A Key-based Adaptive Transactional Memory Executor[*]

Tongxin Bai   Xipeng Shen[‡]   Chengliang Zhang
William N. Scherer III [†]   Chen Ding   Michael L. Scott

Computer Science Department, University of Rochester
[‡]Computer Science Department, The College of William and Mary
[†]Computer Science Department, Rice University

{bai,zhangchl,cding,scott}@cs.rochester.edu
xshen@cs.wm.edu
bill.scherer@cs.rice.edu

Abstract

Software transactional memory systems enable a programmer to easily write concurrent data structures such as lists, trees, hashtables, and graphs, where non-conflicting operations proceed in parallel. Many of these structures take the abstract form of a dictionary, *in which each transaction is associated with a search key. By regrouping transactions based on their keys, one may improve locality and reduce conflicts among parallel transactions.*

In this paper, we present an executor that partitions transactions among available processors. Our key-based adaptive partitioning monitors incoming transactions, estimates the probability distribution of their keys, and adaptively determines the (usually nonuniform) partitions. By comparing the adaptive partitioning with uniform partitioning and round-robin key-less partitioning on a 16-processor SunFire 6800 machine, we demonstrate that key-based adaptive partitioning significantly improves the throughput of fine-grained parallel operations on concurrent data structures.

1   Introduction

Task-level parallelism is becoming increasingly important for general-purpose programs as modern desktop and server machines are equipped with multicore and multithreaded processors. Though essential to increased throughput on such machines, concurrent programs are notoriously difficult to write, from both performance and correctness perspectives. One performance issue with highly concurrent multithreaded programs is the overhead of thread creation and destruction. This overhead becomes particularly apparent when a server program is written in a thread-per-request model with a large number of very short requests. The overhead problem may be addressed by arranging for a pool of threads to share a "bag of tasks," as implemented, for example, by the Java 5 `java.util.concurrent Executor` framework.

Although a thread pool helps to efficiently execute short concurrent tasks, it shares with the thread-per-request model the usual pitfalls of lock-based synchronization, including deadlock, priority inversion, and "wrong lock" bugs. *Transactional memory* [9] addresses these pitfalls by allowing programmers simply to label blocks of code *atomic*, and to count on the underlying system to execute them in parallel whenever possible.

For applications consisting of many small tasks, it seems natural to combine executors and transactional memory. The former minimizes the overhead of threads; the latter ensures atomicity, consistency, and mutual isolation. In this paper, we present a *key-based* transactional memory executor that takes soft-

ware transactions as inputs and schedules them across a pool of worker threads. The main contribution lies in the scheduling policy, which uses search keys or other transaction parameters to increase memory locality, reduce transaction conflicts, and adaptively balance load. We focus in particular on *dictionary* operations, in which the likelihood that two transactions will access similar data correlates with the numerical proximity of keys.

To improve temporal and spatial locality, we can execute transactions with similar keys on the same processor, so successive transactions find data in the cache, reducing access latency and load on the processor-memory interconnect. On a machine with multicore processors and shared chip-level caches, locality among transactions that run on the same chip (if not on the same core) may still improve performance by reducing the number of loads that go to memory.

Transaction conflicts happen when concurrent transactions violate Bernstein's condition [1, 2]: they access the same data, and at least one them is a writer. To reduce such conflicts, we can avoid concurrent execution of transactions with similar keys by scheduling them to the same work thread. Fewer conflicts mean less time wasted on aborted transactions, conflict detection and contention management.

To balance load among worker threads, we first dynamically sample keys of transactions. Then we estimate the distribution of keys and adaptively partition keys across the available worker threads.

The remainder of this paper is organized as follows: Section 2 describes our executor model. Section 3 describes the adaptive partitioning of keys. Section 4.1 discusses our implementation. We empirically validate the feasibility and benefits of key-based adaptation in Section 4.4. For lack of space we present only one of the tests and leave the rest in a technical report [**?**]. We discuss related work in Section 5 and conclude in Section 6.

2 Executor Models

In our model of executors (Figure 1), a *producer* thread generates transactions. A *worker* thread executes transactions, which it retrieves from a private buffer. The *executor*, if present, is called by the producer to choose the worker that should execute a transaction. By sorting the entries in the worker's buffer, the executor could also control the order in which the worker will execute waiting transactions, though we do not use this capability in our experiments.

No executor. In Figure 1 (a), each thread is both a producer and a worker. It generates and executes

(a) no executor

(b) single executor

(c) multiple executors
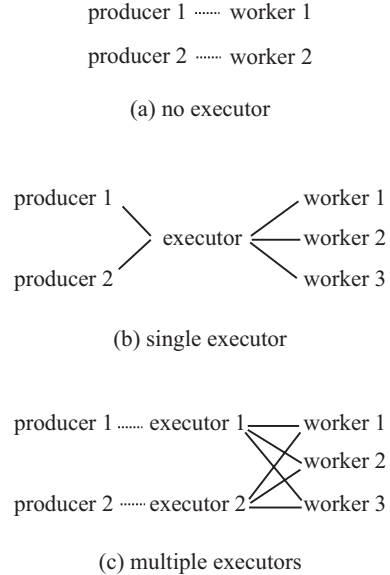
Figure 1. Executor models.

transactions one at a time. The dotted lines in the Figure indicate intra-thread data passing. The synchronous execution is both a feature—there is no queuing overhead—and a problem—neither load balancing nor parallel execution of producers and workers is possible. Parallelism can be increased, of course, by increasing the number of producers, but since the producer is part of the user application, this may or may not be easy. For a fixed number of producers, performance cannot scale beyond the number of threads, even though the machine may have more processors available.

Centralized executor. An executor, implemented as a separate thread, adds overhead for transaction queuing and dispatching but exploits the potential concurrency between the producer and the worker, and admits the possibility of load balancing among workers, as shown in Figure 1 (b). A set of producer threads can be served by an arbitrary number of worker threads. In fact, the executor can change the number of worker threads in response to any change in available resources. The executor also enables policy control by, for example, giving priorities to different types of transactions. More importantly for this work, the executor can reorder transactions to improve throughput.

Parallel executors. A single executor may be a scalability bottleneck. Parallel executors, as shown in Figure 1 (c), take tasks generated by producers and dispatch them to workers in parallel. If the number of executors is the same as the number of producers, the

executor can be a part of the producer thread, so the data passing becomes thread local, as shown by the dotted lines in the Figure.

Load balancing. For maximum speedup, load on the worker threads should be balanced. Many factors affect the speed of a worker thread, including the size of the transactions, their temporal and spatial locality with respect to other transactions in the same worker, and the chance of conflicts with transactions in other workers. Many of these factors have been systematically studied for parallel tasks other than STM transactions. A common solution is *work stealing* [3], where a worker thread with an empty queue grabs work from other queues. An alternative method is for the executor to dynamically change the distribution of transactions.

For concurrent data structure updates, the size of transactions may not vary much. In such a case, an executor can control load balance by assigning the same number of transactions to each worker thread. When work is plentiful, the number transactions in each queue can be expected to be roughly equal. For transactions that have more complex sizes, some type of work stealing may be needed, although the overhead of the executor scheme may increase because the work queue is more complex to manage.

3   Key-based Scheduling

In preceding Sections we have implied that transactions on a dictionary data structure should be reordered based on dictionary keys. In actuality, it is valuable to distinguish between dictionary keys and "transaction keys", which the executor uses for scheduling. Transaction keys are a more general concept, applicable to a wider variety of workloads. By creating a separate notion of transaction key we arrive at a two-phase scheduling scheme. The first phase maps transaction inputs, whatever they may be, into a linear key space. The second phase dispatches transactions based on their location in this space.

3.1   Mapping Transactions to Key Space

The central requirement for keys is that numerical proximity should correlate strongly (though not necessarily precisely) with data locality (and thus likelihood of conflict) among the corresponding transactions. One possible way to find a good mapping from inputs to keys is to profile an application on a representative set of inputs. Alternatively, users might provide appropriate mapping functions for each application. Given a priori knowledge, such manual specification may be both easier and more accurate than profiling. We use manual specification in our experiments.

In generating keys, we have two conflicting goals. On the one hand, we want our keys to be *accurate*: transactions with similar data access patterns should have similar keys. On the other hand, key generation should be *efficient*. Perfect accuracy could be obtained by fully executing a transaction and generating a Gödel number that encodes its data accesses, but this would obviously defeat the purpose of scheduling altogether! Instead, we seek the best estimate of a transaction's accesses that we can obtain without incurring unreasonable overhead.

Some mappings may be trivial. In a stack, for example, we are more interested in the fact that every push and pop will begin by accessing the top-of-stack element than in the exact address of this element. So for this simple case, the hint we provide to the scheduler (the key) is constant for every transactional access to the same stack. This allows the executor to recognize that transactions will race for the same data and schedule them appropriately. This example may seem uninteresting, but it demonstrates how summaries of transactional data access patterns can be meaningfully used by a scheduler that requires no other information about the contents of transactions.

For the benchmark results shown in this paper, we have manually coded functions that generate keys for transactions. Although one would obviously prefer to generate them automatically, we focus for now on demonstrating the potential benefits to system throughput that can be obtained by managing locality. We believe that automatic key generation will prove to be an interesting challenge for future work.

3.2   Scheduling Given Keys

We have experimented with three schemes to schedule transactions, using the executor model of Figure 1 (c). The baseline scheme is a *round robin* scheduler that dispatches new transactions to the next task queue in cyclic order. The second scheme is a *key-based fixed scheduler* that addresses locality by dividing the key space into w equal-sized ranges, one for each of w workers. There is no guarantee, however, that a mapping function that captures data locality will lead to a uniform distribution of keys. The third scheme is a *key-based adaptive scheduler* that samples the input distribution and partitions the key space adaptively in order to balance load.

To simplify the explanation of the adaptive scheduler, we assume that a transaction is some operation on an integer, which we may treat as the transaction key.

During the early part of program execution, the scheduler assigns transactions into worker queues according to a fixed partition. At the same time, it collects the distribution of key values. Once the number of transactions exceeds a predetermined *confidence threshold*, the scheduler switches to an adaptive partition in which the key ranges assigned to each worker are no longer of equal width, but contain roughly equal numbers of transactions.

More specifically, our executor employs a Probability Distribution-based partition (*PD-partition*) [19], which is similar to the extended *distributive partitioned sorting* of Janus and Lamagna [10]. It requires time linear in the number of samples. Figure 3.2 explains the *PD-partition* algorithm. Given an unknown distribution (a), the algorithm constructs a histogram (b) that counts the number of samples in ranges of equal width. It then adds the numbers in successive ranges to obtain cumulative probabilities (c) and uses these to obtain a piece-wise linear approximation of the cumulative distribution function (CDF) (d). Finally, by dividing the probability range into equal-size buckets and projecting down onto the x axis, it obtains the desired adaptive ranges (e).
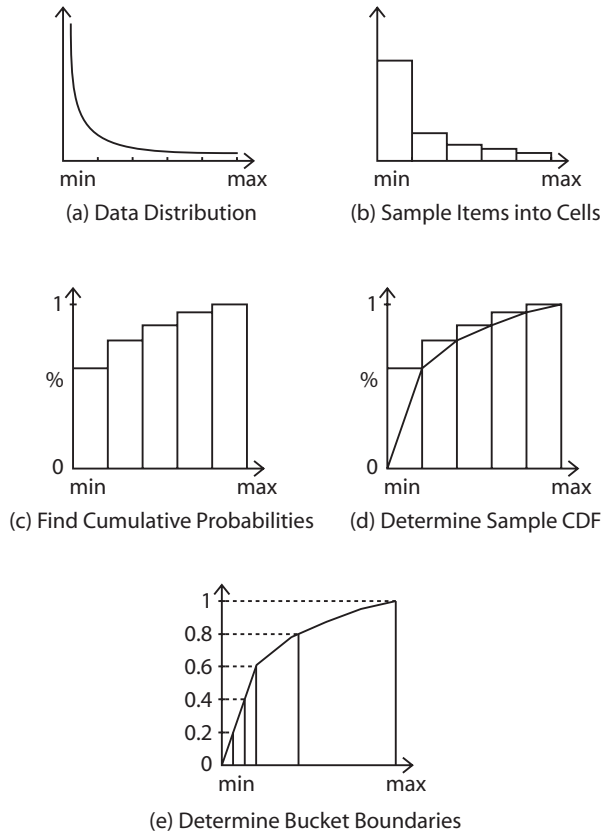

(a) Data Distribution   (b) Sample Items into Cells

(c) Find Cumulative Probabilities   (d) Determine Sample CDF

(e) Determine Bucket Boundaries

Figure 2. Adaptive calculation of key ranges.

The number of samples determines the accuracy of CDF estimation, which in turn determines the *PD-partition* balance. Shen and Ding show how to determine the sample size required for a guaranteed bound on accuracy by modeling the estimation as a multinomial proportion estimation problem [19]. In this work, we use a threshold of 10,000 samples, which guarantees with 95% confidence that the CDF is 99% accurate.

4 Evaluation

4.1 Implementation

Our implementation is based on the Java dynamic software transactional memory (DSTM) system of Herlihy et al. [8]. We have implemented the parallel executor scheme of Figure 1(c) with two separate classes of threads: producer threads, which generate transactions (the producer part in Figure 1(c)) and then dispatch them (the executor part in Figure 1(c)); and worker threads, which execute transactions within the DSTM. These are connected by task queues, one for each worker. The queues are instances of `java.util. concurrent.ConcurrentLinkedQueue`, which implements the Michael & Scott concurrent queue [14].

The entire system is orchestrated by a test driver thread, which selects the designated benchmark, starts the producer threads, records the starting time, starts the worker threads, and stops the producer and worker threads after the test period (10 seconds in our experiments). After the test is stopped, the driver thread collects local statistics from the worker threads and reports the cumulative throughput.

A producer thread operates in a loop. It first checks with the test driver ensuring that it should continue. Then it generates the next transaction. In our experiments, we use four parallel producers (eight for the hash table benchmark to prevent worker threads being hungry). For efficiency we insert the parameters of a transaction rather than the transaction itself into the task queue.

A worker thread follows a simple regimen: check with the test driver for permission to continue, obtain the next transaction, execute it, retry in the case of a failure until successful, and then increment the local counter of complete transactions. We assume that there is no need to report the result of a transaction back to the producer. The effect has already taken place in the shared data structure.

4.2 Microbenchmarks

We test three benchmarks, each of which performs insertions and deletions on concurrent data structures that implement an abstract dictionary with 16-bit search keys. We name the benchmarks by the type of data structure: hash table, red-black tree, and sorted linked list. To emphasize the impact of conflicts we do not include lookup operations. Although lookup operations would exhibit locality, they do not conflict with one another. For lack of space we report results for only the hash table benchmark. The tree and list benchmarks and their results are given in [?].

Hash table: Uses external chaining from an array of 30031 buckets (a prime number close to half the value range). The hash function is the hash key modulo the number of buckets. The benchmark uses the same number of inserts and deletes, so the load factor at stable state is around 1. A conflict occurs when the same bucket is modified by two transactions. We use the output of the hash function (not the dictionary key) as the value of the transaction key. With this particular hash function the dictionary key would also ensure spatial locality among transactions with similar keys, but this would not be the case for other, more "random" hash functions. Because hash table transactions are extremely fast, we use twice the usual number of producers to keep the workers busy.

4.3 Data Collection

All results were obtained on a SunFire 6800, a cache-coherent symmetric multiprocessor with 16 1.2Ghz UltraSPARC III processors and 8MB of L2 cache for each processor. We tested in Sun's Java 1.5.0_06, server version, using the HotSpot JVM augmented with a JSR 166 update `jar` file from Doug Lea's web site [11]. The dynamic software transactional memory (DSTM) implementation came from Herlihy et al. [8]. We used the "Polka" contention manager to arbitrate among conflicting transactions. It combines randomized exponential back-off with a priority accumulation mechanism that favors transactions in which the implementation has already invested significant resources [18].

Memory management has a non-trivial cost in Java. We use the default generational mark-sweep garbage collector. The test driver thread explicitly invokes a full-heap garbage collection after starting the producers and the DSTM system and before recording the starting time. A typical 10-second test period invokes stop-the-world garbage collection two to five times at a total cost of 0.1–0.5 seconds.

For each executor on each benchmark, we take the mean throughput of ten runs. Though we have endeavored to keep the environment as uniform as possible, random factors cannot be fully eliminated. Randomizing factors include memory management, the execution of kernel daemons, and the shutdown of threads at the end of the test. With the exception of a few outliers, all experiments consumed between 10.0 and 10.1 seconds.

4.4 Performance Comparison

We generate transactions of three distributions in a 17-bit integer space. The first 16 bits are for the transaction content (i.e., the dictionary key) and the last is the transaction type (insert or delete). The first distribution is *uniform*. The second is *Gaussian*, with a mean of 65,536 and a variance of 12,000, which means that 99% of the generated values lie among the 72,000 (55%) possibilities in the center of the range. The third is *exponential*. It first generates a random double-precision floating-point number r in range $[0, 1)$ and then takes the last 17 bits of $-\log(1 - r)/0.001$. The parameter 0.001 determines the narrowness of the distribution. In this case, 99% of the generated values lie between 0 and 6907, which is a little over 5% of the range.

Figure 3 presents throughput, in transactions per second, for the hash table for each of the three distributions. Each individual graph includes three curves, one each for the round-robin, fixed key-based and adaptive key-based executors. As supporting data, we have collected the frequency of contentions, which are given in [?].

For the uniform distribution, both key-based executors outperform round robin, clearly showing the benefit of partitioning. The main effect comes from locality since the number of conflicts is negligible in a table of this size and the load balance of round robin is perfect.

The throughput of the fixed executor is 25% higher than round robin for two workers, and the gap increases with additional workers. Note, however, that the fixed executor does not completely balance load, because the modulo function produces 50% "too many" values at the low end of the range. The adaptive executor balances the load via uneven partitioning of the index range. As a result, the adaptive executor outperforms the fixed executor up through about ten threads; at this point the fixed number of producers are unable to satisfy the processing capacity of additional workers.

For the nonuniform distributions, the throughput of the fixed executor suffers as transactions are concentrated in a narrower range. In the extreme (exponential) case, the fixed executor shows no speedup be-
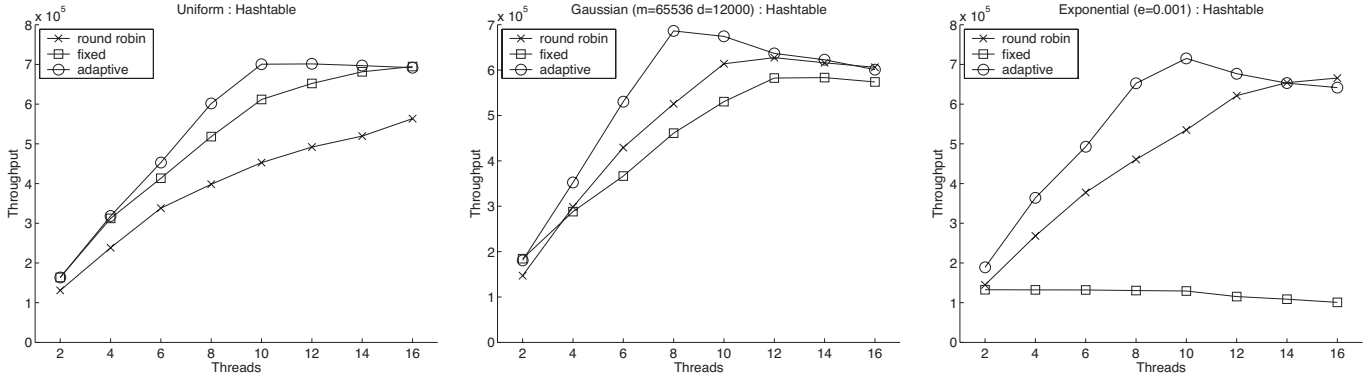
Figure 3. Throughput (txn/s) of the hash table microbenchmark with a uniform (left), Gaussian (middle), or exponential (right) distribution of transaction keys.

yond two workers. The adaptive executor, by contrast, shows mostly scalable performance as long as producers are able to keep up with new workers. Clearly the adaptive executor's advantage over round robin also comes from better locality.

Since load is equally balanced in the round robin and the adaptive executors, the main factor separating their performance would appear to be locality. The hash table here is a clear evidence. Adaptive partitioning also reduces conflicts, but the total amount of contention is so low even in round robin as to be essentially irrelevant. It seems reasonable to expect that conflict reduction will pay off in high-contention applications. The nonuniform input distribution results show the advantage of adaptive partitioning. By sampling key values, the executor can balance load among workers.

In the hash table and the uniform and Gaussian distributions of the sorted list, the total number of contention instances is small enough (less than 1/100th the number of completed transactions) to have a negligible effect on performance. Even in the red black tree and the exponential distribution of the sorted list, fewer than one in four transactions encounters contention. In less scalable applications, contention may be more of an issue for executor throughput.

Executor Overhead Executor overhead is a potential factor in overall performance. It includes the cost of queuing synchronization, both blocking and non-blocking; cache misses due to queue operations in different threads; and the garbage collection required to recover queue nodes and other metadata. To measure this overhead in isolation we devised an extreme case test that compares the throughput of a simple trans-
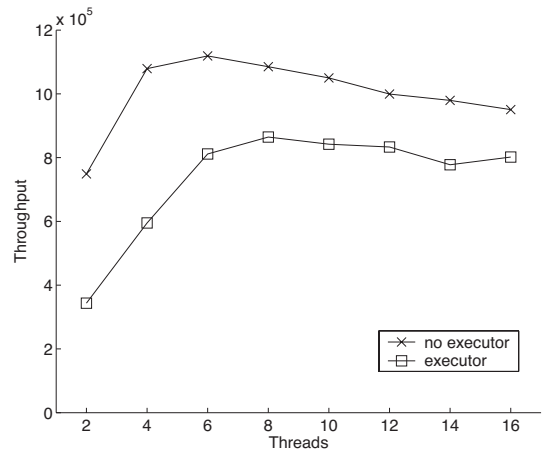
Figure 4. Throughput of empty threads and executor tasks.

actional executor with k workers to that of a collection of k threads executing transactions in a loop. For executor mode, we constantly use six producers. Figure 4 shows the result. With only two worker threads, the executor doubles the overhead of transactions. At higher thread counts the ratio is less pronounced. Naturally it is much less pronounced when transactions are of nontrivial length. In our experiments, executor overhead has a noticeable impact on the hash table benchmark, but almost none on the sorted list.

Overall, our results for the three benchmarks (tree and list shown in [?]) show a clear performance benefit from key-based adaptive scheduling in the hash table and red-black tree benchmarks, where the transaction key accurately predicts a transaction's data access pattern. The advantage is smaller in the sorted list bench-

mark, were the prediction is significantly weaker. We still see a significant benefit in the exponential distribution, however, and in all cases the throughput of the adaptive executor is either the best or essentially tied for the best. Fixed partitioning is clearly unscalable for nonuniform key distributions.

5 Related Work

The basic framework of the executor bears a close resemblance to various flavors of locality-based thread scheduling. In memory-conscious scheduling, Markatos and LeBlanc used application knowledge to give priority to threads that referenced the same data and ran them on the same processor [12]. In CacheMiner [23], Yan et al. defined a general programming interface and relied on the programmer to partition the computation into fine-grain tasks and declare their access pattern as the starting address of data access in each array. The performance was significantly better for parallel sparse matrix multiply, for which static compiler optimization such as tiling was not applicable. Philbin et al. used a similar approach to improve uniprocessor cache performance for N-body simulation [16]. While both the latter two methods used equal-size partitions of array data and assigned the tasks using a hash table, the size of the partition was determined by the number of processors in CacheMiner and by the cache size in Philbin et al.

In comparison, we use adaptive partitioning based on a run-time estimate of the probability distribution. The context is very different because transactions are executed on a stack of software layers, they use dynamically allocated data, and they are not completely parallel.

We were originally motivated by ideas from dynamic computation regrouping. For N-body simulation, irregular mesh, and sparse matrix computations, a number of studies have showed that run-time reordering can improve sequential running time despite its cost. Ding and Kennedy used lexicographic grouping [6]. Han and Tseng used lexicographic sorting [7]. Mitchell et al. divided the data space into fixed-size tiles and grouped computations in cache-sized data sub-regions [15]. Mellor-Crummey et al. organized computation into space-filling curves for computations whose data had spatial coordinates [13]. Strout et al. compared lexicographic grouping, sorting, and sparse tiling and gave a run-time framework for automating them [20]. Strout and Hovland gave a cost model based on hyper graphs [21]. Pingali et al. manually regrouped work in integer computations, including tree traversals [17]. Dynamic regrouping is also part of the inspector-executor model, originally studied by Saltz and others for partitioning irregular computations on distributed-memory parallel machines [5]. Most of these studies considered data transformation in conjunction with computation reordering.

Since transactions are generated concurrently, a separate inspection step is impractical. Key-based executors interleave inspection with transaction processing. The adaptive partitioning has low run-time overhead. We characterize locality with a key, which is readily determined for dictionary operations and should, we conjecture, be readily available for other important classes of applications. The partitioning is fully automatic because of the transaction semantics. In this work we do not address the issue of data layout. The adaptive scheme uses data sampling to estimate the distribution of the workload. Other groups have used code-based sampling to identify data streams and strides [4, 22]. While most past techniques exploit dynamic locality for sequential execution, we focus on parallel execution.

6 Conclusions

In this paper, we have augmented the dynamic software transactional memory system of Herlihy et al. with locality-aware partitioning. We presented an adaptive partitioning method based on run-time estimation of the probability distribution of the input. The adaptive executor shows clear improvement when the key accurately predicts the data access by a transaction. Otherwise, the adaptive executor performs comparable to round robin transaction scheduling. Locality-aware partitioning also has the potential to significantly reduce the rate at which transactions conflict with one another, though this effect was not a principal determinant of performance in our experiments.

Data conflicts among tasks are potentially problematic with traditional (nontransactional) executors, given the potential of lock-based code to trigger deadlock. The automatic atomicity of transactions would seem to broaden the class of applications to which executors are applicable. At the same time, while our results were obtained in the context of transactional memory, we believe that adaptive scheduling has significant promise for nontransactional executors as well.

Acknowledgements

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers, October 2001.

[2] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, Oct. 1966.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, 1995.

[4] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[5] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, Sept. 1994.

[6] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[7] H. Han and C. W. Tseng. A comparison of locality transformations for irregular codes. In *Lecture notes in computer science 1915 (Proceedings of LCR'2000)*, pages 70–84, Rochester, NY, 2000. Springer-Verlag.

[8] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22th PODC*, pages 92–101, Boston, MA, Jul. 2003.

[9] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th ISCA*, pages 289–300, San Diego, CA, May 1993.

[10] P. J. Janus and E. A. Lamagna. An adaptive method for unknown distributions in distributive partitioned sorting. *IEEE Transactions on Computers*, c-34(4):367–372, April 1985.

[11] D. Lea. Concurrency jsr-166 interest site. `http://gee.cs.oswego.edu/dl/concurrency-interest/`.

[12] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel Distributed Systems*, 5(4):379–400, 1994.

[13] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. *International Journal of Parallel Programming*, 29(3), June 2001.

[14] M. M. Michael and M. L. Scott. An efficient algorithm for concurrent priority queue heaps. In *Proceedings of the PODC*, pages 267–275, 1996.

[15] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, California, October 1999.

[16] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[17] V. S. Pingali, S. A. McKee, W. C. Hsieh, and J. B. Carter. Computation Regrouping: Restructuring Programs for Temporal Data Cache Locality. In *International Conference on Supercomputing*, New York, NY, June 2002. best student paper.

[18] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th PODC*, Las Vegas, NV, Jul. 2005.

[19] X. Shen and C. Ding. Adaptive data partition for sorting using probability distribution. In *Proceedings of International Conference on Parallel Processing*, Montreal, Canada, August 2004.

[20] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[21] M. M. Strout and P. Hovland. Metrics and models for reordering transformations. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Memory System Performance*, Washington DC, June 2004.

[22] Y. Wu. Efficient discovery of regular stride patterns in irregular programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[23] Y. Yan, X. Zhang, and Z. Zhang. Cacheminer: A runtime approach to exploit cache locality on *smp*. *IEEE Transactions on Parallel Distributed Systems*, 11(4):357–374, 2000.