific cores for each of these actions and migrates the transactions over multiple cores using core assignment decisions that aim to maximize instruction locality for each action.

The contributions of this work are the following:

- We characterize the memory behavior of the TPC OLTP benchmarks [25] (TPC-B, TPC-C, TPC-E) during traditional transaction execution. We observe that same-type transactions exhibit 53% to 98% overlap in their instruction footprint while the data overlap is at most 6%.
- We design ADDICT, a transaction scheduling mechanism that views transactions as a composition of the actions from the database operations they execute. Therefore, it departs from the traditional way of scheduling transactions, which views them as an indivisible unit.
- Our experimental evaluation shows that ADDICT reduces L1 instruction cache misses by 85%, while also reducing the rate of the long-latency data misses by 20%. Even though ADDICT slightly increases L1 data cache misses and average transaction latency, the improved instruction locality leads to a 45% and 15% gains in total execution time on average on shallow and deep cache hierarchies, respectively.

The rest of the paper is organized as follows. Section 2 details typical database operations and presents the findings of our memory characterization study. Section 3 explains ADDICT's algorithm and its implementation. Section 4 evaluates our ADDICT prototype. Finally, Section 5 surveys related work and Section 6 concludes.

2. INSIDE TRANSACTIONS

Each transaction satisfies a different request in terms of its high-level functionality. However, underneath, transactions execute a series of actions from the same predefined set of database operations. These operations dictate the way of interaction with the storage manager components. This section first details some of the common database operations, and then, investigates the instruction and data overlap across their different instantiations in a workload mix.

2.1 Database Operations

Most transactional workloads have five major operations: index probe, index scan, update tuple, insert tuple, delete tuple. In the rest of this section we discuss their main characteristics; we omit delete tuple because of its similarity to insert tuple. To guide the discussion, Figure 1 sketches the high-level call flow for each operation including the percentage of the instruction footprint for each significant code path in it. In Figure 1, an arrow from box A to box B with label $X\%$ indicates that $X\%$ of the instruction footprint of A comes from executing the routine B. Solid arrows represent calls that are always made whereas dashed arrows represent calls that are not always made, i.e., they depend on a branch condition. The footprint is measured as the unique 64byte cache blocks requested by each operation when running 1000 transactions from the transaction mix of TPC-C (see Section 4.1 for the experimental setup).

Index Probe is the most common operation in transaction processing and is read-only. Its input parameters are an *index identifier* and a *key*. If the key exists in the index, index probe returns the tuple corresponding to the given key value in the index. Otherwise, index probe returns a flag indicating the key is not found. From Figure 1, we see that index probe follows a predictable call path. It starts with a call to the storage manager API, *find key*, which calls the *lookup* routine for the corresponding index. Then, it *traverse*s the index pages from top to bottom to find the desired key and interacts with the lock manager to acquire the *lock* for the record that maps the searched key.

Index Scan is the other read-only operation used in transactions. It takes as input an *index identifier*, two *key* values for the boundaries of the scan, and two *flag*s indicating the inclusiveness of the boundary keys. It returns

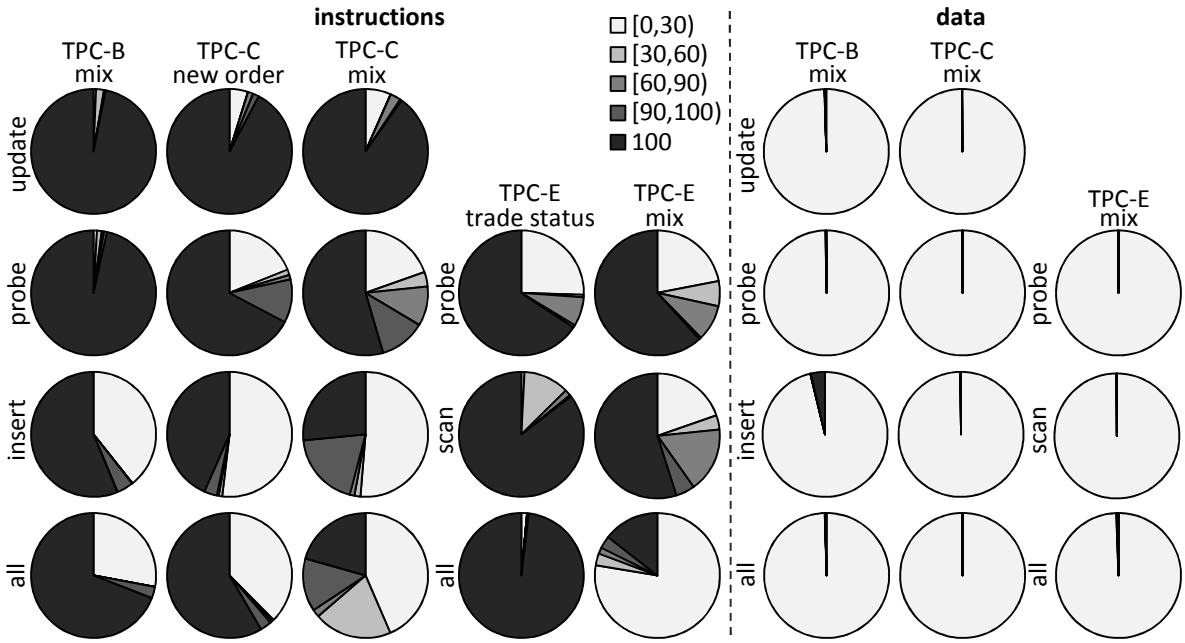

Figure 2: Overlaps in instruction and data footprints across different instantiations of the transactions in a workload mix, transactions of the same type, or database operations. Each pie represents the instruction or data footprint for the indicated transaction, database operation, or workload mix. The legend represents the frequency of an operation, transaction, or workload mix using the corresponding slice of the overall footprint. For example, the darkest slices (*100%*) represent the instructions and data that are executed in all instances, whereas the lightest slices (*[0-30)%*) represent the instructions and data that are common in less than 30% of the instances.

the set of tuples mapping to the key values within the given boundaries. As Figure 1 shows, index scan has two main parts. *Initialize cursor* first finds the position on the index leaf pages to be used as the starting point for the scan. This routine forms 75% of the instruction footprint of index scan. Then, *fetch next* fetches all the tuples until it reaches the scan's ending boundary. The instruction footprint of this last, tuple fetching code part is three times smaller compared to the instruction footprint of *initialize cursor*; *fetch next* has just a short loop that reads the tuples in sequence.

Update Tuple takes as input a *tuple identifier* and the *updated tuple*. Then, it rewrites the part of the data page in the database that corresponds to the tuple identifier. It is a relatively short operation and follows a more predictable execution compared to the other database operations. It has two major routines as shown in Figure 1: *pin record page* pins the page that has the tuple to be updated in the buffer pool, and *update page* updates the record and inserts a log entry for the update.

Insert Tuple takes a *table identifier* and a *tuple* as inputs. *Create record* adds the tuple to one of the data pages that belong to the given table and has sufficient space. *Create index entry* inserts the index entries for this record to all the indexes associated with the table. Figure 1 shows that these two routines almost equally contribute to the instruction footprint. Therefore, inserting a tuple to a table that has indexes, results in a significantly different instruction stream compared to inserting a tuple to a table with no indexes. Similarly, if none of the data pages allocated for the given table has space, then a new data page is created (*allocate page*). This process requires almost half of the instructions in *create record*. Further deviation in the instruction stream might be caused by the instructions needed to handle structural modifications in an index (e.g., index page splits, merges, or new index root creation). Such modifications form 65% of all the instructions needed to create an index entry. Overall, the insert tuple operation exhibits the most variety in its instruction flow compared to the other database operations.

2.2 Commonalities across Transactions

Considering the database operations transactions share, we expect to see significant overlap in the code executed by different transactions as well as some common data accesses. To quantify this intuition, we analyze the memory behavior of the transactions from the standard TPC [25] benchmarks for OLTP, i.e., TPC-B, TPC-C, and TPC-E (Section 4.1 details the experimental setup). We mark each instruction and data cache block accessed by each database operation or transaction. Then, we check how often these blocks appear in other instances of that database operation or transaction. Our goal is to examine instruction and data overlap at three different granularities: a) within the whole transaction mix, b) within each transaction of the same type, and c) in each database operation.

Figure 2 depicts the highlights of the overall analysis. Each pie-chart represents the whole instruction or data footprint for the indicated workload, transaction, or database operation called within that workload or transaction. Next, Section 2.2.1 and Section 2.2.2 detail the results in Figure 2 for instruction and data overlaps, respectively.

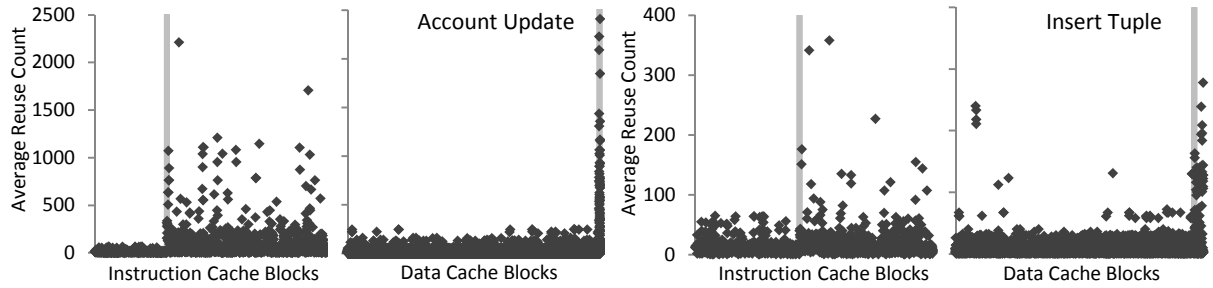

Figure 3: The average number of accesses to each memory address per instance of the TPC-B's `AccountUpdate` transaction and insert tuple operation. The x-axis places the addresses in the order of their commonality across different transaction instances in the workload. The addresses to the right of the vertical light-gray line are the ones that are used in all instances.

2.2.1 Instruction Overlap

The left-hand side of Figure 2 reports the instruction overlap results. For simplicity, Figure 2 only shows the most frequent operations invoked by the most frequent transaction type in the mix in addition to the results from the overall transaction mix for all the workloads. The pie slices group instructions based on the appearance frequency across the instances of each database operation or transaction. For example, the darkest slice (*100%*) represents the instructions that are executed in all instances, whereas the lightest slice (*[0-30)%*) represents the instructions that are common in less than 30% of the instances.

Since TPC-B has only one transaction in its mix, Figure 2 shows the results only for the workload mix (the leftmost pies). The instruction footprint overlap across all *probe* and *update* operation instances exceeds 90%. The overlap across all *insert* operation instances is at 60%. TPC-B has a single transaction type, `AccountUpdate`, that inserts a tuple to the `History` table, which has no index. Investigating where that 40% of uncommonly executed code comes from shows that these instructions come from the part of the *insert* operation code that creates a new data page. Even though there are only six `AccountUpdate` instances out of the 1000 that require this routine, the large instruction footprint of the routine (see Figure 1) causes a high deviation in the whole instruction stream.

The TPC-C charts show similar trends to TPC-B. Within individual transactions, e.g., `NewOrder`, the instruction overlaps in *probe* and *update* operations are high: at least 70% of the instructions accessed are the same. For the *insert* operation, however, around half of the instructions are not so common. `NewOrder` performs inserts to tables with indexes. This code has more branches compared to TPC-B's `AccountUpdate` since it also needs to execute the routine for creating an index entry (see Figure 1).

Since `NewOrder` forms almost half of the TPC-C transaction mix, the charts for each operation in the mix are similar to the charts for `NewOrder`. The slight differences are due to the different tables accessed by the transactions in the mix. For example, `Payment`, which together with `NewOrder` contributes to the 88% of the mix, inserts a tuple to a table with no indexes. Therefore, the instructions for creating an index entry are not common in the overall mix. Furthermore, the degree of overlap is lower in the whole transaction mix (third column, fourth row in Figure 2) compared to the individual operations. This is expected since *probe* is the only database operation code shared by all TPC-C transactions.

Since almost 80% of the TPC-E mix is read-only, Figure 2 presents the results for *probe* and *scan* for TPC-E. TPC-E has 10 transaction types in its mix, twice the number of TPC-C, and the most frequent transaction, `TradeStatus`, accounts for only 19% of the mix. Therefore, the instruction overlap is less in the overall TPC-E mix (fifth column, fourth row in Figure 2) compared to the other two benchmarks. However, among same-type transactions instruction overlap is still significant; different `TradeStatus` instances observe a 98% instruction overlap.

2.2.2 Data Overlap

The right-hand side of Figure 2 presents data overlap results for the transaction mixes only since the conclusions are the same for individual transaction types. Figure 2 clearly depicts that the overlap in data is very low, at most 6%.

The dataset used while collecting the traces is around 100GB for each workload. Therefore, there is almost no overlap on the data that represent database records or lower-levels of the indexes. On the other hand, investigating the sources of the few, very frequently used data shows that metadata information, lock manager, buffer pool structures, and index root pages are commonly accessed (mostly read) across different transactions. Such data mainly stem from the tables that are accessed in all the transactions of a workload's mix, e.g., the `Warehouse` table in TPC-C, or used by all the instances of a particular database operation, e.g., the inserts to `History` table in TPC-B.

2.3 Average Reuse in an Instance

Figure 2 demonstrates the frequency of instruction reuse across different instances of transactions or database operations. It does not indicate how frequently a memory address is reused within each instance. Therefore, we also measure the average per instruction and per data address accesses within one instance of each transaction and database operation. Figure 3 shows the results for the `AccountUpdate` transaction and the *insert tuple* operation in TPC-B. The results for TPC-C and TPC-E and the other database operations share similar trends.

Figure 3 omits the address labels on the x-axis, but places the addresses based on their frequency across different transaction instances (from left to right the frequency increases). The addresses on the right of the light-gray vertical line appear in all the instances. Figure 3 highlights that the frequently reused addresses across transaction and operation instances are also frequently reused within each instance.

2.4 Summary and Conclusions

The memory characterization study demonstrates that:

- Transactions exhibit high instruction overlap because of the common database operations they execute, especially among same-type transactions. This offers an opportunity to achieve better L1-I cache locality by scheduling transactions in a way that would enable instruction reuse across transactions based on their common actions.
- The percentage of the data that is common across transactions is very low due to infrequent reuse of the database tuples. The few frequently used data are small-sized and mostly read-only. Accordingly it may be possible to *pin* them in the caches to improve data cache locality.
- The cache blocks that are highly common across different transaction instances tend to be more frequently reused within each instance. Therefore, any technique for improving cache locality for the common instructions and data across different instances also has potential to improve cache locality within each transaction instance.

3. ADDICT

Section 2 emphasizes that transactions exhibit high instruction commonality whereas the data commonality is low. Based on this finding, we design an alternative method to schedule transactions to maximize instruction cache locality. ADDICT, an advanced instruction chasing mechanism for transactions, departs from the traditional way of scheduling transactions, which sees a transaction as one big task. ADDICT rather considers a transaction as a combination of the database operations it calls and migrates transactions over cores based on the actions their operations are about to execute.

ADDICT consists of two steps, which are detailed in the subsequent subsections. Step 1 determines the migration points in each database operation (Section 3.1) and Step 2 spreads the execution of a transaction over multiple cores based on the migration points picked in the previous step (Section 3.2). Step 2 is always dynamic since it orchestrates transaction execution during the actual run, whereas Step 1 can be either static or dynamic depending on the application's needs.

3.1 Finding Migration Points

To be able to determine when and where to move a transaction at run-time, ADDICT first needs to decide on the migration points in each database operation. ADDICT picks these points separately for each transaction type since the code paths each database operation takes might change based on the tables accessed in a particular transaction, as we observe in Section 2.2.1.

3.1.1 Algorithm

Algorithm 1 shows the details of ADDICT's initial step that finds the migration points for a workload. It takes a list of *indicators* to identify the transactions and database operations in the workload. These indicators can be function names or instruction addresses that correspond to the entry and exit points of the transactions or operations.

In lines 1-16 of Algorithm 1, ADDICT records the sequences of instructions that cause an eviction from each database operation invoked in a particular transaction type as migration point candidates. In parallel, it collects the occurrence count for each of these sequences. Since ADDICT aims to migrate transactions at the granularity of actions from database operations that can fit in an L1-I cache, it resets the L1-I cache upon transaction or database operation entry and exit points in this step. After collecting the candidates, ADDICT picks the most frequent sequence of instructions for each database operation from each transaction type as migration points (line 17 in Algorithm 1).

Algorithm 1 Finding migration points.
Inputs: list of transactions and database operations.
Output: a list of instruction sequences that indicate the migration points picked for each database operation invoked by each transaction.

```
1: m → keeps possible migration points
2: for instruction access addr in workload do
3:    if a transaction entry/exit then
4:       empty the L1-I cache
5:       if transaction entry then
6:          xct = current transaction type
7:    else if a database operation entry/exit then
8:       empty the L1-I cache
9:       if operation entry then
10:         op = current operation
11:         create empty sequence
12:      else
13:         m[xct][op][sequence] + +
14:   else if addr request requires an eviction then
15:      empty the L1-I cache
16:      sequence.append(addr)
17: return the sequence with the highest value for each
    m[xct][op]
```

3.1.2 Example

In line 17 of Algorithm 1, ADDICT has information similar to the following in map m:

```
1) xct1→insert→0x8b5f5f 0x899397→9
2) xct1→insert→0x9bd97f 0x8b5fbf 0x94ffde→1
3) xct2→probe→0x98560e 0x8d97bc→10
4) xct2→update→0x9557f0→5
```

ADDICT goes over this information to figure out the most frequent sequence of migration points. In this example, these are (1) for *insert* operation in xct1, (3) for *probe* operation in xct2, and (4) for *update* operation in xct2. Migration points in (2) represent a corner case in the *insert* tuple operation since they only appear once among all instances of xct1. For *probe* and *update* operations in xct2, however, there are no alternative migration points to the ones in (3) and (4). If there are multiple sequences of migration points that are the most frequent for an operation, ADDICT picks one of them randomly. However, we do not observe such cases for the workloads we evaluate in Section 4.

3.1.3 Implementation

There are several ways of deploying Algorithm 1 in practice. Adopting ADDICT as a pure dynamic approach requires integrating Algorithm 1 with the actual workload

run. ADDICT can perform this step as a part of the ramp-up time (a few seconds) without making any specialized scheduling decisions for transactions and then switch to migrating transactions based on the information collected in this step. On the other hand, Step 1 of ADDICT can be static and performed *a priori* as well. In this case, ADDICT would migrate transactions over the dedicated cores as soon as the real workload run starts.

In this step, ADDICT detects cache-sized chunks from each database operation. Therefore, given an empty L1-I cache, ADDICT should track the instructions that cause cache evictions within each database operation. To track such instructions at run-time, ADDICT can use either the hardware counters on the target hardware or mechanisms like informing memory operations [10]. Upon a transaction/operation entry/exit or eviction, ADDICT must flush the L1-I contents to reset the instruction cache and determine the next cache-sized code chunk in the current operation.

In addition, within the storage manager, there might be functions/routines, where one should avoid migrating at. For example, migrating within short-critical sections or lock acquisitions/releases would increase the duration of these routines. Therefore, Algorithm 1 can take additional input that indicates such functions and avoid picking migration points within these functions.

3.2 Migrating Transactions

After determining the migration points, ADDICT applies its scheduling principles during regular transaction execution. Since it picks the migration points separately for each transaction, it batches same-type transactions to maximize instruction cache locality. Furthermore, while processing a batch, ADDICT adjusts the core assignments based on the needs of the application, i.e., it assigns more cores to a migration point if it is more frequently used.

3.2.1 Algorithm

Algorithm 2 shows the core assignment and transaction migration principles of ADDICT's Step 2. Algorithm 2 takes as input the migration points found by Algorithm 1. It first assigns cores to each of the migration points (lines 1-14). Then, it migrates transactions based on the core assignments (lines 16-31).

Lines 1-14 of Algorithm 2 handle the core assignments on the target hardware. As in Algorithm 1, ADDICT considers each transaction separately. Therefore, each transaction takes $core_0$ as their entry core (lines 3-6). For the remaining core assignments, ADDICT incrementally assigns a unique core ID to each database operation in a transaction (lines 7-10) and its corresponding migration points (lines 11-14). Section 3.2.3 describes how ADDICT handles the cases where the number of migration points does not exactly match the number of available cores. Algorithm 2 omits these details for simplicity.

Lines 16-31 of Algorithm 2 perform the actual transaction execution. To maximize cache locality, in lines 16-17, same-type transactions from the list of client requests form a batch. The batch size is equal to the number of available cores on the current hardware to not to increase average transaction latency drastically. Then, for each instruction to be executed, ADDICT checks whether the transaction should migrate to another core based on the prior core assignment decisions (lines 20-26). If destination core ID has a different value than the current core ID of the transaction being executed, ADDICT migrates the transaction provided that there is an available destination core for the current migration point (lines 27-31).

To ensure the instruction stream is on a path that matches the migration points sequence in the input, ADDICT also tracks the previous migration addresses for each migration point. It migrates a transaction upon encountering a migration point only if that transaction has already executed the previous migration point in the sequence (line 25). An instruction address might be used several times during the execution of a database operation. However, it might lead to migration only if it is called through a specific path. Therefore, ADDICT must check for such order dependencies in the migration sequence.

Algorithm 2 Migrating transactions.
Input: migration points (output of Step 1) m.

```
 1: cores → keeps core assignments
 2: prev → keeps previous migration point
 3: for each transaction type xct in m do
 4:    core = 0, op = 0, prev = 0
 5:    addr = entry instruction for xct
 6:    cores[xct][op][addr] =< core, prev >
 7:    for each operation op in m[xct] do
 8:       core + +, prev = 0
 9:       addr = entry instruction for op
10:       cores[xct][op][addr] =< core, prev >
11:       for each migration address addr in m[xct][op] do
12:          core + +
13:          cores[xct][op][addr] =< core, prev >
14:          prev = addr
15:
16: for each transaction type xct in the list of requests do
17:    group num_cores transactions of type xct in batch
18:    for each core do m_xct = cores[xct], op = 0, prev = 0
19:    for each transaction t in batch do
20:       for each instruction access addr in t do
21:          core_dest = core_curr
22:          if addr is in m_xct then
23:             op = addr, prev = 0
24:          if addr is in m_xct[op] then
25:             if prev == m_xct[op][addr].prev then
26:                core_dest = m_xct[op][addr].core, prev = addr
27:          if core_dest! = core_curr then
28:             if core_dest is available then
29:                migrate t to core_dest
30:             else
31:                steal an idle core from another migration
                   point or wait in the work queue of core_dest
```

3.2.2 Example

Let's assume that Algorithm 2 takes as input the output of the example in Section 3.1.2. At line 15 of Algorithm 2, *cores* would have the assignments given below:

```
xct1→<core0,0>
xct1→insert→<core1,0>
xct1→insert→0x8b5f5f→<core2,0>
xct1→insert→0x899397→<core3,0x8b5f5f>
```

```
xct2→<core0,0>
xct2→probe→<core1,0>
xct2→probe→0x98560e→<core2,0>
xct2→probe→0x8d97bc→<core3,0x98560e>
xct2→update→<core4,0>
xct2→update→0x9557f0→<core5,0>
```

After deciding on the core assignments, ADDICT starts batching transactions. Let's assume that it initially batches requests of xct1 and one of the transactions in that batch has the following instruction sequence:

```
xct1_entry_instr ...  insert_entry_instr ...
0x899397 0x89939c 0x89939e ...  0x8b5f5f
0x8b5f62 ...  0x899397 ...
```

Upon xct1 and *insert* operation entry, ADDICT migrates the transaction to $core_0$ and $core_1$, respectively. When the instruction 0x899397 is accessed for the first time, since its previous migration point, 0x8b5f5f, is not yet encountered, ADDICT keeps the transaction on the same core. When the transaction uses the instruction 0x8b5f5f, since it is the first migration point in the *insert* operation, ADDICT migrates the transaction to $core_2$. When the instruction 0x899397 is reused, since it comes after a migration to $core_2$ due to 0x8b5f5f, ADDICT now migrates the transaction to $core_3$.

3.2.3 Load Balancing

Algorithm 2 presents a simplified version of the actual ADDICT algorithm as it just assigns one core per migration point. In a typical OLTP workload running on modern server hardware, there are (1) database operations that are more frequently used than others and (2) more or fewer cores than the number needed by a transaction. We describe how ADDICT deals with such cases below.

More migration points than cores: If the migration points for a transaction require more cores than what is available in the system, ADDICT starts ignoring the internal migration points in less frequent database operations starting from the last migration point. For example, in Section 3.2.2, if there were only four cores in the system, there would not be any cores assigned to 0x9557f0 in *update* and 0x8d97bc in *probe* for xct2. 0x9557f0 in *update* is ignored prior to 0x8d97bc in *probe* since *update* operation occurs less (5 vs. 10 in Section 3.1.2). 0x8d97bc in *probe* is ignored since there are no more internal migration points to ignore in *update*. In our experiments in Section 4, this situation arises for some TPC-C and TPC-E transactions.

If there are too few cores available for a workload, e.g., if number of cores is even less than the number of operations executed by a transaction type, ADDICT can either fallback to traditional scheduling or switch to a scheduling technique that optimizes instruction locality for a single-core [2, 8].

Fewer migration points than cores: When a transaction requires fewer cores than what is available on the machine, which is the common case in the era of multi-socket multicores, ADDICT distributes the remaining cores based on the frequency of operations. For example, in Section 3.2.2, if there were ten cores in the system, there would be two cores assigned to each migration point in *probe* operation since it is more frequent than *update*. The remaining one core would be given to the entry point of *update*.

In the case of having enough cores to assign to the migration points from multiple transactions, ADDICT can run multiple batches of transactions in parallel.

Dynamic reassignment of cores: After the initial core assignments, ADDICT deploys a dynamic approach. Whenever the destination core to be migrated to is not available, i.e., occupied by another transaction (line 31 of Algorithm 2), there are two options: (1) if there are any idle cores that belong to another migration point, ADDICT reassigns one of these idle cores to the current migration point, (2) if there are no idle cores, then the transaction waits in the work-queue of the destination core.

3.2.4 Implementation

We design ADDICT to be a software-guided hardware mechanism. We think of the migration points picked by the Step 1 of ADDICT as the software hints used by the Step 2 of ADDICT at the hardware side. Therefore, while Step 1 can use the already existing hardware features of modern hardware, Step 2 requires some additional features from the hardware side. These additional features stem from two things: (1) keeping track of the migration points and (2) performing fast and exact thread migrations.

To be able to decide on when and where to migrate a transaction, each core must keep the list of migration points for that transaction as well as an indicator for the current database operation and the previous migration point. ADDICT distinguishes both database operations and migration points using instructions addresses. Therefore, we can calculate ADDICT's space cost mainly based on the space cost of an instruction, which is 48bits on modern servers. Keeping the current database operation and the previous migration point would require 92bits per core. For each migration point, we need to map a <*database operation*, *migration point*> pair to a <*core id*, *previous migration point*>. In this mapping, except for the *core id* value, the rest of three values are instructions. We can keep the *core id*s as 8bit integers since 8bits already give us 256 distinct values. As a result, 152bits would be enough to keep a migration point. This way, a core can keep up to 50 migration points in less than 1KB space, which is a feasible space cost per core on most server hardware.

On the other hand, the hardware cost of the thread migrations is mainly algorithmic, which Atta et. al. [1] describe in detail and show that it is also feasible. We estimate the time required per thread migration to be ∼90 cycles; the cost of writing/reading a thread's state (e.g., the register values, last program counter, etc.) from/to the last-level-cache (∼6 cache lines).

Deploying ADDICT as a pure software mechanism would be less straightforward than our design. Dictating which transactions should run on which cores, is harder and less efficient at the software side. Modifying the context-switching code in the current platform in order to perform fast context-switches, like STEPS does [8], would help to some extent. However, this still does not guarantee that threads are going to migrate exactly to the cores ADDICT wants them to migrate. The functions that set a thread's core affinity (e.g., `pthread_setaffinity_np` in POSIX library) only work well provided that the destination core is idle. Otherwise, the OS scheduler schedules the thread to one of the underutilized cores automatically. To prevent such undesired migrations and cache thrashing, ADDICT requires a more drastic

Table 1: System Parameters.

Processing Cores	16 OoO cores, 2.5GHz 6-wide Fetch/Decode/Issue 128-entry ROB, 80-entry LSQ BTAC (4-way, 512-entry) TAGE (5-tables, 512-entry, 2K-bimod)
Private L1 Caches	32KB, 64B blocks, 8-way 3-cycle load-to-use, 32 MSHRs MESI-coherence for L1-D
L2 NUCA Cache	Shared, 1MB per core, 16-way 64B blocks, 16 banks 16-cycle hit latency, 64 MSHRs
Interconnect	2D Torus, 1-cycle hop latency
Memory	DDR3 1.6GHz, 800MHz Bus, 42ns latency 2 Channels / 1 Rank / 8 Banks 8B Bus Width, Open Page Policy
Latencies	CAS(10), RCD(10), RAS(35) RC(47.5), WR(15), WTR(7.5) RTRS(1), CCD(4), CWD(9.5)

design change at the software-side if a software-only design is more desirable. Deploying an execution model similar to staged databases [9] and assigning stages to each database operation would allow us to pin each stage to a core, send requests to each stage's work queue, and give ADDICT more control over the core affinities.

3.2.5 Effect on Database Components

Under ADDICT, a transaction goes through the same database components as it does under traditional scheduling. ADDICT only involves multiple cores in the execution of a transaction. However, it does not change what a transaction executes. Therefore, ADDICT's migrations have no effect on ACID properties, concurrency control mechanisms, or the logging subsystem. In addition, since ADDICT does not batch more transactions than the number of available cores in the system, it does not change the data contention patterns.

For the cases outside the regular workload run, such as recovery or database population, ADDICT can either fallback to traditional scheduling or find new migration points for the specific operations or routines executed during such periods of execution.

4. EVALUATION

This section demonstrates: (1) the stability of the migration points ADDICT picks across different number of transaction instances (Section 4.2), (2) ADDICT's effect on instruction and data misses at different levels of the memory hierarchy (Section 4.3), (3) ADDICT's impact on performance (Section 4.4), (4) the effect of changing server load on ADDICT's performance (Section 4.5), (5) ADDICT's effectiveness under deeper cache hierarchies (Section 4.6), (6) ADDICT's impact on power (Section 4.7), and (7) ADDICT's overheads (Section 4.8).

4.1 Setup and Methodology

Since ADDICT is a software-guided hardware mechanism (Section 3.2.4), the evaluation uses full timing simulation. We collect x86 execution traces from transactions using Pin [16]. We replay these traces on the Zesto x86 multicore architecture simulator [15], modeling the timing of all events. Table 1 details the hardware parameters in our simulation.

The traces are extracted from three standard transaction processing benchmarks [25]; TPC-B, TPC-C, and TPC-E, while running their workload mix after a warm-up period on the Shore-MT storage manager [12, 20]. Scaling factors are set big enough to have a 100GB dataset right after database population, and the buffer-pool is configured to keep the whole database in memory. To run the most scalable configuration for all the benchmarks, we enable all the logging [13] and locking [11] optimizations of Shore-MT. Since we simulate 16 cores, there are 16 worker threads executing transactions during the trace collection.

We collect 11000 transaction traces for each workload. The initial step of ADDICT (Algorithm 1) uses the first 1000 of the traces (from 1 to 1000) to determine the migration points. Section 4.2 uses all the traces after the first 1000 (from 1001 to 11000), whereas the rest of the sections use the next batch of 1000 traces (from 1001 to 2000) while evaluating the different scheduling mechanisms.

We compare *ADDICT* against three transaction scheduling mechanisms: (1) *Baseline*, the traditional transaction scheduling, where each transaction starts and finishes its execution on one core provided that no context-switching occurs due to I/O, waiting for lock, etc., (2) *STREX* [2], which time-multiplexes a batch of transactions on the same core to enable instruction reuse among the transactions in the batch, (3) *SLICC* [1], which spreads the computation of transactions over several cores to localize common instructions to caches without any software hints. We implement all four scheduling mechanisms on the Zesto simulator. Except for *Baseline*, all the mechanisms rely on batching same-type transactions. *ADDICT* picks a batch size that is equal to the number of available cores by default. Therefore, except for Section 4.5, the batch size is 16 in our experiments.

4.2 Migration Points

As Section 3.1 describes, *ADDICT* picks the most common migration point sequences among all the possible migration points for a transaction type. In our experimental evaluation, *ADDICT* determines the migration points based on a run with 1000 transaction traces (Section 4.1). This section investigates the *stability* of these migration points across all the instances of a transaction. It also shows how stability changes as we drastically increase the total number of transaction instances. A transaction instance has *stable* migration points if *ADDICT*'s core migration selection algorithm, when ran directly on this transaction instance alone, picks migration points that match the migration points chosen by *ADDICT* during the initial profiling phase of the 1000 transaction instances. Figure 4 shows the results for TPC-B's `AccountUpdate` and TPC-C's `NewOrder` and `Payment` transactions. The results are very similar for the other transaction types.

Except for the *insert tuple* operation in TPC-C, the migration points *ADDICT* determines for each database operation is stable in at least 90% of all the transactions. As Section 2.1 notes, *insert tuple* is the operation that has the most variety in its instruction stream across different instantiations. Therefore, it is expected that even the most frequent migration sequence for *insert* does not satisfy almost

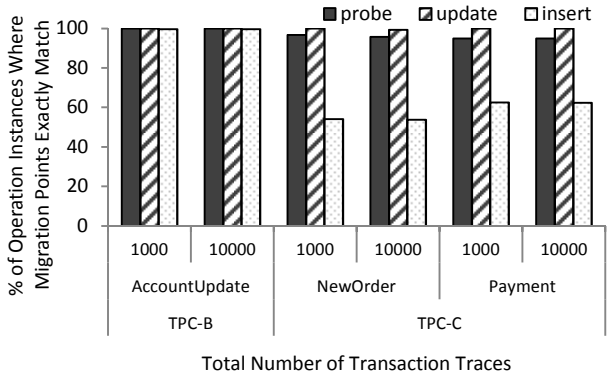

Figure 4: Percentage of database operation instances where the migration points picked by *ADDICT* have an exact match as we increase the number of transaction instances.

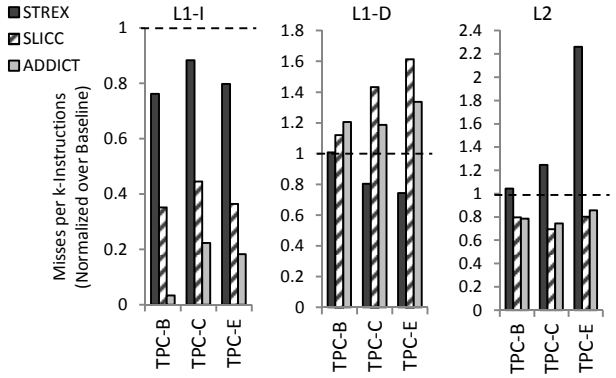

Figure 5: *ADDICT*'s impact on instruction and data misses. Y-axes show the number of misses per 1000 instructions normalized over *Baseline* (=1 on Y-axis).

half of the instances for some transaction types.

Furthermore, Figure 4 shows that the percentage of stability of the migration points stays the same when we move from 1000 to 10000 traces. This demonstrates that the 1000 transaction traces is sufficient enough to capture the differences across multiple instantiations of a transaction type for the workloads we evaluate. Therefore, the rest of the experiments in this section use 1000 transaction traces that is different from the 1000 transaction traces used for determining the migration points (see Section 4.1).

4.3 Instruction and Data Misses

This section quantifies *ADDICT*'s impact on the instruction and data misses at the various cache hierarchy levels. More specifically, this section measures the number of instruction and data misses per 1000 instructions (MPKI) at the L1-I, L1-D, and L2 caches as we run the workloads with different scheduling techniques. Figure 5 reports the MPKI values for *ADDICT*, *STREX*, and *SLICC* normalized over the MPKI values from the *Baseline*.

L1-I: As Figure 5 illustrates, all scheduling mechanisms reduce the L1-I misses. However, *ADDICT* is more effective in reducing the instruction misses compared to the other hardware-techniques. Specifically, *ADDICT* reduces instruction misses by 85% on average over *Baseline* whereas the reduction is 20% and 60% with *STREX* and *SLICC*, respectively. *ADDICT* makes more precise scheduling decisions while chasing instruction locality for transactions because of the software-guidance.

TPC-B benefits the most from *ADDICT* since its transaction mix has only one transaction type. The migration points picked for TPC-B are suitable for all transactions. Therefore, after the initial set of transactions the instructions are spread over the various instruction caches and remain mostly resident for all other transactions. For TPC-C and TPC-E, however, if the new batch of transactions is of different type than the ones in the previous batch, the non-overlapping instruction footprint must be first loaded in the instruction caches by the first few transactions.

L1-D: The L1-D MPKI results in Figure 5 show that the techniques that are based on computation spreading, *SLICC* and *ADDICT*, hinder data locality. When a transaction migrates from one core to another, it leaves its data behind. Therefore, *SLICC* and *ADDICT* increase data misses by 40% and 25% on average over the *Baseline*, respectively. *STREX*, on the other hand, leads to constructive data sharing for the few overlapped read-only data cache blocks (see Section 2.2.2).

Recent studies show that the data misses OLTP suffers the most from are the long-latency data misses from the last-level cache [23]. These misses result to off-chip accesses that require a trip to main-memory. Modern out-of-order (OoO) processor cores are capable of hiding the latency of a few additional L1 data misses that end up being serviced by the on-chip memory hierarchy. Moreover, it is harder to overlap L1 instruction miss stalls compared to L1 data misses on a modern superscalar OoO processor, like the one we model (Section 4.1). Therefore, the slight increase in L1-D MPKI does not overweigh the benefits of reducing the L1-I MPKI as long as we avoid increasing the misses from the last-level cache. Section 4.4 supports this claim.

L2: *ADDICT* and *SLICC* both reduce the L2 MPKI by ∼20% whereas *STREX* increases it by 50% on average. Due to batching transactions on one core, *STREX* runs more transactions concurrently, which increases the stress on the requests to the last-level cache. However, *STREX* still improves the performance as Section 4.4 shows emphasizing the importance of reducing the instruction misses. On the other hand, since all the techniques batch the same type of transactions, they access the same tables concurrently. Therefore, the reduction in L2 MPKI for *ADDICT* and *SLICC* stems from the constructive sharing of the read-only metadata information and higher-levels of the B-tree indexes for the same tables.

4.4 Performance Impact

This section measures how performance varies with *ADDICT*. It uses two performance metrics: (1) total execution time to complete all traces and (2) average time to complete a single transaction. Figure 6 presents the results.

Total execution cycles: Figure 6 shows that *ADDICT* reduces the total execution time by 45% over the *Baseline*. *ADDICT* is better than *STREX* and *SLICC*, which respectively improve performance by 17% and 35% on average over the *Baseline*. *ADDICT* manages to better utilize the instruction caches boosting instruction cache locality (see Figure 5).

Latency: While *STREX*, *SLICC*, and *ADDICT* reduce

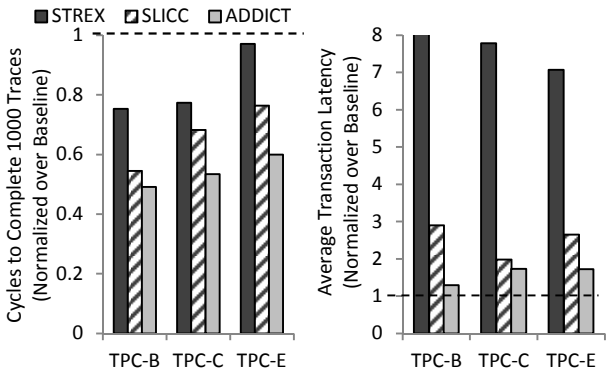

Figure 6: Impact of different scheduling techniques on performance; total execution cycles (left-hand side) and average transaction latency (right-hand side). Y-axes are normalized over *Baseline* (=1 on Y-axis).

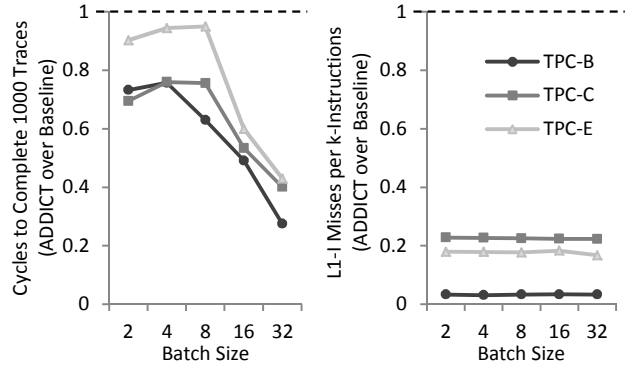

Figure 7: Impact of changing server load (or batch size) on *ADDICT*; total execution cycles (left-hand side) and instruction cache misses (right-hand side). Y-axes are normalized over *Baseline* (=1 on Y-axis).

the total execution time and improve throughput, they all depend on transaction batching. As a result they increase the average transaction latency in all the workloads. However, *ADDICT* exhibits the lowest transaction latency overhead compared to *STREX* and *SLICC* increasing average transaction latency by 60% over the *Baseline*, whereas the latency increase is 7-8X by *STREX* since it overloads cores with multiple transactions.

4.5 Effect of Changing Loads

By default, *ADDICT* picks a batch size that is equal to the number of available cores in the system. This section investigates *ADDICT*'s behavior under different batch sizes, in parallel observing the effect of changing server load on *ADDICT*. Figure 7 reports how well *ADDICT* reduces the total execution cycles and L1-I misses as a function of batch size, i.e., the number of concurrent transactions in the system, from two (lightly-loaded system) to 32 (heavily-loaded system).

Figure 7 shows that while the reduction in L1-I MPKI remains the same the total execution time improves for larger batch sizes. This is expected since the transactions from the previous batch might prefetch the instructions needed for current batch. Therefore, *ADDICT*'s effect on L1-I MPKI

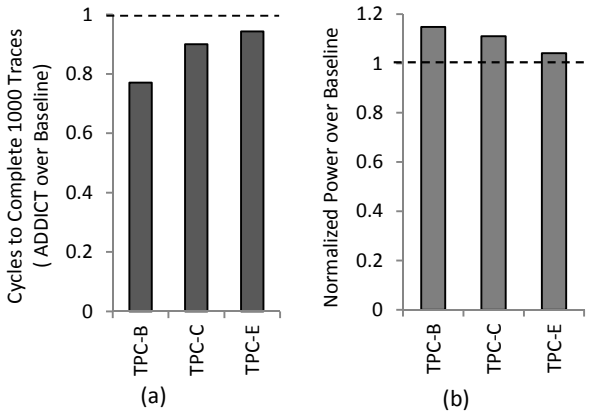

Figure 8: Impact of deeper cache hierarchies on *ADDICT* (a) and *ADDICT*'s impact on power (b). Y-axes plot normalized *ADDICT* values over *Baseline* (=1 on Y-axis).

does not change as we increase the batch size. On the other hand, as we increase the batch size more transactions exploit the improved L1-I locality at a time. As a result, the reduction in the total execution time increases starting from a batch size of 8.

4.6 On Deeper Memory Hierarchies

This section considers a deeper memory hierarchy, which is representative of certain popular modern chip multiprocessors. Specifically, the experiments of this section introduce an additional 256KB per core L2 cache with 7 cycles of access latency. The previously considered shared L2 now appears as a shared L3 and the L1 caches remain the same. Figure 8(a) shows the total execution cycles for *ADDICT* normalized over the *Baseline*.

The reduction in L1-I MPKI and LLC(=L3) MPKI are similar to those for the shallower memory hierarchy (Figure 5). *ADDICT* remains effective at improving overall performance. As expected the overall performance improvements are lower compared to the shallower memory hierarchy since now the penalty for an L1-I cache miss is lower; the new L2 cache now handles some of the instruction cache misses. Considering that Shore-MT has an instruction footprint of 128KB-256KB, most L1-I misses will now be served by the 256KB L2 cache, which effectively keeps the whole instruction footprint. However, the instruction footprint for commercial database management systems would be higher than the instruction footprint of Shore-MT.

4.7 Impact on Power

We also measure *ADDICT*'s impact on power using McPAT [14]. Figure 8(b) shows the average per core power required by *ADDICT* normalized over the numbers for *Baseline*. From Figure 8(b), we see that *ADDICT* requires around 10% more power than *Baseline*.

4.8 Overheads

All three hardware mechanisms we evaluate have one major run-time overhead. They require frequent context-switches either due to time-multiplexing transactions on a single core (*STREX*) or thread migrations across multiple cores (*SLICC* and *ADDICT*). Figure 9 compares the three mechanisms in terms of this overhead. More specifically, we first measure

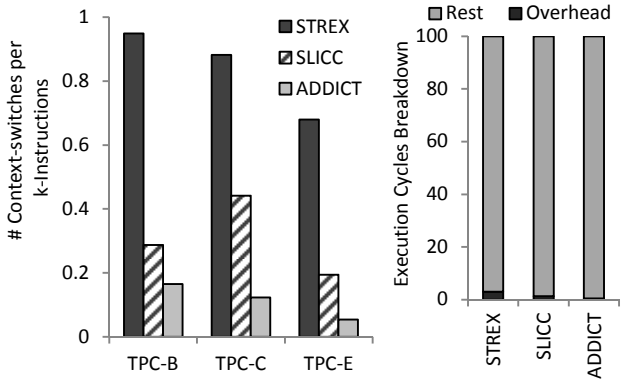

Figure 9: Number of context-switches/thread migrations per 1000 instructions (left-hand side) and execution cycles breakdown (right-hand side).

the number of times they context-switch transactions per 1000 instructions. Then, we report the contribution of this overhead on total execution cycles.

As the graph on the left-hand side of Figure 9 shows, *ADDICT* achieves its better performance through fewer migrations compared to both *STREX* and *SLICC*; 85% and 60%, respectively. Therefore, its run-time overhead due to context-switches is lower compared to the other two mechanisms. Nevertheless, the graph on the right-hand side of Figure 9 shows the execution cycles breakdown averaging the results for all the workload runs. It demonstrates that none of the mechanisms suffer due to the additional context-switches they incur. Even in the case of *STREX*, only 3% of the overall cycles go to these context-switches (labeled *Overhead*).

4.9 Summary

The evaluation shows that ADDICT is able to make effective decisions on the migration points for a variety of transaction types. As a result, it significantly reduces the instruction misses since it optimizes transaction scheduling to maximize instruction locality. ADDICT encounters 85% fewer instruction misses for typical OLTP benchmarks compared to traditional scheduling. As a result, it reduces the total execution time by 45% under shallower cache hierarchies and 15% under deeper cache hierarchies. In addition, it incurs lower run-time cost and performs better than the current state-of-the-art hardware scheduling mechanisms for transactions (e.g., STREX [2] and SLICC [1]).

5. RELATED WORK

The related work consists of two parts: (1) memory behavior characterization for OLTP workloads and (2) improving instruction locality in L1-I.

5.1 Characterization of OLTP Workloads

Previous workload characterization studies performed on traditional data management systems investigate OLTP workloads at the micro-architectural level [5, 19, 22, 24]. They all highlight that OLTP exploits aggressive micro-architectural features very poorly; i.e., spends most of the execution cycles on memory, especially instruction, stalls and exhibits low IPC. However, all of these studies consider the data management system as a black-box. None of them maps the sources of the hardware underutilization to the components of a typical data management system.

On the other hand, Wenisch et. al. [26] attribute the temporal streams in data cache misses to the application components such as various kernel activities, SQL interpreter, storage manager, etc. Tözün et. al. [23] go one step further and focus only on the storage manager. They map both the data and instruction misses coming from the different levels of the cache hierarchy of a modern commodity server to storage manager components and database operations. Our analysis is complementary to these works since we investigate the sources of memory access overlaps not cache misses, within transactions and database operations.

Finally, Atta et. al. [1, 2] briefly quantify the instruction overlaps in TPC-C and TPC-E transactions. We expand their study by exploring more transaction types, studying the overlaps for both data and instructions, and looking at the overlaps at the granularity of databases operations.

5.2 Improving Instruction Cache Locality

There is a large body of work on reducing instruction stalls through improving instruction cache locality. Here we survey the ones that target OLTP workloads specifically.

Smart static or dynamic compilation techniques [18] can optimize the code layout to mainly minimize the conflict misses. However, as Tözün et. al. [23] show, even if we minimize the conflict misses with any code optimization technique, there is a significant amount of capacity misses that we have to reduce for more efficient OLTP execution.

On the other hand, instruction prefetching proposals designed for OLTP-like applications has emerged from simple stream buffers [19] to highly sophisticated stream predictors [6] that trade simplicity for accuracy. For example, PIF [6] requires ∼40KB of extra storage per core. Therefore, modern commodity servers still prefer the low-cost next-line prefetcher, which sequentially fetches the memory addresses, for L1-I. Nevertheless, both the code optimization and instruction prefetching techniques are orthogonal to ADDICT and can be combined with it.

In addition to these techniques, there is a line of recent work that aim to improve instruction locality through exploiting the code commonality among concurrent transactions. These span proposals from batching transactions and time-multiplexing their execution on one core, STEPS [8] and STREX [2], to spreading the computation of transactions across multiple cores, computation spreading [3] and SLICC [1]. Similarly to ADDICT, they all rely on the initial/leader thread to miss the instructions it needs as it would during traditional transaction execution, and the rest of the threads to reuse the instructions already brought into cache(s) by the initial thread. However, except for STEPS, they are all oblivious to software. They cannot prevent migrations or context switches during lock acquisitions or releases. In addition, even though their hardware costs are low, ADDICT minimizes the space and functionality required by these two pure hardware techniques since it determines its migration decisions through software hints. On the other hand, STEPS is unable to exploit multicore hardware and requires cumbersome modifications to be able to perform fast context switching at the software.

ADDICT aims to achieve the best of both SLICC and STEPS; spread the computation of a transaction over multiple cores to enable an ample cache capacity for instructions

and get the insights for when and where to migrate from the software-side to better localize the instructions in L1-I. In parallel, ADDICT attempts to reduce the migration costs and the transaction latency incurred by the two techniques.

6. CONCLUSIONS

L1 instruction miss stalls are among the main causes of the hardware underutilization when running transaction processing applications on today's hardware. To overcome this problem, we design ADDICT. ADDICT assigns cores to the actions of each database operation in each transaction at a granularity that matches the size of the L1 instruction cache being used. It dynamically spreads the execution of transactions over multiple cores based on the core assignments to maximize the locality for instructions. Our evaluation shows that ADDICT's efforts in improving the instruction cache locality offers great potential in terms of performance and hardware utilization because of the high reuse frequency of instructions both within one and across different transactions and database operations.

We envision ADDICT as a task scheduler on emerging heterogeneous many-core processors where cores are specialized for various database functionalities. In such a setting, ADDICT can also guide developers while making decisions about which granularity each database operations should be specialized at. Finally, in addition to OLTP workloads, ADDICT can benefit any application that suffers from instruction stalls and have concurrent requests executing a series of actions from a predefined set.

7. REFERENCES

[1] I. Atta, P. Tözün, A. Ailamaki, and A. Moshovos. SLICC: Self-Assembly of Instruction Cache Collectives for OLTP Workloads. In *MICRO*, pages 188–198, 2012.

[2] I. Atta, P. Tözün, X. Tong, A. Ailamaki, and A. Moshovos. STREX: Boosting Instruction Cache Reuse in OLTP Workloads through Stratified Transaction Execution. In *ISCA*, pages 273–284, 2013.

[3] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-Fly. In *ASPLOS*, pages 283–292, 2006.

[4] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal Prefetching B+-Trees: Optimizing Both Cache and Disk Performance. In *SIGMOD*, pages 157–168, 2002.

[5] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *ASPLOS*, pages 37–48, 2012.

[6] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive Instruction Fetch. In *MICRO*, pages 152–162, 2011.

[7] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey. Cache-Conscious Frequent Pattern Mining on Modern and Emerging Processors. *The VLDB Journal*, 16(1):77–96, 2007.

[8] S. Harizopoulos and A. Ailamaki. Improving Instruction Cache Performance in OLTP. *ACM TODS*, 31(3):887–920, 2006.

[9] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD*, pages 383–394, 2005.

[10] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing Memory Operations: Memory Performance Feedback Mechanisms and Their Applications. *ACM TOCS*, 16(2):170–205, 1998.

[11] R. Johnson, I. Pandis, and A. Ailamaki. Improving OLTP Scalability Using Speculative Lock Inheritance. *PVLDB*, 2(1):479–489, 2009.

[12] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *EDBT*, pages 24–35, 2009.

[13] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: A Scalable Approach to Logging. *PVLDB*, 3:681–692, 2010.

[14] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO*, pages 469–480, 2009.

[15] G. H. Loh, S. Subramaniam, and Y. Xie. Zesto: A Cycle-Level Simulator for Highly Detailed Microarchitecture Exploration. In *ISPASS*, pages 53–64, 2009.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, pages 190–200, 2005.

[17] D. Porobic, E. Liarou, P. Tözün, and A. Ailamaki. ATraPos: Adaptive Transaction Processing on Hardware Islands. In *ICDE*, pages 688–699, 2014.

[18] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, and M. Valero. Code Layout Optimizations for Transaction Processing Workloads. In *ISCA*, pages 155–164, 2001.

[19] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *ASPLOS*, pages 307–318, 1998.

[20] Shore-MT Official Website. http://diaswww.epfl.ch/shore-mt/.

[21] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-Temporal Memory Streaming. In *ISCA*, pages 69–80, 2009.

[22] R. Stets, K. Gharachorloo, and L. Barroso. A Detailed Comparison of Two Transaction Processing Workloads. In *WWC*, pages 37–48, 2002.

[23] P. Tözün, B. Gold, and A. Ailamaki. OLTP in Wonderland – Where do cache misses come from in major OLTP components? In *DaMoN*, pages 8:1–8:6, 2013.

[24] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From A to E: Analyzing TPC's OLTP Benchmarks – The obsolete, the ubiquitous, the unexplored. In *EDBT*, pages 17–28, 2013.

[25] Transcation Processing Performance Council (TPC). http://www.tpc.org.

[26] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal Streams in Commercial Server Applications. In *IISWC*, pages 99–108, 2008.