Concurrency control in distributed caching

Kashinath Dev Rada Y. Chirkova
kdev,chirkova@csc.ncsu.edu

Abstract

Replication and caching strategies are increasingly being used to improve performance and reduce delays in distributed environments. A query can be answered more quickly by accessing a cached copy than making a database round trip. Numerous techniques have been proposed to achieve caching and replication in various contexts. In our context of flat cluster-based networks, we have observed that none of the schemes prove to be optimal for all scenarios. In this technical report we look at concurrency control techniques for achieving consistency in distributed caching in flat cluster-based networks. We then come up with heuristics to choose some concurrency control mechanisms over others, depending on the parameters such as the number of data requests and the ratio of read to write requests.

1 Introduction

1.1 Overview

In client-server architectures, data are usually stored in a central database server. The central server interacts with several client servers, which interface with end users. If these client servers are remote, then data from the central server will have to be retrieved over long distances. This type of data retrieval contributes to the shortage of resources such as communication bandwidth, and typically results in an unacceptably long response time. One mechanism for solving this problem, called *caching*, is to store frequently retrieved data at the point of usage or along the way. Caching is a tried and tested mechanism for dramatically speeding up applications [8]. In the client server architecture, the cache resides on the local machine and the database is on the remote server. When data are cached locally on machines, the local cache services the request for data. This lessens retrieval time, since the number of roundtrips to the database are reduced. Data are often distributed and replicated over several caches to improve memory and CPU utilization [26]. Since the data are stored locally in this case, local data updates on these individual caches may lead to situations where the information in the database and the cache drift out of synchronization. This may cause invalid data and results being returned. Concurrency control deals with issues involved with allowing multiple end users simultaneous access to shared entities, such as objects or data records. If we wish to achieve the goal of consistency of data in caches, we need to use concurrency control methods, so that multiple nodes can simultaneously and correctly access the data for reading or updating.

The goal of this paper is to come up with heuristics to choose some concurrency control mechanisms over others, depending on scenarios such as the number of data requests and the ratio of read to write requests. We adapt existing concurrency control schemes from distributed databases, distributed systems and web server caches for the context of replicated distributed caching in flat cluster-based networks.

1.2 Motivation

Distributed caching is used in many applications that run in distributed environments. The primary advantages of using caching are improvements in the availability of data and overall performance. In the J2EE environment [10], which is a platform for developing and deploying distributed multi-tiered enterprise applications, there are many provisions provided for caching in many parts of the architecture. Entity bean caching is one such provision [22]. Many existing application servers provide caching to improve application performance. Websphere Application Server [21], Oracle Application Server [2], BEA WebLogic Server [19], JBoss Application Server [20] are a few of the many application servers that implement caching.

There are scenarios where additional application level caching needs to be implemented, over what is already provided by the application servers. Products such as Tangosol [28] and the JBoss Cache [7] have implemented caching to improve performance of J2EE applications.

In a typical J2EE architecture without application caching, the data in the data store are normally encapsulated by an Enterprise Java Bean (EJB) [9]. EJBs are reusable and portable software components that model business objects and processes. For every request for EJB data, a roundtrip to the database has to be made. This can prove costly because of the delay in answering the request. When caching is involved, the same query is likely to be answered more quickly.

1.3 Contributions

The main goal of this paper is to help decide which concurrency control algorithms would perform better for maintaining cache coherency for several scenarios in distributed caching. There is a lot of existing work done in the fields of concurrency control for distributed databases, distributed systems and web caches. This paper focuses on providing guidance for selecting a concurrency control mechanism for small cluster-based distributed environments. These schemes can be used in scenarios where caching is used to store data that is read and updated often, but not critical enough to protect using transactions. The results of this paper could be useful to many real world applications such as application servers running on the clustered J2EE technology and in other enterprise applications that make expensive roundtrips to the database for data access. Some other scenarios that can benefit from our results include client-server systems and distributed database systems.

2 Related Work

Caching has been used for quite some time to speed up data exchanges between the database and application systems [8]. While the single-cache model is the traditional approach to caching, the distributed caching model is a relatively new architecture [17]. The general architecture of distributed caching is described along with its advantages and disadvantages in [17, 5, 28]. In [24], replication and distribution of data are two out of the four design recommendations for improving cache performance.

[17] explains some common terms associated with distributed caching. Some related problems such as accessing replicated objects, can be found at [18] and [13]. Once the need for caching has been established and distributed caching is seen as an important component in architectures for providing caching in cluster-based application severs, the next logical question is how to maintain consistency among caches. In contrast to traditional caching, this problem occurs in distributed caching as objects are replicated at many servers. A lot of work has been done in the field of cache consistency for web caching and distributed systems [3, 26, 29]. At the same time, very little work has been done in the field of distributed caching.

[25, 3, 4, 26] are a few of the studies similar to ours. [25] and [3] are in the setting of databases, [4] is in the context of distributed databases and [26] is in the context of web caching. The authors of [3] compare three approaches of concurrency control in databases, namely locking with blocking, locking with immediate-restart and an optimistic algorithm. However these approaches deal with handling transaction conflicts, whereas in our scenario we are not concerned with transactions, but rather individual requests. The authors of [4] compare the structure and correctness of many variations of concurrency control schemes. However they do not provide performance analysis of the schemes. The authors of [26] compare various schemes used in maintaining web cache coherence. However these schemes are implemented by cache polling each time and by checking for consistency, whereas our work involves using more complex consistency control schemes. [4] is a study of cache coherence schemes in a client-server DBMS architecture. At the same time this paper used locking schemes, such as 2-phase locking which provides ACID properties. The focus of our work is different in that we compare more lightweight schemes, which do not guarantee ACID properties.

3 Preliminaries

Recall that we had defined caching earlier as the concept of locally storing copies of remote data objects for faster retrieval. In this and the following sections, we are using a setting that is same as that used by Pooja

Kohli in [14]. In fact Pooja Kohli and we used the same setting to study two different aspects of distributed caching.

3.1 Pure distributed caching

In *pure distributed caching*, only one copy of each data object is maintained across all the cache servers. The objects are distributed among the existing cache servers. Each cache server is responsible for maintaining the objects it caches. When an object needs to be accessed, the cache server that has the object is contacted, and the object is retrieved. While this scheme makes the task of cache consistency easy, it makes the lookups relatively more expensive. This scheme could also lead to uneven distribution of workload when one object becomes much more popular than the others, resulting in an increase in the number of queries targeted toward that particular cache server.

3.2 Replication

In *replication*, each object is replicated over all cache servers, thus each cache server maintains a copy of each object. Replication makes the system redundant and available. It also eliminates the single point of failure, which occurs when the 'one copy per object' scenario is implemented as mentioned in the previous subsection. Replication also improves response time, as the workload can be distributed among multiple servers caching the objects. At the same time, this introduces complexity in maintaining cache consistency, as all the servers are now accountable for the same data object.

4 Replicated distributed caching

We use the term *(replicated) distributed caching* to refer to schemes that contain some elements of pure distributed caching *and* replication. Each of pure distributed caching and replication offer two extremes. While pure distributed caching offers a scenario that is easy to maintain, it has drawbacks. First, it usually hinders performance as the server with the copy of the object needs to be located and contacted, making lookups relatively more expensive. Second, it may lead to uneven workload distribution and, finally presents a single point of failure. With replication, the system has increased redundancy and availability, but maintaining consistency among the various replicas is more complex. By combining the two we avoid the drawbacks offered when any one of these schemes is applied just by itself. To implement distributed caching, we replicate the objects at more than one but not all cache servers. We refer to the number of servers an object is replicated at as *degree of replication*.

5 Comparison of Concurrency control schemes

The schemes compared are all variations on the primary based remote write protocol, as shown in Figure 1.

5.1 Approach based on the non-locking model

5.1.1 Single thread model

This scheme is an implementation of the remote read and remote write or single server model, proposed in [12]. The cache provides sequential consistency by implementing the single thread model scheme. Sequential consistency is a slightly weaker consistency model than strict consistency. In general, a data store is said to be *sequentially consistent* when it satisfies the following condition (we borrow this definition from [23]):
The result of any execution is the same as if the (read and write) operations by all the processes on the data store were executed in some sequential order and the operations of each individual process appeared in this sequence in the order specified by its program.

In this scheme, concurrency control is implemented by a single thread to handle the read and write/update requests. The requests to the cache are buffered and sent to the origin server, which hosts the primary copy. The origin server buffers the requests and processes them in serial order. No locking is involved as there are

no concurrent operations. This scheme is the simplest to implement but does not offer good performance. This scheme cannot take advantage of multi-CPU systems.

We discuss the meaning of the terms *cache server, origin server, lock manager* in full detail in section 6.1.1. The cache server is a remote server that implements an object cache. The origin server is the server where the original stored data resides. The lock manager resides on the origin server and is responsible for assigning the locks.

Pseudocode for cache server:

```
1. While (no request){
         do nothing
}

2. If (message == 'new request') {
    send request to the origin server
}

3. If (message == 'success message from origin server') {
     execute request
}

4. If (message == 'cache update message from origin server') {
    update cache with data
}

5. Goto step 1
```

Pseudocode for origin server:

```
1. While (no message){
         do nothing
}

2. If (message == 'new request'){
    add to buffer queue
}

3 While (buffer queue is not empty){

    read request from front of the queue

     If (request == 'READ')
     {
        send success message back to cache
     }
    Else If (request == 'WRITE')
    {
        update cache and database
        send success message back to cache server
        send cache update message to all other cache servers
    }

}
```

```
4. Goto step 1
```

5.2 Approaches based on the locking model

In this scheme [4], read requests and write requests are synchronized by explicitly detecting and preventing conflicts between concurrent operations. The schemes in this model are examples of the weak consistency model [23].

We borrow the following definitions from [23]. A synchronization variable S has only a single associated operation `synchronize(S)`, which synchronizes all local copies of the data store. When the data store is synchronized, all local writes by process P are propagated to the other copies, whereas writes by other processes are brought in to P's copy. Using synchronization variables to partially define consistency leads to the so-called weak consistency. Weak consistency models have three properties: 1) Accesses to synchronization variables associated with a data store are sequentially consistent. 2) No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere. 3) No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

Before a read operation, a request must own a read lock and before writing a value, it should own a write lock on the data item. All schemes used in the locking model use a *thread pool* at the origin server to handle requests [11]. In this model, a group of threads are pre-spawned during initialization to form the thread pool. This eliminates the overhead of creating a new thread for each incoming request. When a request arrives at the origin server, one thread from the pool is selected to handle the request. After the request has been handled, the thread is returned to the pool.

5.2.1 Locking with write operations preferred over reads

This scheme was proposed in [6]. [15] describes an implementation of it. In this scheme, any number of requests can read the data concurrently, whereas the write requests are serialized and handled one at a time. The operation of this lock is as follows. The first read request that requests the lock will get it. Subsequent read requests also get the lock, and all of them are allowed to read the data concurrently. When a write request tries to acquire a lock, it is put on a write sleep queue until all the read requests exit. A second write request will also be put on the write sleep queue. Should a new read request show up at this point, it will be put on the read request sleep queue until all the write requests have completed. Further write requests will be placed on the write sleep queue (hence, in front of the waiting read request), meaning that write requests are always favored over read requests. The write requests will obtain the lock one at a time, each waiting for the previous write request to complete.

When all the write requests have completed, the entire set of sleeping read requests are awakened and can then attempt to acquire the lock.

This scheme is used primarily in situations where there are a large number of read requests and very few write requests.

Pseudocode for cache server:

```
1. While (no message){
         do nothing
}

2. If (message == 'new request') {
    If (buffer_queue is empty)
    {
        send request to origin server
    }
    else
    {
        add new request to buffer_queue
```

```
    }
}

3. If (message == 'success message from origin server') {
        If (message == 'read success') {
            read data from cache
        }
        Else If (message == 'write success') {
            write data to cache
        }
    send first request from buffer_queue to origin server
}

4. If (message == 'cache update message from origin server') {
    update cache with data
}

5. Goto step 1
```

Pseudocode for origin server:

```
1. Initialize the thread pool and cache;

2. While (no message){
        do nothing
}

3 If (new message) {
    select thread from thread pool
}

4. If (message == 'READ request') {
    acquire read_lock from lock manager

    if (success) {
        then send success message to cache
        release read_lock
        return thread to thread pool
    }
}

5 if (message == 'WRITE request') {
    acquire write_lock from lock manager

    if (success) {
        update cache and database
        send success message back to cache server
        send cache update message to all other cache servers
        release write_lock
        return thread to thread pool
    }
}

6. Goto step 2
```

Pseudocode for lock manager:

```
1. function lockWrite {
    If (no write lock assigned or no read lock assigned){
        assign write lock
    }
    Else{
        wait in write queue
    }
}

2. function lockRead {
    If (no waiting write requests){
        assign read lock
    }
    Else{
        wait in read queue
    }
}

3. function releaseLock {
    If (waiting write requests and no read locks assigned){
            wake 1 write request
    }
    Else If (no waiting write requests and there are waiting read requests){
            wake all read requests
    }
}
```

5.2.2 Locking with no preference

This scheme is an implementation of primary copy locking or remote read and remote write or single server model.

In this scheme [27], concurrency control is implemented by using mutex locks. Before a request can perform a read or write operation, it must acquire a mutex lock for that operation. There are two types of locks for a resource, a read lock and a write lock. Requests to the cache are buffered and dispatched one at a time to obtain a lock on the primary copy which resides on the origin server. The lock manager on the origin server is responsible for assigning the locks. The lock manager has two wait queues: the read wait queue and the write wait queue. When a request cannot be assigned a lock it is placed in the wait queue. When a lock is released, the next lock request should be granted to one request from one of the queues. To avoid favoring one over the other, a competition based policy is used. This allows the interested requests to race to grab the resource. In this scheme, one request from the read wait queue and one request from the write wait queue are dispatched. They compete to grab the resource. The first request to acquire the lock on the resource gets it, the other request is put back on the queue.

Pseudocode for cache server:

```
1. While (no message){
         do nothing
}

2. If (message == 'new request') {
    If (buffer_queue is empty)
    {
        send request to origin server
    }
    Else
```

```
    {
        add new request to buffer_queue
    }
}

3. If (message == 'success message from origin server') {
    execute request
    send request from front of buffer_queue to origin server
}

4. If (message == 'cache update message from origin server') {
    update cache with data
}

5. Goto step 1
```

Pseudocode for origin server:

```
1. Initialize the thread pool and cache;

2. While (no message){
         do nothing
}

3 If (new message) {
    select thread from thread pool
}

4. If (message == 'READ') {
    acquire read_lock from lock manager

    If (success) {
        then send success message to cache
        release read_lock
        return thread to thread pool
    }
}

5 If (message == 'WRITE') {
    acquire write_lock from lock manager

    If (success) {
        update cache and database
        send success message back to cache server
        send cache update message to all other cache servers
        release write_lock
        return thread to thread pool
    }
}

6. Goto step 2
```

Pseudocode for lock manager:

```
1. function wakeReader {
```

```
    wake one read request from read queue
}

2. function wakeWriter {
    wake one write request from write queue
}

3. function releaseLock {
    If (no more owners){
        then wakeReader() and wakeWriter()
    }
}

4. function lockWrite {
    If (no locks on object){
         then assign write_lock
    }
    Else{
        add to write queue
    }
}

5. function lockRead {
    If (no locks on object){
         then assign read_lock
    }
    Else{
        add to read queue
    }
}
```

5.2.3 Locking for updates with write-behind cache

The concept of the write-behind approach is mentioned in [16]. Tangosol [28] uses the write-behind approach and implements its caching in such a way that all read-only operations occur locally, all concurrency control operations involve at most one other cluster node, and only update operations require communicating with all other cluster nodes. This approach, shown in Figure 2, results in scalable performance. The scheme uses the concept of an *issuer*. An issuer is a node that is responsible for maintaining the consistency of the data it is caching. The issuer must issue the lock for the resource. This concept is similar to the other locking schemes evaluated so far where in the caches try to get a lock from the primary copy.

The scheme implemented in Tangosol is different from the other schemes in that it employs the write-behind approach to caching whereas the other schemes all use the write-through approach. In the write-behind approach updates to a cache are not immediately written back to the database, instead they are held in the cache. After a certain interval of time the data are written back to the database. The advantages of this scheme are as follows:

- Application performance tends to improve because users do not have to wait for the data to be written to the underlying data store. (The data are written later using a different execution thread.)
- The application is likely to experience reduced database load, resulting from the reduction in the number of read and write operations. The reads are reduced by caching, as with any other caching approach. The writes-typically much more expensive operations-are often reduced in number because multiple changes to the same object within the write-behind interval are coalesced and only written once to the underlying database.

Pseudocode for cache server:

```
1. While (no message){
        do nothing
}

2. If (message == ‘new read request’) {
    If (buffer_queue is not empty)
    {
        add to buffer_queue
    }
    Else
    {
        read data from cache
    }
}

3. If (message == ‘new write request’) {
    If (buffer_queue is empty)
    {
        send request to origin server
    }
    Else
    {
        add new request to buffer_queue
    }
}

4. If (message == ‘success message from origin server’) {
    execute request
    process_buffer_queue
}

5. Function process_buffer_queue {
    remove first message from buffer_queue
    while (message == ‘read request’){
        read data from cache
        remove first message from buffer_queue
    }
    send request to origin server
}

6. If (message == ‘cache update message from origin server’) {
    update cache with data
}

7. Goto step 1
```

Pseudocode for origin server:

```
1. Initialize the thread pool and cache;

2. While (no message){
        do nothing
}

3 If (new message) {
```

```
    select thread from thread pool
}

4 If (message == 'WRITE') {
    acquire write_lock from lock manager

    If (success) {
        update cache
        send success message back to cache server
        send cache update message to all other cache servers
        release write_lock
        return thread to thread pool
    }
}

5. If (timestamp expired){
        update database
}

6. Goto step 2
```

Pseudocode for lock manager:

```
1. function lockWrite {
    If (no write lock assigned){
        assign write lock
    }
    Else {
        wait in queue
    }
}

2. function releaseLock {
    If (there are waiting write requests){
        wake one write request
    }
}
```

6 Implementation

6.1 Testbed description

We simulate a distributed caching environment with a central server and a cluster of workstations serving as client servers. The basic testbed for simulating distributed caching is implemented on three workstations, each with a 863MHz Intel Pentium III Processor, 128 MB RAM and an internal 13.9GB hard drive. The testbed also includes an IBM xSeries 335 server with a 2GHz Intel Xeon CPU, 1GB RAM and two internal 16.9 GB hard drives. All the servers run Microsoft Windows 2000 Professional 5.00.2195 Service Pack 4. The IBM xSeries 335 server acts as the origin server; it hosts the IBM DB2 (Version 7.0) database-management system. All the components are connected via a LAN to form a cluster.

6.1.1 The basic building blocks of the testbed

Origin Server The server on which the original stored data resides or is to be created is called the *origin server*. Whenever a cache server needs to cache an object, it contacts the origin server and retrieves the object to be cached. For each scheme, the origin server hosts the central controller responsible for maintaining

coherency, the primary copy and the issuer. There is one origin server in the environment; it hosts the central database for all the cache servers.

Cache Servers These are the remote servers that interface with end users. Each cache server implements an object cache which stores one object. The cache server is responsible for maintaining the state of the object it caches. Whenever a client needs to access an object, the query is addressed by the nearest cache server that is maintaining the object. There are multiple cache servers attached to the origin server to simulate a distributed network. We do not consider the problems of choosing the caching policy, i.e. deciding which object to cache, or of cache replacement-we assume cache-size and caching policies to be independent complex problems which are studied extensively in the literature.

Objects In our scenario, objects are Java objects encapsulating tuples of a database residing on the origin server. The objects in our experiments were of type integer.

Cache Each cache server implements an object cache. The object cache is implemented as a Java Hash Table data structure [1]. Each stored object has a key and value pair. The key is generated on the primary key defined for the object in the database. Using this key, the lookup of the value of the object can be done in constant time.

Request A request is defined as a read or write/update operation for an object. Requests are simulated as SQL queries or update statements.
An example of a read request(i.e. query) is:

```
SELECT age FROM employee_table WHERE salary > 1000000;
```

An example of a write request(i.e. update statement) is:

```
UPDATE employee_table SET dept='Accounting' WHERE dept='Auditing'
```

Figure 3 describes how the above mentioned modules interact with each other. The database resides on the origin server. The user interacts with the cache manager. The cache manager in turn interacts with the cache manipulator and with the cache managers residing on other cache servers. It also sends and retrieves objects from the origin server. The cache manipulator is mainly responsible for maintaining the local cache and performs initialization, retrieval and updates of the cache..

6.1.2 Design assumptions and simplifications

- Cache size is infinite: Recall that we do not consider the problems of choosing the caching policy, i.e. deciding which object to cache, or of cache replacement.
- For all the schemes studied, the origin server sends the updated data to the other caches whenever an update has been completed. We chose this scheme because it generates the lowest number of traffic in terms of the number of messages passed. This scheme was used in combination with all the schemes compared in this paper.
- To minimize the number of workstations needed for simulating cache servers, in our network a single workstation is capable of mapping to up to five caches. This is required to simulate concurrent operations on multiple caches.
- Network traffic is not considered as a criterion for evaluating performance, since the cluster size is never assumed to be very large. The number of cache servers in our cluster is no more than ten cache servers.

6.1.3 Testbed design details

Cache Manipulator This is a Java class written to do operations on the local cache; it handles the following operations:

- **Cache initialization:** Creation of cache and allocation of data structures. This is done at the time the cache is first instantiated on a workstation.
- **Filling the cache:** Once the cache has been simulated, the objects that need to be cached are retrieved from the origin server.
- **Object retrieval and updates:** When a client requests to read or update an object, the cache manipulator accesses the local cache and addresses these requests.

Cache Manager This is the module that implements the concurrency control algorithm used by the cache.

Prototype design and implementation The prototype for each concurrency control protocol evaluated in our experiments is built on top of the testbed described above. For each scheme, new methods specific to the scheme are integrated into the cache manager module.

6.2 Evaluation Methodology

6.2.1 Parameters

- **Number of requests:** A *request* is defined as a read or update of an object requested by a query. All requests are grouped in batches and run together; this is indicative of the load run at the servers. The server load is classified as follows:

  Low The number of requests is kept at 10

  Medium The number of requests is kept at 100

  High The number of requests is kept at 250

  Since we didn't have an industry standard to base these values on, we chose these values based on information we received from our contacts at IBM, Art Jolin and Yang Lei.

- **Degree of replication:** To observe the scalability behavior in our small cluster size environment, we varied the degree of replication between the values 2, 3, 5 and 10. These numbers were chosen as they reflect the typical number of caches in small cluster-based networks.
- **Ratio of read to write requests:** We varied the ratio of read requests to write requests from 1:1 to 4:1 to 9:1. These ratios should reflect the typical ratios in applications where (1) the number of reads is equal to the number of writes (2) the number of reads is greater than the number of writes, and (3) the number of reads dominates the number of write requests.
- **Latency:** Is defined as end-to-end delay (measured in milliseconds) required to complete a batch of requests by a cache server.

6.3 Evaluation

The schemes of section 5 were evaluated by measuring performance in terms of average latency experienced while completing a batch of requests. The requests were in the form of SQL queries which either read or update an object. We batched the SQL queries together and ran them on a client server machine. We varied the number of requests in each batch to measure the performance of the scheme under different load settings. Specifically, for each scheme the average latency values were recorded for the low, medium and high number of requests. The ratio of read to write requests was also varied. The degree of replication of the caches was varied as well. The data points for the experiments can be found in Appendix A.

6.4 Experimental results

6.4.1 Cumulative results

Figures 4 through 6 combine the results of all the schemes and compare the average latency times experienced when a client server completes read and update requests in the ratio 1:1. These graphs are indicative of a low load of 10 requests, a medium load of 100 requests and a high load of 250 requests running on the server respectively.

Figures 7 through 9 combine the results of all the schemes and compare the average latency times experienced when a client server completes requests of reads and writes in the ratio 4:1. These graphs are indicative of a low load of 10 requests, a medium load of 100 requests and a high load of 250 requests running on the server respectively.

Figures 10 through 12 combine the results of all the schemes and compare the average latency times experienced when a client server completes requests of reads and writes in the ratio 9:1. These graphs are indicative of a low load of 10 requests, a medium load of 100 requests and a high load of 250 requests running on the server respectively.

6.4.2 Comparison of three mutex locking schemes

Figures 13 through 15 show a comparison of the three mutex locking schemes for the average latency times experienced when a client server completes read and write requests in the ratio 1:1 for varying thread pool sizes. From these three figures, we can see that the medium sized thread pool performed better than the small and high size thread pools.

Figures 16 through 24 show a comparison of the three mutex locking schemes. Performance of caching with the write-behind approach is maximized when the ratio of the number of read requests to write requests is high. The mutex locking schemes in all low, medium and high load settings give similar performance. For all scenarios, the write-behind approach performs better than locking with writes preferred over reads, the performance of the latter scheme is similar to that of locking with no preference.

6.5 Summary

We have compared four classes of schemes - single thread model, two mutex locking schemes based on primary based remote write protocol and one mutex locking with write-behind caching model, which were described in section 5. An investigation of the results indicates that mutex-based locking schemes are very close in terms of performance compared to the single thread model based scheme, which is more expensive. This is especially true when the load of requests is high. In our experiments, the single thread model performed the worst in terms of elapsed time. For our scenario and experiments, the single thread model performs similar to the baseline case when there is no caching done. We also implemented a basic version of Timestamp ordering based concurrency control and studied its performance. We found that the performance of this scheme was worse than all the other schemes. We have not included the results of it in our work.

From our experiments we conclude that a medium sized thread pool of 100 threads performs better than the low size (10 threads) and high size (250 threads) thread pools. For low loads, the performance of small and medium size thread pools were comparable. For medium and high loads, the medium size thread pool performed better than the small and high size thread pools. A possible reason why a medium size thread pool is better than a high size thread pool size is that in our experiments the requests were short-running and were handled well by the medium thread pool. In that case the overhead of having a larger thread pool size is unnecessary.

7 Conclusions and Future Work

In our experiments we examined four basic classes of schemes in distributed caching: schemes based on the single thread model, two mutex-locking schemes based on primary based remote write protocol and one mutex-locking scheme using the write behind caching model. Our experiments show that performance of the single thread model performs the worst among the schemes we compared. In our experiments it performs on average ten times worse than the other schemes. In our experiments we found that it's performance

is comparable to case where no caching is used. The single thread model is not scalable and cannot take advantage of multi-CPU systems.

The scheme which used the cache write-behind approach consistently performed better than all the other schemes. However the scheme has to have other special built in communication if the database is changed by some other application which is not using the cache. We have implemented this built in communication by using timestamps, similarly to what is implemented in Tangosol [28]. A mechanism for cache failure should also be in place. Performance of caching using the write-behind approach performs the best when the ratio of the number of read requests to the number of write requests is high. This scheme is most suitable for applications where access to the database is for the most part through the cache.

Locking with writes preferred over reads has better performance than the other scheme, when there is a large number of read requests and very few write requests (e.g. the ratio of 9:1 in our experiments)

In our experiments we used various thread pool sizes which corresponded to the number of requests. The use of an optimally sized thread pool strategy removes thread creation overhead and minimizes the resource utilization of the server. Future work would be to find a method to choose an optimal thread pool size.

Our implementation of a basic version of the timestamp ordering scheme performed worse than the single thread model. In future, the performance of a more advanced implementation of timestamp ordering needs to be studied and compared with the other schemes.

These schemes which we implemented for small-size cluster networks support either sequential consistency or weak consistency with synchronization variables. These schemes scale sufficiently well when the degree of replication is increased up to 10 caches. A future line of study would be to compare these schemes for larger networks.

A Data Points for Latency Measurements

All the schemes described in section 5 were tested for small, medium and high batch loads. The tests were run for 10 vs. 100 vs. 250 requests on the same data object. The requests comprised three cases, where the ratio of reads to writes was 1:1, 4:1 and 9:1. Each scheme was tested for 2, 3, 5 and 10 caches. The results are summarized as below. For the three schemes using the thread pool model, the number of threads in the pool varied from 10 to 100 to 250.

Scheme	Latency for 2 caches	Latency for 3 caches	Latency for 5 caches	Latency for 10 caches
Mutex locking with write preferred over read (threadpool of 10 threads)	146.8	216.53	306.88	536.14
Mutex locking with write preferred over read (threadpool of 100 threads)	114	170.73	252.48	384.72
Mutex locking with write preferred over read (threadpool of 250 threads)	226.6	265.66	338.8	451.88
Mutex locking with no preference (threadpool of 10 threads)	165.5	223.13	293.68	505.6
Mutex locking with no preference (threadpool of 100 threads)	109.5	183.3	241.28	331.62
Mutex locking with no preference (threadpool of 250 threads)	231.2	302.06	351.32	409.7
Single thread model	984.3	1444.73	2364.52	4411.26
Mutex locking with write-behind cache (threadpool of 10 threads)	121.9	214.6	297.56	464.06
Mutex locking with write-behind cache (threadpool of 100 threads)	111	138.6	185.68	275.64
Mutex locking with write-behind cache (threadpool of 250 threads)	190.7	235.22	317.4	369.34

Table 1: Results for 10 requests with write/read request ratio of 1:1 (Latency is measured in milliseconds)

Scheme	Latency for 2 caches	Latency for 3 caches	Latency for 5 caches	Latency for 10 caches
Mutex locking with write preferred over read (threadpool of 10 threads)	1079.6	1592.6	2637.52	5098.42
Mutex locking with write preferred over read (threadpool of 100 threads)	525.1	700	876.28	1380.9
Mutex locking with write preferred over read (threadpool of 250 threads)	1568.6	2021.86	2015.73	2298.5
Mutex locking with no preference (threadpool of 10 threads)	1079.5	1662.46	2687.48	5308.1
Mutex locking with no preference (threadpool of 100 threads)	631.5	813.46	1067.4	2092.12
Mutex locking with no preference (threadpool of 250 threads)	1612.6	1989.6	1975.68	2304.02
Single thread model	10247	15404.2	25677.32	50784.08
Mutex locking with write-behind cache (threadpool of 10 threads)	1071.8	1585.53	2597.56	5075.6
Mutex locking with write-behind cache (threadpool of 100 threads)	386	433.33	514.96	746.28
Mutex locking with write-behind cache (threadpool of 250 threads)	1481.4	1887.46	1941.45	2185.58

Table 2: Results for 100 requests with write/read request ratio of 1:1 (Latency is measured in milliseconds)

Scheme	Latency for 2 caches	Latency for 3 caches	Latency for 5 caches	Latency for 10 caches
Mutex locking with write preferred over read (threadpool of 10 threads)	2610.9	3900.06	6401.8	12569.06
Mutex locking with write preferred over read (threadpool of 100 threads)	1039.1	1277.93	1971.24	2877.14
Mutex locking with write preferred over read (threadpool of 250 threads)	3662.5	4365.6	4628.84	5510.84
Mutex locking with no preference (threadpool of 10 threads)	2703.2	4016.53	6633.16	13179.02
Mutex locking with no preference (threadpool of 100 threads)	1259.3	1678.2	2311.16	4620.56
Mutex locking with no preference (threadpool of 250 threads)	3629.8	3295.8	4187.48	4833.14
Single thread model	25807.8	38573.2	63370.56	124650.84
Mutex locking with write-behind cache (threadpool of 10 threads)	2567.1	3853.13	6316.2	12439.63
Mutex locking with write-behind cache (threadpool of 100 threads)	750.1	830.13	900.08	1512.14
Mutex locking with write-behind cache (threadpool of 250 threads)	3342.1	4270.71	4592.6	4870.9

Table 3: Results for 250 requests with write/read request ratio of 1:1 (Latency is measured in milliseconds)

Scheme	Latency for 2 caches	Latency for 3 caches	Latency for 5 caches	Latency for 10 caches
Mutex locking with write preferred over read (threadpool of 10 threads)	148.5	210.46	288.84	511.02
Mutex locking with write preferred over read (threadpool of 100 threads)	121.8	170.8	204.44	268.4
Mutex locking with write preferred over read (threadpool of 250 threads)	223.4	297.93	339.24	409.92
Mutex locking with no preference (threadpool of 10 threads)	143.9	214.6	301.24	508.78
Mutex locking with no preference (threadpool of 100 threads)	140.7	151.93	242.48	342.46
Mutex locking with no preference (threadpool of 250 threads)	229.6	271.86	383.76	535.96
Single thread model	976.3	1455.26	2341.4	4445.64
Mutex locking with write-behind cache (threadpool of 10 threads)	128.1	197.8	287.55	501.975
Mutex locking with write-behind cache (threadpool of 100 threads)	117.3	131.13	201.24	262.56
Mutex locking with write-behind cache (threadpool of 250 threads)	217.1	259.5	336.53	391.56

Table 4: Results for 10 requests with write/read request ratio of 1:4 (Latency is measured in milliseconds)

Scheme	Latency for 2 caches	Latency for 3 caches	Latency for 5 caches	Latency for 10 caches
Mutex locking with write preferred over read (threadpool of 10 threads)	1085.9	1613.66	2585.64	5081.5
Mutex locking with write preferred over read (threadpool of 100 threads)	498.4	637.66	709.96	1054.08
Mutex locking with write preferred over read (threadpool of 250 threads)	1508.6	1744.8	2381.2	2385.55
Mutex locking with no preference (threadpool of 10 threads)	1096.8	1638.46	2739.36	5274.68
Mutex locking with no preference (threadpool of 100 threads)	634.4	712.29	1111.28	2217.12
Mutex locking with no preference (threadpool of 250 threads)	1592.1	1499.93	2001.88	2107.14
Single thread model	10234.5	15402.13	25901.36	50768.96
Mutex locking with write-behind cache (threadpool of 10 threads)	1076.5	1596.8	2583.4	5057.3
Mutex locking with write-behind cache (threadpool of 100 threads)	370.2	426	639.15	988.43
Mutex locking with write-behind cache (threadpool of 250 threads)	1456.2	1645.814	2264.85	2316.9

Table 5: Results for 100 requests with write/read request ratio of 1:4 (Latency is measured in milliseconds)

Scheme	Latency for 2 caches	Latency for 3 caches	Latency for 5 caches	Latency for 10 caches
Mutex locking with write preferred over read (threadpool of 10 threads)	2587.7	3888.6	6373.24	12495.58
Mutex locking with write preferred over read (threadpool of 100 threads)	876.5	1032.4	1581.88	2044.1
Mutex locking with write preferred over read (threadpool of 250 threads)	3722.375	4601.83	3783.16	3071.88
Mutex locking with no preference (threadpool of 10 threads)	2696.8	4026.88	6627.48	13027.62
Mutex locking with no preference (threadpool of 100 threads)	1274.9	1714.73	2318.8	6701.28
Mutex locking with no preference (threadpool of 250 threads)	3957.7	3605.2	3520.44	5251.52
Single thread model	25815.6	38640.46	63258.16	125592.22
Mutex locking with write-behind cache (threadpool of 10 threads)	2582.9	3876.06	6331.26	12414.53
Mutex locking with write-behind cache (threadpool of 100 threads)	796.8	944.08	1179.06	1669.625
Mutex locking with write-behind cache (threadpool of 250 threads)	3648.33	4569.44	3772.55	3046.825

Table 6: Results for 250 requests with write/read request ratio of 1:4 (Latency is measured in milliseconds)

Scheme	Latency for 2 caches	Latency for 3 caches	Latency for 5 caches	Latency for 10 caches
Mutex locking with write preferred over read (threadpool of 10 threads)	158	218.73	299.92	478.44
Mutex locking with write preferred over read (threadpool of 100 threads)	142	180.06	166.84	282.16
Mutex locking with write preferred over read (threadpool of 250 threads)	201.7	263.53	340	579.04
Mutex locking with no preference (threadpool of 10 threads)	162.6	179.33	312.56	523.72
Mutex locking with no preference (threadpool of 100 threads)	120.2	151.93	243.68	320.52
Mutex locking with no preference (threadpool of 250 threads)	212.5	280.13	389.36	475.68
Single thread model	976.5	1458.33	2342.44	4491.24
Mutex locking with write-behind cache (threadpool of 10 threads)	123.5	199.06	296.2	476.075
Mutex locking with write-behind cache (threadpool of 100 threads)	117.3	148	164.15	274.12
Mutex locking with write-behind cache (threadpool of 250 threads)	195.4	221.66	338.72	451.58

Table 7: Results for 10 requests with write/read request ratio of 1:9 (Latency is measured in milliseconds)

Scheme	Latency for 2 caches	Latency for 3 caches	Latency for 5 caches	Latency for 10 caches
Mutex locking with write preferred over read (threadpool of 10 threads)	1056.2	1620.8	2612.33	5093.12
Mutex locking with write preferred over read (threadpool of 100 threads)	454.7	632.26	624.4	919.54
Mutex locking with write preferred over read (threadpool of 250 threads)	1494.35	1875.82	2332.44	2307.7
Mutex locking with no preference (threadpool of 10 threads)	1106.2	1662.6	2726.88	5349.4
Mutex locking with no preference (threadpool of 100 threads)	589.1	855.06	1371.2	2283.42
Mutex locking with no preference (threadpool of 250 threads)	1695.2	1539.66	1668.12	2281.04
Single thread model	10234.6	15393.86	25722	50944.68
Mutex locking with write-behind cache (threadpool of 10 threads)	1051.6	1601	2611.24	5067.975
Mutex locking with write-behind cache (threadpool of 100 threads)	381.3	427.06	590	852.52
Mutex locking with write-behind cache (threadpool of 250 threads)	1481.4	1867.92	2280.35	2146.95

Table 8: Results for 100 requests with write/read request ratio of 1:9 (Latency is measured in milliseconds)

Scheme	Latency for 2 caches	Latency for 3 caches	Latency for 5 caches	Latency for 10 caches
Mutex locking with write preferred over read (threadpool of 10 threads)	2596.8	3875.13	6353.76	12343.12
Mutex locking with write preferred over read (threadpool of 100 threads)	848.5	898.86	1115.6	1634.94
Mutex locking with write preferred over read (threadpool of 250 threads)	3798.875	4244.8	5410.72	3290.975
Mutex locking with no preference (threadpool of 10 threads)	2688.9	4031.13	6612.48	12970.36
Mutex locking with no preference (threadpool of 100 threads)	1300	1689.66	2485.68	5869.06
Mutex locking with no preference (threadpool of 250 threads)	3726.5	3889.6	3099.44	4955.34
Single thread model	25862.5	38576.93	63229.36	124528.54
Mutex locking with write-behind cache (threadpool of 10 threads)	2587.4	3871.86	6330.35	12280.23
Mutex locking with write-behind cache (threadpool of 100 threads)	828.16	833.46	1086	1601.25
Mutex locking with write-behind cache (threadpool of 250 threads)	3642.83	3782.11	4664.44	3152.28

Table 9: Results for 250 requests with write/read request ratio of 1:9 (Latency is measured in milliseconds)

References

[1] Java hash map implementation. http://java.sun.com/j2se/1.4.2/docs/api/java/util /HashMap.html.

[2] Oracle Application Server 10g. http://www.oracle.com/technology/software/products/ias/devuse.html.

[3] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. In *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, pages 58–105. 1996.

[4] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[5] Kyle Brown. The distributed cache pattern. IBM Software Services for WebSphere.

[6] Randal C. Burns, Robert M. Rees, and Darrell D. E. Long. Efficient data distribution in a web server farm. *IEEE Internet Computing*, 5(4):56–65, 2001.

[7] JBoss Cache. http://www.jboss.com/products/jbosscache.

[8] Michael J. Carey, Michael J. Franklin, Miron Livny, and Eugene J. Shekita. Data caching tradeoffs in client-server dbms architectures. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 357–366, New York, NY, USA, 1991. ACM Press.

[9] EJB concepts. http://java.sun.com/products/ejb/.

[10] J2EE concepts. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/.

[11] J. Hu, I. Pyarali, and D. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks, 1997.

[12] James C. Hu, Sumedh Mungee, and Douglas C. Schmidt. Techniques for developing and measuring high performance web servers over high speed ATM networks. In *INFOCOM (3)*, pages 1222–1231, 1998.

[13] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[14] Pooja Kohli and Rada Y. Chirkova. Cache invalidation and propagation in distributed caching. Technical report, NCSU.

[15] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Java Technology.* Sun BluePrints Program. 2000.

[16] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. *ACM Trans. Comput. Syst.*, 12(2):123–164, 1994.

[17] Markus Pizka Oliver Theel. Distributed caching and replication. Thirty-second Annual Hawaii International Conference on System Sciences-Volume 8, 1999.

[18] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[19] BEA WebLogic Server. http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic/server.

[20] JBoss Application Server. http://www.jboss.com/products/jbossas.

[21] WebSphere Application Server. http://www-306.ibm.com/software/webservers/appserv/was/.

[22] Akara Sucharitakul. Optimizing entity beans. http://java.sun.com/developer/ technicalArticles/ebeans/sevenrules/.

[23] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[24] Renu Tewari, Michael Dahlin, Harrick Vin, and John Kay. Beyond hierarchies: Design considerations for distributed caching on the Internet. In *IEEE ICDCS'99*, 1999.

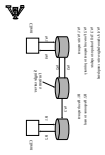
[25] Alexander Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Computing Surveys*, 30(1):70–119, 1998.

[26] Jia Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, 1999.

[27] Thomas Wang. Java thread programming: Implement read and write locks. http://www.concentric.net/ Ttwang/tech/rwlock.htm.

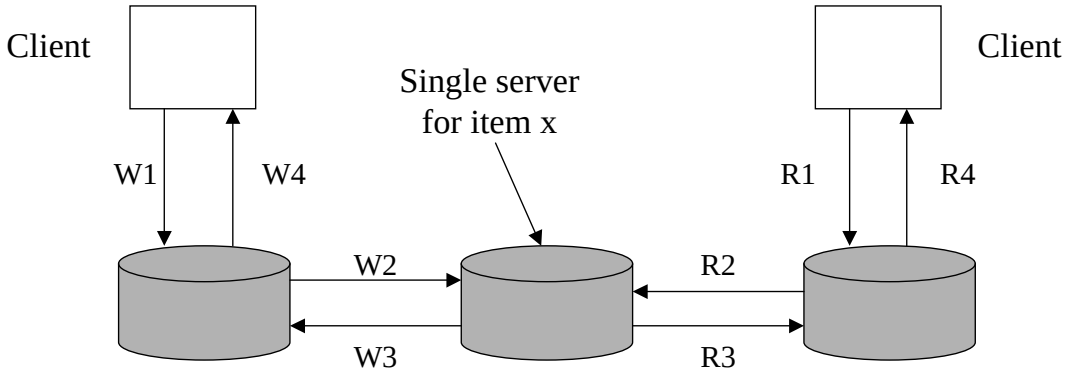
[28] www.tangosol.com. http://www.tangosol.com/coherence-featureguide.pdf.

[29] Lin K. Yu P., Wu K. and Son S. On real-time databases: Concurrency control and scheduling. *Proceedings of IEEE, Special Issue on Real-Time Systems*, 82(1):140–157, 1994.



Primary based remote-write protocol

Primary-based remote-write protocol with a fixed server to which all read and write operations are forwarded



W1. Write request

W2. Forward request to server for x

W3. Acknowledge write completed

W4. Acknowledge write completed

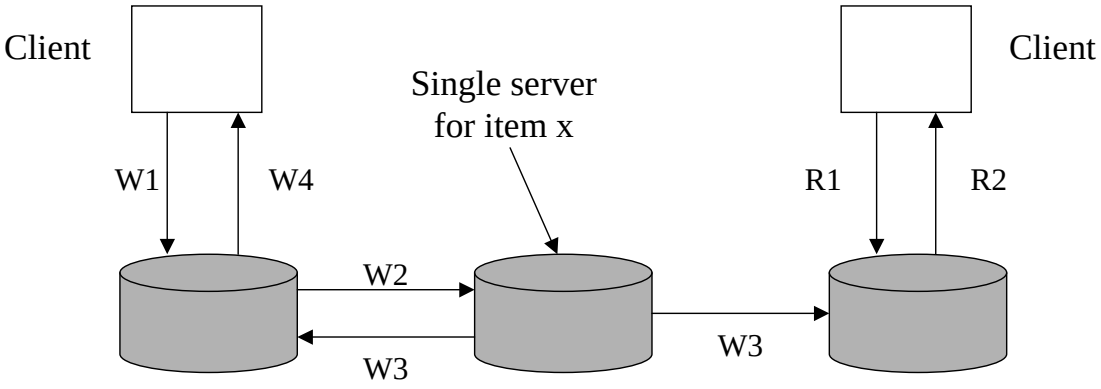
R1. Read request

R2. Forward request to server for x

R3. Return response

R4. Return response

Figure 1: Approaches based on the primary based remote write protocol



W1. Write request

W2. Forward request to primary

W3. Tell backups to update

W4. Acknowledge write completed

R1. Read request

R2. Response to read

Figure 2: Variation of a primary based remote write protocol (Tangosol version)

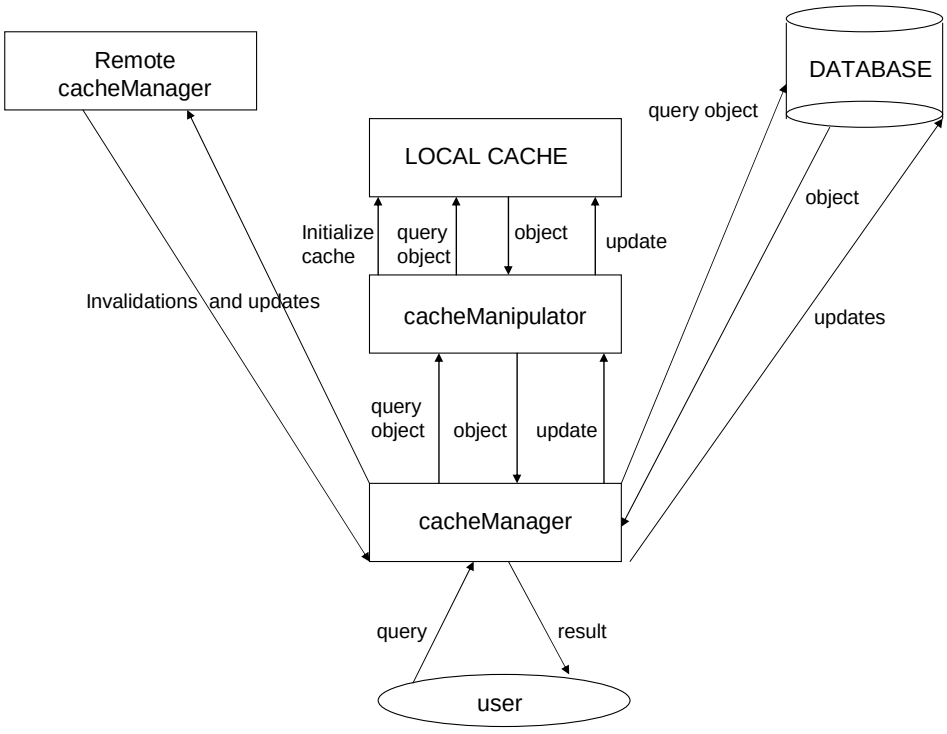


Figure 3: Interprocess communication

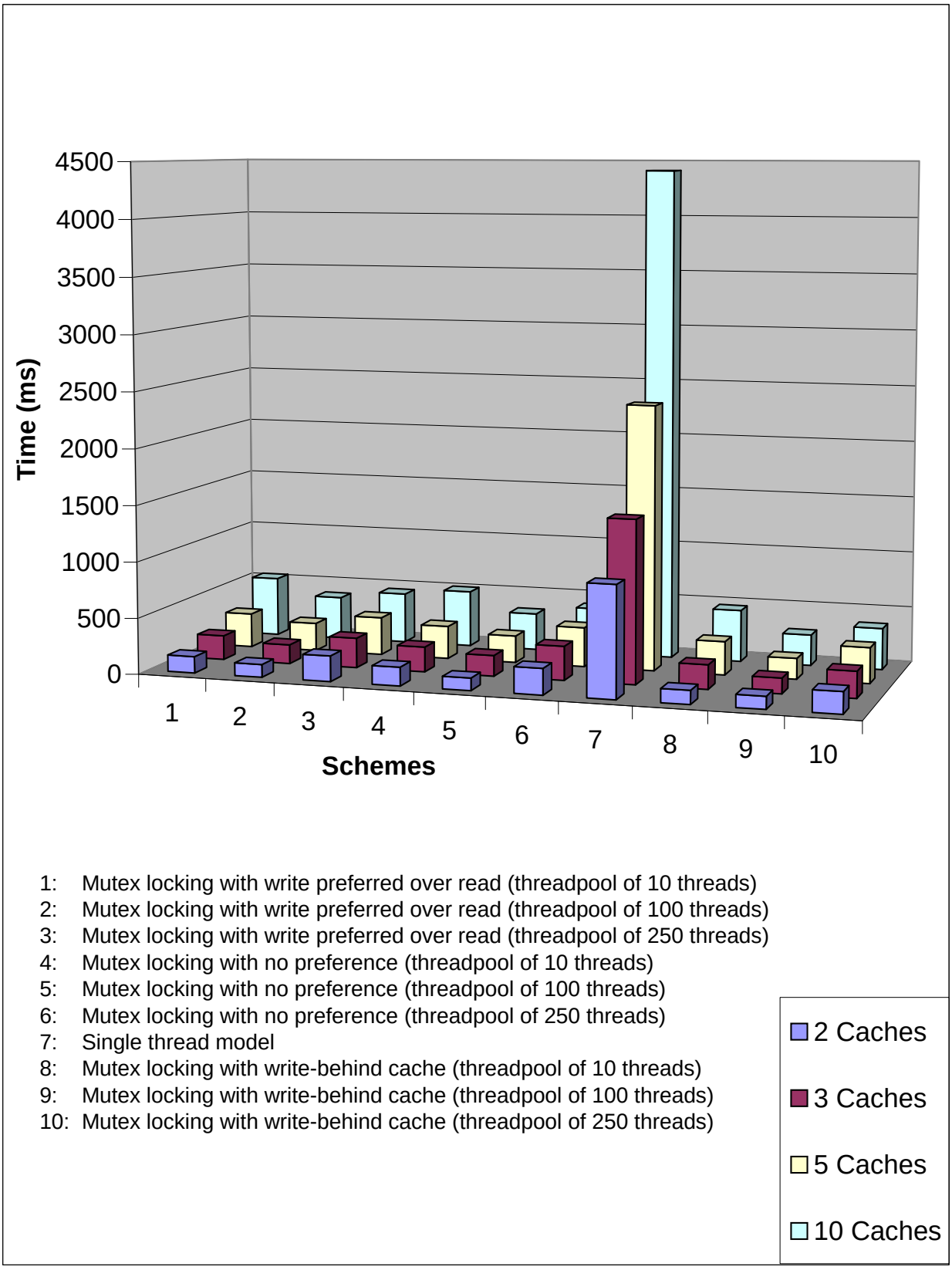


Figure 4: Consolidated results for low load: 5 writes and 5 reads

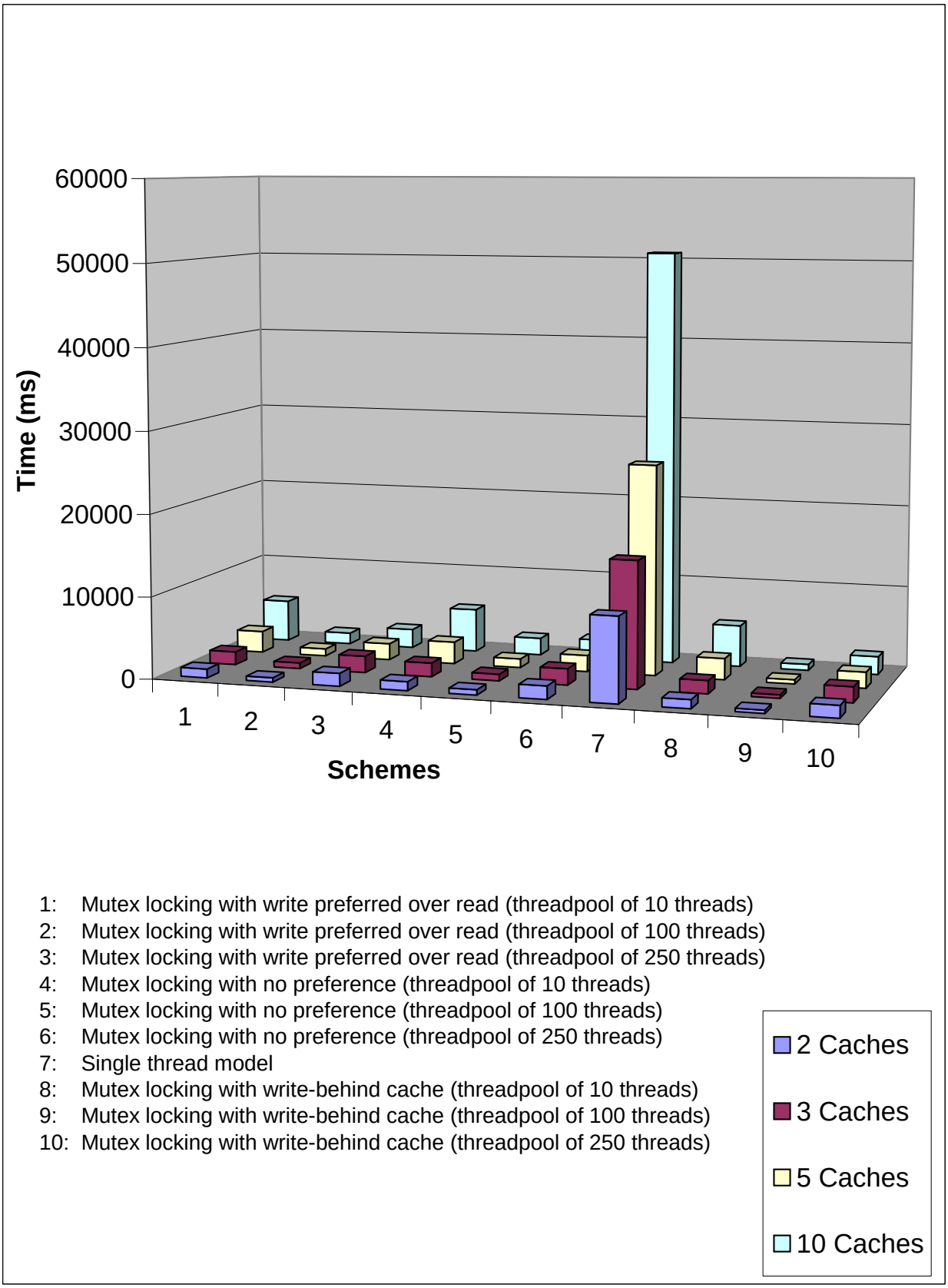


Figure 5: Consolidated results for medium load: 50 writes and 50 reads

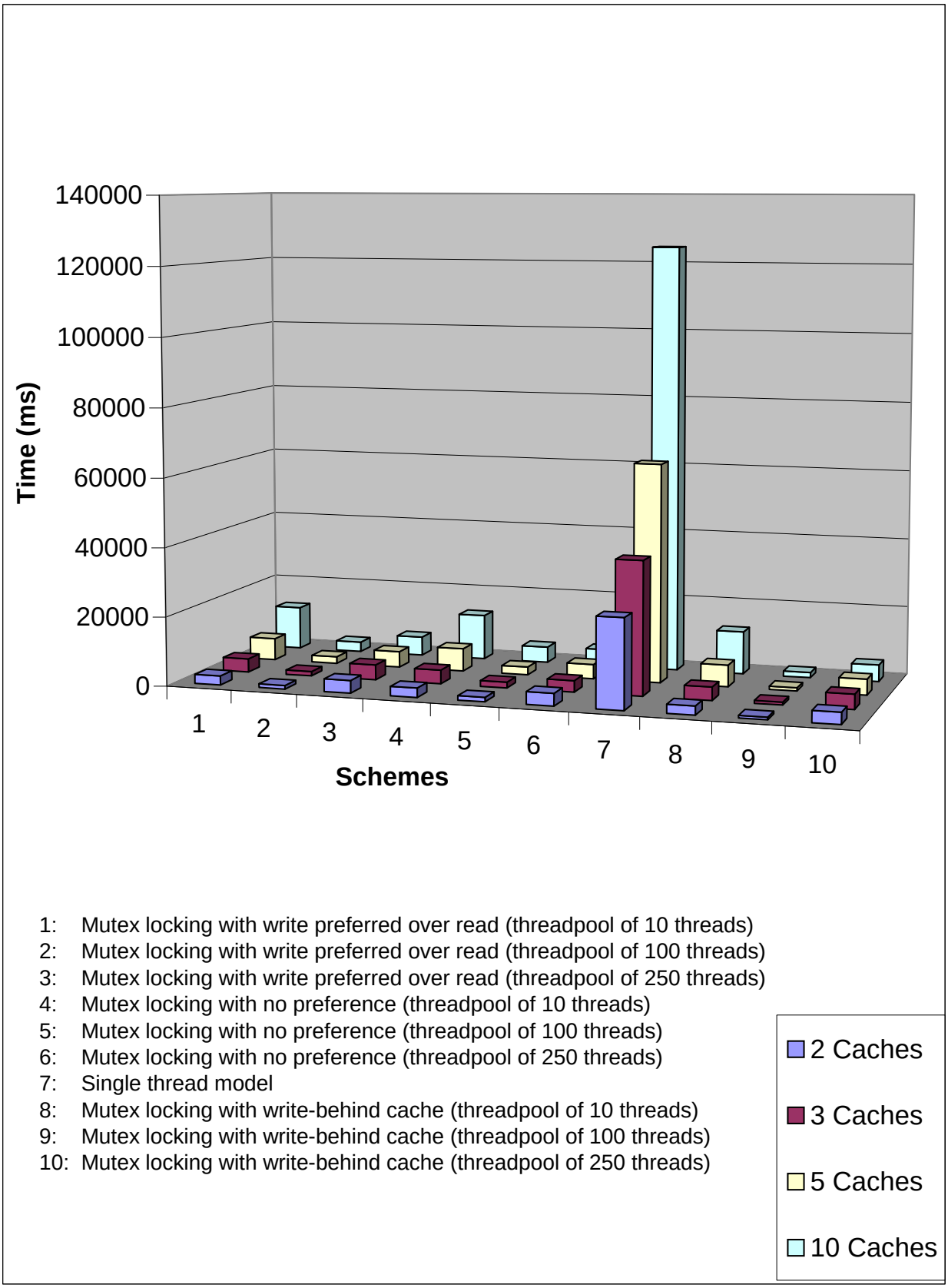


Figure 6: Consolidated results for high load: 125 writes and 125 reads

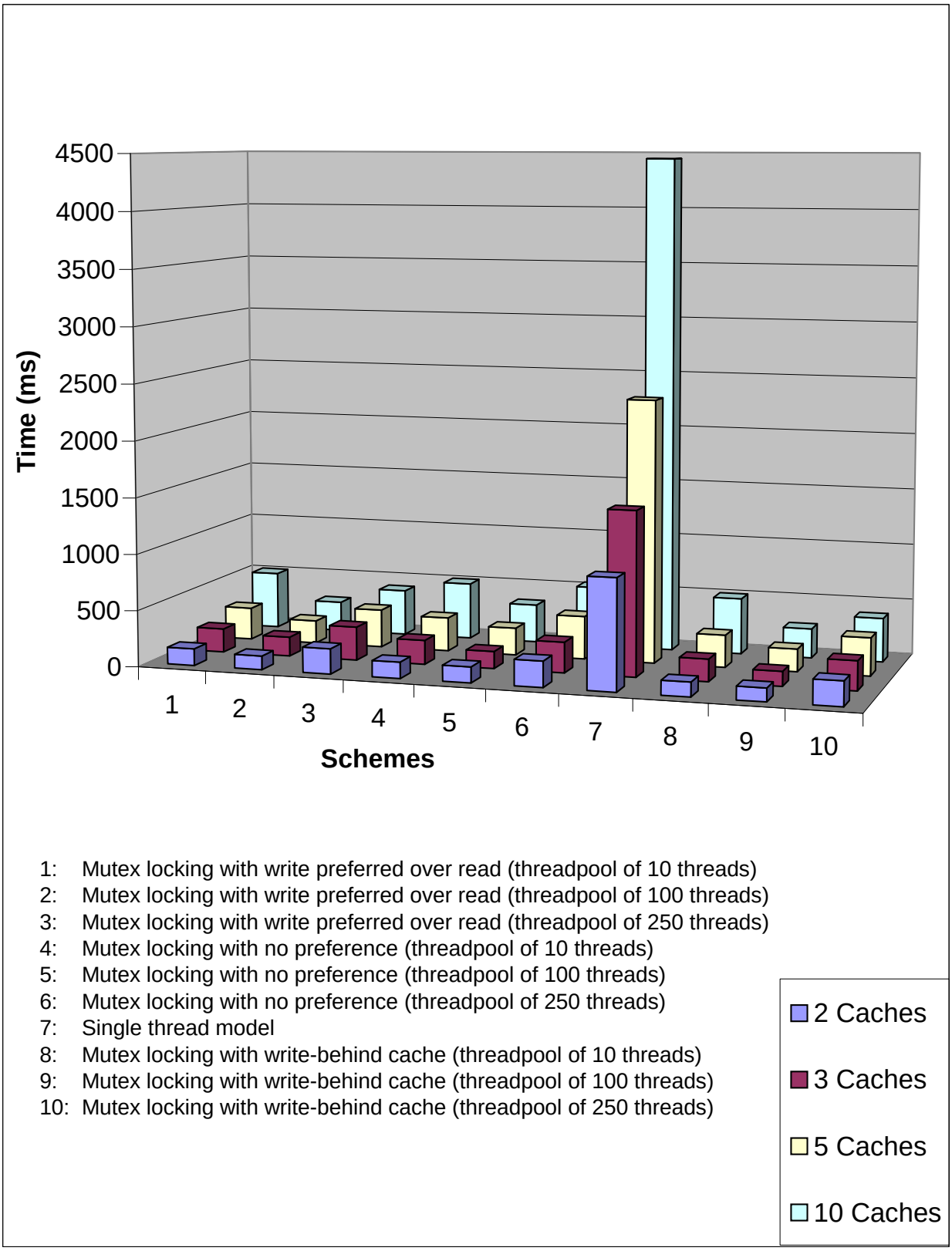


Figure 7: Consolidated results for low load: 2 writes and 8 reads

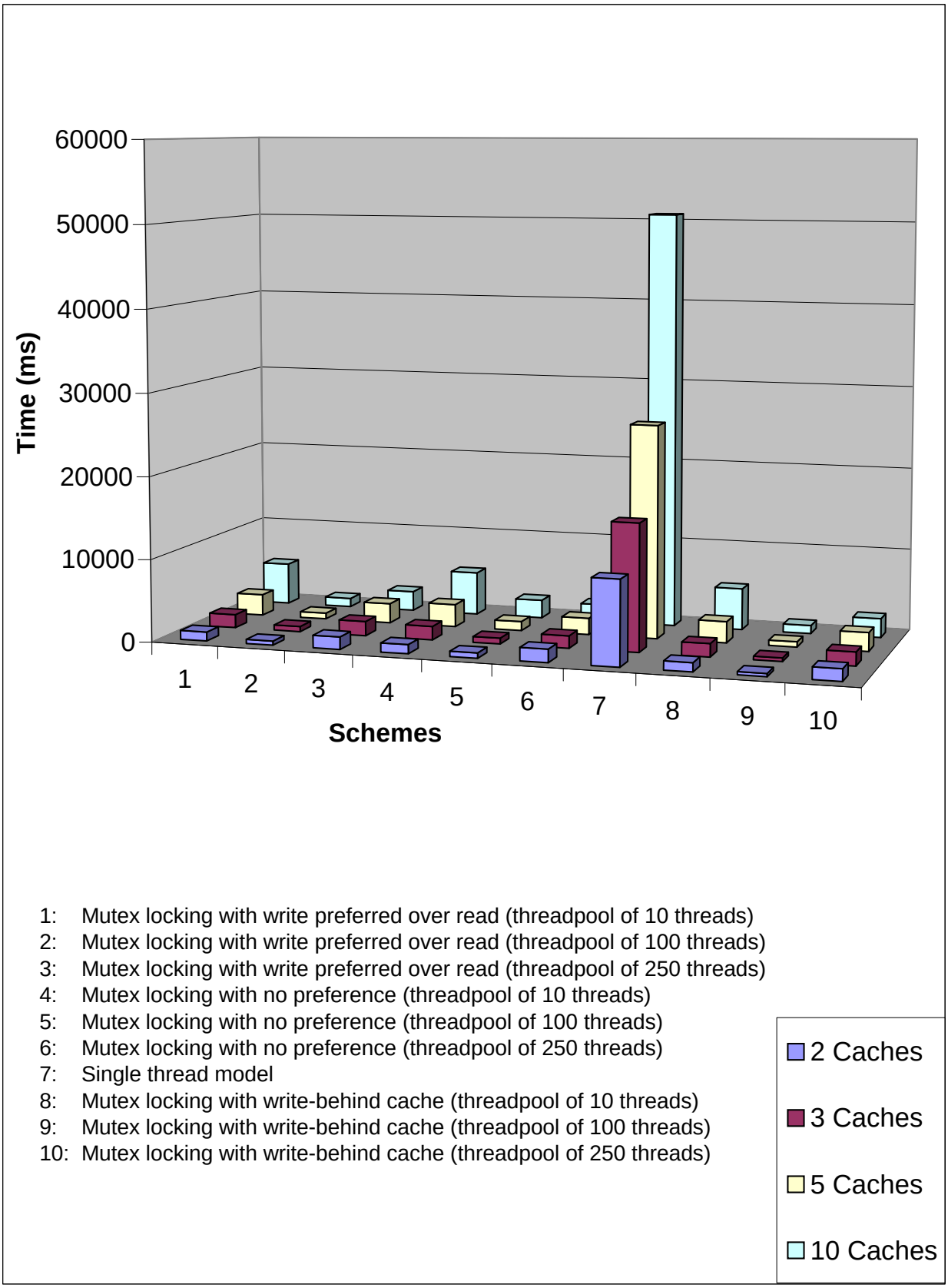


Figure 8: Consolidated results for medium load: 20 writes and 80 reads

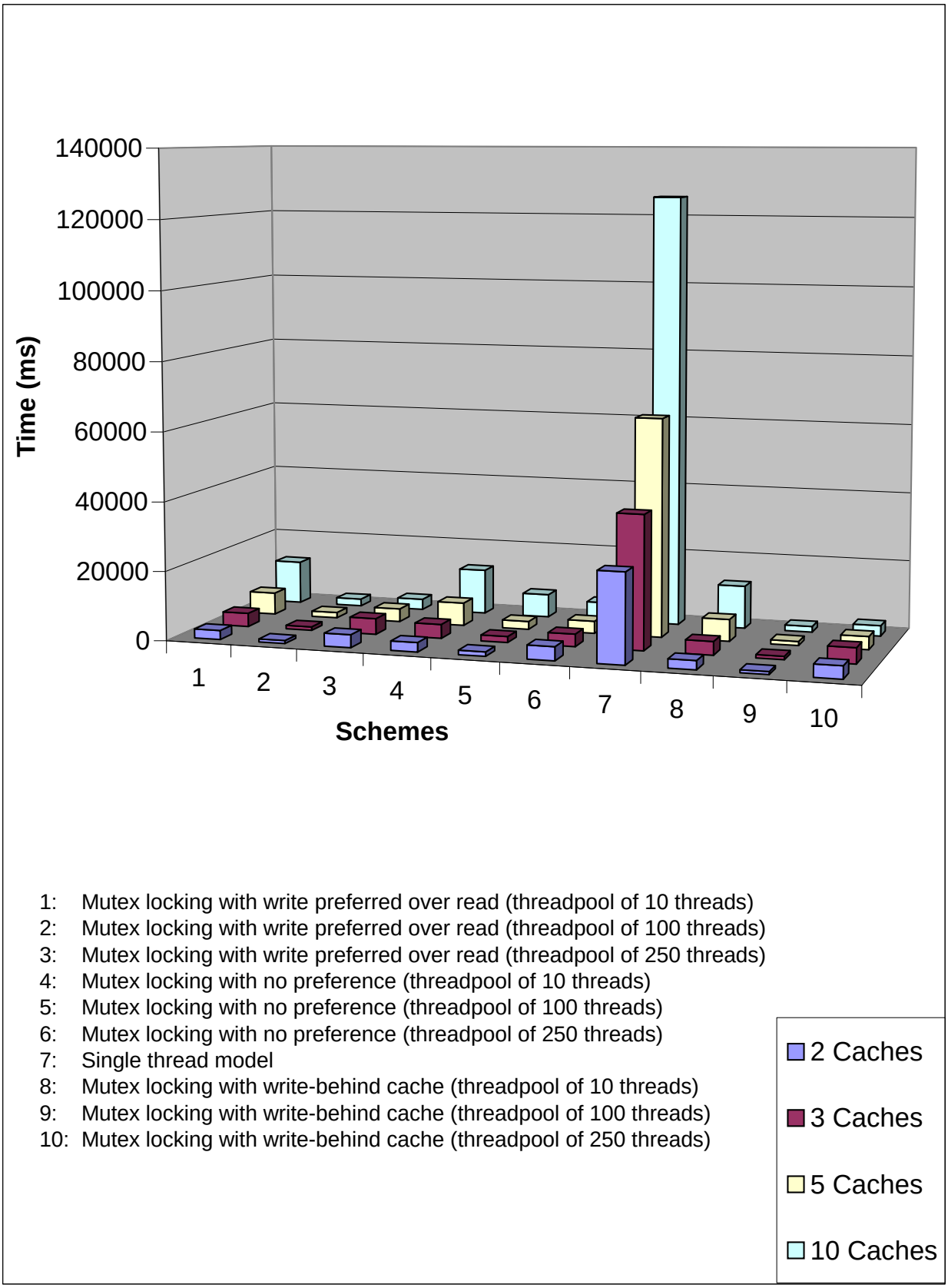


Figure 9: Consolidated results for high load: 50 writes and 200 reads

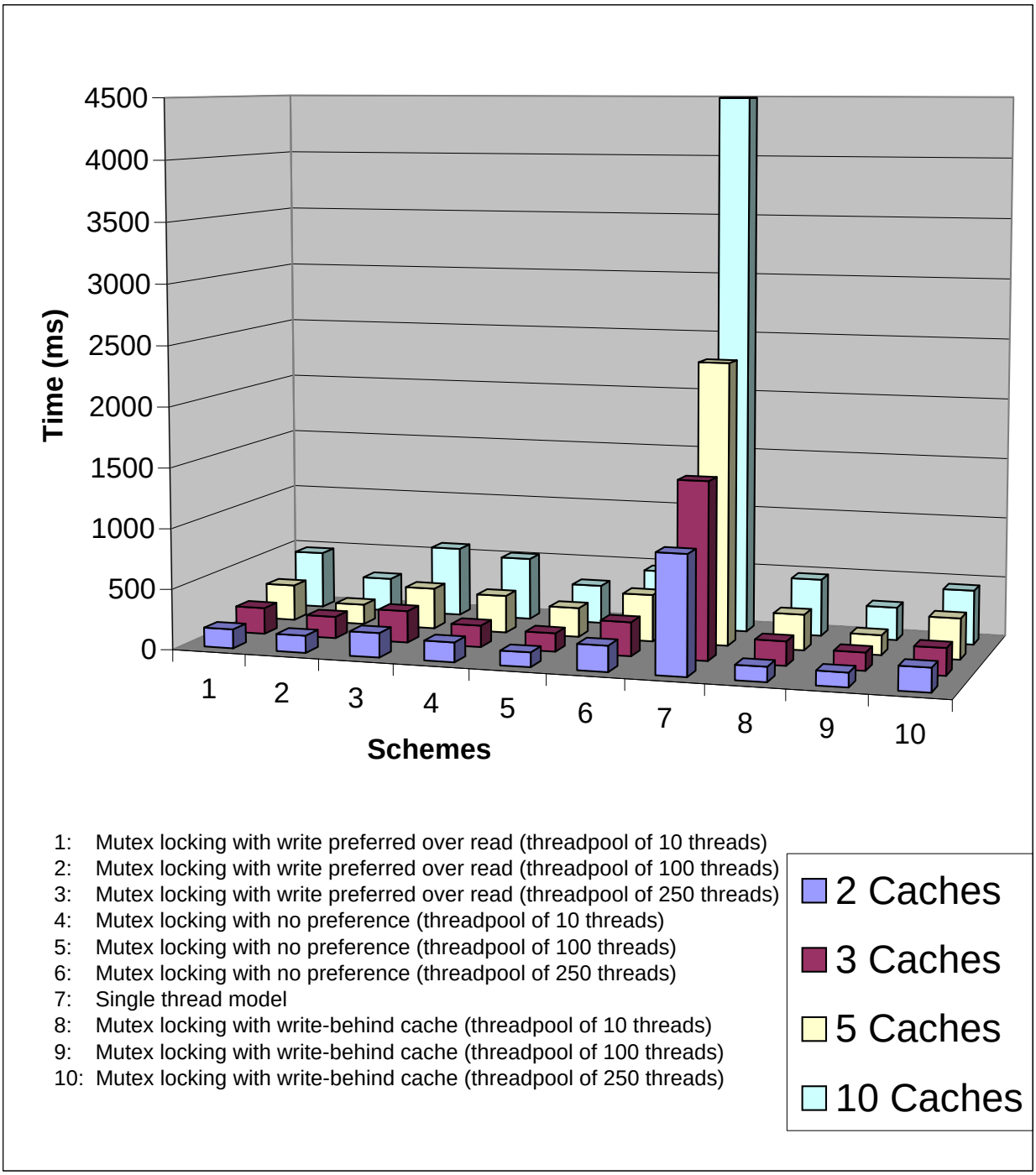


Figure 10: Consolidated results for low load: 1 write and 9 reads

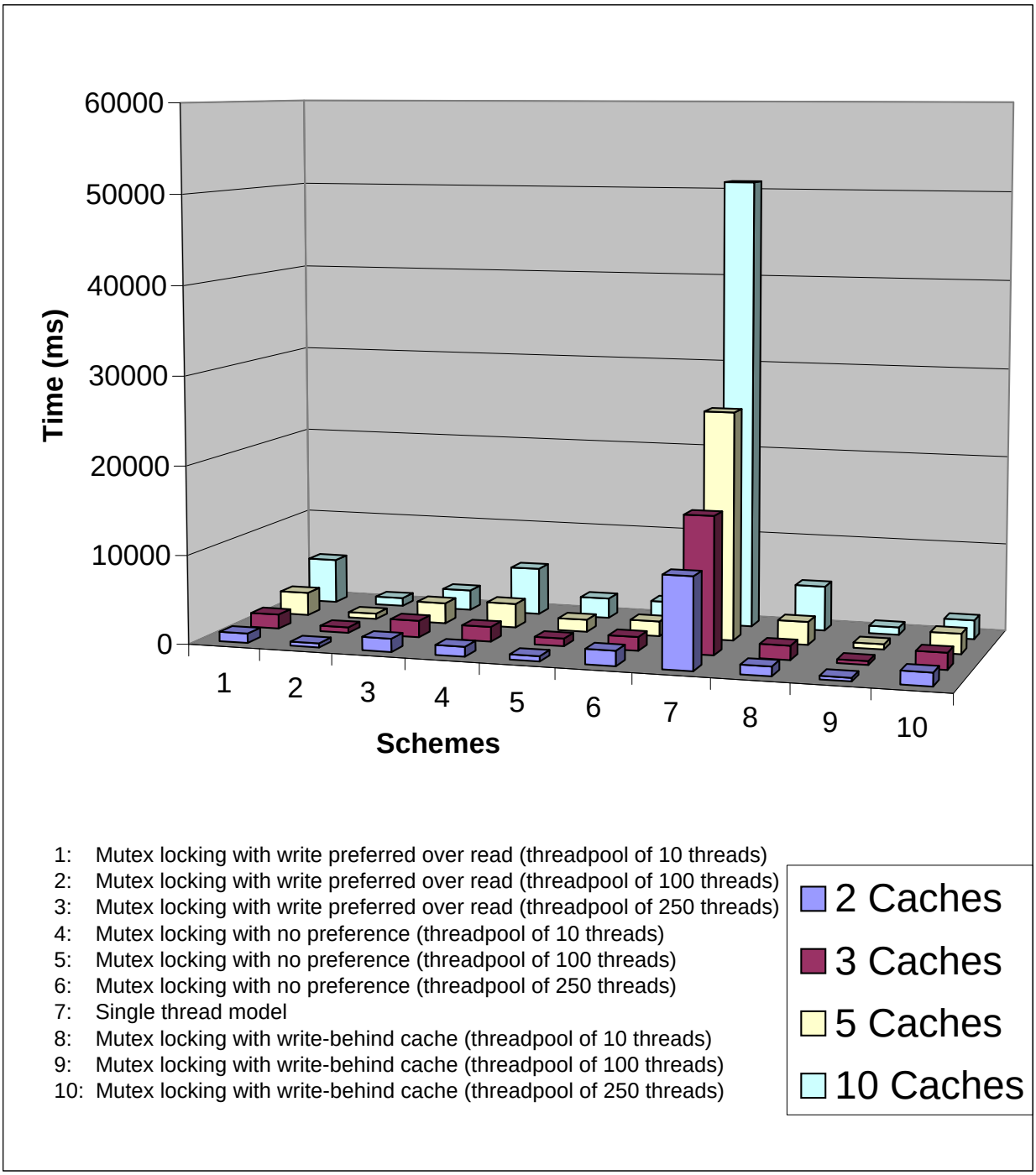


Figure 11: Consolidated results for medium load: 10 writes and 90 reads

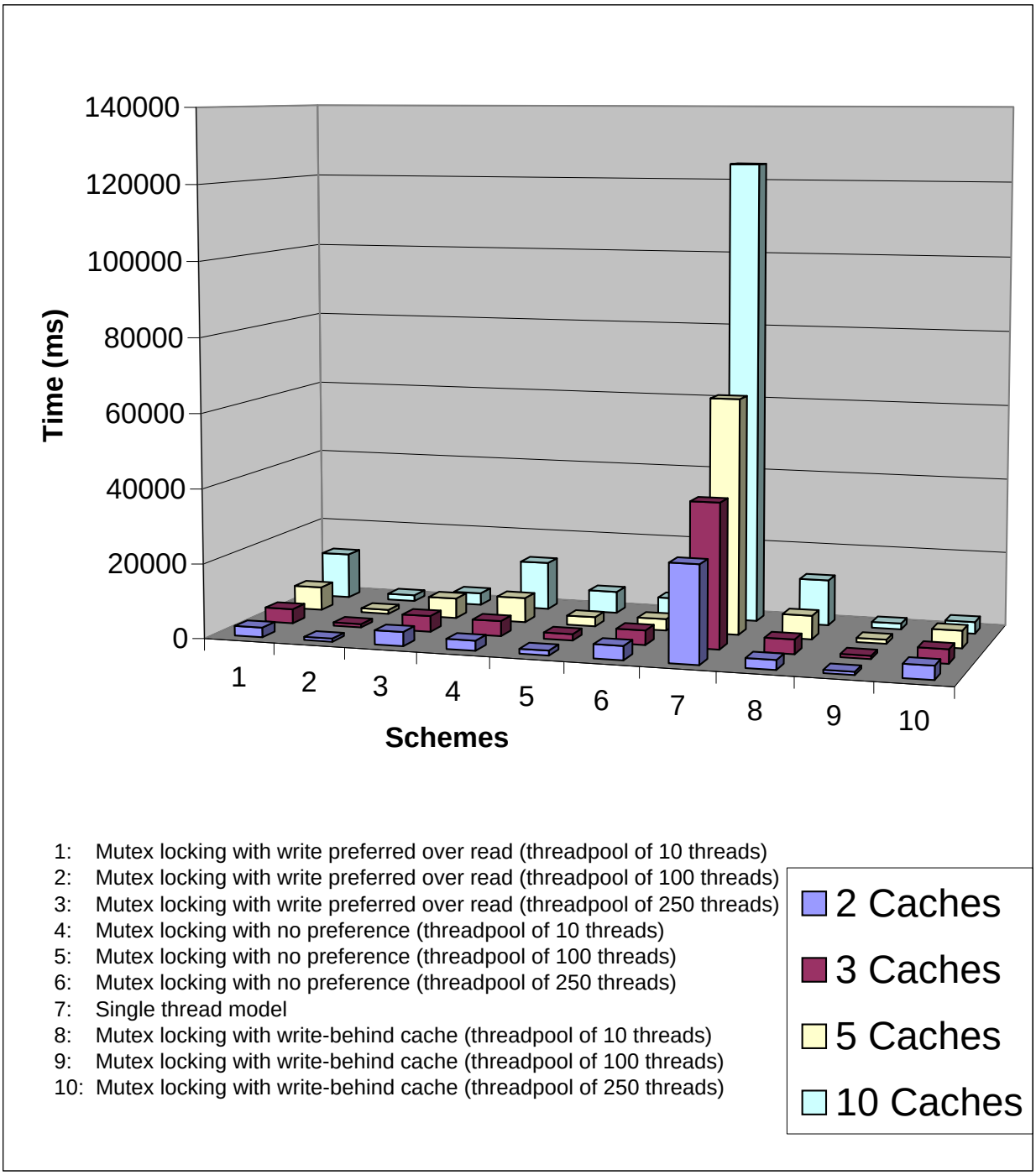


Figure 12: Consolidated results for high load: 25 writes and 225 reads

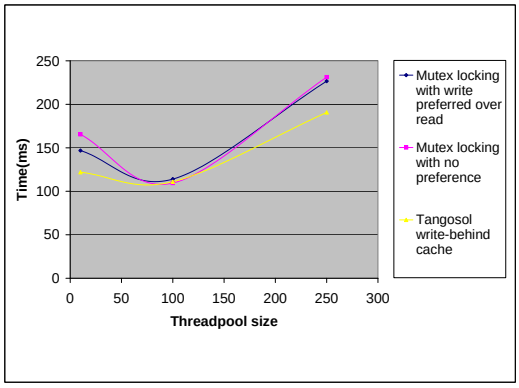


Figure 13: Comparison of three mutex locking schemes with 10 requests for write/read request ratio of 1:1

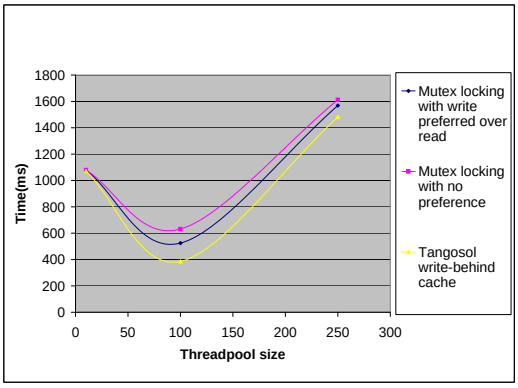


Figure 14: Comparison of three mutex locking schemes with 100 requests for write/read request ratio of 1:1

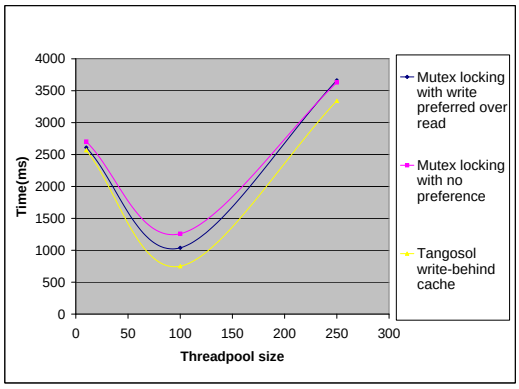


Figure 15: Comparison of three mutex locking schemes with 250 requests for write/read request ratio of 1:1

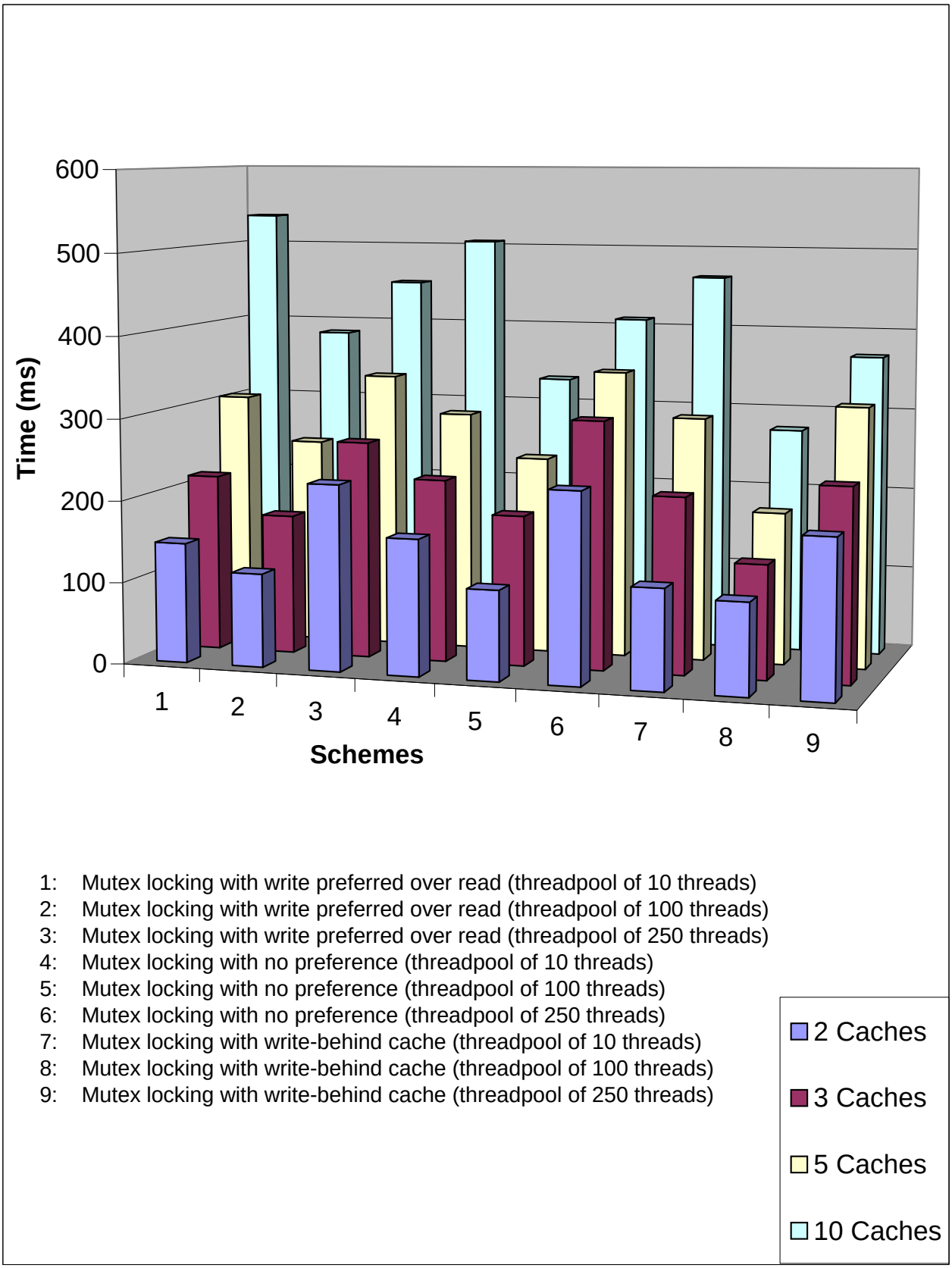


Figure 16: Consolidated results of locking schemes for low load: 5 writes and 5 reads

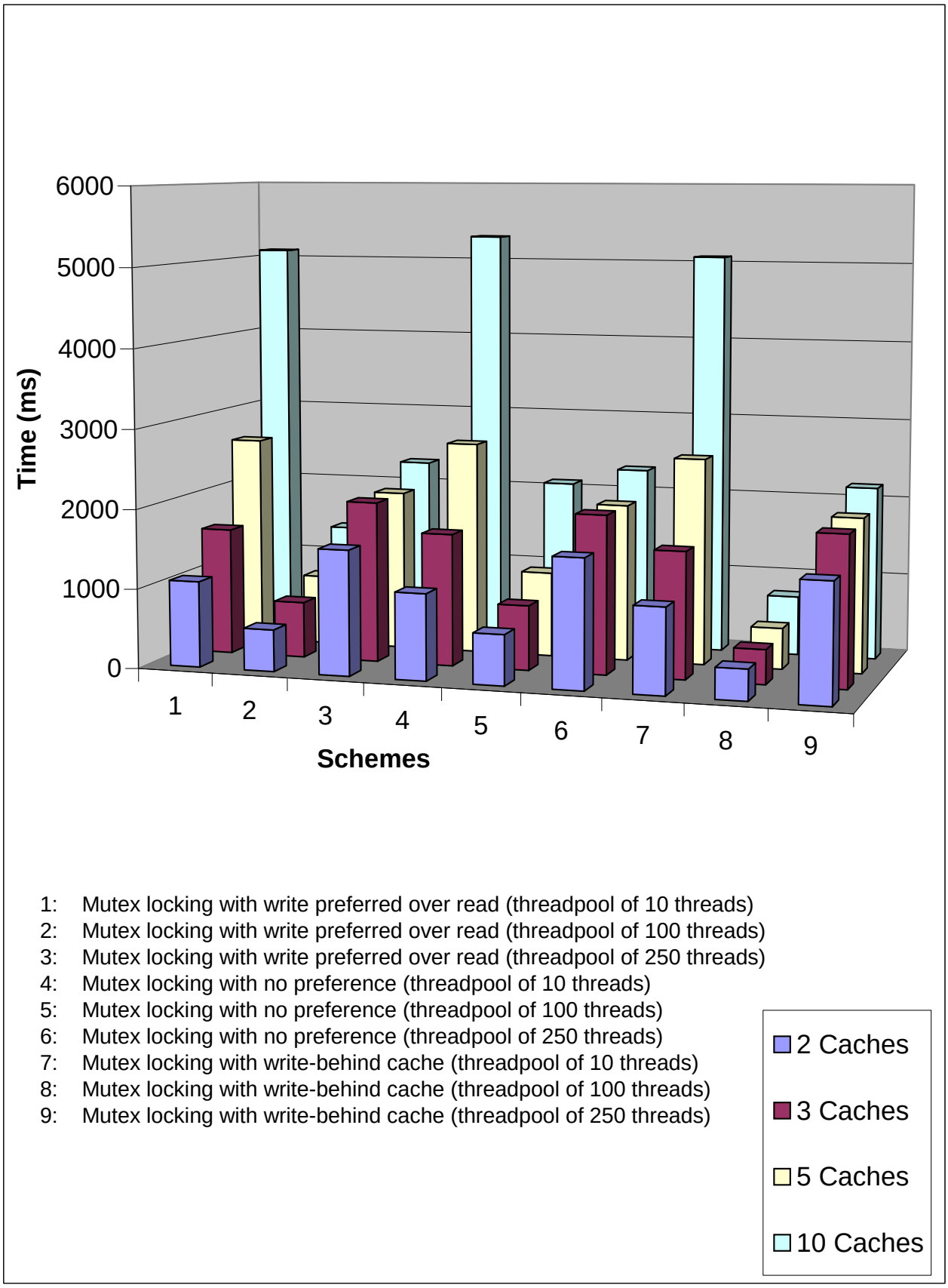


Figure 17: Consolidated results of locking schemes for medium load: 50 writes and 50 reads

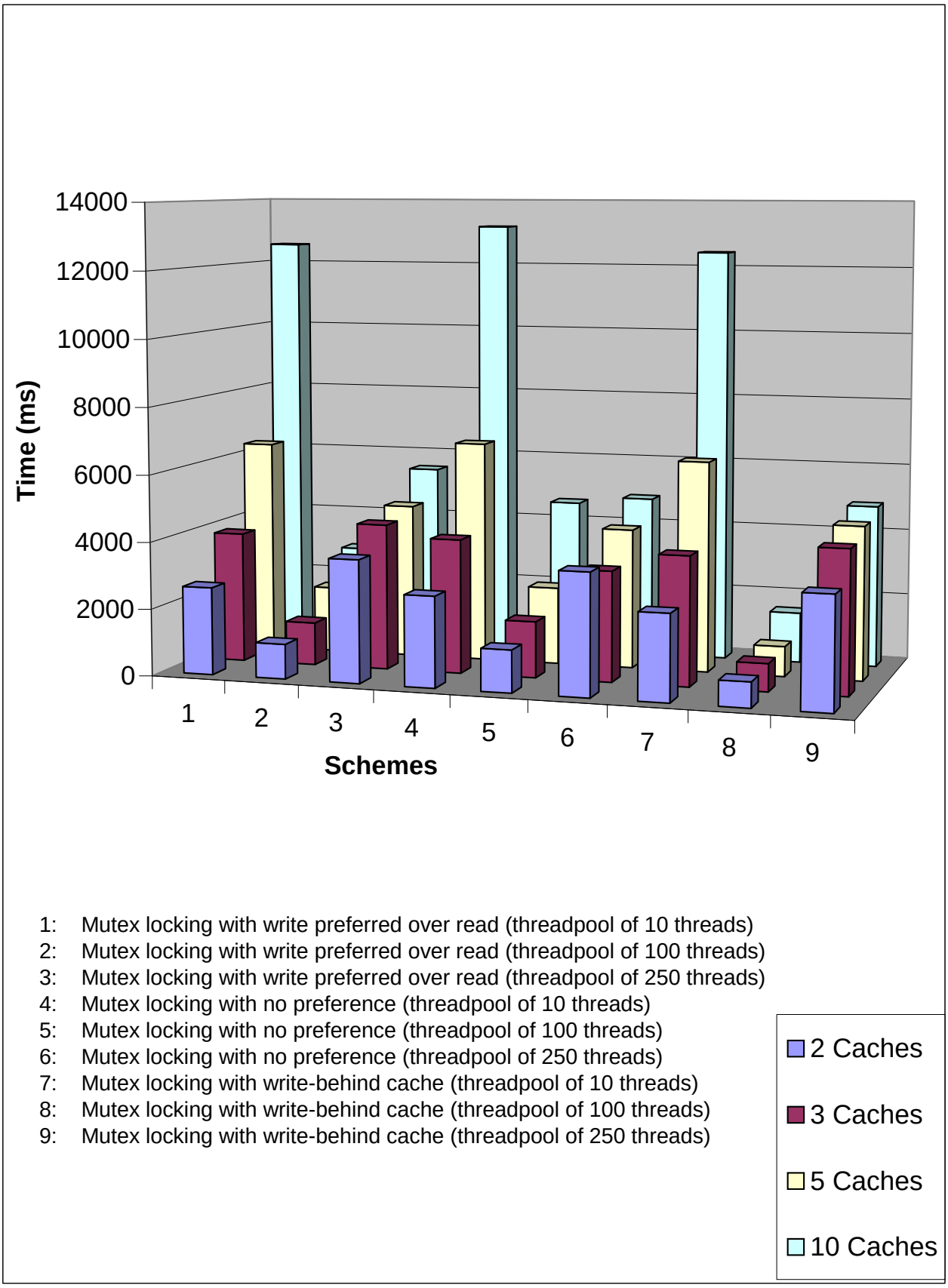


Figure 18: Consolidated results of locking schemes for high load: 125 writes and 125 reads

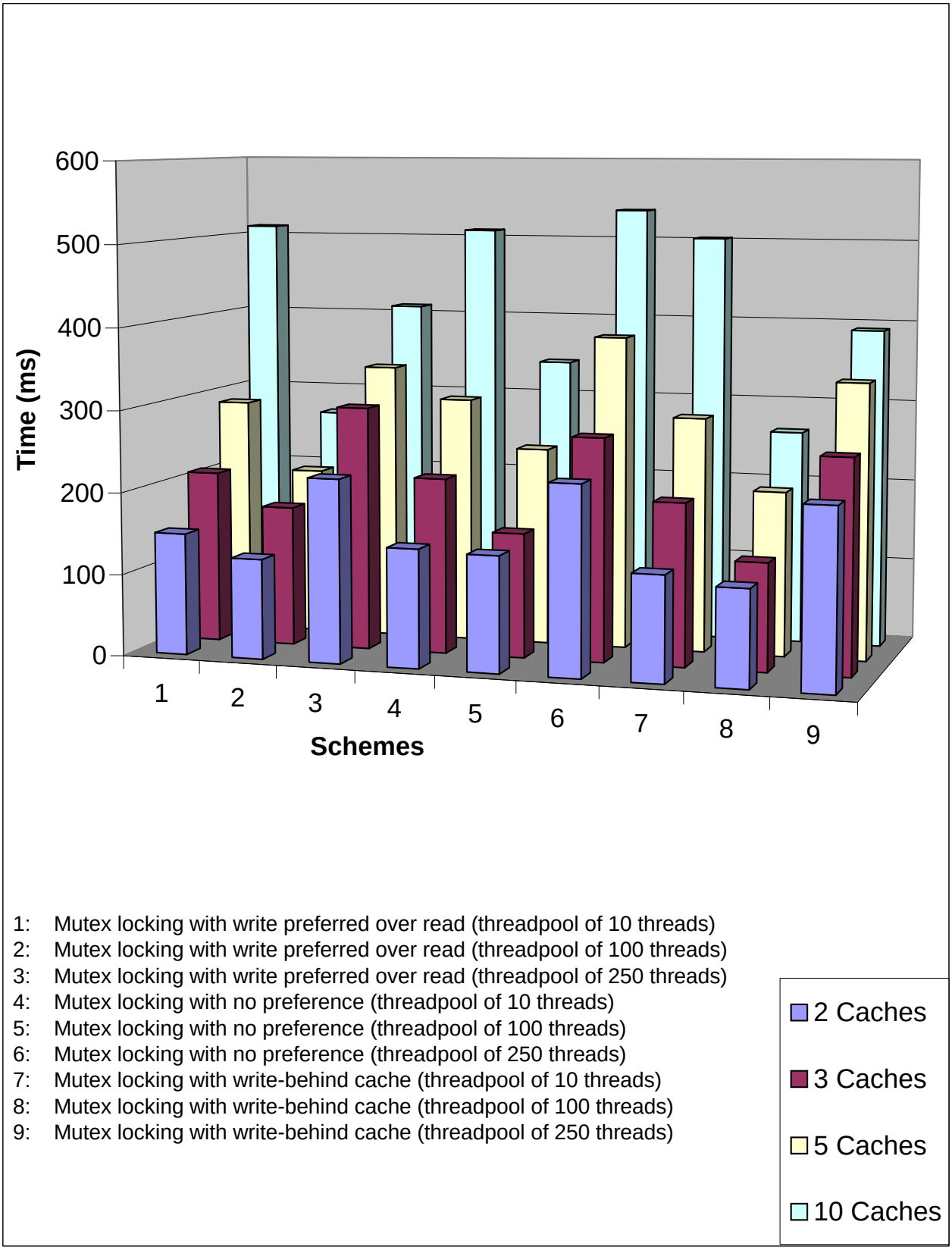


Figure 19: Consolidated results of locking schemes for low load: 2 writes and 8 reads

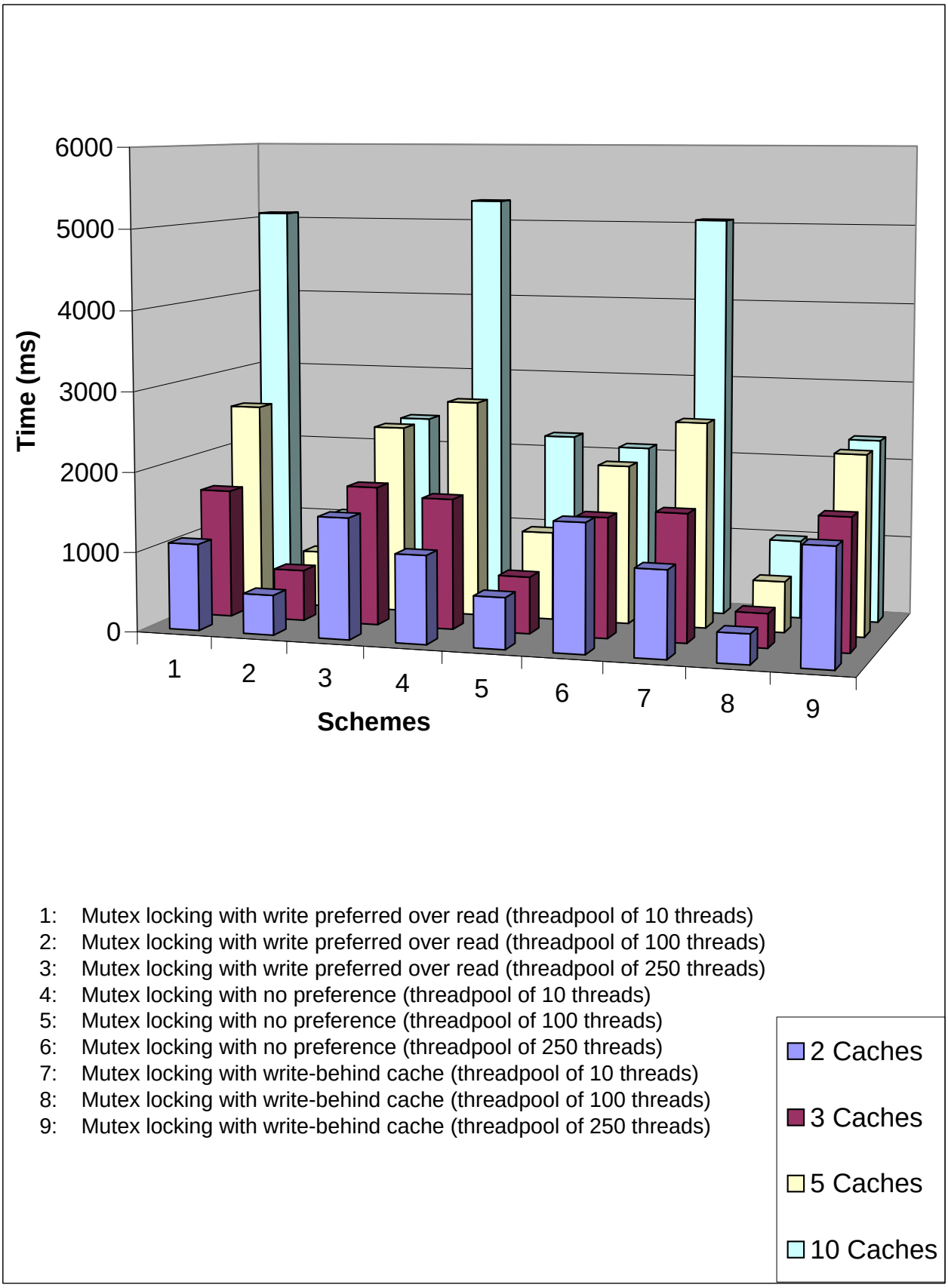


Figure 20: Consolidated results of locking schemes for medium load: 20 writes and 80 reads

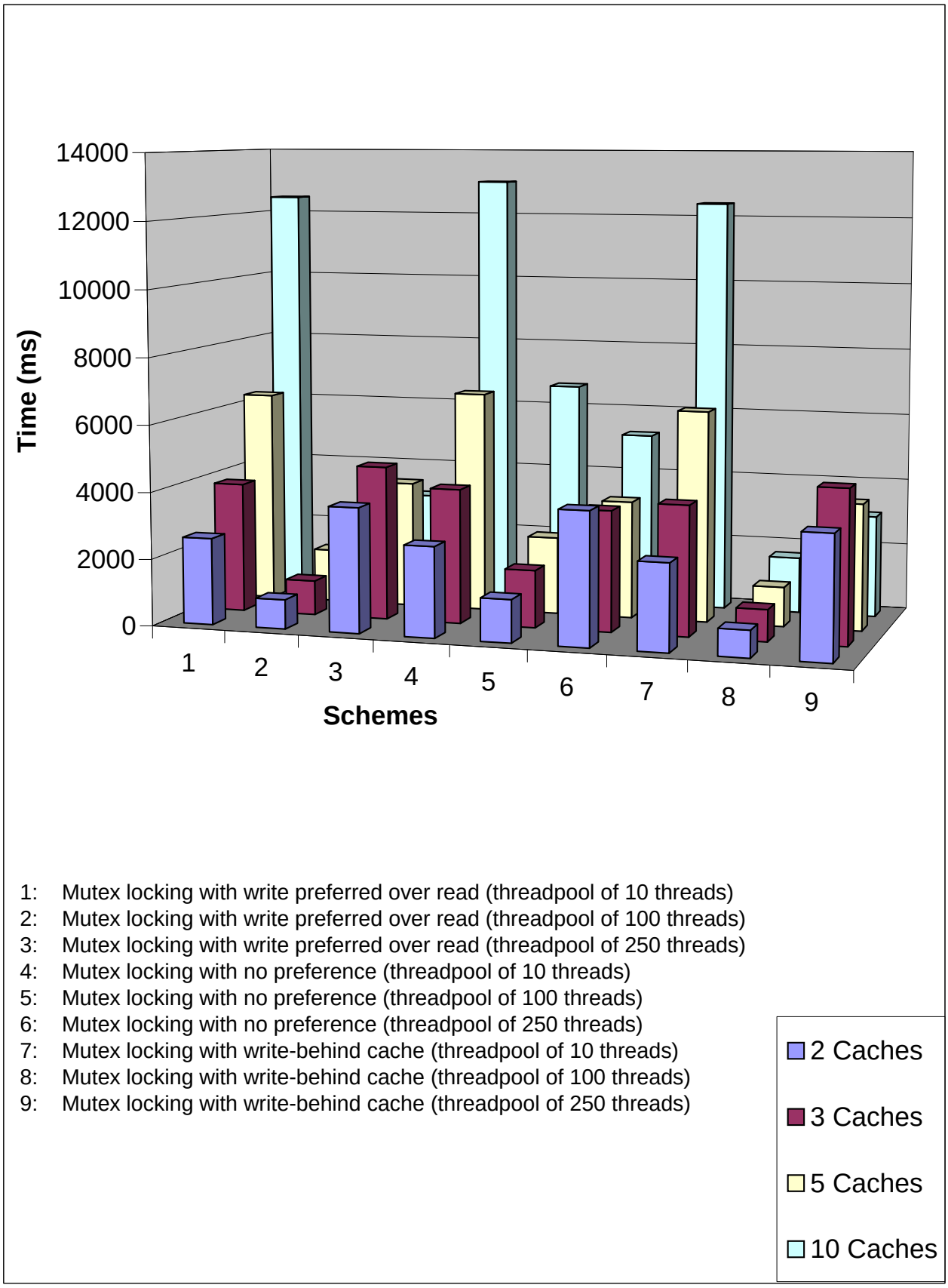


Figure 21: Consolidated results of locking schemes for high load: 50 writes and 200 reads

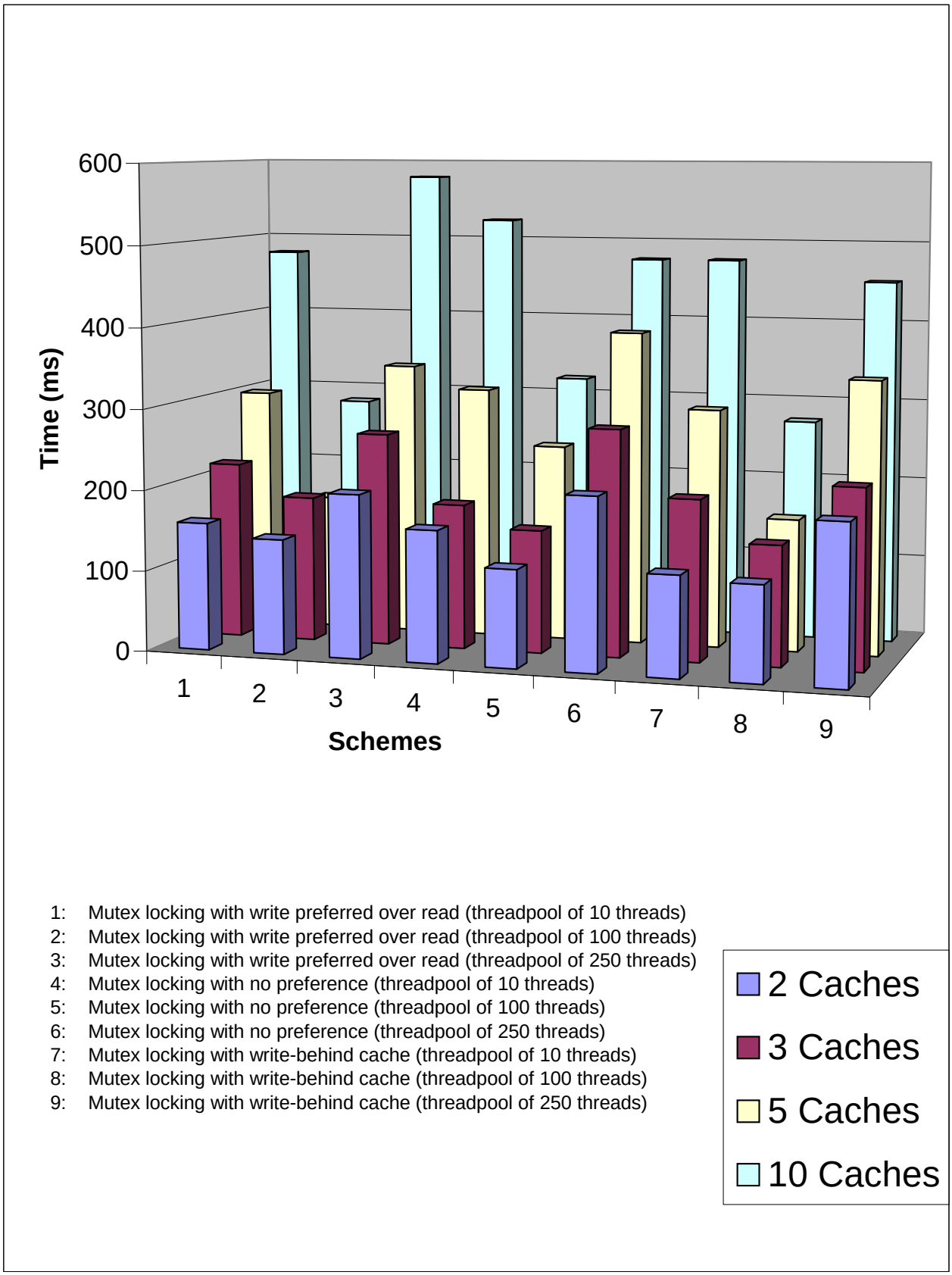


Figure 22: Consolidated results of locking schemes for low load: 1 write and 9 reads

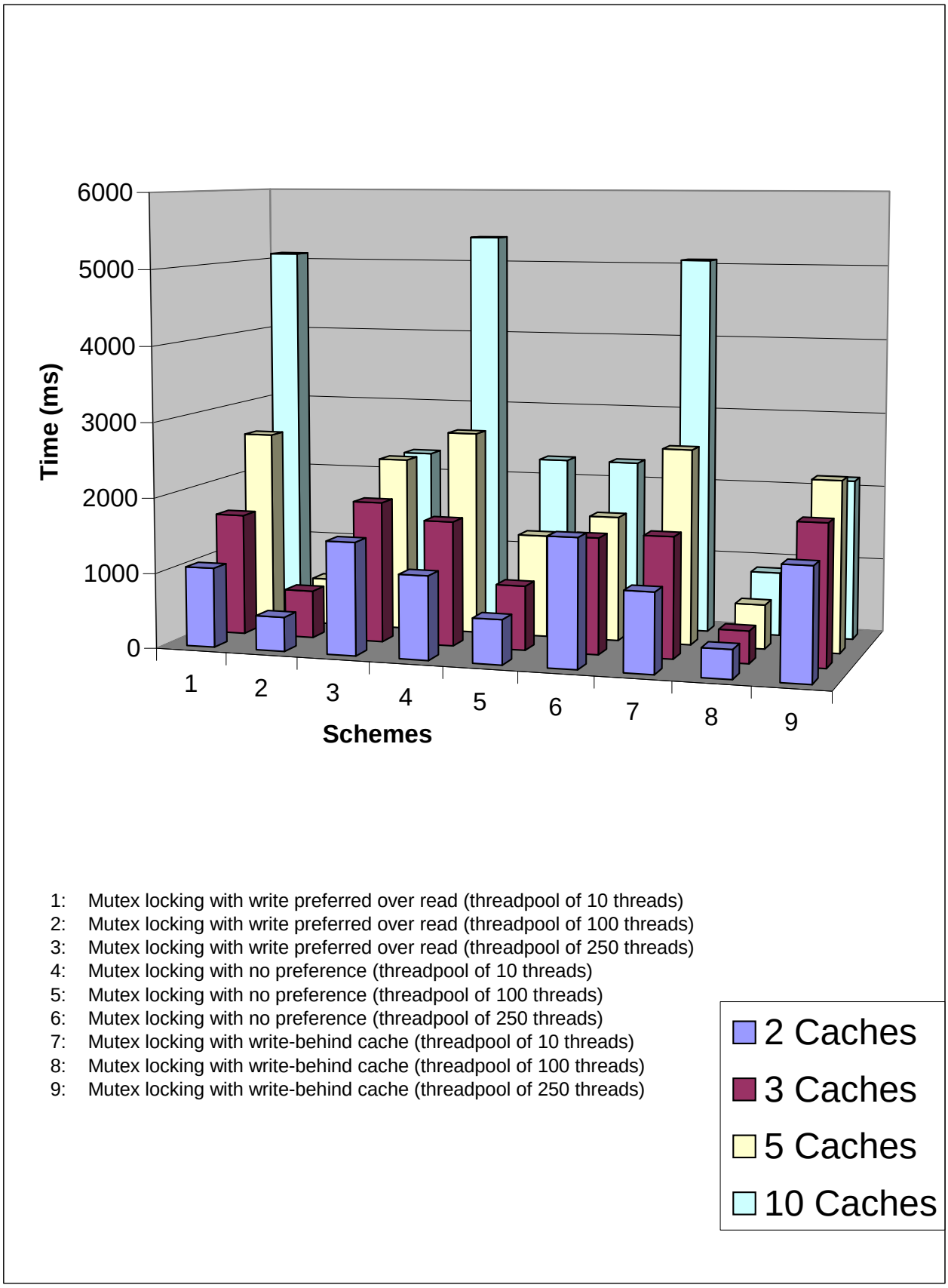


Figure 23: Consolidated results of locking schemes for medium load: 10 writes and 90 reads

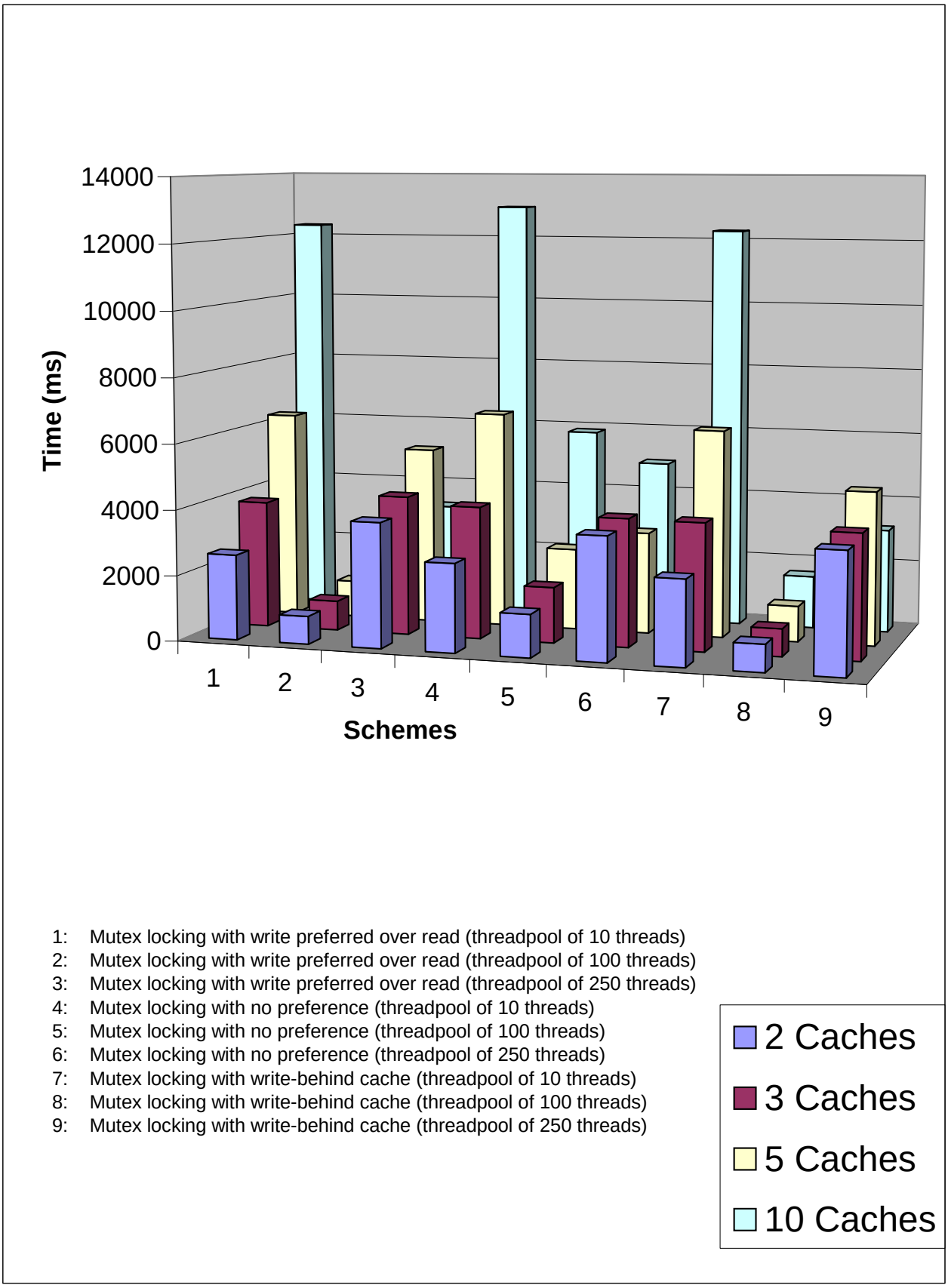


Figure 24: Consolidated results of locking schemes for high load: 25 writes and 225 reads