Parallel Event Processing for Content-Based Publish/Subscribe Systems*

Amer Farroukh, Elias Ferzli, Naweed Tajuddin, and Hans-Arno Jacobsen
Dept. of Electrical and Computer Engineering, Middleware Systems Research Group
University of Toronto, Toronto, Canada
{amer.farroukh, elias.ferzli, naweed.tajuddin, arno.jacobsen}@utoronto.ca

ABSTRACT

Event processing systems are a promising technology for enterprise-scale applications. However, achieving scalability yet maintaining high performance is a challenging problem. This work introduces a parallel matching engine which leverages current chip multi-processors to increase throughput and to reduce the matching time. We present three parallelization techniques, as well as lock-based and software transactional memory-based implementations of each technique, and discuss their impact. The results show a 74% reduction of the average matching time and an improved throughput of over 1600 events/second when using eight processors.

1. INTRODUCTION

Event processing systems are becoming increasingly ubiquitous in mission-critical domains where high performance is of the utmost importance [18]. Such systems often span the enterprise, supporting thousands of users and processing millions of events. Achieving scalability and high performance under excessive scale and load is a challenging problem.

Fast event processing at Gigabit per second speed is of great importance in future network monitoring and analysis technologies [5]. Recently, intrusion detection systems have adopted application-layer protocol analysis techniques to ensure reliable detection of attacks and violations of security policies [8]. These approaches strive to achieve near wire-speed processing when filtering network streams. As data processing at the application level is not fast enough to handle Gigabit per second traffic on inexpensive equipment, parallel event processing that can run on commodity hardware is a promising solution.

The most computation-intensive function in event processing systems is the processing and matching engine that handles the important task of connecting decoupled entities. Finding efficient algorithms to improve matching performance has been an active area of research; however, current work is limited in that the focus has been on developing *sequential* algorithms [9, 3]. Sequential matching algorithms, while effective at increasing event throughput and reducing matching time, fail to exploit the parallel processing capabilities of today's chip multi-processors (CMPs). Since chip manufacturers are migrating towards multi-processor architectures, the next generation of matching algorithms must be capable of concurrent operation in order to improve efficiency.

While CMPs are widely available, exploiting them to create high-performance concurrent applications is a daunting task due to the inherent difficulty in ensuring thread-safe operations. Traditionally, lock-based synchronization mechanisms have been used to protect the integrity of shared data in parallel programs. Leveraging lock-based synchronization mechanisms, however, requires considerable skill on the part of the programmer. Coarse-grained locks provide simplicity in implementation but do not scale well [12]. Fine-grained locks are difficult to implement and prone to deadlocks and priority inversion [17]. Recently, transactional memory has gained momentum as a way to facilitate parallelization of applications [17, 16]. Transactional memory simplifies the application programming interface by treating accesses to shared data as transactions. In a software-based implementation of transactional memory (STM), programmers simply identify shared variables and critical sections, and the STM system optimistically executes transactions. If a conflict is detected, changes are rolled back instead of being committed.

In this paper, we present a parallel content-based publish/subscribe matching engine. We parallelize the matching engine using three techniques. In the multiple events independent processing (ME-IP) technique, threads are assigned to events and execute the entire matching for their respective events from start to end. In the single event collaborative processing (SE-CP) technique, threads are assigned to specific parts of the matching process of a single event. The third technique, called the Hybrid technique, is a hybrid combination of the first two techniques.

Additionally, we implement the parallelization techniques using three synchronization paradigms. We use a static approach where each processor is assigned a fixed number of events/predicates. We also parallelize using traditional lock-based synchronization, as well as with software transactional memory.

Our contributions are thus three-fold. First, we present the notion of parallelization in the event processing context

*This is the extended version of the DEBS'09 paper.

DEBS'09, July 6.9, Nashville, TN, USA.

by parallelizing a content-based publish/subscribe matching engine [9]. Second, we present three parallelization techniques and discuss the scope of performance gains that can be attained by parallelization. Third, we discuss the effect of the synchronization paradigm on throughput and matching time.

The remainder of this paper is organized as follows. In Section 2, we survey sequential matching algorithms and discuss work related to parallel event processing. In Section 3, we present the language and data model used and provide an overview of the centralized matching algorithm. Section 4 describes our three parallelization techniques and Section 5 presents the different synchronization paradigms used to implement those techniques. In Section 6, we provide extensive performance evaluations. Finally, we conclude in Section 7.

2. RELATED WORK

The field of Event Processing is still in its infancy and accordingly techniques for improving throughput and matching time have been addressed in several approaches [9, 3, 7]. These main memory algorithms can be categorized into two classes. However, they are based on a centralized implementation. To the best of our knowledge, we are among the first to experiment with a parallel matching engine that directly leverages chip multi-processors (CMPs).

The first class consists of two-phase algorithms: Predicates are evaluated in the first phase and matched subscriptions are computed in the second phase. SIFT [21] uses a counting algorithm for Phase 2 where users subscribe to text documents by setting weights to keywords. YFilter [7] stores user queries in non-deterministic finite automation (NFA) data structures to match against extensible markup language (XML) files. The subscriptions targeted by our system are simpler where each subscription is represented by a set of predicates. Fabret *et al.* in [9] group subscriptions in clusters based on common predicates, called access predicates. Access predicates are evaluated first, and must match in order to grant access to individual clusters. Pereira *et al.* [19, 20] use a similar matching algorithm but the algorithm presented in [9] offers better performance through support for pre-fetching, multi-attribute hash tables, access predicates and dynamic clustering.

The second class compiles predicates into a test network (i.e., a tree structure) [11, 13, 3, 14]. The internal nodes store predicates to be tested and leaf nodes represent subscriptions. The publication is evaluated starting at the root of the tree and traverses through the matched internal nodes. Each subscription is represented by one leaf node as in [3] or in multiple leaf nodes as in [11]. These algorithms suffer from several drawbacks when compared to the two phase algorithms. They do not leverage spatial or temporal locality, they are space consuming, and the data structures used are complex and costly to maintain.

We parallelize the matching algorithm presented in [9] as it supports large workloads, flexibility for changing subscriptions, high publication rates, and fast matching times. The data structure used is space efficient and cache aware to reduce false sharing.

Software transactional memories (STMs) are an active research topic and several research prototypes have been developed [17, 16, 15]. In keeping with the spirit of transactional memory - ease of use - we chose to parallelize using the LibTM library by Lupei *et al.* [15]. LibTM provides compiler support to minimize manual code changes, but is also configurable to enable performance evaluation under different STM implementation strategies.

Parallel event processing is slowly gaining traction. Biger *et al.* describe event processing networks (EPN) implemented using a stratification approach [4]. Parallelism can be achieved by grouping autonomic event processing agents into distributed stratum operating on different machines. Fahmy *et al.* have looked at computing a worst-case bound on the response time of tasks in real-time distributed systems utilizing STM for concurrency control [10]. IBM is developing products for high performance, large-scale event processing [2, 1].

This work presents a parallel matching engine for a broker running on a chip multi-processor (CMP) machine to improve event-processing performance.

3. BACKGROUND

In this section we present a general system overview and briefly discuss the data structures used, and their impact on the system. We also discuss the different techniques used to parallelize the matching algorithm.

3.1 Language and Data Model

A subscription is a collection of predicates, each of which consists of 3 characteristics: An attribute name, a value and a relational operator ($<, \leq, =, \neq, \geq, >$). An event is a collection of pairs, where each pair consists of an attribute name and a value. An event's pair, say (<attribute name> x, <value> y), matches a subscription's predicate (<attribute name> a, <value> b, <operation> c) only when x = a, and y <operation c> b. In other words, for an event to satisfy a subscription, every predicate in the subscription should be matched by some pair in the event [9].

For each attribute name, a 2-dimensional table is used to store the subscription predicates. Each row in the table represents an operator type, where the columns represent the values. For example, to store the subscription (<attribute> price, <value> 10, <operation> =), the attribute name is hashed to determine which table should be accessed. The operator of the subscription is used to index rows, whereas the value is used to index columns. Each entry in the table has a unique predicate ID, where predicates in the system are represented by integers. This scheme facilitates the matching process, minimizes memory consumption, and reduces the complexity of storing subscriptions [9].

3.2 Centralized Matching Algorithm

As discussed above, a subscription S is defined by a set of predicates of the form (<attribute name> a, <value> b, <operation> c) and an event is defined by a set of (<attribute name> x, <value> y). A predicate p is called an access predicate when a set of subscriptions cannot be satisfied unless this predicate is matched. Our implementation of the algorithm uses a linked list of access predicates to store subscriptions. Every access predicate controls access to its corresponding cluster of subscriptions.

The centralized matching algorithm is based on the two-phase algorithm presented in [9]. In the first phase, a bit vector is used to track all predicates matched by an event regardless of their corresponding subscriptions. The bit vector is first initialized to '0'. For every attribute of an event,

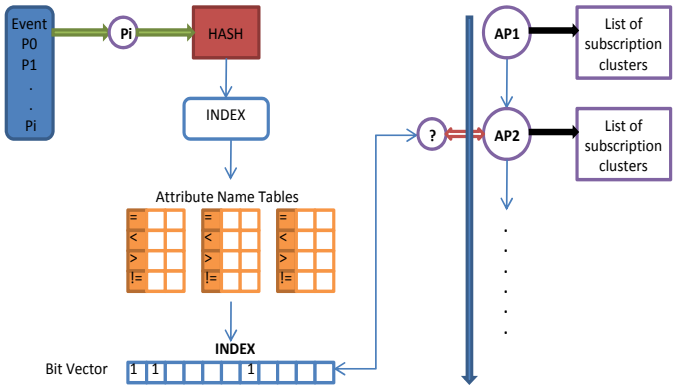

Figure 1: Overview of Matching Engine.

the attribute name is hashed to determine the table holding the attribute and the resulting table is accessed. Entries matched by this attribute are used to index the bit vector and the corresponding bits are set to '1', indicating a match. Figure 1 presents an overview of the data structure used and illustrates the steps invoked when matching publications. In the second phase of the algorithm, the access predicate list is traversed and evaluated against the bit vector. When an access predicate is matched (i.e., corresponding bit vector position is '1'), each cluster pertaining to the access predicate is evaluated. If the access predicate is not matched, then there is no need to process the corresponding subscription clusters.

The algorithm implementation is cache aware. It minimizes cache misses and false sharing. The implementation leverages spatial and temporal locality. Array traversal is optimized to increase spatial locality by storing the subscriptions column-wise, and temporal locality is achieved by reusing the same set of variables consecutively.

4. PARALLEL EVENT PROCESSING

In this section, we present the three parallelization techniques: multiple events independent processing (ME-IP), single event collaborative processing (SE-CP), and multiple events collaborative processing (ME-CP) or, in short, Hybrid. The aim of ME-IP is to increase throughput while SE-CP is used to reduce the average matching time of a single event. The Hybrid approach is a combination of ME-IP and SE-CP and it targets both metrics.

4.1 Multiple Events Independent Processing

The aim of this parallelization technique is to increase system throughput (i.e., number of events processed per second). Threads work independently on separate events to reduce the total matching time and thus increase throughput. Each thread is allocated a block of events to process as shown in Figure 2. For each event, it executes Phase 1 and computes its own bit vector. It then proceeds to Phase 2, where it traverses the access predicate list and saves the matched subscriptions. Once all events in the block are processed by a single thread, it fetches the next available set of events. Minimal synchronization is needed in ME-IP as threads are only required to keep track of the next available block of events to process. Independent processing can be achieved by a barrier-free implementation which favours scalability as the number of threads increases. On the other hand, ME-IP does not reduce the matching time of a single event since only one thread is allocated per event. Thread collaboration is key to reduce the matching time of a single event which is presented next.

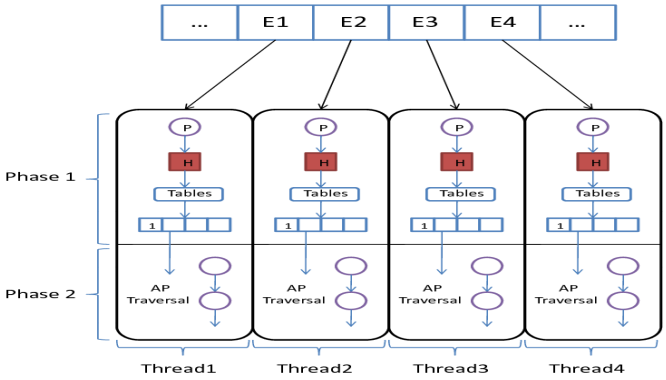

Figure 2: Multiple Events Independent Processing

4.2 Single Event Collaborative Processing

The reduction of the average matching time per event is the primary concern of SE-CP as all threads work collaboratively on a single event. Figure 3 describes the various synchronization and processing steps of this technique. For every event, each thread takes a block of predicates, evaluates them and sets the corresponding bits in its bit vector to '1' . As Phase 1 is completed, all threads update a global bit vector with their local bit vector and wait for other threads to update it as well. After all bit vectors are merged into the global bit vector, all threads move to Phase 2 of the matching algorithm. In the second phase, the access predicate list is divided into blocks and each thread is assigned to a single block. This approach divides the work of Phase 2 into several smaller tasks that can be executed in parallel. Threads wait for the completion of Phase 2 before processing a new event. It is worth mentioning that the number of global bit vector updates is directly proportional to the number of threads. This synchronization overhead reduces scalability as shown in our experiments.

4.3 Multiple Events Collaborative Processing

The multiple events collaborative processing (ME-CP) or Hybrid technique is a combination of ME-IP and SE-CP which gives the flexibility of reducing the matching time of a single event or increasing the overall throughput. Processors are divided into groups, and each group is assigned to a single event. Throughput is increased by multiple groups processing events in parallel, and matching time is reduced by leveraging the processors within a group to parallelize the matching algorithm. For example, if the matching time of a single event is more important, then more threads could be allocated to each group. One the other hand, reducing the number of threads per group results in higher throughput. The flexibility of dynamically allocating threads to different groups in the Hybrid/ME-CP algorithm eases the burden of meeting service-level-agreements (SLAs) as more subscriptions are entered into the system. Additional resources could be dedicated to a single event to maintain a constant average matching time as subscriptions flood the broker. Thus,

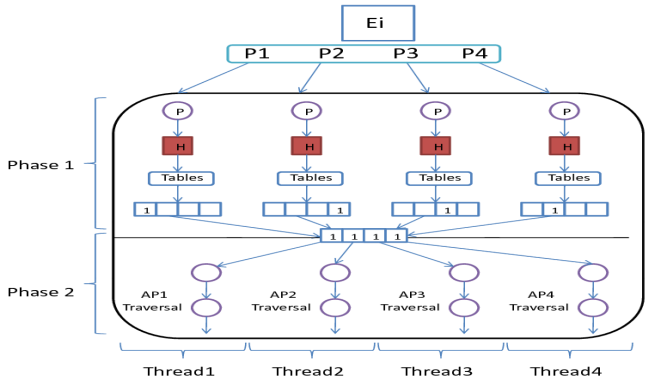

Figure 3: Single Event Collaborative Processing

gradually moving from one thread per event, ME-IP thread allocation, to eight threads per event, SE-CP thread allocation, stabilizes the average matching time in order to meet the required SLAs.

5. IMPLEMENTATION

This section presents three different implementations of the techniques described above. We first present the static approach that *a priori* divides the processing load on all threads. Next, we describe a dynamic thread-allocation implementation that is based on fine-grained locking. Finally, we show an STM-based implementation of the different techniques mentioned above.

5.1 Static-based

The first implementation is based on static thread allocation. Each thread is allocated a predetermined portion of the work to process. This approach is lock-free and requires minimal synchronization. Although, it appears optimal to use, it severely suffers with non-uniform workloads. The evident downside of this scheme is the idle time incurred by free threads as some threads might be able to finish before others in case of unbalanced workloads.

Events and predicates are evenly distributed among all threads in ME-IP and SE-CP, respectively. This distribution is based on the number of events/predicates to process and not on the corresponding processing time.

5.2 Lock-Based

We also present a dynamic thread allocation implementation based on traditional locks. The workload is divided into chunks where each thread processes the next available chunk. This scheme however involves using shared counters that keep track of workload sizes and available chunks. To keep the shared counters in check, locks are used to ensure atomicity of an operation. Consequently, when a thread needs to update a shared variable, it has to acquire the lock, update the variable, and then release the lock. When compared to the static implementation, this scheme achieves better resource utilization; however, data synchronization incurs lock contention when multiple threads require simultaneous updates to shared variables.

5.3 STM-Based

The advent of transactional memory promises a future of ease in parallel programming. This technology redeems programmers from the impediment of fine-grained locking and deadlock resolution. To cope with this new technology, we present a software transactional memory-based (STM) implementation of the matching engine. Transactions are defined in STM as a series of read and write instructions to be executed atomically as they replace critical sections in lock-based implementations. Threads execute transactions by modifying shared variables independently and each thread records the reads and writes that it has executed. Before a transaction is committed, the thread checks if any of the variables used during the transaction has been modified by a peer thread. If some of the variables have been changed by another thread, then the transaction is aborted and changes are rolled back; otherwise changes are committed. A transaction is re-executed until success. Thus, STM trades off performance for ease of implementation.

The left side in Figure 4 shows an example of a critical section in a lock-based implementation where multiple locks need to be handled. On the right end, the same function is implemented in STM. In the latter case, shared variables are declared as STM type and they are protected by the function BEGIN_TRANSACTION() and ended with the function COMMIT_TRANSACTION().

5.4 Block Size

As the number of threads increases, lock contention might hinder scalability. We introduce a block size parameter as an optimization metric to minimize lock contention. Therefore, a static allocation of threads will incur idle time that could be avoided by a more efficient choice of block size. The effect of block size is governed by a balance between lock-contention and idle time. Thus, an optimal block size reduces lock contention, decreases idle time, and achieves a uniform distribution of workload among all threads as shown in the experiments.

6. EXPERIMENTAL RESULTS

We evaluated the performance of our parallel event processing engine by examining the scalability of each technique. Specifically, we show the gains achievable in both throughput and average matching time as the matching engine scales from a sequential system of one processor to a fully parallel system of eight processors. We compare each implementation of each technique and report on the performance of each synchronization paradigm.

Additionally, we evaluated the dependence of matching time on the block size parameter. We experimentally showed the effect of block size on lock contention and thread idle time.

Experiments were run on a machine with 2.33GHz/1333-MHz FSB quad-core Xeon processors based on the "Core" micro-architecture. It has a total of eight processors where each thread is allocated to a single processor. Each core has 32KB data and instruction L1 caches. Each core shares a 4MB unified L2 cache. There is pre-fetching into the L2 cache. The L1 and L2 cache line sizes are 64B and 128B, respectively.

A workload generator was used to generate subscriptions and events. Both subscriptions and events were input to the system in a single batch (first subscriptions were loaded followed by events). Table 1 specifies the three workloads used. The *scalability workload* was used to evaluate through-

```
pthread_mutex_lock(&mutex);
    Update_Bit_Vector();
pthread_mutex_unlock(&mutex);

pthread_mutex_lock(&counter_mutex);
    counter++;
pthread_mutex_unlock(&counter_mutex);
```

```
// Beginning of Critical Section
BEGIN_TRANSACTION();
    Update_Bit_Vector();
    tm_counter++;

COMMIT_TRANSACTION();
// End of Critical Section
```

Figure 4: An example of Critical Section translation from Locks (left) to STM (right)

put and average matching time performance. The *block size workload* was used to examine the effect of block size, and the *subscriptions workload* was used to study the effect of large numbers of subscriptions on the matching time.

The workloads were chosen based on the workloads used in [9] and modified to model a real event processing system. As such, the workloads consisted of a significant number of subscriptions and events of varying size. Moreover, results were averaged over five runs.

6.1 Scalability

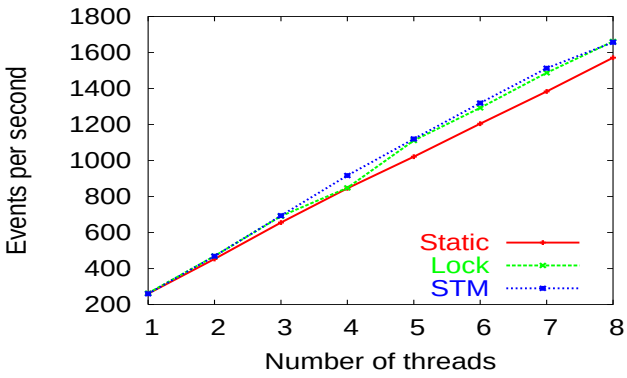

Figure 5: Throughput of ME-IP based on Scalability Workload in Table 1.

ME-IP Analysis: Figure 5 shows the number of events processed per second (throughput) for ME-IP under the static, lock-based, and STM implementations. As the graph shows, increasing the number of processors from 1 to 8 results in a near-linear increase in throughput for all implementations (263 events/s to over 1600 events/s). The linear increase is due to the fact that threads process events independently, resulting in only minimal synchronization overhead.

Additionally, we notice that all three implementations produced similar results. As above, this similarity is due to threads working independently of each other and minimal synchronization overhead.

Figure 6 shows the average matching time of a single event as the number of threads increase for the three implementations. In the ideal case, the average matching time would remain constant. In practice, however, we notice a slight increase in matching time for all three implementations, most prominently in the increase from one thread to two threads. This was expected as synchronization overhead (however minimal) increases with the number of threads. The STM approach had a slightly longer matching time for two threads due to wasted work prior to transaction rollbacks. The STM approach, however, remains promising as the throughput and the matching time are comparable to the other techniques. It is easier to imlplement than locks and it is more suitable to dynamic workloads than the static approach.

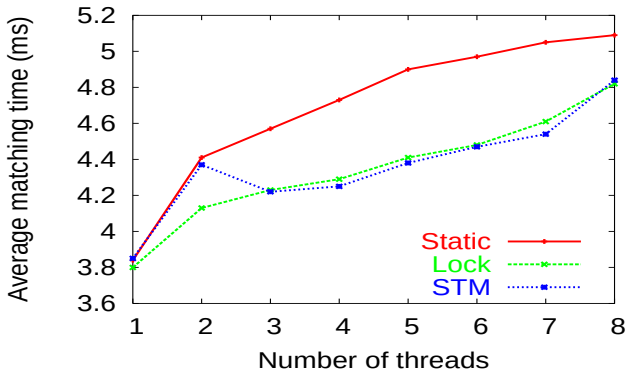

Figure 6: Average Matching Time of ME-IP based on Scalability Workload in Table 1.

In summary, ME-IP scales well with the number of threads, achieving a near-linear performance gain. The results show that it is highly beneficial to parallelize event processing systems, especially when events are independent of each other and are of uniform size. Since all implementations show similar gains, such event processing systems can be parallelized using any of the three synchronization paradigms.

SE-CP Analysis: Figure 7 depicts the time required to match a single event under the three implementations of SE-CP. For all techniques, we notice a steady decrease in matching time as the number of threads increases from one to four. When increasing the number of threads any further, no significant speedup was achieved. Beyond four threads, the matching times of the three techniques tend to stay constant. Recall that in SE-CP, threads work together on accelerating the two phases and updating the global bit vector; as a result, the barrier between the two phases impedes scalability and performance. This is not a hindering issue when

Table 1: Workload Specification

Parameter	Scalability Workload	Block Size Workload	Subscriptions Workload
Number of Subscriptions	6,000,000	2,000,000	1,000,000-5,000,000
Average Predicates per Subscription	10	6	10
Subscription Predicate Value Range	1-15	1-15	1-15
Number of Events	5000	5000	5000
Average Attributes per Event	50	30	50
Event Attribute Value Range	1-15	1-15	1-15
Number of Distinct Predicates	1500	1500	1500
Number of Distinct Attributes	100	100	100

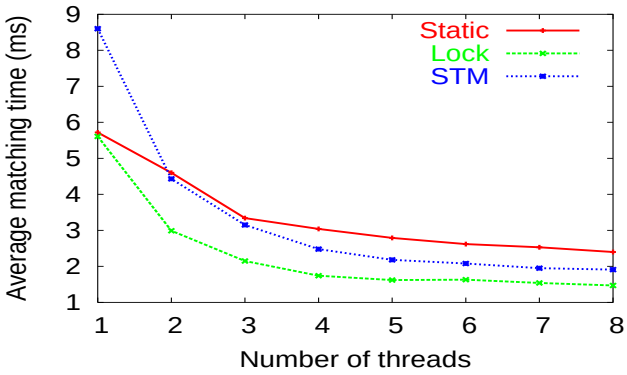

Figure 7: Average Matching Time of SE-CP based on Scalability Workload in Table 1.

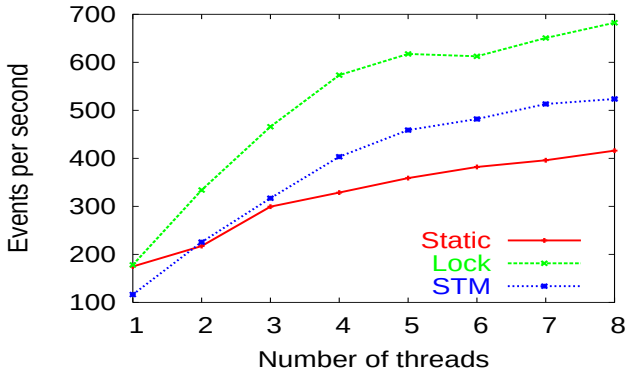

Figure 8: Throughput of SE-CP based on Scalability Workload in Table 1.

the number of threads is relatively small (one to four). In general, the matching time was reduced by approximately 78% for all three implementations.

Due to the nature of both the algorithm and the STM implementation, accessing shared variables takes longer as checks are needed before committing. This explains why STM matching time for one thread is worse than the other implementations. For the static and lock-based approaches, as the number of processors increases, updating the bit vector and waiting on barriers becomes a major bottleneck. Lock contention and barrier wait time neutralize the speedup. As for STM, the bottleneck is the transaction rollbacks. As threads write to memory and then check their logs for data corruption, transactions might be aborted and re-executed. We notice for all three approaches, at eight threads, the matching times roughly converge.

The results are similar for throughput performance of SE-CP (Figure 8). The throughput is directly related to the matching time, and thus, while there exists a substantial increase in throughput as the number of threads scale from one to four, the gain is not as great as when the active threads scale from four to eight as explained above.

SE-CP demonstrates that matching time of a single event can be decreased through parallel processing. This implicitly results in improved throughput as well but it is still less than that of ME-IP. This approach, however, does not scale as the number of threads increase due to wait times at the barrier between Phase 1 and Phase 2 as discussed next.

Time Profiling: Figure 9 shows the wait time and Phase 1 time for the lock-based implementation of SE-CP. We notice that when increasing the number of threads, the matching time decreases slightly and eventually settles between 0.02 and 0.03 ms. Recall that in Phase 1, each thread computes its own bit vector. When the number of threads is relatively large (greater than 4), combining the bit vectors into one common bit vector is a time consuming task. Each thread tries to gain atomicity when updating the common bit vector, after which it waits at a barrier for all threads to reflect their updates as well. Therefore, the time saved by allowing threads to compute different parts of the bit vector is counter-balanced by the process of updating the common bit vector. Figure 10 shows Phase 2 time for the lock-based implementation of SE-CP. Recall that in Phase 2 the access predicate list is traversed and compared to the bit vector to determine a match. If the access predicate is matched, the corresponding subscription clusters are evaluated for a potential match. With SE-CP, each thread is responsible for checking a number of access predicates. As the graph shows, with an increasing number of threads, the matching time in Phase 2 decreases because threads process the access predicate list faster. This time profile shows that Phase 2 continues to decrease as more threads are added to the system. However, Phase 1 time is almost constant due to the trade-off between splitting the workload among threads and updating the common bit vector. The wait time increases with the number of threads which is another factor for performance gain degradation. A different workload with non-uniform predicate processing time might result in a more scalable Phase 2 time. We intend to study the subscription and event workload effects in future work.

ME-CP/Hybrid Analysis: The Hybrid implementation is a versatile technique that combines both ME-IP and SE-CP.

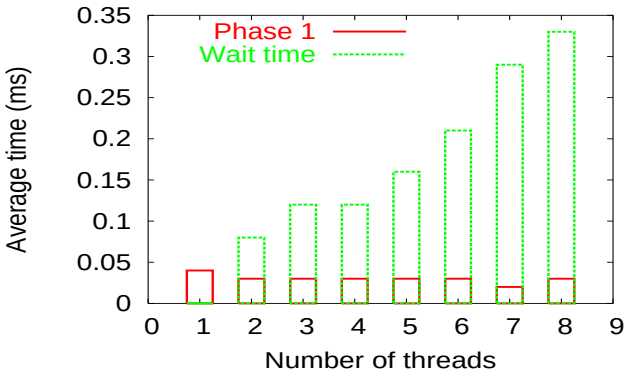

Figure 9: Phase 1 Matching Time and Wait Time of SE-CP in the Lock-Based Implementation.

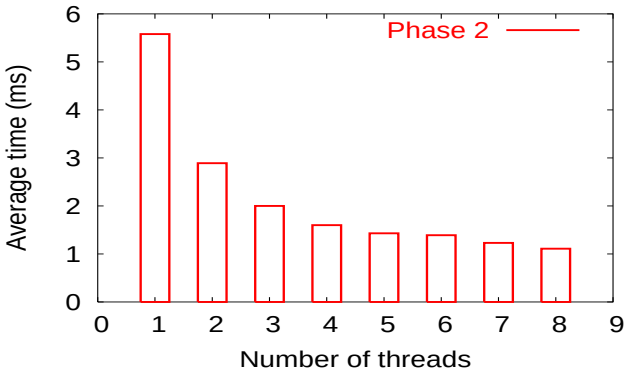

Figure 10: Phase 2 Matching Time of SE-CP in the Lock-Based Implementation.

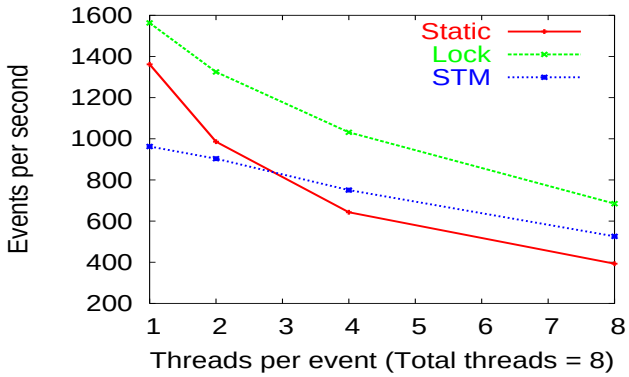

Figure 11: Throughput of Hybrid based on Scalability Workload in Table 1.

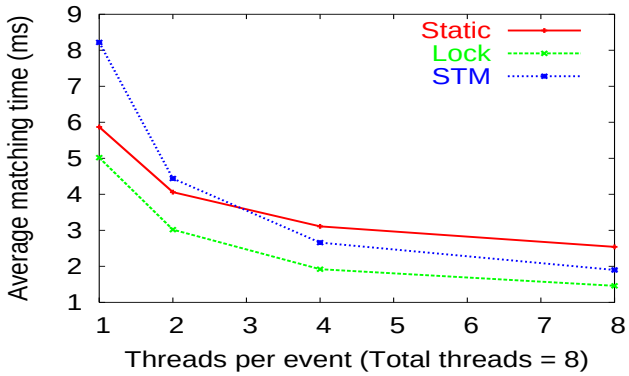

Figure 12: Average Matching Time of Hybrid based on Scalability Workload in Table 1.

Figure 11 plots the number of events processed per second against the number of threads per event. Recall that the machine used to run the experiments has eight processors. The hybrid approach divides the available threads into groups. Each group is responsible for processing an event. When there is only one thread per event, the hybrid behaviour is analogous to ME-IP with eight threads running. It can be noticed from Figure 11 that when using one thread per event, a high throughput of 1362, 1562, and 962 events/second was achieved for static, lock-based, and STM, respectively. with one thread per event (approximately 1600 for static, 1830 for lock-based, 980 for STM). Comparing the values against ME-IP, we realize that with all three implementations, the throughput achieved in the hybrid approach is less than the throughput of ME-IP. This decrease is justified by the complicated checks needed to ensure proper inter and intra-group communication between threads. Thus, throughput decreases as more threads collaborate to process a single event. When eight threads are working on a single event, the hybrid approach becomes similar to SE-CP with eight threads executing.

Figure 12 shows the average matching time for an event when 1, 2, 4, and 8 threads process a single event. As the number of threads per events increases, the average matching time decreases. It can be concluded from the results that the Hybrid technique balances throughput and average matching time. This technique is suitable for maintaining the average matching time constant when large numbers of subscriptions overwhelm the broker.

6.2 Blocksize

Figure 13 shows the relationship between workload and block size. The workload was fixed at two million subscriptions, with an average of six predicates per subscription. We only show the results of the lock-based implementation of SE-CP with seven processors executing in parallel as other techniques have similar behaviour. We fixed the block size for Phase 1 and varied the block size for Phase 2 (i.e., evaluating access-predicate clusters).

Additionally, access predicates are computed to optimize performance which results in a non-uniform distribution of clusters [9]. Therefore, a static allocation of threads (where block size is equal to the number of access predicates divided by the number of threads) will incur idle time that could be avoided by a more efficient choice of block size. The effect of block size is governed by a balance between lock-contention and idle time. Thus, an optimal block size (i.e., 12 in Figure 13) achieves a uniform distribution of workload among all threads.

If the block size is too small (less than 10), while there may be increased uniformity in the workload distribution, average matching time suffers because there is too much lock contention as threads quickly process their load and wait

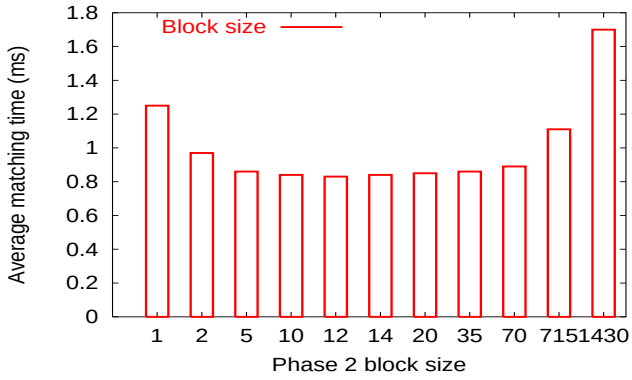

Figure 13: Effects of Block Size on Average Matching Time based on Block Size Workload in Table 1.

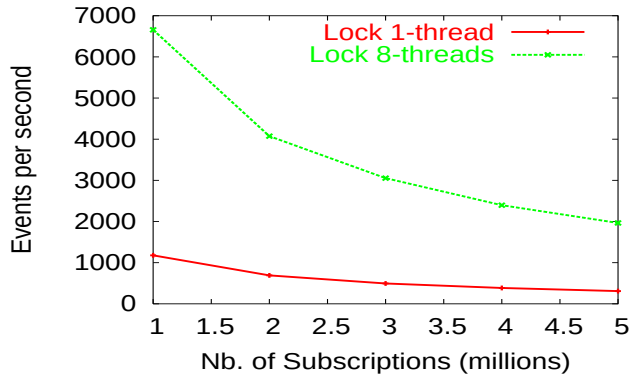

Figure 14: Effects of Number of Subscriptions on ME-IP Throughput based on Subscriptions Workload in Table 1.

for the next series of events. On the other hand, if the block size is too large (greater than 35), while lock contention is minimized, average matching time still suffers because of increased idle time. This is due to the fact that at the end of the run, a subset of threads may have to wait for the remaining threads to complete execution of their final blocks.

The poorest performing block sizes were the extreme cases of 1 and 1430 (the size of the access predicate data structure). In the former case, lock contention impedes performance; and the latter case is equivalent to sequential execution. Thus block sizes between 10 and 30 proved to be the most efficient in this experiment.

6.3 Subscriptions

Figure 14 shows the effect of increasing the number of subscriptions on system throughput. As more subscriptions are inserted into the system the throughput of the single-threaded approach or the multiple-threaded (ME-IP) approach decreases. This is expected, as increasing the number of subscriptions results in larger bit vectors, more dense clusters, and longer matching times.

The results of Figure 15 illustrate the effects of large numbers of subscriptions on the average matching time of the centralized algorithms and SE-CP with eight threads. The matching time is increased almost five times when using eight threads and only 1.5 times for the centralized case.

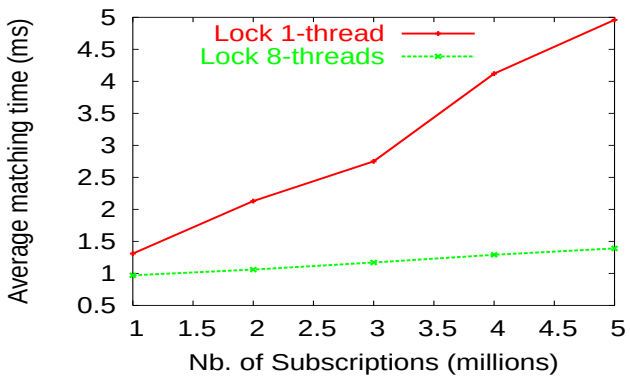

Figure 15: Effects of Number of Subscriptions on SE-CP Average Matching Time based on Subscriptions Workload in Table 1.

This is due to synchronization time wasted when updating larger bit-vectors.

7. CONCLUSIONS

In this paper, we have presented three parallelization techniques to reduce the matching time of a single event and to increase throughput. The results have shown that when using eight processors, the throughput increased from 263 to over 1600 events/second in ME-IP and the matching time is reduced by 74% in SE-CP for the given workloads. We also presented a hybrid approach that is a combination of both techniques. The hybrid approach is driven by service-level-agreement (SLA) requirements. As more subscriptions are entered into the system, the average processing time of a single event increases which might violate SLAs. This approach gives the flexibility to the administrator to dedicate more resources to a single event and maintain a constant average matching time as subscriptions flood the broker. Thus, moving from one thread per event (ME-IP) to eight threads per event (SE-CP) stabilizes the average matching time to meet the required SLA.

We also presented three implementation schemes for the parallel matching engine. The static-based scheme is the simplest where threads are allocated to a pre-determined workload. The static allocation used in this technique suffers from idle time as the workload is not split evenly among all threads; it is not, however, vulnerable to lock contention.

Second, we presented a fine-grained lock-based implementation that favours dynamic allocation of threads to reduce idle time. This technique uses a block size parameter to reduce lock contention while keeping idle time minimal. It was noted that small block sizes result in high lock contention and large block sizes sacrifice parallelism. Thus, an adequate block size would balance both trade-offs. Despite the evident performance gain achieved by this implementation, fine-grained locking is still a challenging task for most programmers due to its difficulty in deadlock resolution.

Third, since transactional memory has emerged as a promising solution for ease of synchronization, we also presented a software transactional memory-based implementation. This implementation would be comparable to the static-based implementation in terms of performance, if not prone to wasted work due to rollbacks. While transaction rollbacks are ex-

pensive in software, the aim of this implementation is to ease the burden of handling locks. It is also inspired by the hope that hardware transactional memory (HTM) [6] will soon be adopted by chip manufacturers; thus transactions would be executed much faster and TM will be comparable to the performance of lock-based implementations.

The results showed that ME-IP is highly scalable but the performance gain of SE-CP has an is upper bound of four threads. The bit vector update between the two phases in the latter technique is a function of the number of threads, which hinders scalability and performance. On the other hand, in case of more subscriptions and heavier workloads, dense access-predicate clusters or high matching predicate values might compensate for the idle time wasted at bit vector updates. However, manipulating the workload to result in more scalable systems is not discussed in this paper as we intend to solve this problem in future work.

8. REFERENCES

[1] Business event processing from IBM. http://www-01.ibm.com/software/solutions/soa/business_event_processing.html.

[2] Websphere business events extreme scale. http://www-01.ibm.com/software/integration/wbexs/.

[3] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC*, 1999.

[4] A. Biger, O. Etzion, and Y. Rabinovich. Stratified implementation of event processing network. Fast abstract on DEBS, 2008.

[5] C. Cranor, T. Johnson, and O. Spataschek. Gigascope: a stream database for network applications. In *SIGMOD*, 2003.

[6] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *SIGOPS*, 2006.

[7] Y. Diao, P. Fischer, M. J. Franklin, and R. To. Yfilter: Efficient and scalable filtering of xml documents. In *ICDE*, 2002.

[8] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer. Dynamic application-layer protocol analysis for network intrusion detection. In *USENIX Security Symposium*, 2006.

[9] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, 2001.

[10] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Response time analysis of software transactional memory-based distributed real-time systems. In *SAC*, 2009.

[11] J. Gough and G. Smith. Efficient recognition of events in distributed systems. In *ACSC*, 1995.

[12] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. In *SIGOPS*, 2007.

[13] E. N. Hanson. Rule condition testing and action execution in Ariel. In *SIGMOD*, 1992.

[14] G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *ICDCS*, 2005.

[15] D. Lupei, A. Czajkowski, C. Segulja, M. Stumm, and C. Amza. Automatic adaptation of transactional memory state management to application conflict patterns. In *Interact*, 2009.

[16] V. J. Marathe, W. S. Iii, and M. L. Scott. Adaptive software transactional memory. In *DISC*, 2005.

[17] M. Olszewski, J. Cutler, and J. G. Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *PACT*, 2007.

[18] Oracle. Complex event processing in the real world. White Paper, 2007.

[19] J. A. Pereira, F. Fabret, F. Llirbat, R. Preotiuc-Pietro, K. A. Ross, and D. Shasha. Publish/subscribe on the web at extreme speed. In *VLDB*, 2000.

[20] J. A. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In *CooplS*, 2000.

[21] T. W. Yan and H. Garcia-Molina. The SIFT information dissemination system. In *TODS*, 1999.