Logging Scheme for Transactional Memory

Laurel Emurian

August 2010

1 Abstract

Transactional memory is a speculative scheme used for managing memory contention in multiprocessing systems. In hardware transactional memory, we can use a transactional cache to store all original copies of data modified during a transaction and the modified data. In case of data contention among processors, we must abort, or revert each thread back to its original state using the original data we stored in the transactional cache. Unfortunately, the transactional cache consumes approximately 30% of the total energy used during execution, partially due to storing two copies of data. The transactional cache also frequently overflows, causing serialization and slower execution.

We are proposing the use of a transactional logging scheme to move private data out of the transactional cache. In the transactional logging scheme, we only need to save the original copy of private data in case of contention. By reducing the amount of data stored on the transactional cache, we aim to reduce overflows and energy consumed. The transactional logging scheme is more efficient than the transactional cache because it is a simple stack. We can improve the efficiency of the log by keeping track of the types of accesses we make to private data and only storing the addresses and data that we need. To do so, we use a filter cache. The filter cache understands accesses to memory, so it can estimate if we will need to restore the data. Preliminary results show that the logging scheme is promising and may reduce power while retaining performance.

2 Logging Scheme

In the current transactional scheme, all transactional data is stored on the transactional cache. The transactional cache must store two copies of data: the original data and data that a transaction may modify. If there is a transactional abort, all modified data is invalidated and the original data is restored. If there is a commit, then the backup copies are invalidiated. Since the transactional cache must hold two copies of transactional data, it consumes roughly 30% of total energy consumed during execution. The transactional cache is also subject to a large amount of overflows, which can contribute to increased cycles spent on a transactional because the execution is serialized once an overflow occurs. [1]

To combat the energy consumption and overflows of the transactional cache, we implemented the transactional logging scheme. The transactional logging scheme stores only private transactional data, allowing us to store only shared transactional data on the transactional cache. Since we only store shared data in the transactional cache, we reduce the amount of data stored in the transactional cache. We only store the original copy of the private transactional data on the log. The private data is also treated as normal, nontransactional data and placed in the L1 cache in case of modification. If an address is modified more than once, the log may store redundant copies of addresses and data. We can improve the efficiency of the log by keeping track of the types of accesses we make to transactional data. To reduce these redundancies, we added a filter cache. The simplest implementation of the filter cache, the static filter cache keeps track of the addresses that a transaction modifies, so only the first modification of an address is written in the log. A more complicated implementation of the filter cache, the dynamic filter cache, filters addresses that have been read and written. By using the dynamic filter cache, we ensure that only modified data that we have read first will be saved in the log. There are three versions of the logging scheme: No filter cache, static filter cache, and dynamic filter cache.

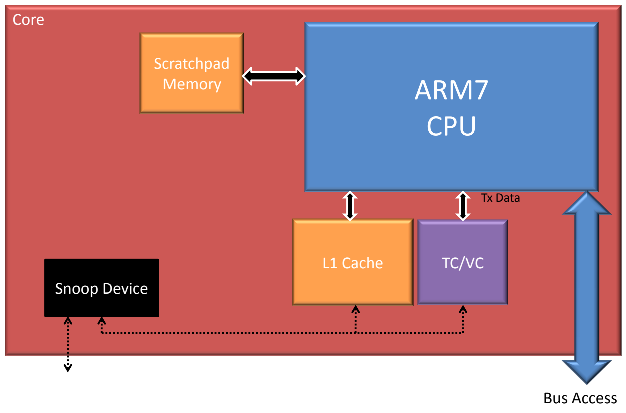

Figure 1: Architecture overview of one core.

2.1 Logging - No filter cache

The traditional logging scheme uses a log stored on the scratch pad memory to store private data. The log operates as a stack. Each time we need to save private data, we save the data on the scratch pad memory at a saved index, and increase the index.

When using the logging scheme, we only write private transactional data to the log. The shared transactional data is still written to the transactional cache, as in the original data scheme. We save the original value of the private data to the log only when the data is modified. In case of a commit, we discard the data in the log by resetting the log index to 0. In case of an abort, we write all of the data on the log back to memory, starting with the most recently written private data.

2.2 Static Filter Cache

Applications that have a large amount of private, transactional data can easily overflow

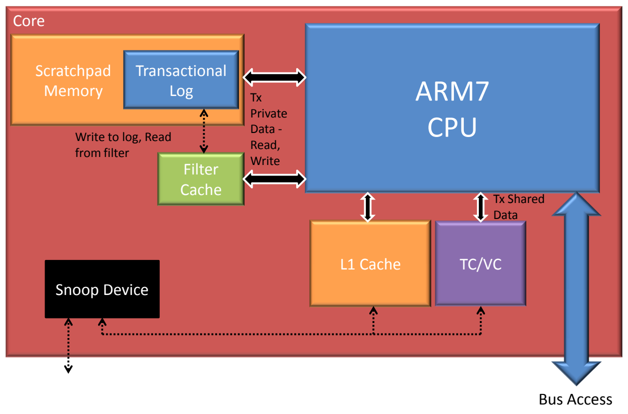

Figure 2: Architecture overview including the transactional log and filter cache.

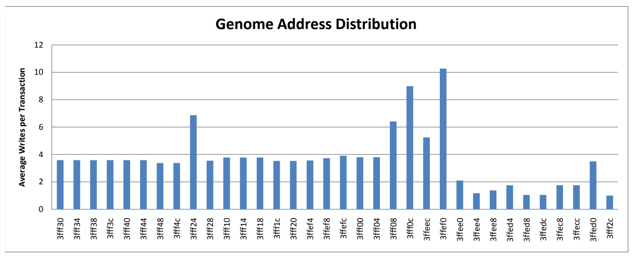

Figure 3: Address Distribution for Genome benchmark.

the scratchpad memory. If an address is already written to the log, however, we do not need to write the address and corresponding data to the log again because we only need an address and its original data in case of an abort. If an application writes to a private address multiple times, we could potentially save space in the log by only saving an address once. Figure 3 shows an address distribution for the genome benchmark. Since genome writes to multiple private addresses multiple times, a log filtering method could save a large amount of space on the scratch pad memory.

The filter cache is a cache that only stores addresses that we write to the log; it does not store any data. We use the filter cache to 'filter' all of the log entries. When we store in address in the log, we also store that address in the filter cache. Each time we want to update the log with an address and data, we check the filter cache for that address. If the address is not in the filter cache then we update the log. If the address is in the filter cache we do not update the log. Since the original data is the data the we will save to the log the first time the address is written to, we do not need to save the address and its modified data multiple times. If the transaction commits or aborts, we invalidate all lines in the filter cache.

2.3 Dynamic Filter Cache

The dynamic filter cache only saves addresses that are read from memory. If we only write data to an address during a transaction, we do not need to save that address and data because the data is not used by other data during a transaction. Therefore, we only need to store addresses that have been read and are then written later on in the transaction. The scheme for saving addresses that are read and then written to the log is slightly more complicated than the static filter cache.

In the dynamic filter cache scheme, we use two bits to identify the data stored in a cache line. A dirty bits state of 01 indicates that a line is invalid. A state of 00 indicates that the line has been read, but not written. A state of 10 indicates that the line has been written to the log.

If we read an address, we need to check if the address is in the filter cache. If it is, then we do not need to do anything else. If the address is not already in the filter cache, we must check to see if the line in the filter cache that the current data will be replacing contains valid data. If the line in the filter cache contains valid data, then we must check the line state. If the line state is 00 we must force update the log: write the current address and data already in the filter cache to the log before we replace it with the new data. We need to save the data to the log because if the data that we have just evicted is written to and then read again, we will save incorrect data to the log. If the line state is 10, then we can overwrite the line and save the new address in the filter cache, with state 00.

If we write an address, we also need to check if the address is in the filter cache. If we have a cache hit, then we need to save the address and data to the log and change the line state in the filter cache to 10. If we have a cache miss, then we either had to force update the log with the data or we have not read the data, so we do not need to save it. Figure 4 shows a flow chart of the dynamic filter cache scheme.

2.4 Architecture Overview

The filter cache is a direct mapped cache. The direct mapped cache provides us with the ability to quickly access addresses. A downfall of using a direct mapped cache as a filter cache is line replacement. If an address has the same tag as an address that is already in the filter cache, the line will be replaced when the filter cache probably is not full, so we are not using the full filter cache. To solve this problem, we could use a fully associative cache. The fully associative cache would allow us to fill the filter cache completely before any line is replaced. A fully associative cache, however, does not allow for a fast search and access time, which very important to find addresses in our logging scheme.

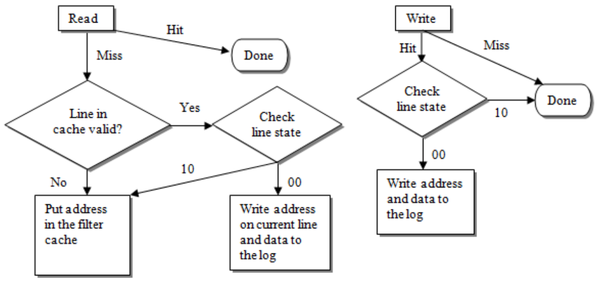
Read
Hit
Done
Miss
Line in cache valid?
Yes
Check line state
No
10
00
Put address in the filter cache
Write address on current line and data to the log
Write
Hit
Miss
Check line state
Done
10
00
Write address and data to the log

Figure 4: Scheme for writing data in the log when using the dynamic filter cache.

The transactional logging and filter cache scheme requires a few additions to the overall architecture of the core, as seen in Figure 2. The transactional log is located on the scratchpad memory, which still receives both read and write signals from the CPU. We also note that only transactional shared data is now written to the transactional cache and private transactional data is written to the log on the scratchpad memory. We have also added a cache, the filter cache, to the architecture. The signals between the filter cache and the log differ, depending on whether or not the filter cache policy is static or dynamic. If we are using the static filter cache, then the filter cache receives the write signal from the CPU and checks if the address is a hit or a miss. If the address is a miss, then the filter cache stores the address and sends the private data to be stored on the transactional log. In the case of the dynamic filter cache, private data reads and writes are sent through the filter cache to the log. The read and write scheme and cases for which the address and data are sent to the log can be seen in Figure 4. If the filter cache size is 0 the transactional log receives read and write requests. All read and write requests sent through the log and filter cache are also sent to the L1 cache.

2.5 Related Work

LogTM-SE [2] is also a transactional logging scheme. LogTM-SE provides a logging scheme that stores modified data on a log, resents data pointers on a commit and writes the log back to memory on an abort. Our logging scheme uses also uses a log that stores modified data and provides a fast commit. Our logging scheme also uses the implementation of the filter cache to ensure that the only data stored on the log is data that needs to be restored if transaction aborts. LogTM-SE offers a speedup in performance as well as a fast means in recovery of data in case of an abort. The transactinal logging method that we have implemented speeds up performance as well, but it also offers a reduction in the amount of overflows in the transactional cache as well as improvements in energy efficiency.

3 Remaining Work

3.1 Log Overflow

Currently, if there is a log overflow in the simulator we send an assert and ask that the user increase the size of the scratchpad memory. Since the scratchpad memory has limited size, this is not an optimal solution. There are several solutions in case of an overflow on the log: write the private data into the transactional cache or extend the log into memory.

In case of a log overflow, we could put log data in the transactional cache. If the log overflows, then we check the transactional cache for space. If there is room in the transactional cache, then we can write the private data in the transactional cache, and overflow when the transactional cache overflows. If there is no more room in the transactional cache, then we can trigger a transactional overflow and serialize the transaction.

We could also extend the log into memory in case of a log overflow. We would extend the log into memory by creating a pointer to a location in memory where the log would be stored. Extending the log to memory would be more expensive in terms of having to read and write to memory every time we update the log, but since we reset the log every time we abort or commit this expense has the potential to be small. Extending the log would also not require us to serialize, as saving the data in the transactional cache in case of an overflow would.

3.2 Bugs

We encountered the majority of the bugs in the simulator when using the victim cache configuration. Since transactional data is stored in the L1 cache when using the victim cache, there were several extra cases that we needed to consider in case of an L1 cache miss. Problems occurred when the data that we were trying to evict from L1 cache was marked as transactional. Different problems occurred depending on whether or not the transactional data in the L1 cache was modified or unmodified.

3.2.1 Removing Unmodified Transactional Data

When using the victim cache configuration, we cannot just remove transactional data from the L1 cache. We need to save the transactional data in the victim cache. In the simulator, the methods `read_trans()` and `write_trans()` handle the removal of L1 cache lines marked as transactional. When we use the log, however, we never invoke `read_trans()` or `write_trans()` when we encounter private data, because `read_trans()` and `write_trans()` will mark private data as transactional, in which case using the log becomes pointless.

The solution to removing unmodified transactional data from the L1 cache is to handle the problem outside of the `read_trans()` and `write_trans()` methods. We chose to place the solution in `mem_ctrl.cpp`. If there is an L1 cache miss with private data, we first check to see if the line that the private data will replace is transactional. If the line is transactional, then we check for space in the victim cache. If there is space in the victim, then we write the L1 line that we want to replace in the victim cache, and replace the line. If there is not space in the victim cache, we read the private data from memory and leave the line in the L1 alone.

3.2.2 Removing Transactional Modified data

Our solution to removing transactional data from the L1 cache triggered another problem: removing transactional data from the L1 cache that has been modified.

We did not realize this problem until we tried to run a simulation with a 1-way 1KB data cache. By replacing lines in the L1 cache, we were saving modified transactional data as a commit line in the victim cache. These modified transactional lines were being written back to memory during a transaction. If the transaction needed to abort, we were restoring the modified data instead of the original data. In cases of simple benchmarks such as count, restoring the modified data caused count to skip counting numbers. In case of more complicated benchmarks such as genome this problem caused us to read a bad address from memory.

Our solution was to leave the modified transactional data in the L1 cache. Instead, in the case of an L1 miss with private data, we write the private data in the victim cache and mark the private data and normal, commit transactional and exclusive. This solution created a few more problems. Luckily, the simulator already handles victim cache hits on normal transactional data. We still needed to handle victim cache hits for private data in a memory write. If there is a victim cache hit on private data, then we write the data in the commit cache and set the dirty bits to normal and modified.

4 Results

As shown in Figure 6, the logging scheme greatly reduces the amount of overflows for the vanilla configuration. There is an increase in abort rate for all benchmarks (seen in Figure 5. The increase in abort rate may be due to the decrease in overflows, as the longer a transaction lasts, the larger chance it has of being interrupted by another transaction.

Execution Cycles, shown in Figure 7, drop in the case of skiplist 4 core victim configuration. They increase in patricia 4core victim, with a small dynamic filter cache. This change could be due to a larger amount of forced updates for the dynamic filter cache (Figure 10, because each forced update causes another read to memory for a log update. This read causes increased cycles. Note: Read-filter refers to the dynamic filter cache. Write-filter refers to the static filter cache.

The transactional log and filter cache statistics shows a decrease in writes to the log when the filter cache is used, as seen in Figure 9.

References

[1] Cesare Ferri, Samantha Wood, Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Embedded-tm: Energy and complexity-effective hardware transactional memory for embedded multicore systems. *Journal of Parallel and Distributed Computing*, In Press, Corrected Proof:–, 2010.

[2] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *In HPCA 13*, pages 261–272, 2007.

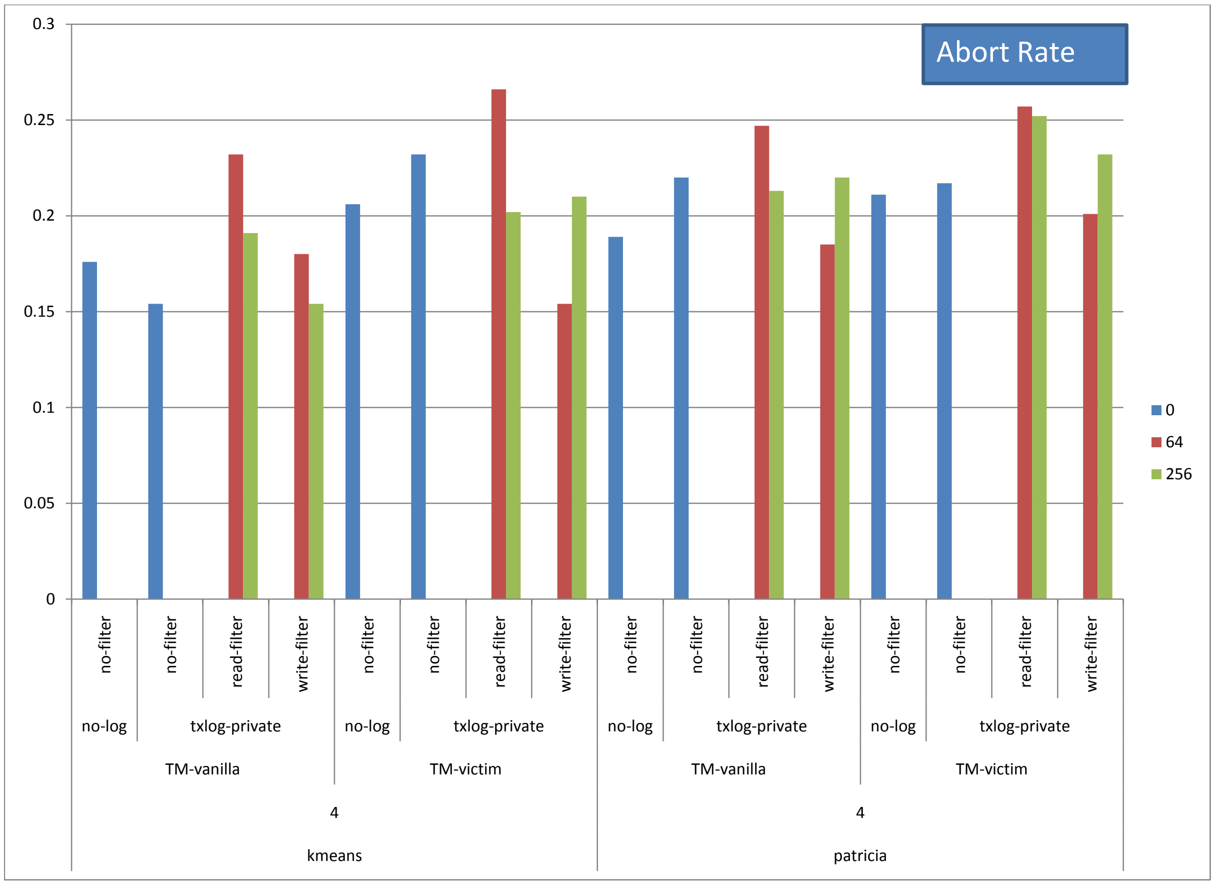

Figure 5: Abort Rate for Patricia and Kmeans. Skiplist has 0 aborts.

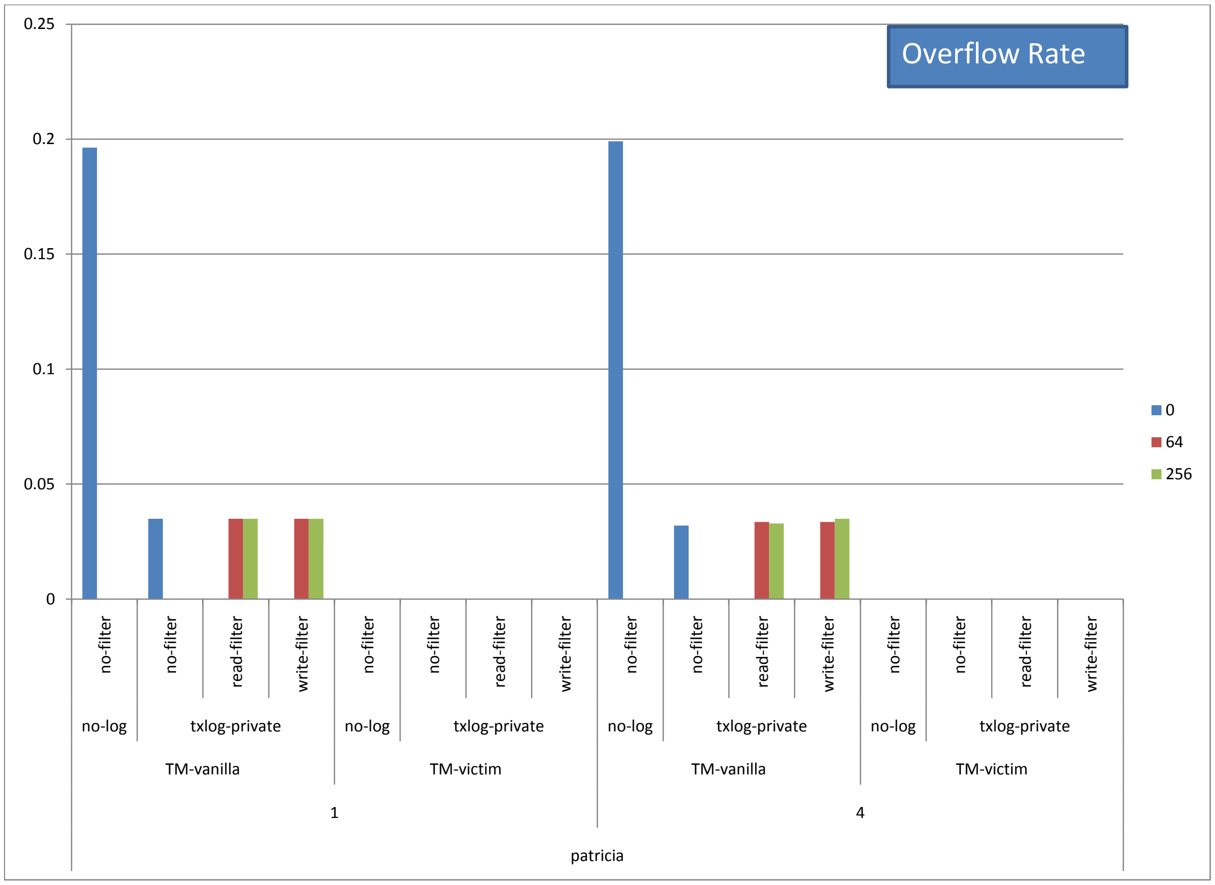

Figure 6: Overflow Rate for Patricia. Neither Kmeans nor skiplist overflowed.

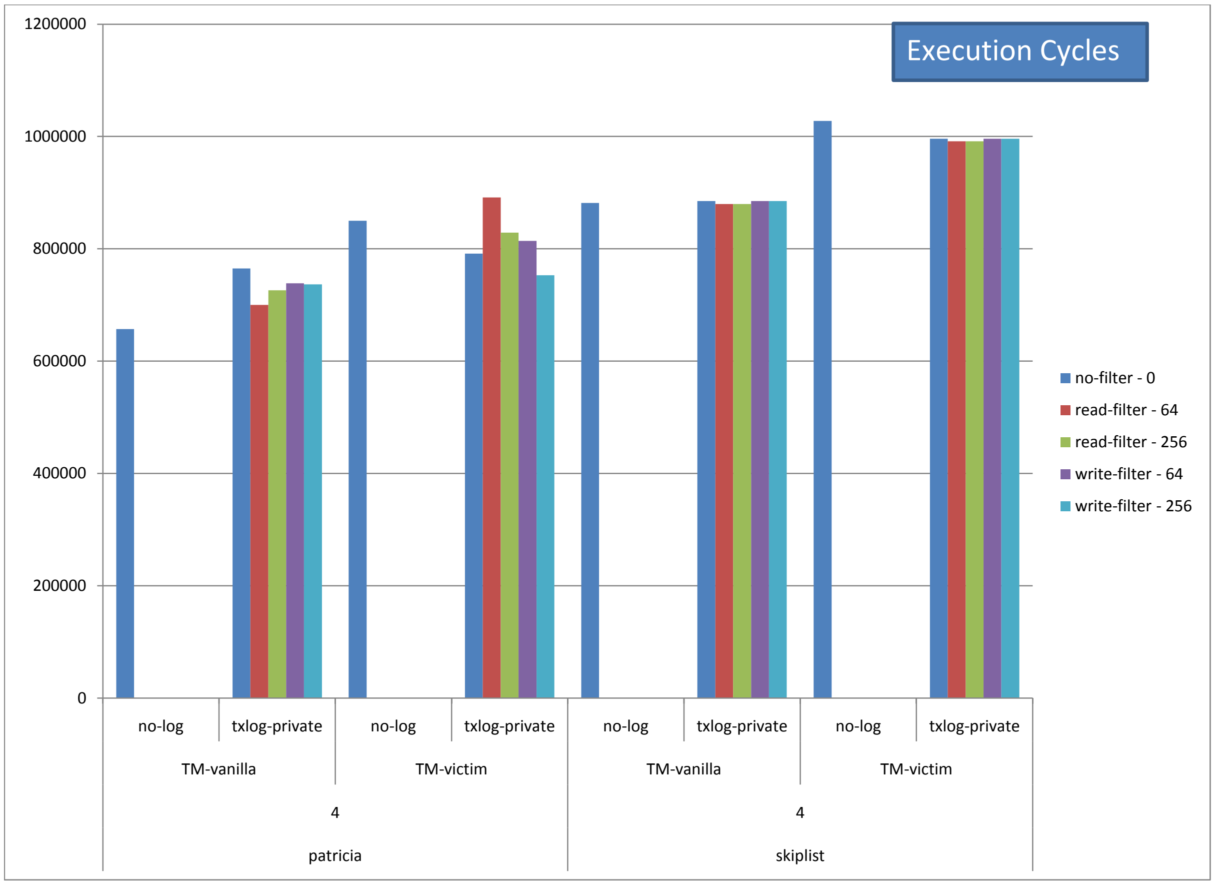

Figure 7: Execution cycles for patricia and skiplist.

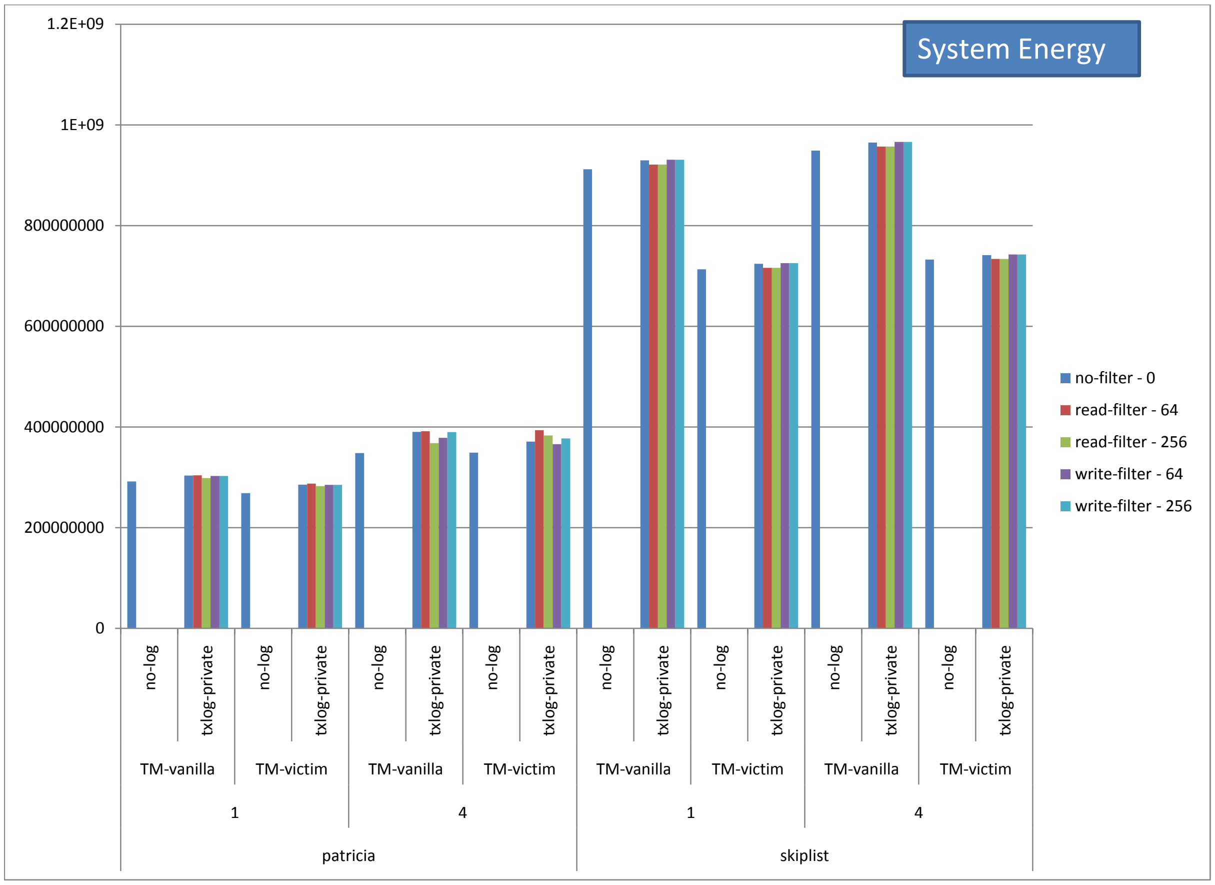

Figure 8: System Energy for patricia and skiplist.

Cores	Config	Log	Filter	Filter Size	Energy	Cycles
1	Victim	no-log	no-filter	0	3.83E09	28546668
1	Victim	txlog	no-filter	0	3.87E09	28593698
1	Victim	txlog	dynamic-filter	64	3.87E09	28577696
1	Victim	txlog	dynamic-filter	256	3.85E09	28545099
1	Victim	txlog	static-filter	64	3.87E09	28541282
1	Victim	txlog	static-filter	256	3.85E09	28543454
1	Vanilla	no-log	no-filter	0	6.26E09	28329353
1	Vanilla	txlog	no-filter	0	6.28E09	28559222
1	Vanilla	txlog	dynamic-filter	64	6.29E09	28537394
1	Vanilla	txlog	dynamic-filter	256	6.27E09	2850472
1	Vanilla	txlog	static-filter	64	6.28E09	28541282
1	Vanilla	txlog	static-filter	256	6.26E09	28504994
4	Victim	no-log	no-filter	0	4.53E09	9294252
4	Victim	txlog	no-filter	0	4.59E09	9300128
4	Victim	txlog	dynamic-filter	64	4.6E09	9313659
4	Victim	txlog	dynamic-filter	256	4.57E09	9283152
4	Victim	txlog	static-filter	64	4.58E09	9290102
4	Victim	txlog	static-filter	256	4.56E09	9284104
4	Vanilla	no-log	no-filter	0	7.41E09	9192030
4	Vanilla	txlog	no-filter	0	7.45E09	9283213
4	Vanilla	txlog	dynamic-filter	64	7.48E09	9318782
4	Vanilla	txlog	dynamic-filter	256	7.43E09	9274955
4	Vanilla	txlog	static-filter	64	7.45E09	9277301
4	Vanilla	txlog	static-filter	256	7.43E09	9275166

Table 1: Kmeans system energy and execution cycles.

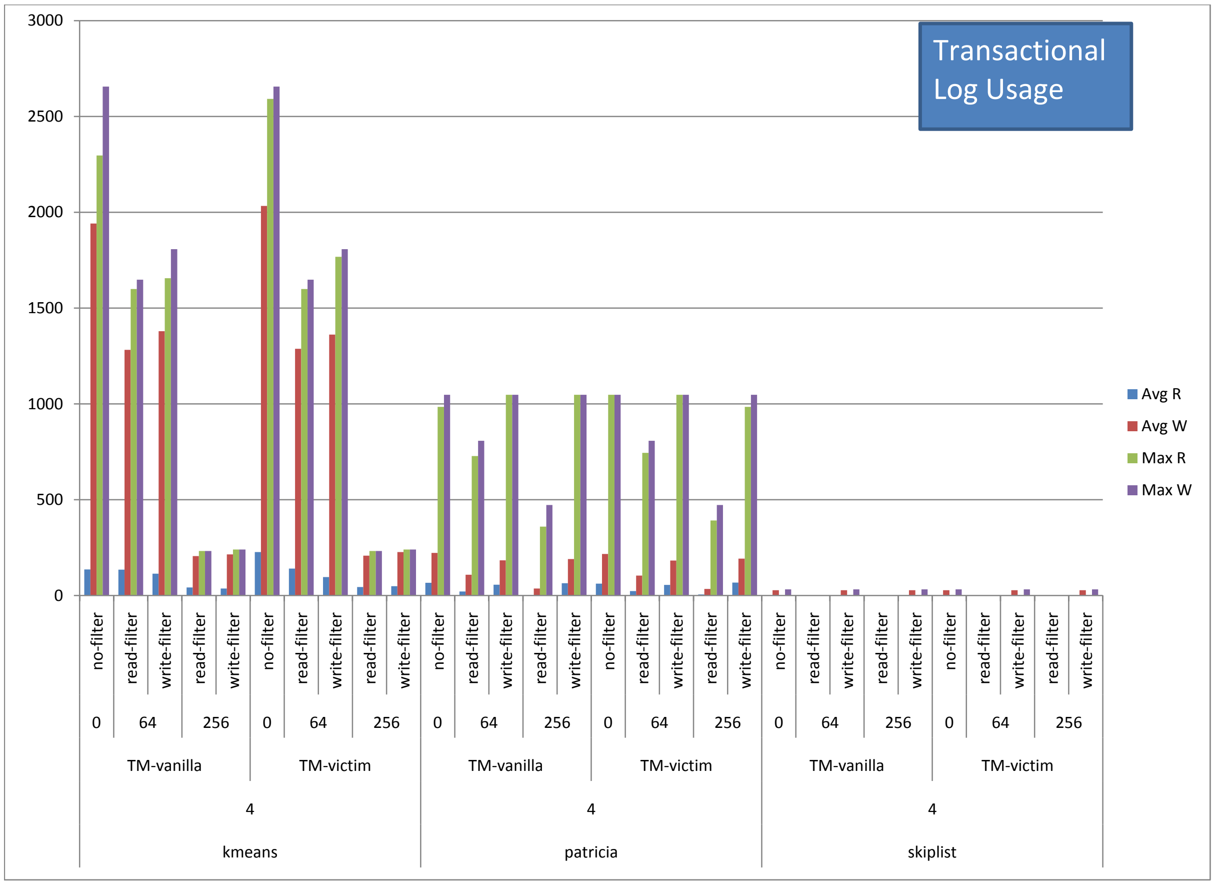

Figure 9: Average and Maximum reads and writes on the log per transaction.

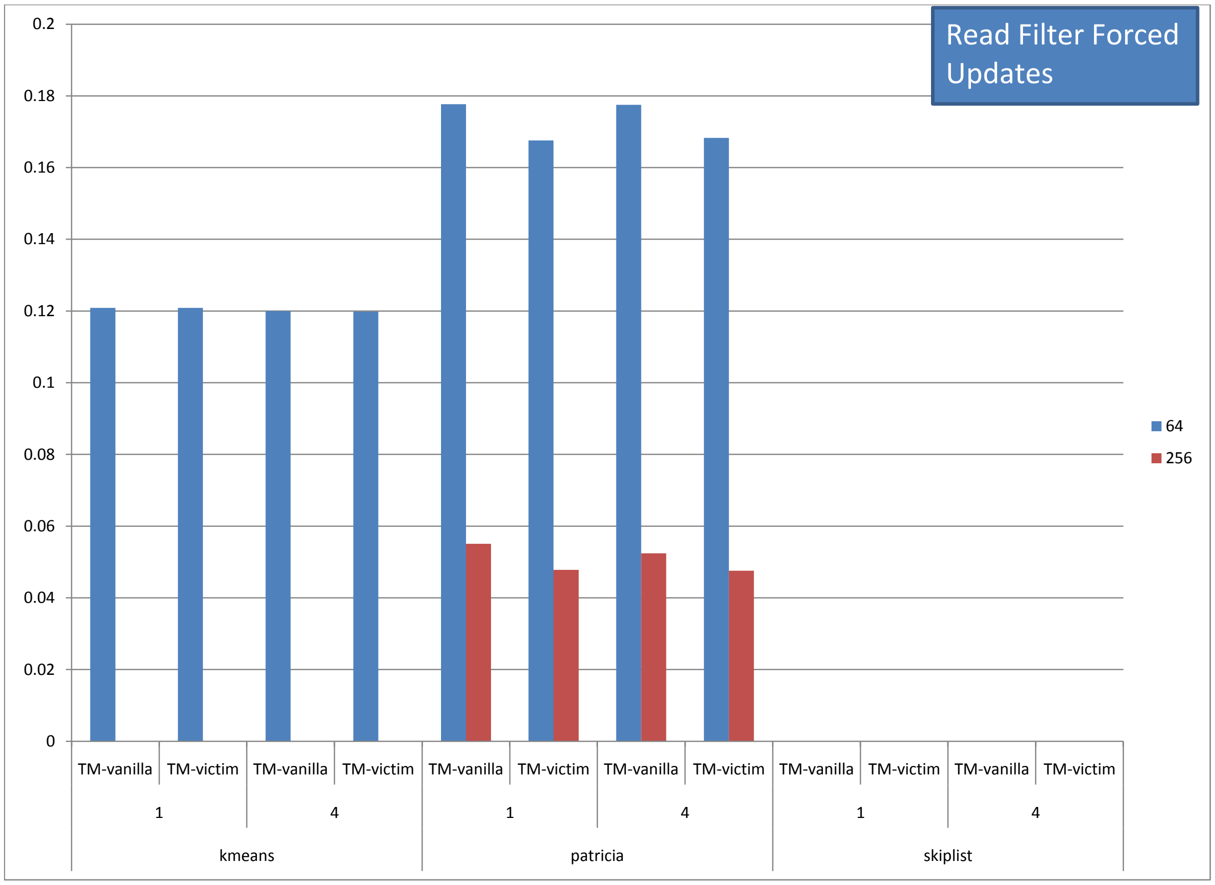

Figure 10: Percentage of Forced Updates needed in the dynamic filter cache.

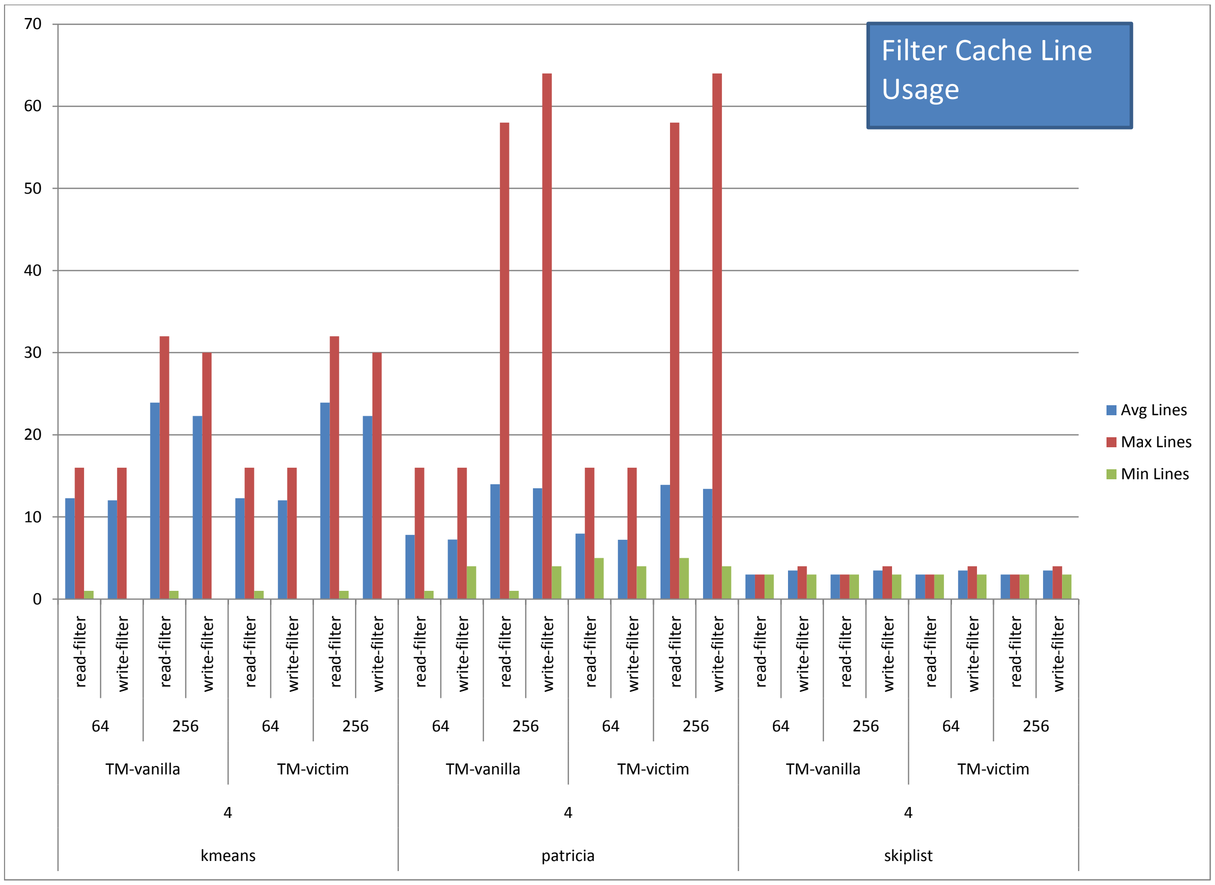

Figure 11: Maximum, Average, and Minimum lines used in the filter cache per transaction.

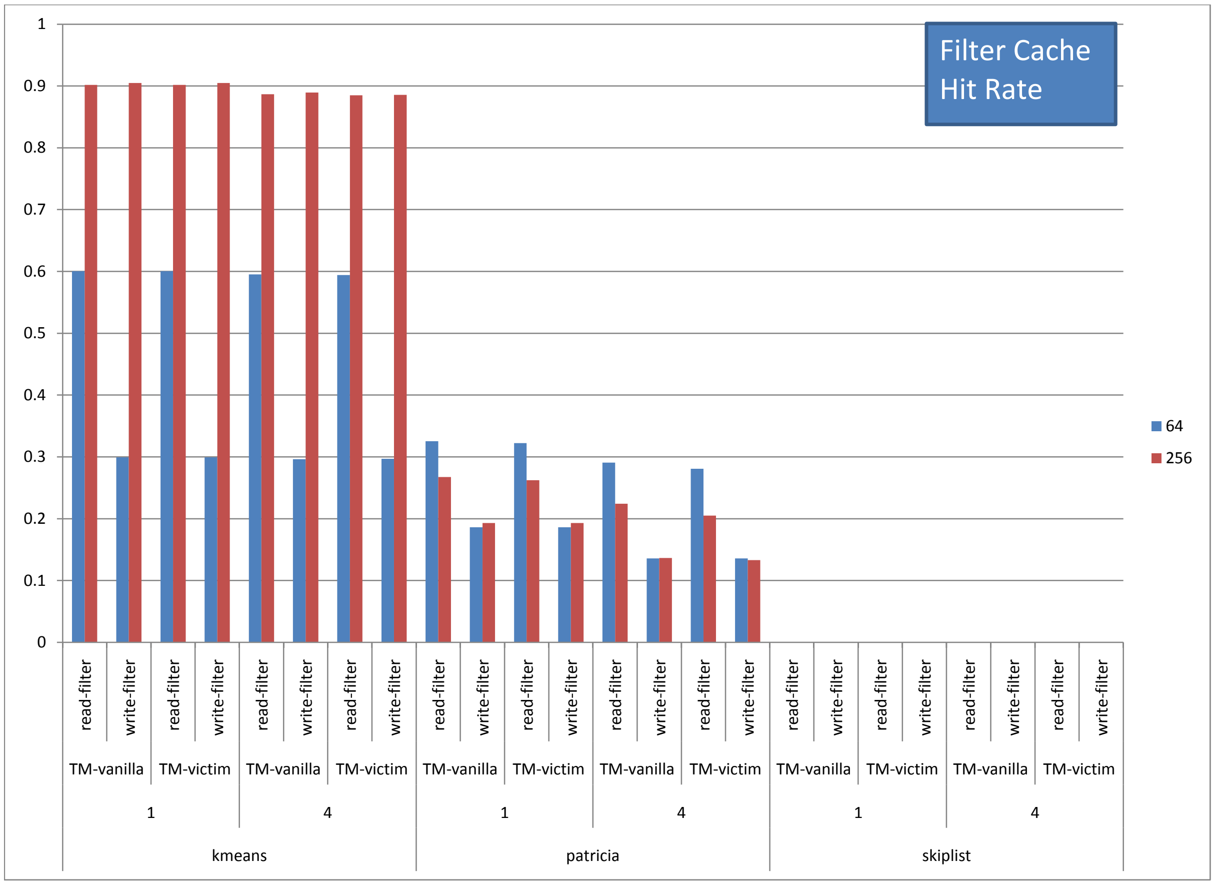

Figure 12: Filter cache hit rate for dynamic and static filter cache.